Python Pandas - Sorting

There are two kinds of sorting available in Pandas. They are −

- By label
- By Actual Value

Let us consider an example with an output.

```
import pandas as pd
import numpy as np

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

Its **output** is as follows −

```
        col2       col1
1  -2.063177   0.537527
4   0.142932  -0.684884
6   0.012667  -0.389340
2  -0.548797   1.848743
3  -1.044160   0.837381
5   0.385605   1.300185
9   1.031425  -1.002967
8  -0.407374  -0.435142
0   2.237453  -1.067139
7  -1.445831  -1.701035
```

In **unsorted_df**, the **labels** and the **values** are unsorted. Let us see how these can be sorted.

By Label

Using the **sort_index()** method, by passing the axis arguments and the order of sorting, DataFrame can be sorted. By default, sorting is done on row labels in ascending order.

```
import pandas as pd
import numpy as np

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

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

Its **output** is as follows −

```
        col2       col1
0   0.208464   0.627037
1   0.641004   0.331352
2  -0.038067  -0.464730
3  -0.638456  -0.021466
4   0.014646  -0.737438
5  -0.290761  -1.669827
6  -0.797303  -0.018737
7   0.525753   1.628921
8  -0.567031   0.775951
9   0.060724  -0.322425
```

Order of Sorting

By passing the Boolean value to ascending parameter, the order of the sorting can be controlled. Let us consider the following example to understand the same.

```
import pandas as pd
import numpy as np

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

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

Its **output** is as follows −

```
         col2        col1
9    0.825697    0.374463
8   -1.699509    0.510373
7   -0.581378    0.622958
6   -0.202951    0.954300
5   -1.289321   -1.551250
4    1.302561    0.851385
3   -0.157915   -0.388659
2   -1.222295    0.166609
1    0.584890   -0.291048
0    0.668444   -0.061294
```

Sort the Columns

By passing the axis argument with a value 0 or 1, the sorting can be done on the column labels. By default, axis=0, sort by row. Let us consider the following example to understand the same.

```
import pandas as pd
import numpy as np

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

```
   mns = ['col2','col1'])

sorted_df=unsorted_df.sort_index(axis=1)

print sorted_df
```

Its **output** is as follows −

```
         col1        col2
1   -0.291048    0.584890
4    0.851385    1.302561
6    0.954300   -0.202951
2    0.166609   -1.222295
3   -0.388659   -0.157915
5   -1.551250   -1.289321
9    0.374463    0.825697
8    0.510373   -1.699509
0   -0.061294    0.668444
7    0.622958   -0.581378
```

By Value

Like index sorting, **sort_values()** is the method for sorting by values. It accepts a 'by' argument which will use the column name of the DataFrame with which the values are to be sorted.

```
import pandas as pd
import numpy as np

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
```

Its **output** is as follows −

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

Observe, col1 values are sorted and the respective col2 value and row index will alter along with col1. Thus, they look unsorted.

'by' argument takes a list of column values.

```
import pandas as pd
import numpy as np

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
```

Its **output** is as follows −

```
  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.

```
import pandas as pd
import numpy as np

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
```

Its **output** is as follows −

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

Python Pandas - Working with Text Data

we will discuss the string operations with our basic Series/Index. In the subsequent chapters, we will learn how to apply these string functions on the DataFrame.

Pandas provides a set of string functions which make it easy to operate on string data. Most importantly, these functions ignore (or exclude) missing/NaN values.

Almost, all of these methods work with Python string functions (refer: https://docs.python.org/3/library/stdtypes.html#string-methods). So, convert the Series Object to String Object and then perform the operation.

Let us now see how each operation performs.

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.

Let us now create a Series and see how all the above functions work.

