5.1 Introduction to pandas Data Structures

Pandas has two main data structures. Series and DataFrame.

Series

A one dimensional array like object containing a sequency of types. similar to NumPy types.

In [2]:

```
import pandas as pd
import numpy as np
obj = pd.Series([4, 7, -5, 3])
obj
```

Out[2]:

```
0    4
1    7
2   -5
3    3
dtype: int64
```

The left side shows the index as one was not specified. You can specify your own index.

In [3]:

```
obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
obj2
```

Out[3]:

```
d    4
b    7
a   -5
c    3
dtype: int64
```

In [4]:

```
obj2.index #returns information on the Series
```

Out[4]:

```
Index(['d', 'b', 'a', 'c'], dtype='object')
```

Compared to Numpy arrays you can use labels in the index when selecting single or a set of values

In [5]:

```
obj2[['c', 'a', 'd']]
```

Out[5]:

```
c    3
a   -5
d    4
dtype: int64
```

In [6]:

```
obj2[obj2 > 0]
```

Out[6]:

```
d    4
b    7
c    3
dtype: int64
```

In [7]:

```
np.exp(obj2)
```

Out[7]:

```
d      54.598150
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

a way to think aobut a Series is a fixed-length, ordered dict, as it is a mapping of the index values to data. It can be used in many contexts where you may use a dict

In [8]:

```
'b' in obj2
```

Out[8]:

```
True
```

should you have a dict of data you can create a series with the folowing code

In [9]:

```
sdata = {'Ohio':35000, 'Texas':7100, 'Oregon':16000, 'Utah':5000}
obj4 = pd.Series(sdata)
obj4
```

Out[9]:

```
Ohio      35000
Texas      7100
Oregon    16000
Utah       5000
dtype: int64
```

Should you want them in a particular order you can pass a list of index's

In [10]:

```
states = ['California', 'Ohio', 'Oregon', 'Texas']
obj4 = pd.Series(sdata, index=states)
obj4
```

Out[10]:

```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas          7100.0
dtype: float64
```

the .isnull and .notnull function should be used to detect missing data

In [11]:

```
pd.isnull(obj4)
```

Out[11]:

```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

In [12]:

```
pd.notnull(obj4)
```

Out[12]:

```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

A useful Series feature is that it automatically aligns by index label in arithmetic operations

In [13]:

```
obj4 + obj4
```

Out[13]:

```
California        NaN
Ohio          70000.0
Oregon        32000.0
Texas         14200.0
dtype: float64
```

In [14]:

```
obj4.name = 'population' # gives the object a name
obj4.index.name = 'state' #gives the index a name
```

.
.
.
.
.
.
.
.

DataFrame

In [15]:

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'], 'year':[2000,2
001,2002,2001,2002,2003], 'pop':[1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
frame.head() #shows the first 5 rows
```

Out[15]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9

if you specify a sequence of columns, the dataframes columns will be arranged in that order.

In [16]:

```
pd.DataFrame(data, columns=['year', 'state', 'pop'])
```

Out[16]:

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9
5	2003	Nevada	3.2

selecting data from the DataFrame

In [17]:

```
frame2 = pd.DataFrame(data, columns= ['year', 'state', 'pop', 'debt'], index=['one', 't
wo', 'three', 'four', 'five', 'six'])
frame2.head()
```

Out[17]:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN

In [18]:

```
frame['state']
```

Out[18]:

```
0      Ohio
1      Ohio
2      Ohio
3    Nevada
4    Nevada
5    Nevada
Name: state, dtype: object
```

In [19]:

```
frame.year
```

Out[19]:

```
0    2000
1    2001
2    2002
3    2001
4    2002
5    2003
Name: year, dtype: int64
```

In [20]:

```
frame.loc[1] #gets row 1
```

Out[20]:

```
state    Ohio
year     2001
pop       1.7
Name: 1, dtype: object
```

to assign data to an empy column

In [21]:

```
frame2['debt'] = np.arange(6)
frame2.head()
```

Out[21]:

	year	state	pop	debt
one	2000	Ohio	1.5	0
two	2001	Ohio	1.7	1
three	2002	Ohio	3.6	2
four	2001	Nevada	2.4	3
five	2002	Nevada	2.9	4

