5.2 Essential Functionality - pandas

In [1]:
```
import numpy as np
import pandas as pd
```

Reindexing

```
Use reindex() function to reindex the dataframe.
- By default values in the new index that do not have corresponding
records in the dataframe are assigned NaN
- We can fill in the missing values by passing a value to the keyword
fill_value.
```

In [2]:
```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
obj
```

Out[2]:
```
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

In [3]:
```
obj2 = obj.reindex(['a', 'b', 'c', 'e'])
obj2
```

Out[3]:
```
a   -5.3
b    7.2
c    3.6
e    NaN
dtype: float64
```

In [ ]:
```

```

In [4]:
```
obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
obj3
```

Out[4]:
```
0      blue
2    purple
4    yellow
dtype: object
```

```
In [5]: obj3.reindex(range(6), method='ffill')
```

```
Out[5]: 0      blue
        1      blue
        2    purple
        3    purple
        4    yellow
        5    yellow
        dtype: object
```

```
In [6]: obj3.reindex(range(6), method='bfill')
```

```
Out[6]: 0      blue
        1    purple
        2    purple
        3    yellow
        4    yellow
        5       NaN
        dtype: object
```

```
In [7]: obj33 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 1, 2])
        obj33
```

```
Out[7]: 0      blue
        1    purple
        2    yellow
        dtype: object
```

```
In [8]: obj33.reindex(range(6), method='ffill')
```

```
Out[8]: 0      blue
        1    purple
        2    yellow
        3    yellow
        4    yellow
        5    yellow
        dtype: object
```

```
In [9]: obj33.reindex(range(6), method='bfill')
```

```
Out[9]: 0      blue
        1    purple
        2    yellow
        3       NaN
        4       NaN
        5       NaN
        dtype: object
```

With DataFrame, reindex can alter either the (row) index, columns, or both.

In [10]:
```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
                     index=['a', 'c', 'd'],
                     columns=['Ohio', 'Texas', 'California'])
frame
```

Out[10]:

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

When passed only a sequence, it reindexes the rows in the result

In [11]:
```
frame2 = frame.reindex(['a', 'b', 'c', 'd'])
frame2
```

Out[11]:

	Ohio	Texas	California
a	0.0	1.0	2.0
b	NaN	NaN	NaN
c	3.0	4.0	5.0
d	6.0	7.0	8.0

In [12]:
```
frame
```

Out[12]:

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

The columns can be reindexed with the columns keyword

In [13]:
```
states = ['Texas', 'Utah', 'California']
f1 = frame.reindex(columns=states)
f1
```

Out[13]:

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

you can reindex more briefly by label-indexing with loc

In [14]:

```
frame
```

Out[14]:

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

In [15]:

```
#frame.loc[['a', 'b', 'c', 'd']]  #giving error ***to check
```

Dropping Entries from an Axis

drop method will return a new object with the indicated value or values deleted from an axis

In [16]:

```
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
obj
```

Out[16]:

```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

In [17]:

```
new_obj = obj.drop('c')
new_obj
```

Out[17]:

```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

In [18]:

```
obj.drop(['d', 'c'])
```

Out[18]:

```
a    0.0
b    1.0
e    4.0
dtype: float64
```

With DataFrame, index values can be deleted from either axis

In [19]:

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
.....: index=['Ohio', 'Colorado', 'Utah', 'New York'],
.....: columns=['one', 'two', 'three', 'four'])
data
```

Out[19]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Calling drop with a sequence of labels will drop values from the row labels (axis 0)

In [20]:

```
data.drop(['Colorado', 'Ohio'])
```

Out[20]:

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

You can drop values from the columns by passing axis=1 or axis='columns'

In [21]:

```
data.drop('two', axis=1)
```

Out[21]:

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

In [22]:

```
data.drop(['two', 'four'], axis='columns')
```

Out[22]:

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

- Many functions, like drop, which modify the size or shape of a Series or DataFrame, can manipulate an object in-place without returning a new object
- One should be careful with the inplace, as it destroys any data that is dropped.