```
import pandas as pd
import numpy as np

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t', np.nan,
'1234','SteveSmith'])

print s
```

Its **output** is as follows −

```
0              Tom
1     William Rick
2             John
3          Alber@t
4              NaN
5             1234
6      Steve Smith
dtype: object
```

lower()

```
import pandas as pd
import numpy as np

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t', np.nan,
'1234','SteveSmith'])

print s.str.lower()
```

Its **output** is as follows −

```
0              tom
1     william rick
2             john
```

```
3          alber@t
4              NaN
5             1234
6      steve smith
dtype: object
```

upper()

```
import pandas as pd
import numpy as np

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t', np.nan,
'1234','SteveSmith'])

print s.str.upper()
```

Its **output** is as follows −

```
0              TOM
1     WILLIAM RICK
2             JOHN
3          ALBER@T
4              NaN
5             1234
6      STEVE SMITH
dtype: object
```

len()

```
import pandas as pd
import numpy as np

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t', np.nan,
'1234','SteveSmith'])
print s.str.len()
```

Its **output** is as follows −

```
0     3.0
1    12.0
2     4.0
3     7.0
4     NaN
5     4.0
6    10.0
dtype: float64
```

strip()

```
import pandas as pd
import numpy as np
s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])
print s
print ("After Stripping:")
print s.str.strip()
```

Its **output** is as follows −

```
0             Tom
```

```
1    William Rick
2            John
3         Alber@t
dtype: object

After Stripping:
0             Tom
1    William Rick
2            John
3         Alber@t
dtype: object
```

split(pattern)

```
import pandas as pd
import numpy as np
s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])
print s
print ("Split Pattern:")
print s.str.split(' ')
```

Its **output** is as follows −

```
0             Tom
1    William Rick
2            John
3         Alber@t
dtype: object

Split Pattern:
0    [Tom, , , , , , , , , , ]
1    [, , , , , William, Rick]
2    [John]
3    [Alber@t]
dtype: object
```

cat(sep=pattern)

```
import pandas as pd
import numpy as np

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print s.str.cat(sep='_')
```

Its **output** is as follows −

```
Tom _ William Rick_John_Alber@t
```

get_dummies()

```
import pandas as pd
import numpy as np

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print s.str.get_dummies()
```

Its **output** is as follows −

```
   William Rick   Alber@t    John    Tom
0              0         0       0      1
1              1         0       0      0
2              0         0       1      0
3              0         1       0      0
```

contains ()

```
import pandas as pd

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print s.str.contains(' ')
```

Its **output** is as follows −

```
0    True
1    True
2    False
3    False
dtype: bool
```

replace(a,b)

```
import pandas as pd
s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])
print s
print ("After replacing @ with $:")
print s.str.replace('@','$')
```

Its **output** is as follows −

```
0    Tom
1    William Rick
2    John
3    Alber@t
dtype: object

After replacing @ with $:
0    Tom
1    William Rick
2    John
3    Alber$t
dtype: object
```

repeat(value)

```
import pandas as pd

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print s.str.repeat(2)
```

Its **output** is as follows −

```
0    Tom              Tom
1    William Rick    William Rick
```

```
2                   JohnJohn
3                   Alber@tAlber@t
dtype: object
```

count(pattern)

```
import pandas as pd

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print ("The number of 'm's in each string:")
print s.str.count('m')
```

Its **output** is as follows −

```
The number of 'm's in each string:
0     1
1     1
2     0
3     0
```

startswith(pattern)

```
import pandas as pd

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print ("Strings that start with 'T':")
print s.str. startswith ('T')
```

Its **output** is as follows −

```
0   True
1   False
2   False
3   False
dtype: bool
```

endswith(pattern)

```
import pandas as pd
s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])
print ("Strings that end with 't':")
print s.str.endswith('t')
```

Its **output** is as follows −

```
Strings that end with 't':
0   False
1   False
2   False
3   True
dtype: bool
```

find(pattern)

```
import pandas as pd

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])
```

```
print s.str.find('e')
```

Its **output** is as follows −

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

"-1" indicates that there no such pattern available in the element.

findall(pattern)

```
import pandas as pd

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print s.str.findall('e')
```

Its **output** is as follows −

```
0 []
1 []
2 []
3 [e]
dtype: object
```

Null list([ ]) indicates that there is no such pattern available in the element.

swapcase()