In [22]:

```
del frame2['debt']
frame2.head() #deletes column
```

Out[22]:

	year	state	pop
one	2000	Ohio	1.5
two	2001	Ohio	1.7
three	2002	Ohio	3.6
four	2001	Nevada	2.4
five	2002	Nevada	2.9

Arithmetic and data Alignment

when you add objects together, if index pairs are not the same, the respective index in the result will be the union of the index pairs.

In [23]:

```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5,], index=['a', 'b', 'c', 'd'])
s2 = pd.Series([1.3, -5.0, 2.2, 7.4, 7.7], index=['c', 'd', 'e', 'f', 'g'])
s1
```

Out[23]:

```
a    7.3
b   -2.5
c    3.4
d    1.5
dtype: float64
```

In [24]:

```
s2
```

Out[24]:

```
c    1.3
d   -5.0
e    2.2
f    7.4
g    7.7
dtype: float64
```

In [25]:

```
s1 + s2
```

Out[25]:

```
a    NaN
b    NaN
c    4.7
d   -3.5
e    NaN
f    NaN
g    NaN
dtype: float64
```

if the cell does not exist in one of the two Series of df objects you are adding. The result will be NaN .

.

Operations between DataFrame and Series

Broadcasting. Subtracting a subdimension or row from a series/data frame will subtract the result from all dimensions/rows

In [26]:

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

Out[26]:

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

In [27]:

```
arr[0]
```

Out[27]:

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

In [28]:

```
arr - arr[0]
```

Out[28]:

```
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

In [29]:

```
# now for the DataFrame
frame = pd.DataFrame(np.arange(12.).reshape((4,3)), columns=list('bde'), index=['Utah',
'Ohio', 'Texas', 'Oregon'])
series = frame.iloc[0]
```

In [30]:

```
frame
```

Out[30]:

	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 [31]:

```
series
```

Out[31]:

```
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

In [32]:

```
frame - series
```

Out[32]:

	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

Function Application and Mapping

numpy ufuncs are also applicable to DataBase objects

In [33]:

```
frame = pd.DataFrame(np.random.randn(4,3), columns=list('bde'), index=['East Sussex',
'Hampshire', 'Kent', 'Shropshire'])
frame
```

Out[33]:

	b	d	e
East Sussex	1.223321	-0.932108	0.281089
Hampshire	-0.767113	0.349402	0.952245
Kent	-1.714094	-0.037945	0.477059
Shropshire	-1.643877	1.008616	-0.727541

In [34]:

```
frame.abs()
```

Out[34]:

	b	d	e
East Sussex	1.223321	0.932108	0.281089
Hampshire	0.767113	0.349402	0.952245
Kent	1.714094	0.037945	0.477059
Shropshire	1.643877	1.008616	0.727541

In [35]:

```
f = lambda x: x.max() - x.min()
frame.apply(f) #finds the maximum difference between the values in columns
```

Out[35]:

```
b    2.937415
d    1.940724
e    1.679785
dtype: float64
```

In [36]:

```
#should you want the function to be applied to rows instead.
frame.apply(f, axis='columns')
```

Out[36]:

```
East Sussex    2.155429
Hampshire      1.719357
Kent           2.191153
Shropshire     2.652493
dtype: float64
```

In [37]:

```
mean = lambda x: x.mean()
frame.apply(mean, axis='columns')
```

Out[37]:

```
East Sussex    0.190767
Hampshire      0.178178
Kent          -0.424993
Shropshire    -0.454267
dtype: float64
```

Element-wise Python functions can be used too. you can do this with applymap()

In [38]:

```
form = lambda x: '%.2f' % x
frame.applymap(form)
```

Out[38]:

	b	d	e
East Sussex	1.22	-0.93	0.28
Hampshire	-0.77	0.35	0.95
Kent	-1.71	-0.04	0.48
Shropshire	-1.64	1.01	-0.73

.
.
.
.