In [23]:
```
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
obj
```

Out[23]:
```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

In [24]:
```
obj.drop('c', inplace=True)
obj
```

Out[24]:
```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

Indexing, Selection, and Filtering

- Series indexing (obj[...]) works analogously to NumPy array indexing,
- except you can use the Series’s index values instead of only integers

In [25]:
```
obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
obj
```

Out[25]:
```
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

In [26]:
```
obj['b']
```

Out[26]:
```
1.0
```

In [27]:
```
obj[1]
```

Out[27]:
```
1.0
```

In [28]:
```
obj[2:4]
```

Out[28]:
```
c    2.0
d    3.0
dtype: float64
```

```
In [29]: obj[['b', 'a', 'd']]
```

```
Out[29]: b    1.0
         a    0.0
         d    3.0
         dtype: float64
```

```
In [30]: obj[[1, 3]]
```

```
Out[30]: b    1.0
         d    3.0
         dtype: float64
```

```
In [31]: obj[obj < 2]
```

```
Out[31]: a    0.0
         b    1.0
         dtype: float64
```

** Slicing with labels behaves differently than normal Python slicing in that the endpoint is inclusive

```
In [32]: obj['b':'c']
```

```
Out[32]: b    1.0
         c    2.0
         dtype: float64
```

Setting using these methods modifies the corresponding section of the Series

```
In [33]: obj['b':'c'] = 5
         obj
```

```
Out[33]: a    0.0
         b    5.0
         c    5.0
         d    3.0
         dtype: float64
```

Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence

In [34]:
```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
                    columns=['one', 'two', 'three', 'four'])
data
```

Out[34]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [35]:
```
data['two']
```

Out[35]:
```
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int32
```

In [36]:
```
data[['three', 'one']]
```

Out[36]:

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

In [37]:
```
data[:2]
```

Out[37]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

In [38]:
```
data['three'] > 5
```

Out[38]:
```
Ohio        False
Colorado     True
Utah         True
New York     True
Name: three, dtype: bool
```

In [39]:
```
data[data['three'] > 5]
```

Out[39]:

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [40]:
```
data[:2]
```

Out[40]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

In [41]:
```
data
```

Out[41]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [42]:
```
data < 5
```

Out[42]:

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

In [43]:
```
data[data < 5] = 0
data
```

Out[43]:

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Selection with loc and iloc

They enable you to select a subset of the rows and columns from a DataFrame with NumPy-like notation using either:

- axis labels (loc)
- or integers (iloc)

Also refer: [https://www.geeksforgeeks.org/difference-between-loc-and-iloc-in-pandas-dataframe/amp/ (https://www.geeksforgeeks.org/difference-between-loc-and-iloc-in-pandas-dataframe/amp/)](https://www.geeksforgeeks.org/difference-between-loc-and-iloc-in-pandas-dataframe/amp/)

In [44]:
```
data
```

Out[44]:

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [45]:
```
data.loc['Colorado']
```

Out[45]:
```
one      0
two      5
three    6
four     7
Name: Colorado, dtype: int32
```

In [46]:
```
# example using loc: selecting single row and multile column

data.loc['Colorado', ['two', 'three']]
```

Out[46]:
```
two      5
three    6
Name: Colorado, dtype: int32
```

In [47]:
```
# example using iloc: selecting single row and multile column
data.iloc[2, [3, 0, 1]]
```

Out[47]:
```
four    11
one      8
two      9
Name: Utah, dtype: int32
```

In [48]:
```
data.iloc[2]
```

Out[48]:
```
one       8
two       9
three    10
four     11
Name: Utah, dtype: int32
```

In [49]:
```
data.iloc[[1, 2], [3, 0, 1]]
```

Out[49]:

	four	one	two
Colorado	7	0	5
Utah	11	8	9

Both indexing functions work with slices in addition to single labels or lists of labels

In [50]:
```
data
```

Out[50]:

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [51]:
```
data.loc[:'Utah', 'two']
```

Out[51]:
```
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int32
```

In [52]:
```
data.iloc[:2, 1]
```

Out[52]:
```
Ohio        0
Colorado    5
Name: two, dtype: int32
```

In [53]:
```
data.iloc[:, :3]
```

Out[53]:

	one	two	three
Ohio	0	0	0
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

In [54]:
```
data.three > 5
```

Out[54]:
```
Ohio        False
Colorado     True
Utah         True
New York     True
Name: three, dtype: bool
```

In [55]:
```
data.iloc[:, :3][data.three > 5] ##
```

Out[55]:

	one	two	three
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

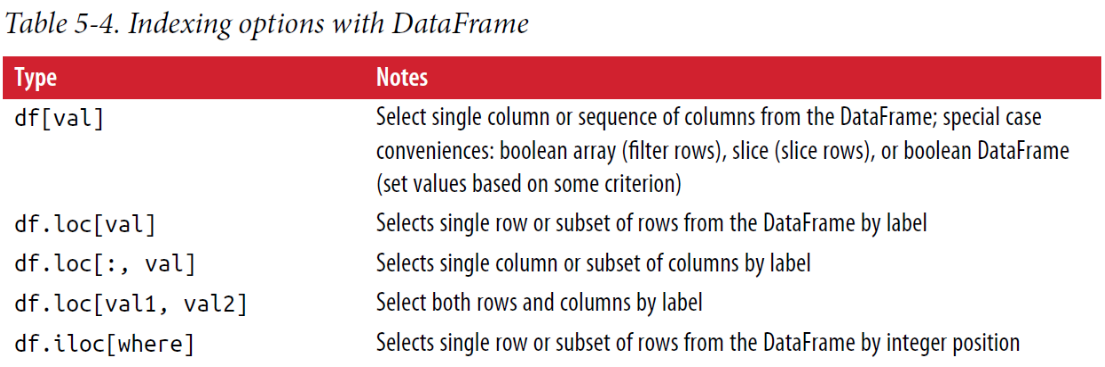


Table 5-4. Indexing options with DataFrame

Type	Notes
`df[val]`	Select single column or sequence of columns from the DataFrame; special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion)
`df.loc[val]`	Selects single row or subset of rows from the DataFrame by label
`df.loc[:, val]`	Selects single column or subset of columns by label
`df.loc[val1, val2]`	Select both rows and columns by label
`df.iloc[where]`	Selects single row or subset of rows from the DataFrame by integer position

Integer Indexes

In [56]:
```
# Here index containing 0, 1, 2
# but inferring what the user wants (label-based indexing or position-based
ser = pd.Series(np.arange(3.))
ser
```

Out[56]:
```
0    0.0
1    1.0
2    2.0
dtype: float64
```

In [57]:
```
ser[2]
```

Out[57]:
```
2.0
```

In [58]:
```
#ser[-1]  #Key error
```

In [59]:
```
# with a non-integer index, there is no potential for ambiguity
ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])
ser2
```

Out[59]:
```
a    0.0
b    1.0
c    2.0
dtype: float64
```

In [60]:

```
ser2[-1]
```

Out[60]:

```
2.0
```

In [61]:

```
ser
```

Out[61]:

```
0    0.0
1    1.0
2    2.0
dtype: float64
```

NOTE:

- To keep things consistent, if you have an axis index containing integers, data selection will always be label-oriented.
- For more precise handling, use loc (for labels) or iloc (for integers)

In [62]:

```
ser[:1]
```

Out[62]:

```
0    0.0
dtype: float64
```

In [63]:

```
ser.loc[:1]
```

Out[63]:

```
0    0.0
1    1.0
dtype: float64
```

In [64]:

```
ser.iloc[:1]
```

Out[64]:

```
0    0.0
dtype: float64
```

In [ ]:

```

```

Operations between DataFrame and Series

As with NumPy arrays of different dimensions, arithmetic between DataFrame and Series is also defined

In [65]:

```
arr = np.arange(12.).reshape((3, 4))
arr
```

Out[65]:

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

Broadcasting

- Broadcasting describes how arithmetic works between arrays of different shapes.
- It can be a powerful feature, but one that can cause confusion, even for experienced users.
- This is referred to as broadcasting and is explained in more detail as it relates to general NumPy arrays in Appendix A.3.

In [66]:
```
# Simple case
# the scalar value 4 has been broadcast to all of the other elements in the
arr * 4
```

Out[66]:
```
array([[ 0.,  4.,  8., 12.],
       [16., 20., 24., 28.],
       [32., 36., 40., 44.]])
```

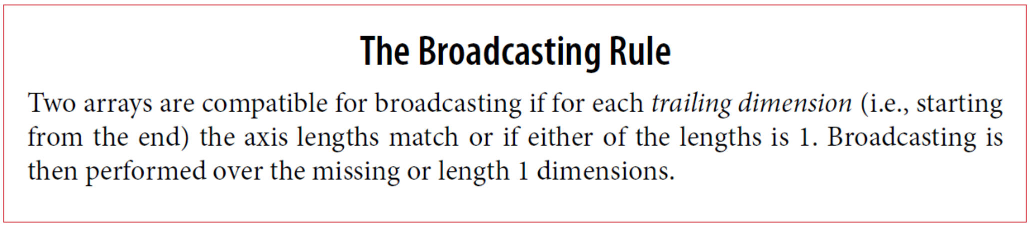
In [67]:
```
arr[0]
```

Out[67]:
```
array([0., 1., 2., 3.])
```

In [68]:
```
# Broadcasting
# When we subtract arr[0] from arr, the subtraction is performed once for e
arr - arr[0]
```

Out[68]:
```
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

See Appendix A.3



The Broadcasting Rule

Two arrays are compatible for broadcasting if for each *trailing dimension* (i.e., starting from the end) the axis lengths match or if either of the lengths is 1. Broadcasting is then performed over the missing or length 1 dimensions.

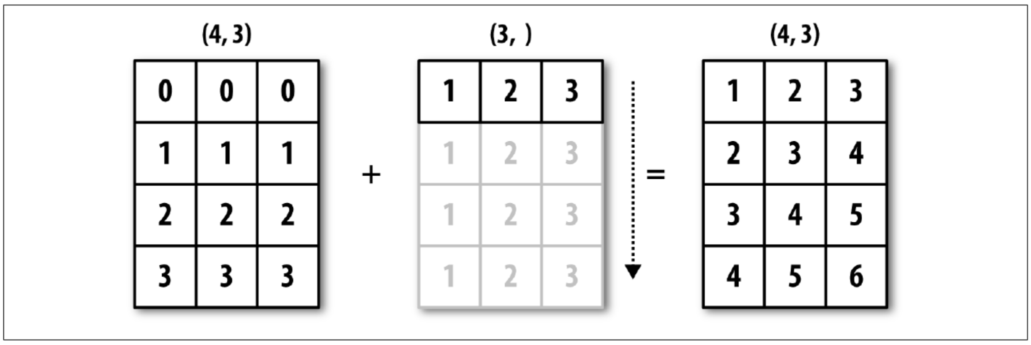


Figure A-4. Broadcasting over axis 0 with a 1D array

Operations between a DataFrame and a Series are similar

In [69]:
```
frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
.....: columns=list('bde'),
.....: index=['Utah', 'Ohio', 'Texas', 'Oregon'])
frame
```

Out[69]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [70]:
```
series = frame.iloc[0]
series
```

Out[70]:
```
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

By default, arithmetic between DataFrame and Series matches the index of the Series on the DataFrame's columns, broadcasting down the rows

In [71]:
```
frame - series
```

Out[71]:

	b	d	e
Utah	0.0	0.0	0.0
Ohio	3.0	3.0	3.0
Texas	6.0	6.0	6.0
Oregon	9.0	9.0	9.0

If an index value is not found in either the DataFrame's columns or the Series's index, the objects will be reindexed to form the union

In [72]:
```
series2 = pd.Series(range(3), index=['b', 'e', 'f'])
series2
```

Out[72]:
```
b    0
e    1
f    2
dtype: int64
```

In [73]:
```
frame
```

Out[73]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [74]:
```
frame + series2
```

Out[74]:

	b	d	e	f
Utah	0.0	NaN	3.0	NaN
Ohio	3.0	NaN	6.0	NaN
Texas	6.0	NaN	9.0	NaN
Oregon	9.0	NaN	12.0	NaN

If instead broadcast over the columns, matching on the rows:

- use one of the arithmetic methods

In [75]:
```
series3 = frame['d']
series3
```

Out[75]:
```
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64
```

In [76]:
```
frame
```

Out[76]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [77]:

```
# The axis number that you pass is the axis to match on
# In this case,  match on the DataFrame’s row index (axis='index' or axis=0
frame.sub(series3, axis='index')  #****
```

Out[77]:

	b	d	e
Utah	-1.0	0.0	1.0
Ohio	-1.0	0.0	1.0
Texas	-1.0	0.0	1.0
Oregon	-1.0	0.0	1.0

Python lambda function

https://www.geeksforgeeks.org/python-lambda-anonymous-functions-filter-map-reduce/ (https://www.geeksforgeeks.org/python-lambda-anonymous-functions-filter-map-reduce/)

https://www.geeksforgeeks.org/python-pandas-apply/ (https://www.geeksforgeeks.org/python-pandas-apply/)

https://www.geeksforgeeks.org/python-pandas-series-apply/ (https://www.geeksforgeeks.org/python-pandas-series-apply/)

https://www.geeksforgeeks.org/difference-between-map-applymap-and-apply-methods-in-pandas/ (https://www.geeksforgeeks.org/difference-between-map-applymap-and-apply-methods-in-pandas/)

Applying lambda function

https://www.geeksforgeeks.org/python-pandas-apply/ (https://www.geeksforgeeks.org/python-pandas-apply/)

https://www.geeksforgeeks.org/python-pandas-series-apply/ (https://www.geeksforgeeks.org/python-pandas-series-apply/)

https://www.geeksforgeeks.org/difference-between-map-applymap-and-apply-methods-in-pandas/ (https://www.geeksforgeeks.org/difference-between-map-applymap-and-apply-methods-in-pandas/)

In [78]:

```
x=3
f=lambda x: x**2 -1
print(f(x))
```

```
8
```

Function Application and Mapping

In [79]:

```
np.random.randn(4, 3)
```

Out[79]:

```
array([[ 1.16343418, -0.31729283,  0.46234441],
       [ 0.22095103, -0.0375128 ,  1.2626555 ],
       [ 0.77051086, -1.88862728,  0.78014138],
       [ 0.42730043, -1.12092883,  0.02458483]])
```

In [80]:

```
list('bde')
```

Out[80]:

```
['b', 'd', 'e']
```

In [81]:

```
frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
.....: index=['Utah', 'Ohio', 'Texas', 'Oregon'])
frame
```

Out[81]:

	b	d	e
Utah	0.230516	-0.830978	1.516567
Ohio	0.261709	-0.608631	-0.040429
Texas	-0.842504	-0.257702	0.520605
Oregon	1.749651	-1.189138	0.311340

In [82]:

```
np.abs(frame)
```

Out[82]:

	b	d	e
Utah	0.230516	0.830978	1.516567
Ohio	0.261709	0.608631	0.040429
Texas	0.842504	0.257702	0.520605
Oregon	1.749651	1.189138	0.311340

In [83]:

```
frame
```

Out[83]:

	b	d	e
Utah	0.230516	-0.830978	1.516567
Ohio	0.261709	-0.608631	-0.040429
Texas	-0.842504	-0.257702	0.520605
Oregon	1.749651	-1.189138	0.311340

apply() method of DataFrame

- applying a function on one-dimensional arrays to each column or row.

```
Example: Here the function f, which computes the difference between the
maximum and minimum of a Series, is invoked once on each column in
frame.
- The result is a Series having the columns of frame as its index.
```

In [84]:

```
f = lambda x: x.max() - x.min()
frame.apply(f)
```

Out[84]:

```
b    2.592155
d    0.931436
e    1.556996
dtype: float64
```

In [85]:

```
0.543010 +0.622371
```

Out[85]:

```
1.165381
```

If axis='columns' is passed to apply, the function will be invoked once per row instead

In [86]:

```
frame.apply(f, axis='columns')
```

Out[86]:

```
Utah      2.347544
Ohio      0.870340
Texas     1.363109
Oregon    2.938789
dtype: float64
```

In [87]:

```
x
```

Out[87]:

```
3
```

- Many of the most common array statistics (like sum and mean) are DataFrame methods, so using apply is not necessary.
- The function passed to apply need not return a scalar value; it can also return a Series with multiple values

In [88]:

```
def f1(x):
    return pd.Series([x.min(), x.max()], index=['min', 'max']) #**
```

In [89]:

```
frame.apply(f1)
```

Out[89]:

	b	d	e
min	-0.842504	-1.189138	-0.040429
max	1.749651	-0.257702	1.516567

In [90]:
```
frame.apply(f1, axis='columns')
```

Out[90]:
```
             min       max
Utah   -0.830978  1.516567
Ohio   -0.608631  0.261709
Texas  -0.842504  0.520605
Oregon -1.189138  1.749651
```

https://www.geeksforgeeks.org/python-pandas-series-rank/
(https://www.geeksforgeeks.org/python-pandas-series-rank/)
https://www.geeksforgeeks.org/python-pandas-dataframe-rank/
(https://www.geeksforgeeks.org/python-pandas-dataframe-rank/)

applymap() method of DataFrame - Element-wise Python function

Example:

- Suppose you wanted to compute a formatted string from each floating-point value in frame.
- use method applymap()

In [91]:
```
format = lambda x: '%.2f' % x
frame.applymap(format)
```

Out[91]:
```
           b      d      e
Utah    0.23  -0.83   1.52
Ohio    0.26  -0.61  -0.04
Texas  -0.84  -0.26   0.52
Oregon  1.75  -1.19   0.31
```

Series map() method for applying an element-wise function

In [92]:
```
frame
```

Out[92]:
```
               b          d          e
Utah    0.230516  -0.830978   1.516567
Ohio    0.261709  -0.608631  -0.040429
Texas  -0.842504  -0.257702   0.520605
Oregon  1.749651  -1.189138   0.311340
```

In [93]:

```
frame['e'].map(format)
```

Out[93]:

```
Utah       1.52
Ohio      -0.04
Texas      0.52
Oregon     0.31
Name: e, dtype: object
```

Sorting

Sorting a dataset by some criterion using built-in operation

In [94]:

```
obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
obj
```

Out[94]:

```
d    0
a    1
b    2
c    3
dtype: int64
```

sort_index method() - sorts lexicographically by row or column index, and returns a new, sorted object

In [95]:

```
obj.sort_index()
```

Out[95]:

```
a    1
b    2
c    3
d    0
dtype: int64
```

sort_values() method - sort Series by its values

In [96]:

```
obj = pd.Series([4, 7, -3, 2])
obj
```

Out[96]:

```
0    4
1    7
2   -3
3    2
dtype: int64
```

In [97]:

```
obj.sort_values()
```

Out[97]:

```
2   -3
3    2
0    4
1    7
dtype: int64
```

Any missing values are sorted to the end of the Series by default

In [98]:

```
obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
obj
```

Out[98]:

```
0    4.0
1    NaN
2    7.0
3    NaN
4   -3.0
5    2.0
dtype: float64
```

In [99]:

```
obj.sort_values()
```

Out[99]:

```
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

DataFrame sort_index() method: sort by index on either axis

Also refer: https://www.geeksforgeeks.org/python-pandas-dataframe-sort_index/ (https://www.geeksforgeeks.org/python-pandas-dataframe-sort_index/)

In [100]:

```
frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
                     index=['three', 'one'],
                     columns=['d', 'a', 'b', 'c'])
frame
```

Out[100]:

	d	a	b	c
three	0	1	2	3
one	4	5	6	7

In [101]:

```
frame.sort_index()
```

Out[101]:

	d	a	b	c
one	4	5	6	7
three	0	1	2	3

In [102]:
```
frame.sort_index(axis=1)
```

Out[102]:

	a	b	c	d
three	1	2	3	0
one	5	6	7	4

- The data is sorted in ascending order by default
- can be sorted in descending order, use ascending=False

In [103]:
```
frame.sort_index(axis=1, ascending=False)
```

Out[103]:

	d	c	b	a
three	0	3	2	1
one	4	7	6	5

Sorting a DataFrame using sort_values() method

Sorting a DataFrame sort_values - can use the data in one or more columns as the sort keys

- Pass one or more column names to the by option of sort_values()

In [104]:
```
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
frame
```

Out[104]:

	b	a
0	4	0
1	7	1
2	-3	0
3	2	1

In [105]:
```
frame.sort_values(by='b')
```

Out[105]:

	b	a
2	-3	0
3	2	1
0	4	0
1	7	1

To sort by multiple columns, pass a list of names

In [106]:
```
frame.sort_values(by=['a', 'b'])
```

Out[106]:

	b	a
2	-3	0
0	4	0
3	2	1
1	7	1

Ranking

- Ranking assigns ranks from one through the number of valid data points in an array (ranks 1 to n)
- The rank() methods for Series and DataFrame used for this purpose
- by default rank breaks ties by assigning each group the mean rank
- also refer https://www.geeksforgeeks.org/python-pandas-series-rank/ (https://www.geeksforgeeks.org/python-pandas-series-rank/)
- also refer https://www.geeksforgeeks.org/python-pandas-dataframe-rank/ (https://www.geeksforgeeks.org/python-pandas-dataframe-rank/)

In [107]:
```
obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
obj
```

Out[107]:
```
0    7
1   -5
2    7
3    4
4    2
5    0
6    4
dtype: int64
```

In [108]:
```
obj.rank()
```

Out[108]:
```
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

In [109]:
```
obj.rank()
```

Out[109]:
```
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

method='first'

- instead of using the average rank 6.5 for the entries 0 and 2, these have been set to 6 and 7 because label 0 precedes label 2 in the data

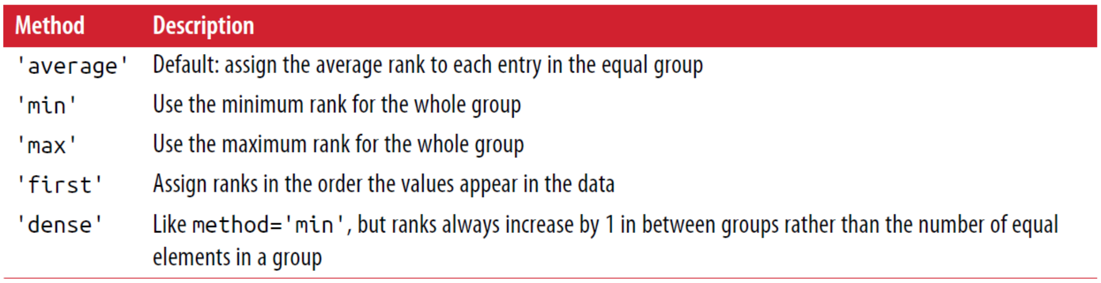
In [110]:
```
obj.rank(method='first')
```

Out[110]:
```
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

In [111]:
```
# Assign tie values the maximum rank in the group
obj.rank(ascending=False, method='max')
```

Out[111]:
```
0    2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6    4.0
dtype: float64
```

Table 5-6. Tie-breaking methods with rank



Method	Description
'average'	Default: assign the average rank to each entry in the equal group
'min'	Use the minimum rank for the whole group
'max'	Use the maximum rank for the whole group
'first'	Assign ranks in the order the values appear in the data
'dense'	Like method='min', but ranks always increase by 1 in between groups rather than the number of equal elements in a group

DataFrame can compute ranks over the rows or the columns

In [112]:

```
frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
                      'c': [-2, 5, 8, -2.5]})
frame
```

Out[112]:

```
      b  a     c
0   4.3  0  -2.0
1   7.0  1   5.0
2  -3.0  0   8.0
3   2.0  1  -2.5
```

In [113]:

```
frame.rank(axis='columns')
```

Out[113]:

```
     b    a    c
0  3.0  2.0  1.0
1  3.0  1.0  2.0
2  1.0  2.0  3.0
3  3.0  2.0  1.0
```