```
import pandas as pd

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t'])
print s.str.swapcase()
```

Its **output** is as follows −

```
0   tOM
1   wILLIAM rICK
2   jOHN
3   aLBER@T
dtype: object
```

islower()

```
import pandas as pd

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t'])
print s.str.islower()
```

Its **output** is as follows −

```
0   False
1   False
2   False
3   False
dtype: bool
```

isupper()

```
import pandas as pd

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t'])

print s.str.isupper()
```

Its **output** is as follows −

```
0   False
1   False
2   False
3   False
dtype: bool
```

isnumeric()

```
import pandas as pd

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t'])

print s.str.isnumeric()
```

Its **output** is as follows −

```
0   False
1   False
2   False
3   False
dtype: bool
```

Python Pandas - Options and Customization

Pandas provide API to customize some aspects of its behavior, display is being mostly used.

The API is composed of five relevant functions. They are −

- get_option()
- set_option()
- reset_option()
- describe_option()
- option_context()

Let us now understand how the functions operate.

get_option(param)

get_option takes a single parameter and returns the value as given in the output below −

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.

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

Its **output** is as follows −

```
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.

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

Its **output** is as follows −

```
20
```

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

set_option(param,value)

set_option takes two arguments and sets the value to the parameter as shown below −

display.max_rows

Using **set_option()**, we can change the default number of rows to be displayed.

```
import pandas as pd

pd.set_option("display.max_rows",80)

print pd.get_option("display.max_rows")
```

Its **output** is as follows −

```
80
```

display.max_columns

Using **set_option()**, we can change the default number of rows to be displayed.

```
import pandas as pd

pd.set_option("display.max_columns",30)

print pd.get_option("display.max_columns")
```

Its **output** is as follows −

```
30
```

reset_option(param)

reset_option takes an argument and sets the value back to the default value.

display.max_rows

Using reset_option(), we can change the value back to the default number of rows to be displayed.

```
import pandas as pd

pd.reset_option("display.max_rows")
print pd.get_option("display.max_rows")
```

Its **output** is as follows −

```
60
```

describe_option(param)

describe_option prints the description of the argument.

display.max_rows

Using reset_option(), we can change the value back to the default number of rows to be displayed.

```
import pandas as pd
pd.describe_option("display.max_rows")
```

Its **output** is as follows −

```
display.max_rows : int
   If max_rows is exceeded, switch to truncate view. Depending on
   'large_repr', objects are either centrally truncated or
printed as
   a summary view. 'None' value means unlimited.

   In case python/IPython is running in a terminal and
`large_repr`
   equals 'truncate' this can be set to 0 and pandas will auto-
detect
   the height of the terminal and print a truncated object which
fits
   the screen height. The IPython notebook, IPython qtconsole, or
   IDLE do not run in a terminal and hence it is not possible to
do
   correct auto-detection.
   [default: 60] [currently: 60]
```

option_context()

option_context context manager is used to set the option in **with statement** temporarily. Option values are restored automatically when you exit the **with block** −

display.max_rows

Using option_context(), we can set the value temporarily.

```
import pandas as pd
with pd.option_context("display.max_rows",10):
   print(pd.get_option("display.max_rows"))
   print(pd.get_option("display.max_rows"))
```

Its **output** is as follows −

```
10
10
```

See, the difference between the first and the second print statements. The first statement prints the value set by **option_context()** which is temporary within the **with context** itself. After the **with context**, the second print statement prints the configured value.

Frequently used Parameters

Sr.No	Parameter & Description
1	**display.max_rows** Displays maximum number of rows to display
2	**2 display.max_columns** Displays maximum number of columns to display
3	**display.expand_frame_repr** Displays DataFrames to Stretch Pages
4	**display.max_colwidth** Displays maximum column width
5	**display.precision** Displays precision for decimal numbers

Python Pandas - Indexing and Selecting Data

we will discuss how to slice and dice the date and generally get the subset of pandas object.

The Python and NumPy indexing operators "[ ]" and attribute operator "." provide quick and easy access to Pandas data structures across a wide range of use cases. However, since the type of the data to be accessed isn't known in advance, directly using standard operators has some optimization limits. For production code, we recommend that you take advantage of the optimized pandas data access methods explained in this chapter.

Pandas now supports three types of Multi-axes indexing; the three types are mentioned in the following table −

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**. When slicing, the start bound is also included. Integers are valid labels, but they refer to the label and not the position.

.loc() has multiple access methods like −

- A single scalar label
- A list of labels
- A slice object
- A Boolean array

loc takes two single/list/range operator separated by ','. The first one indicates the row and the second one indicates columns.

Example 1

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

df = pd.DataFrame(np.random.randn(8, 4),
index = ['a','b','c','d','e','f','g','h'], columns = ['A', 'B',
'C', 'D'])

#select all rows for a specific column
print df.loc[:,'A']
```

Its **output** is as follows −

```
a   0.391548
b  -0.070649
c  -0.317212
d  -2.162406
e   2.202797
f   0.613709
g   1.050559
h   1.122680
Name: A, dtype: float64
```

Example 2

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

df = pd.DataFrame(np.random.randn(8, 4),
index = ['a','b','c','d','e','f','g','h'], columns = ['A', 'B',
'C', 'D'])

# Select all rows for multiple columns, say list[]
print df.loc[:,['A','C']]
```

Its **output** is as follows −

```
            A           C
a    0.391548    0.745623
b   -0.070649    1.620406
c   -0.317212    1.448365
d   -2.162406   -0.873557
e    2.202797    0.528067
f    0.613709    0.286414
g    1.050559    0.216526
h    1.122680   -1.621420
```

Example 3

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

df = pd.DataFrame(np.random.randn(8, 4),
index = ['a','b','c','d','e','f','g','h'], columns = ['A', 'B',
'C', 'D'])

# Select few rows for multiple columns, say list[]
print df.loc[['a','b','f','h'],['A','C']]
```

Its **output** is as follows −

```
           A          C
a   0.391548   0.745623
b  -0.070649   1.620406
f   0.613709   0.286414
h   1.122680  -1.621420
```

Example 4

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

df = pd.DataFrame(np.random.randn(8, 4),
index = ['a','b','c','d','e','f','g','h'], columns = ['A', 'B',
'C', 'D'])

# Select range of rows for all columns
print df.loc['a':'h']
```

Its **output** is as follows −

```
            A           B          C          D
a    0.391548   -0.224297   0.745623   0.054301
b   -0.070649   -0.880130   1.620406   1.419743
c   -0.317212   -1.929698   1.448365   0.616899
d   -2.162406    0.614256  -0.873557   1.093958
e    2.202797   -2.315915   0.528067   0.612482
f    0.613709   -0.157674   0.286414  -0.500517
g    1.050559   -2.272099   0.216526   0.928449
h    1.122680    0.324368  -1.621420  -0.741470
```

Example 5

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

df = pd.DataFrame(np.random.randn(8, 4),
index = ['a','b','c','d','e','f','g','h'], columns = ['A', 'B',
'C', 'D'])

# for getting values with a boolean array
print df.loc['a']>0
```

Its **output** is as follows −

```
A   False
B   True
C   False
D   False
Name: a, dtype: bool
```

.iloc()

Pandas provide various methods in order to get purely integer based indexing. Like python and numpy, these are **0-based** indexing.

The various access methods are as follows −

- An Integer
- A list of integers
- A range of values

Example 1

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

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

# select all rows for a specific column
print df.iloc[:4]
```

Its **output** is as follows −

```
            A          B          C          D
0    0.699435   0.256239  -1.270702  -0.645195
1   -0.685354   0.890791  -0.813012   0.631615
2   -0.783192  -0.531378   0.025070   0.230806
3    0.539042  -1.284314   0.826977  -0.026251
```

Example 2

```
import pandas as pd
import numpy as np

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

# Integer slicing
print df.iloc[:4]
print df.iloc[1:5, 2:4]
```

Its **output** is as follows −

```
            A          B          C          D
0    0.699435   0.256239  -1.270702  -0.645195
1   -0.685354   0.890791  -0.813012   0.631615
2   -0.783192  -0.531378   0.025070   0.230806
3    0.539042  -1.284314   0.826977  -0.026251

            C          D
1   -0.813012   0.631615
2    0.025070   0.230806
3    0.826977  -0.026251
4    1.423332   1.130568
```

Example 3

```
import pandas as pd
import numpy as np

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

# Slicing through list of values
print df.iloc[[1, 3, 5], [1, 3]]
print df.iloc[1:3, :]
print df.iloc[:,1:3]
```

Its **output** is as follows −

```
          B         D
1   0.890791    0.631615
3  -1.284314   -0.026251
5  -0.512888   -0.518930

          A         B         C         D
1  -0.685354    0.890791  -0.813012    0.631615
2  -0.783192   -0.531378   0.025070    0.230806

          B         C
0   0.256239   -1.270702
1   0.890791   -0.813012
2  -0.531378    0.025070
3  -1.284314    0.826977
4  -0.460729    1.423332
5  -0.512888    0.581409
6  -1.204853    0.098060
7  -0.947857    0.641358
```

.ix()

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

Example 1

```
import pandas as pd
import numpy as np

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

# Integer slicing
print df.ix[:4]
```

Its **output** is as follows −

```
          A         B         C         D
0   0.699435   0.256239  -1.270702  -0.645195
1  -0.685354   0.890791  -0.813012   0.631615
2  -0.783192  -0.531378   0.025070   0.230806
3   0.539042  -1.284314   0.826977  -0.026251
```

Example 2

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(8, 4), columns = ['A', 'B',
'C', 'D'])
# Index slicing
print df.ix[:,'A']
```

Its **output** is as follows −

```
0   0.699435
```

```
1   -0.685354
2   -0.783192
3    0.539042
4   -1.044209
5   -1.415411
6    1.062095
7    0.994204
Name: A, dtype: float64
```

Use of Notations

Getting values from the Pandas object with Multi-axes indexing uses the following notation −

Object	Indexers	Return Type
Series	s.loc[indexer]	Scalar value
DataFrame	df.loc[row_index,col_index]	Series object
Panel	p.loc[item_index,major_index, minor_index]	p.loc[item_index,major_index, minor_index]

Note − .iloc() & .ix() applies the same indexing options and Return value.

Let us now see how each operation can be performed on the DataFrame object. We will use the basic indexing operator '[ ]' −

Example 1

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.randn(8, 4), columns = ['A', 'B',
'C', 'D'])
print df['A']
```

Its **output** is as follows −

```
0   -0.478893
1    0.391931
2    0.336825
3   -1.055102
4   -0.165218
5   -0.328641
6    0.567721
7   -0.759399
Name: A, dtype: float64
```

Note − We can pass a list of values to [ ] to select those columns.

Example 2

```
import pandas as pd
```

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

print df[['A','B']]
```

Its **output** is as follows −

```
           A           B
0  -0.478893   -0.606311
1   0.391931   -0.949025
2   0.336825    0.093717
3  -1.055102   -0.012944
4  -0.165218    1.550310
5  -0.328641   -0.226363
6   0.567721   -0.312585
7  -0.759399   -0.372696
```

Example 3

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.randn(8, 4), columns = ['A', 'B',
'C', 'D'])
print df[2:2]
```

Its **output** is as follows −

```
Columns: [A, B, C, D]
Index: []
```

Attribute Access

Columns can be selected using the attribute operator '.'.

Example

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

print df.AStatistical 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.

```
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

```
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.

```
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.

```
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.

```
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

Its **output** is as follows −

```
0   -0.478893
1    0.391931
2    0.336825
3   -1.055102
4   -0.165218
5   -0.328641
6    0.567721
7   -0.759399
Name: A, dtype: float64
```

Python Pandas - 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.

```
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

```
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.

```
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.

```
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.

```
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

Python Pandas - 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.

```
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.

```
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.

```
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.