Sorting and Ranking

Sorting a dataset by some criterion is another built-in operation.

In [39]:

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

Out[39]:

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

In [40]:

```
obj.sort_index()
```

Out[40]:

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

In [41]:

```
frame2 = pd.DataFrame(np.arange(8).reshape((2,4)), index=['three', 'one'], columns=['d'
, 'a', 'b', 'c'])
frame2
```

Out[41]:

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

In [42]:

```
frame2.sort_index() #sorts rows
```

Out[42]:

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

In [43]:

```
frame2.sort_index(axis=1) #sorts columns
```

Out[43]:

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

In [44]:

```
frame2.sort_index(axis=1, ascending=False) #columns sorted in decending order
```

Out[44]:

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

In [45]:

```
frame2.sort_values(by='b', ascending=False) #sorts the row by b Descending
```

Out[45]:

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

Summarizing and Computing Descriptive Statistics

pandas objects have built in tools for gathering summary statistics for a whole series or rows/columns of a dataframe. unlike numpy they have built-in handling for missing data.

In [46]:

```
df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5], [np.nan, np.nan], [0.75, -1.3]], index=[
'a', 'b', 'c', 'd'], columns=['one', 'two'])
df
```

Out[46]:

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

In [47]:

```
df.sum(0)
```

Out[47]:

```
one     9.25
two    -5.80
dtype: float64
```

In [48]:

```
df.sum(axis='columns')
```

Out[48]:

```
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

In [49]:

```
df.idxmax() #returns index value where max or min value exists
df.idxmin()
```

Out[49]:

```
one     d
two     b
dtype: object
```

In [50]:

```
df.cumsum()
```

Out[50]:

	one	two
a	1.40	NaN
b	8.50	-4.5
c	NaN	NaN
d	9.25	-5.8

In [52]:

```
df.describe() #provides multiple statistics
```

Out[52]:

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	-2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

Descriptive Statistics built-in

count -- numer of non-NA values
describe -- compute set of summary statistics for each df column
min, max -- compute min/max values
argmin, argmax -- compute index locations(integers) at which minimum, maximum value, obtained
idxmin, idxmax -- compute index labels at which minimum or maximum value obtained, respectively
quantile -- Compute sample quantile ranging from 0 to 1
sum -- Sum of values
mean -- Mean of values
median -- Arithmetic median (50% quantile) of values
mad -- Mean absolute deviation from mean value
prod -- Product of all values
var -- Sample variance of values
std -- Sample standard deviation of values
skew -- Sample skewness of values
kurt -- Sample kurtosis of values
cumsum -- Cumulative sum of values
cumin, cummax -- Cumulative minimum or maximum of values, respectively
cumprod -- Cumulative product of values
diff -- Compute first arithmetic difference (useful of time series)
pct_change -- Compute percent changes
.
.
.
.
.
.

Correlation and Covariance

Correlation and Covarience, are computed from pairs of arguments. Lets look at some DataFrames of stock prices and volumnes obtianed form yahoo! Finance using the add-on pandas-datareader package.

In [57]:

```
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker) for ticker in ['AAPL', 'IBM', 'MSFT', 'G
OOG']}
price = pd.DataFrame({ticker: data['Adj Close'] for ticker, data in all_data.items()})
volume = pd.DataFrame({ticker: data['Volume'] for ticker, data in all_data.items()})
```

In [59]:

```
returns = price.pct_change()
returns.tail()
```

Out[59]:

	AAPL	IBM	MSFT	GOOG
Date				
2020-07-15	0.006877	0.019901	-0.001488	-0.004564
2020-07-16	-0.012305	0.008211	-0.019804	0.002880
2020-07-17	-0.002020	0.008870	-0.005100	-0.001614
2020-07-20	0.021074	0.010071	0.042981	0.033103
2020-07-21	0.000341	0.026747	-0.003544	0.002057

The corr method of Series computes the correlation of the overlapping, non-NA aligned-by-index values in two series. Relatedly, cov computes the covarience

In [61]:

```
returns['MSFT'].corr(returns['IBM'])
```

Out[61]:

```
0.5921278584062394
```

In [62]:

```
returns['MSFT'].cov(returns['IBM'])
```

Out[62]:

```
0.00016595179395406258
```

DataFrame's corr and cov methods, on the other hand, return a full correlation or covariance matrix as a DataFrame, respectively:

In [63]:

```
returns.corr()
```

Out[63]:

	AAPL	IBM	MSFT	GOOG
AAPL	1.000000	0.526784	0.715336	0.669699
IBM	0.526784	1.000000	0.592128	0.542973
MSFT	0.715336	0.592128	1.000000	0.783817
GOOG	0.669699	0.542973	0.783817	1.000000

In [64]:

```
returns.cov()
```

Out[64]:

	AAPL	IBM	MSFT	GOOG
AAPL	0.000332	0.000155	0.000227	0.000203
IBM	0.000155	0.000260	0.000166	0.000146
MSFT	0.000227	0.000166	0.000303	0.000227
GOOG	0.000203	0.000146	0.000227	0.000277

DataFrames corrwith method, allows you to compute pairwise correlations between a DataFrame's columns or rows with another Series or DataFrame. passing a Series returns a Series with the correlations of percent changes with volume:

In [65]:

```
returns.corrwith(returns.IBM)
```

Out[65]:

```
AAPL    0.526784
IBM     1.000000
MSFT    0.592128
GOOG    0.542973
dtype: float64
```

Passing a DataFrame computes the correlations of matching column names. Below is computed correlations of percent changes with volume:

In [66]:

```
returns.corrwith(volume)
```

Out[66]:

```
AAPL   -0.132683
IBM    -0.106003
MSFT   -0.061536
GOOG   -0.145615
dtype: float64
```

Unique Values, Value Counts, and Membership

In [ ]:

```

```

In [67]:

```
obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
uniques = obj.unique()
uniques
```

Out[67]:

```
array(['c', 'a', 'd', 'b'], dtype=object)
```

In [68]:

```
obj.value_counts()
```

Out[68]:

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

In [69]:

```
pd.value_counts(obj.values, sort=False)
```

Out[69]:

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

In [70]:

```
obj
```

Out[70]:

```
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object
```

In [71]:

```
mask = obj.isin(['b', 'c'])
mask
```

Out[71]:

```
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
```

In [72]:

```
obj[mask]
```

Out[72]:

```
0    c
5    b
6    b
7    c
8    c
dtype: object
```

Related to sinin is the Index.get_indexer method, which gives you an index array from an array of possibly non-distinct values into another array of distinct values:

In [74]:

```
to_match= pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])
unique_vals = pd.Series(['c', 'b', 'a'])
pd.Index(unique_vals).get_indexer(to_match)
```

Out[74]:

```
array([0, 2, 1, 1, 0, 2], dtype=int64)
```

In the case you want to create a histogram from a data frame you can do the following

In [76]:

```
data = pd.DataFrame({'Qu1': [1,3,4,3,4],
                     'Qu2': [2,3,1,2,3],
                     'Qu3': [1,5,2,4,4]})
data
```

Out[76]:

	Qu1	Qu2	Qu3
0	1	2	1
1	3	3	5
2	4	1	2
3	3	2	4
4	4	3	4

In [79]:

```
result = data.apply(pd.value_counts)
result
```

Out[79]:

	Qu1	Qu2	Qu3
1	1.0	1.0	1.0
2	NaN	2.0	1.0
3	2.0	2.0	NaN
4	2.0	NaN	2.0
5	NaN	NaN	1.0

In [80]:

```
result.fillna(0)
```

Out[80]:

	Qu1	Qu2	Qu3
1	1.0	1.0	1.0
2	0.0	2.0	1.0
3	2.0	2.0	0.0
4	2.0	0.0	2.0
5	0.0	0.0	1.0

In [ ]: