Introduction-Pandas

November 19, 2020

1 Introduction to Pandas and other libraries

(Many thanks to Evimaria Terzi and Mark Crovella for their code and examples)

1.1 Pandas

Pandas is the Python Data Analysis Library.

Pandas is an extremely versatile tool for manipulating datasets, mostly tabular data. You can think of Pandas as the evolution of excel spreadsheets, with more capabilities for coding, and SQL queries such as joins and group-by.

It also produces high quality plots with matplotlib, and integrates nicely with other libraries that expect NumPy arrays.

You can find more details here

1.1.1 Storing data tables

Most data can be viewed as tables or matrices (in the case where all entries are numeric). The rows correspond to objects and the columns correspond to the attributes or features.

There are different ways we can store such data tables in Python

Two-dimensional lists

[1]:
```
D = [[0.3, 10, 1000],[0.5,2,509],[0.4, 8, 789]]
print(D)
```

```
[[0.3, 10, 1000], [0.5, 2, 509], [0.4, 8, 789]]
```

[2]:
```
D = [[30000, 'Married', 1],[20000,'Single', 0],[45000, 'Maried', 0]]
print(D)
```

```
[[30000, 'Married', 1], [20000, 'Single', 0], [45000, 'Maried', 0]]
```

Numpy Arrays

Numpy is a the library of Python for numerical computations and matrix manipulations. It has a lot of the functionality of Matlab but also allows for data analysis operations (similar to Pandas). Read more for Numpy here: http://www.numpy.org/

The Array is the main data structure for numpy. It stores multidimensional numeric tables.

We can create numpy arrays from lists

[3]:
```
import numpy as np

#1-dimensional array
x = np.array([2,5,18,14,4])
print ("\n Deterministic 1-dimensional array \n")
print (x)

#2-dimensional array
x = np.array([[2,5,18,14,4], [12,15,1,2,8]])
print ("\n Deterministic 2-dimensional array \n")
print (x)
```

```
 Deterministic 1-dimensional array

[ 2  5 18 14  4]

 Deterministic 2-dimensional array

[[ 2  5 18 14  4]
 [12 15  1  2  8]]
```

There are also numpy operations that create arrays of different types

[4]:
```
x = np.random.rand(5,5)
print ("\n Random 5x5 2-dimensional array \n")
print (x)

x = np.ones((4,4))
print ("\n 4x4 array with ones \n")
print (x)

x = np.diag([1,2,3])
print ("\n Diagonal matrix\n")
print(x)
```

```
 Random 5x5 2-dimensional array

[[0.91538311 0.36139701 0.36132849 0.79058036 0.91987516]
 [0.75662154 0.10677586 0.92997889 0.18592405 0.29674822]
 [0.76340147 0.76490157 0.33458267 0.20046639 0.98217537]
 [0.58061203 0.27348177 0.19172822 0.05249137 0.6416912 ]
 [0.47860946 0.26316603 0.16611766 0.81903092 0.03064936]]

 4x4 array with ones
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]

 Diagonal matrix

[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

Why do we need numpy arrays? Because we can do different linear algebra operations on the numeric arrays

For example:

[5]:
```
x = np.random.randint(10,size=(2,3))
print("\n Random 2x3 array with integers")
print(x)

#Matrix transpose
print ("\n Transpose of the matrix \n")
print (x.T)

#multiplication and addition with scalar value
print("\n Matrix 2x+1 \n")
print(2*x+1)
```

```
 Random 2x3 array with integers
[[7 2 0]
 [4 1 3]]

 Transpose of the matrix

[[7 4]
 [2 1]
 [0 3]]

 Matrix 2x+1

[[15  5  1]
 [ 9  3  7]]
```

[6]:
```
lx = [list(y) for y in x]
lx
```

[6]:
```
[[7, 2, 0], [4, 1, 3]]
```

Pandas data frames

A data frame is a table in which each row and column is given a label. Very similar to a spreahsheet or a SQL table.

Pandas DataFrames are documented at: http://pandas.pydata.org/pandas-docs/dev/generated/pandas.DataFrame.html

Pandas dataframes enable different data analysis operations

1.1.2 Creating Data Frames

A dataframe has names for the columns and the rows of the tables. The column names are stored in the attribute **columns**, while the row names in the attribute **index**. When these are not speficied, they are just indexed by default with the numbers 0,1,...

There are multiple ways we can create a data frame. Here we list just a few.

[7]:
```
import pandas as pd #The pandas library
from pandas import Series, DataFrame #Main pandas data structures
```

[8]:
```
#Creating a data frame from a list of lists

df = pd.DataFrame([[1,2,3],[9,10,12]])
print(df)

# Each list becomes a row
# Names of columns are 0,1,3
# Rows are indexed by position numbers 0,1
```

```
   0   1   2
0  1   2   3
1  9  10  12
```

[9]:
```
#Creating a data frame from a numpy array

df = pd.DataFrame(np.array([[1,2,3],[9,10,12]]))
print(df)
```

```
   0   1   2
0  1   2   3
1  9  10  12
```

[10]:
```
# Specifying column names
df = pd.DataFrame(np.array([[1,2,3],[9,10,12]]), columns=['A','B','C'])
print(df)
```

```
   A   B   C
0  1   2   3
1  9  10  12
```

[11]:
```
#Creating a data frame from a dictionary
# Keys are column names, values are lists with column values

dfe = pd.DataFrame({'A':[1,2,3], 'B':['a','b','c']})
print(dfe)
```

```
   A  B
0  1  a
1  2  b
2  3  c
```

[12]:
```
# Reading from a csv file:
df = pd.read_csv('example.csv')
print(df)

# The first row of the file is used for the column names
# The property columns gives us the column names
print(df.columns)
print(list(df.columns))

# Reading from a csv file without header:
df = pd.read_csv('no-header.csv',header = None)
print(df)
```

```
   NUMBER CHAR
0       1    a
1       2    b
2       3    c
Index(['NUMBER', 'CHAR'], dtype='object')
['NUMBER', 'CHAR']
   0  1
0  1  a
1  2  b
2  3  c
```

[13]:
```
# Reading from am excel file:
df = pd.read_excel('example.xlsx')
print(df)
```

```
   NUMBER CHAR
0       1    a
1       2    b
2       3    c
```

[14]:
```
#Writing to a csv file:
df.to_csv('example2.csv')
for x in open('example2.csv').readlines():
    print(x.strip())
```

```
# By default the row index is added as a column, we can remove it by seting␣
 ↪index=False
df.to_csv('example2.csv',index = False)
for x in open('example2.csv').readlines():
    print(x.strip())
```

```
,NUMBER,CHAR
0,1,a
1,2,b
2,3,c
NUMBER,CHAR
1,a
2,b
3,c
```

Fetching data

For demonstration purposes, we'll use a library built-in to Pandas that fetches data from standard online sources. More information on what types of data you can fetch is at: https://pandas-datareader.readthedocs.io/en/latest/remote_data.html

We will use stock quotes from IEX. To make use of these you need to first create an account and obtain an API key. Then you set the environment variable IEX_API_KEY to the value of the key as it is snown below

[15]:
```
import os
os.environ["IEX_API_KEY"] =␣
 ↪"pk_4f1eb9a770e04d2ebc44123e297618bb"#"pk_*****************************"
```

[16]:
```
import pandas_datareader.data as web # For accessing web data
from datetime import datetime #For handling dates
```

[17]:
```
stocks = 'FB'
data_source = 'iex'
start = datetime(2018,1,1)
end = datetime(2018,12,31)

stocks_data = web.DataReader(stocks, data_source, start, end)

#If you want to load only some of the attributes:
#stocks_data = web.DataReader(stocks, data_source, start, end)[['open','close']]
```

[18]:
```
# the method info() outputs basic information for our data frame
stocks_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 251 entries, 2018-01-02 to 2018-12-31
Data columns (total 5 columns):
```

```
open      251 non-null float64
high      251 non-null float64
low       251 non-null float64
close     251 non-null float64
volume    251 non-null int64
dtypes: float64(4), int64(1)
memory usage: 11.8+ KB
```

[19]:
```
#the medthod head() outputs the top rows of the data frame
stocks_data.head()
```

[19]:

date	open	high	low	close	volume
2018-01-02	177.68	181.58	177.55	181.42	18151903
2018-01-03	181.88	184.78	181.33	184.67	16886563
2018-01-04	184.90	186.21	184.10	184.33	13880896
2018-01-05	185.59	186.90	184.93	186.85	13574535
2018-01-08	187.20	188.90	186.33	188.28	17994726

[20]:
```
#the medthod tail() outputs the last rows of the data frame
stocks_data.tail()
```

[20]:

date	open	high	low	close	volume
2018-12-24	123.10	129.74	123.02	124.06	22066002
2018-12-26	126.00	134.24	125.89	134.18	39723370
2018-12-27	132.44	134.99	129.67	134.52	31202509
2018-12-28	135.34	135.92	132.20	133.20	22627569
2018-12-31	134.45	134.64	129.95	131.09	24625308

Note that the date attribute is the index of the rows, not an attribute.

[21]:
```
#trying to access the date column will give an error

stocks_data.date
```

```
␣
↪---------------------------------------------------------------------------

       AttributeError                            Traceback (most recent call␣
↪last)

       <ipython-input-21-d893f040ef09> in <module>
         1 #trying to access the date column will give an error
         2
   ----> 3 stocks_data.date
```

```
       C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\generic.py in␣
↪__getattr__(self, name)
      5177             if self._info_axis.
↪_can_hold_identifiers_and_holds_name(name):
      5178                 return self[name]
   -> 5179             return object.__getattribute__(self, name)
      5180
      5181     def __setattr__(self, name, value):


       AttributeError: 'DataFrame' object has no attribute 'date'
```

The number of rows in the DataFrame:

[22]:
```
len(stocks_data)
```

[22]:
```
251
```

[23]:
```
stocks_data.to_csv('stocks_data.csv')
for x in open('stocks_data.csv').readlines()[0:10]:
    print(x.strip())
df = pd.read_csv('stocks_data.csv')
df.head()
```

```
date,open,high,low,close,volume
2018-01-02,177.68,181.58,177.55,181.42,18151903
2018-01-03,181.88,184.78,181.33,184.67,16886563
2018-01-04,184.9,186.21,184.1,184.33,13880896
2018-01-05,185.59,186.9,184.93,186.85,13574535
2018-01-08,187.2,188.9,186.33,188.28,17994726
2018-01-09,188.7,188.8,187.1,187.87,12393057
2018-01-10,186.94,187.89,185.63,187.84,10529894
2018-01-11,188.4,188.4,187.38,187.77,9588587
2018-01-12,178.06,181.48,177.4,179.37,77551299
```

[23]:

	date	open	high	low	close	volume
0	2018-01-02	177.68	181.58	177.55	181.42	18151903
1	2018-01-03	181.88	184.78	181.33	184.67	16886563
2	2018-01-04	184.90	186.21	184.10	184.33	13880896
3	2018-01-05	185.59	186.90	184.93	186.85	13574535
4	2018-01-08	187.20	188.90	186.33	188.28	17994726

Note that in the new dataframe, there is now a date column, while the index values are numbers 0,1,...

1.1.3 Working with data columns

The columns or "features" in your data

[24]:
```
df.columns
```

[24]:
```
Index(['date', 'open', 'high', 'low', 'close', 'volume'], dtype='object')
```

We can also assign a list to the columns property in order to change the attribute names.

Alternatively, you can change the name of an attribute using rename:

[25]:
```
df = df.rename(columns = {'volume':'V'})
print(list(df.columns))
df.columns = ['date', 'open', 'high', 'low', 'close', 'vol']
df.head()
```

```
['date', 'open', 'high', 'low', 'close', 'V']
```

[25]:

	date	open	high	low	close	vol
0	2018-01-02	177.68	181.58	177.55	181.42	18151903
1	2018-01-03	181.88	184.78	181.33	184.67	16886563
2	2018-01-04	184.90	186.21	184.10	184.33	13880896
3	2018-01-05	185.59	186.90	184.93	186.85	13574535
4	2018-01-08	187.20	188.90	186.33	188.28	17994726

Selecting a single column from your data.

It is important to keep in mind that this selection process returns a new data frame.

[26]:
```
df['open'].head()
```

[26]:
```
0    177.68
1    181.88
2    184.90
3    185.59
4    187.20
Name: open, dtype: float64
```

Another way of selecting a single column from your data

[27]:
```
df.open.head()
```

[27]:
```
0    177.68
1    181.88
2    184.90
3    185.59
4    187.20
Name: open, dtype: float64
```

Selecting multiple columns

[28]:
```
df[['open','close']].head()
```

[28]:

	open	close
0	177.68	181.42
1	181.88	184.67
2	184.90	184.33
3	185.59	186.85
4	187.20	188.28

We can use the values method to obtain the values of one or more attributes. It returns a numpy array. You can trasform it into a list, by applying the list() operator.

[29]:
```
df.open.values
```

[29]:
```
array([177.68, 181.88, 184.9 , 185.59, 187.2 , 188.7 , 186.94, 188.4 ,
       178.06, 181.5 , 179.26, 178.13, 180.85, 180.8 , 186.05, 189.89,
       187.95, 187.75, 188.75, 183.01, 188.37, 188.22, 192.04, 186.93,
       178.57, 184.15, 181.01, 174.76, 177.06, 175.62, 173.45, 180.5 ,
       178.99, 175.77, 176.71, 178.7 , 179.9 , 184.58, 184.45, 182.3 ,
       179.01, 173.29, 176.2 , 181.78, 178.74, 183.56, 183.91, 185.23,
       185.61, 182.6 , 183.24, 184.49, 177.01, 167.47, 164.8 , 166.13,
       165.44, 160.82, 156.31, 151.65, 155.15, 157.81, 156.55, 152.03,
       161.56, 157.73, 157.82, 157.93, 165.36, 166.98, 164.58, 165.73,
       165.83, 166.88, 166.2 , 167.79, 167.27, 165.43, 160.15, 173.22,
       176.81, 173.79, 172.  , 174.25, 175.13, 173.08, 177.35, 178.25,
       179.67, 183.15, 184.85, 187.71, 184.88, 183.7 , 182.68, 183.49,
       183.77, 184.93, 182.5 , 185.88, 186.02, 184.34, 186.54, 187.87,
       193.07, 191.84, 194.3 , 191.03, 190.75, 187.53, 188.81, 192.17,
       192.74, 193.1 , 195.79, 194.8 , 196.24, 199.1 , 202.76, 201.16,
       200.  , 197.6 , 199.18, 195.18, 197.32, 193.37, 194.55, 194.74,
       198.45, 204.93, 204.5 , 202.22, 203.43, 207.81, 207.5 , 204.9 ,
       209.82, 208.77, 208.85, 210.58, 215.11, 215.72, 174.89, 179.87,
       175.3 , 170.67, 173.93, 170.68, 177.69, 178.97, 186.5 , 184.75,
       185.85, 182.04, 180.1 , 180.71, 179.34, 180.42, 174.5 , 174.04,
       172.81, 172.21, 173.09, 173.7 , 175.99, 178.1 , 176.3 , 175.9 ,
       177.15, 173.5 , 169.49, 166.98, 160.31, 163.51, 163.94, 163.25,
       162.  , 161.72, 161.92, 159.39, 160.08, 164.5 , 166.64, 161.03,
       161.99, 164.3 , 167.55, 168.33, 163.03, 161.58, 160.  , 161.46,
       159.21, 155.54, 157.69, 156.82, 150.13, 156.73, 153.32, 155.4 ,
       159.56, 158.51, 155.86, 154.76, 151.22, 154.28, 147.73, 145.82,
       148.5 , 139.94, 155.  , 151.52, 151.8 , 150.1 , 149.31, 151.57,
       150.49, 146.75, 144.48, 142.  , 143.7 , 142.33, 141.07, 137.61,
       127.03, 134.4 , 133.65, 133.  , 135.75, 136.28, 135.92, 138.26,
       143.  , 140.73, 133.82, 139.25, 139.6 , 143.88, 143.08, 145.57,
       143.34, 143.08, 141.08, 141.21, 130.7 , 133.39, 123.1 , 126.  ,
       132.44, 135.34, 134.45])
```

[30]:
```
df[['open','close']].values
```

[30]:
```
array([[177.68, 181.42],
       [181.88, 184.67],
       [184.9 , 184.33],
       [185.59, 186.85],
       [187.2 , 188.28],
       [188.7 , 187.87],
       [186.94, 187.84],
       [188.4 , 187.77],
       [178.06, 179.37],
       [181.5 , 178.39],
       [179.26, 177.6 ],
       [178.13, 179.8 ],
       [180.85, 181.29],
       [180.8 , 185.37],
       [186.05, 189.35],
       [189.89, 186.55],
       [187.95, 187.48],
       [187.75, 190.  ],
       [188.75, 185.98],
       [183.01, 187.12],
       [188.37, 186.89],
       [188.22, 193.09],
       [192.04, 190.28],
       [186.93, 181.26],
       [178.57, 185.31],
       [184.15, 180.18],
       [181.01, 171.58],
       [174.76, 176.11],
       [177.06, 176.41],
       [175.62, 173.15],
       [173.45, 179.52],
       [180.5 , 179.96],
       [178.99, 177.36],
       [175.77, 176.01],
       [176.71, 177.91],
       [178.7 , 178.99],
       [179.9 , 183.29],
       [184.58, 184.93],
       [184.45, 181.46],
       [182.3 , 178.32],
       [179.01, 175.94],
       [173.29, 176.62],
       [176.2 , 180.4 ],
       [181.78, 179.78],
       [178.74, 183.71],
```

```
[183.56, 182.34],
[183.91, 185.23],
[185.23, 184.76],
[185.61, 181.88],
[182.6 , 184.19],
[183.24, 183.86],
[184.49, 185.09],
[177.01, 172.56],
[167.47, 168.15],
[164.8 , 169.39],
[166.13, 164.89],
[165.44, 159.39],
[160.82, 160.06],
[156.31, 152.22],
[151.65, 153.03],
[155.15, 159.79],
[157.81, 155.39],
[156.55, 156.11],
[152.03, 155.1 ],
[161.56, 159.34],
[157.73, 157.2 ],
[157.82, 157.93],
[157.93, 165.04],
[165.36, 166.32],
[166.98, 163.87],
[164.58, 164.52],
[165.73, 164.83],
[165.83, 168.66],
[166.88, 166.36],
[166.2 , 168.1 ],
[167.79, 166.28],
[167.27, 165.84],
[165.43, 159.69],
[160.15, 159.69],
[173.22, 174.16],
[176.81, 173.59],
[173.79, 172.  ],
[172.  , 173.86],
[174.25, 176.07],
[175.13, 174.02],
[173.08, 176.61],
[177.35, 177.97],
[178.25, 178.92],
[179.67, 182.66],
[183.15, 185.53],
[184.85, 186.99],
[187.71, 186.64],
```

```
[184.88, 184.32],
[183.7 , 183.2 ],
[182.68, 183.76],
[183.49, 182.68],
[183.77, 184.49],
[184.93, 183.8 ],
[182.5 , 186.9 ],
[185.88, 185.93],
[186.02, 184.92],
[184.34, 185.74],
[186.54, 187.67],
[187.87, 191.78],
[193.07, 193.99],
[191.84, 193.28],
[194.3 , 192.94],
[191.03, 191.34],
[190.75, 188.18],
[187.53, 189.1 ],
[188.81, 191.54],
[192.17, 192.4 ],
[192.74, 192.41],
[193.1 , 196.81],
[195.79, 195.85],
[194.8 , 198.31],
[196.24, 197.49],
[199.1 , 202.  ],
[202.76, 201.5 ],
[201.16, 201.74],
[200.  , 196.35],
[197.6 , 199.  ],
[199.18, 195.84],
[195.18, 196.23],
[197.32, 194.32],
[193.37, 197.36],
[194.55, 192.73],
[194.74, 198.45],
[198.45, 203.23],
[204.93, 204.74],
[204.5 , 203.54],
[202.22, 202.54],
[203.43, 206.92],
[207.81, 207.32],
[207.5 , 207.23],
[204.9 , 209.99],
[209.82, 209.36],
[208.77, 208.09],
[208.85, 209.94],
```

```
[210.58, 210.91],
[215.11, 214.67],
[215.72, 217.5 ],
[174.89, 176.26],
[179.87, 174.89],
[175.3 , 171.06],
[170.67, 172.58],
[173.93, 171.65],
[170.68, 176.37],
[177.69, 177.78],
[178.97, 185.69],
[186.5 , 183.81],
[184.75, 185.18],
[185.85, 183.09],
[182.04, 180.26],
[180.1 , 180.05],
[180.71, 181.11],
[179.34, 179.53],
[180.42, 174.7 ],
[174.5 , 173.8 ],
[174.04, 172.5 ],
[172.81, 172.62],
[172.21, 173.64],
[173.09, 172.9 ],
[173.7 , 174.65],
[175.99, 177.46],
[178.1 , 176.26],
[176.3 , 175.9 ],
[175.9 , 177.64],
[177.15, 175.73],
[173.5 , 171.16],
[169.49, 167.18],
[166.98, 162.53],
[160.31, 163.04],
[163.51, 164.18],
[163.94, 165.94],
[163.25, 162.  ],
[162.  , 161.36],
[161.72, 162.32],
[161.92, 160.58],
[159.39, 160.3 ],
[160.08, 163.06],
[164.5 , 166.02],
[166.64, 162.93],
[161.03, 165.41],
[161.99, 164.91],
[164.3 , 166.95],
```

```
[167.55, 168.84],
[168.33, 164.46],
[163.03, 162.44],
[161.58, 159.33],
[160.  , 162.43],
[161.46, 158.85],
[159.21, 157.33],
[155.54, 157.25],
[157.69, 157.9 ],
[156.82, 151.38],
[150.13, 153.35],
[156.73, 153.74],
[153.32, 153.52],
[155.4 , 158.78],
[159.56, 159.42],
[158.51, 154.92],
[155.86, 154.05],
[154.76, 154.78],
[151.22, 154.39],
[154.28, 146.04],
[147.73, 150.95],
[145.82, 145.37],
[148.5 , 142.09],
[139.94, 146.22],
[155.  , 151.79],
[151.52, 151.75],
[151.8 , 150.35],
[150.1 , 148.68],
[149.31, 149.94],
[151.57, 151.53],
[150.49, 147.87],
[146.75, 144.96],
[144.48, 141.55],
[142.  , 142.16],
[143.7 , 144.22],
[142.33, 143.85],
[141.07, 139.53],
[137.61, 131.55],
[127.03, 132.43],
[134.4 , 134.82],
[133.65, 131.73],
[133.  , 136.38],
[135.75, 135.  ],
[136.28, 136.76],
[135.92, 138.68],
[138.26, 140.61],
[143.  , 141.09],
```

```
       [140.73, 137.93],
       [133.82, 139.63],
       [139.25, 137.42],
       [139.6 , 141.85],
       [143.88, 142.08],
       [143.08, 144.5 ],
       [145.57, 145.01],
       [143.34, 144.06],
       [143.08, 140.19],
       [141.08, 143.66],
       [141.21, 133.24],
       [130.7 , 133.4 ],
       [133.39, 124.95],
       [123.1 , 124.06],
       [126.  , 134.18],
       [132.44, 134.52],
       [135.34, 133.2 ],
       [134.45, 131.09]])
```

1.2 Data Frame methods

A DataFrame object has many useful methods.

[31]:
```
df.mean()
```

[31]:
```
open     1.714547e+02
high     1.736153e+02
low      1.693031e+02
close    1.715110e+02
vol      2.768798e+07
dtype: float64
```

Note that date did not appear in the list. This is because it stores Strings

[32]:
```
df.std()
```

[32]:
```
open     1.968349e+01
high     1.942387e+01
low      2.007437e+01
close    1.997745e+01
vol      1.922117e+07
dtype: float64
```

[33]:
```
df.median()
```

[33]:
```
open          174.89
high          176.98
low           172.83
```

```
close         174.70
vol      21860931.00
dtype: float64
```

[34]:
```
df.open.mean()
```

[34]:
```
171.45466135458165
```

[35]:
```
df.high.mean()
```

[35]:
```
173.61533864541832
```

Use describe to get all statistics for the data

[36]:
```
stocks_data.describe()
```

[36]:

	open	high	low	close	volume
count	251.000000	251.000000	251.000000	251.000000	2.510000e+02
mean	171.454661	173.615339	169.303147	171.510956	2.768798e+07
std	19.683487	19.423868	20.074371	19.977452	1.922117e+07
min	123.100000	129.740000	123.020000	124.060000	9.588587e+06
25%	157.815000	160.745000	155.525000	157.915000	1.782839e+07
50%	174.890000	176.980000	172.830000	174.700000	2.186093e+07
75%	184.890000	186.450000	183.420000	185.270000	3.031384e+07
max	215.720000	218.620000	214.270000	217.500000	1.698037e+08

[37]:
```
stocks_data.sum()
```

[37]:
```
open      4.303512e+04
high      4.357745e+04
low       4.249509e+04
close     4.304925e+04
volume    6.949682e+09
dtype: float64
```

The functions we have seen work on columns. We can apply them to rows as well by specifying the **axis** of the data.

axis = 0 means columns, and it is the default behavior

axis = 1 means rows

[38]:
```
stocks_data.sum(axis=1)
```

[38]:
```
date
2018-01-02    18152621.23
2018-01-03    16887295.66
2018-01-04    13881635.54
2018-01-05    13575279.27
```

```
2018-01-08    17995476.71
                 …
2018-12-24    22066501.92
2018-12-26    39723890.31
2018-12-27    31203040.62
2018-12-28    22628105.66
2018-12-31    24625838.13
Length: 251, dtype: float64
```

Sorting: You can sort by a specific column, ascending (default) or descending. You can also sort inplace.

[103]:
```
stocks_data.sort_values(by = 'open', ascending =False).head()
```

[103]:

date	open	high	low	close	volume
2018-07-25	215.72	218.62	214.27	217.50	64592585
2018-07-24	215.11	216.20	212.60	214.67	28468681
2018-07-23	210.58	211.62	208.80	210.91	16731969
2018-07-18	209.82	210.99	208.44	209.36	15334907
2018-07-20	208.85	211.50	208.50	209.94	16241508

1.2.1 Bulk Operations

Methods like **sum( )** and **std( )** work on entire columns.

We can run our own functions across all values in a column (or row) using **apply( )**.

[39]:
```
df.date.head()
```

[39]:
```
0    2018-01-02
1    2018-01-03
2    2018-01-04
3    2018-01-05
4    2018-01-08
Name: date, dtype: object
```

The **values** property of the column returns a list of values for the column. Inspecting the first value reveals that these are strings with a particular format.

[40]:
```
first_date = df.date.values[0]
first_date
#returns a string
```

[40]:
```
'2018-01-02'
```

The datetime library handles dates. The method strptime transforms a string into a date (according to a format given as parameter).

[41]:
```
datetime.strptime(first_date, "%Y-%m-%d")
```

[41]:
```
datetime.datetime(2018, 1, 2, 0, 0)
```

We will now make use of two operations:

The **apply** method takes a dataframe and applies a function that is given as input to apply to all the entries in the data frame. In the case below we apply it to just one column.

The **lambda** function allows to define an anonymus function that takes some parameters (d) and uses them to compute some expression.

Using the lambda function with apply, we can apply the function to all the entries of the data frame (in this case the column values)

[42]:
```
df.date = df.date.apply(lambda d: datetime.strptime(d, "%Y-%m-%d"))
date_series = df.date # We want to keep the dates
df.date.head()

#Another way to do the same thing, by applying the function to every row (axis
↪= 1)
#df.date = df.apply(lambda row: datetime.strptime(row.date, "%Y-%m-%d"), axis=1)
```

[42]:
```
0   2018-01-02
1   2018-01-03
2   2018-01-04
3   2018-01-05
4   2018-01-08
Name: date, dtype: datetime64[ns]
```

[47]:
```
df.date.head()
```

[47]:
```
0   2018-01-02
1   2018-01-03
2   2018-01-04
3   2018-01-05
4   2018-01-08
Name: date, dtype: datetime64[ns]
```

For example, we can obtain the integer part of the open value

[46]:
```
df.apply(lambda row: int(row.open), axis=1)
```

[46]:
```
0      177
1      181
2      184
3      185
4      187
      ...
```

```
246    123
247    126
248    132
249    135
250    134
Length: 251, dtype: int64
```

Each row in a DataFrame is associated with an index, which is a label that uniquely identifies a row.

The row indices so far have been auto-generated by pandas, and are simply integers starting from 0.

From now on we will use dates instead of integers for indices – the benefits of this will show later.

Overwriting the index is as easy as assigning to the **index** property of the DataFrame.

[48]:
```
df.index = df.date
df.head()
```

[48]:

	date	open	high	low	close	vol
date						
2018-01-02	2018-01-02	177.68	181.58	177.55	181.42	18151903
2018-01-03	2018-01-03	181.88	184.78	181.33	184.67	16886563
2018-01-04	2018-01-04	184.90	186.21	184.10	184.33	13880896
2018-01-05	2018-01-05	185.59	186.90	184.93	186.85	13574535
2018-01-08	2018-01-08	187.20	188.90	186.33	188.28	17994726

Another example using the simple example.csv data we loaded

[49]:
```
dfe
```

[49]:

	A	B
0	1	a
1	2	b
2	3	c

[50]:
```
dfe.index = dfe.B
```

[51]:
```
dfe
```

[51]:

	A	B
B		
a	1	a
b	2	b
c	3	c

Now that we have made an index based on date, we can drop the original `date` column. We will not do it in this example to use it later on.

[54]:
```
df = df.drop(columns = ['date']) #Equivalent to df = df.drop(columns =
 ↪['date']), axis=1)
#axis = 0 refers to dropping labels from rows (or you can use index = labels)
#axis = 1 refers to dropping labels from columns.
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 251 entries, 2018-01-02 to 2018-12-31
Data columns (total 5 columns):
open     251 non-null float64
high     251 non-null float64
low      251 non-null float64
close    251 non-null float64
vol      251 non-null int64
dtypes: float64(4), int64(1)
memory usage: 21.8 KB
```

1.2.2 Accessing rows of the DataFrame

So far we've seen how to access a column of the DataFrame. To access a row we use a different notation.

To access a row by its index value, use the **`.loc()`** method.

[55]:
```
df.loc[datetime(2018,5,7)]
```

[55]:
```
open          177.35
high          179.50
low           177.17
close         177.97
vol      18697195.00
Name: 2018-05-07 00:00:00, dtype: float64
```

To access a row by its sequence number (ie, like an array index), use **`.iloc()`** ('Integer Location')

[56]:
```
df.iloc[10:20] #dataframe with rows from 10 to 20
```

[56]:

date	open	high	low	close	vol
2018-01-17	179.26	179.32	175.80	177.60	27992376
2018-01-18	178.13	180.98	177.08	179.80	23304901
2018-01-19	180.85	182.37	180.17	181.29	26826540
2018-01-22	180.80	185.39	180.41	185.37	21059464
2018-01-23	186.05	189.55	185.55	189.35	25678781
2018-01-24	189.89	190.66	186.52	186.55	24334548
2018-01-25	187.95	188.62	186.60	187.48	17377740
2018-01-26	187.75	190.00	186.81	190.00	17759212
2018-01-29	188.75	188.84	185.63	185.98	20453172

```
2018-01-30  183.01  188.18  181.84  187.12  20858556
```

[57]:
```
df.iloc[0:2,[1,3]] #dataframe with rows 0:2, and the second and fourth columns
```

[57]:
```
              high   close
date
2018-01-02  181.58  181.42
2018-01-03  184.78  184.67
```

[58]:
```
df[['high','close']].iloc[0:2]
```

[58]:
```
              high   close
date
2018-01-02  181.58  181.42
2018-01-03  184.78  184.67
```

1.2.3 To iterate over the rows, use `.iterrows()`

[59]:
```
num_positive_days = 0
for idx, row in df.iterrows(): #returns the index name and the row
    if row.close > row.open:
        num_positive_days += 1

print("The total number of positive-gain days is {}.".format(num_positive_days))
```

```
The total number of positive-gain days is 130.
```

You can also do it this way:

[60]:
```
num_positive_days = 0
for i in range(len(df)):
    row = df.iloc[i]
    if row.close > row.open:
        num_positive_days += 1

print("The total number of positive-gain days is {}.".format(num_positive_days))
```

```
The total number of positive-gain days is 130.
```

Or this way:

[61]:
```
pos_days = [idx for (idx,row) in df.iterrows() if row.close > row.open]
print("The total number of positive-gain days is "+str(len(pos_days)))
```

```
The total number of positive-gain days is 130
```

1.3 Filtering

It is very easy to select interesting rows from the data.

All these operations below return a new DataFrame, which itself can be treated the same way as all DataFrames we have seen so far.

[62]:
```
tmp_high = df.high > 170
tmp_high.head()
```

[62]:
```
date
2018-01-02    True
2018-01-03    True
2018-01-04    True
2018-01-05    True
2018-01-08    True
Name: high, dtype: bool
```

Summing a Boolean array is the same as counting the number of **True** values.

[63]:
```
sum(tmp_high)
```

[63]:
```
149
```

Now, let's select only the rows of **df** that correspond to **tmp_high**

[64]:
```
df[tmp_high].head()
```

[64]:

date	open	high	low	close	vol
2018-01-02	177.68	181.58	177.55	181.42	18151903
2018-01-03	181.88	184.78	181.33	184.67	16886563
2018-01-04	184.90	186.21	184.10	184.33	13880896
2018-01-05	185.59	186.90	184.93	186.85	13574535
2018-01-08	187.20	188.90	186.33	188.28	17994726

Putting it all together, we have the following commonly-used patterns:

[65]:
```
positive_days = df[df.close > df.open]
positive_days.head()
```

[65]:

date	open	high	low	close	vol
2018-01-02	177.68	181.58	177.55	181.42	18151903
2018-01-03	181.88	184.78	181.33	184.67	16886563
2018-01-05	185.59	186.90	184.93	186.85	13574535
2018-01-08	187.20	188.90	186.33	188.28	17994726
2018-01-10	186.94	187.89	185.63	187.84	10529894

[66]:
```
very_positive_days = df[df.close-df.open > 5]
very_positive_days.head()
```

[66]:
```
              open    high     low   close       vol
date
2018-02-06  178.57  185.77  177.74  185.31  37758505
2018-02-14  173.45  179.81  173.21  179.52  28929704
2018-04-10  157.93  165.98  157.01  165.04  58947041
2018-07-17  204.90  210.46  204.84  209.99  15349892
2018-08-02  170.68  176.79  170.27  176.37  32399954
```

[67]:
```
df[(df.high<170)&(df.low>80)]
```

[67]:
```
              open    high     low   close        vol
date
2018-03-23  165.44  167.10  159.02  159.39   53609706
2018-03-26  160.82  161.10  149.02  160.06  126116634
2018-03-27  156.31  162.85  150.75  152.22   79116995
2018-03-28  151.65  155.88  150.80  153.03   60029170
2018-03-29  155.15  161.42  154.14  159.79   59434293
…              …       …       …       …          …
2018-12-24  123.10  129.74  123.02  124.06   22066002
2018-12-26  126.00  134.24  125.89  134.18   39723370
2018-12-27  132.44  134.99  129.67  134.52   31202509
2018-12-28  135.34  135.92  132.20  133.20   22627569
2018-12-31  134.45  134.64  129.95  131.09   24625308

[102 rows x 5 columns]
```

1.3.1 Creating new columns

To create a new column, simply assign values to it. Think of the columns as a dictionary:

[68]:
```
df['profit'] = (df.close - df.open)
df.head()
```

[68]:
```
              open    high     low   close       vol  profit
date
2018-01-02  177.68  181.58  177.55  181.42  18151903    3.74
2018-01-03  181.88  184.78  181.33  184.67  16886563    2.79
2018-01-04  184.90  186.21  184.10  184.33  13880896   -0.57
2018-01-05  185.59  186.90  184.93  186.85  13574535    1.26
2018-01-08  187.20  188.90  186.33  188.28  17994726    1.08
```

[69]:
```
df.profit[df.profit>0].describe()
```

[69]:
```
count    130.000000
mean       2.193308
std        1.783095
min        0.020000
```

```
25%        0.720000
50%        1.630000
75%        3.280000
max        8.180000
Name: profit, dtype: float64
```

[70]:
```
for idx, row in df.iterrows():
    if row.close < row.open:
        df.loc[idx,'gain']='negative'
    elif (row.close - row.open) < 1:
        df.loc[idx,'gain']='small_gain'
    elif (row.close - row.open) < 3:
        df.loc[idx,'gain']='medium_gain'
    else:
        df.loc[idx,'gain']='large_gain'
df.head()
```

[70]:

date	open	high	low	close	vol	profit	gain
2018-01-02	177.68	181.58	177.55	181.42	18151903	3.74	large_gain
2018-01-03	181.88	184.78	181.33	184.67	16886563	2.79	medium_gain
2018-01-04	184.90	186.21	184.10	184.33	13880896	-0.57	negative
2018-01-05	185.59	186.90	184.93	186.85	13574535	1.26	medium_gain
2018-01-08	187.20	188.90	186.33	188.28	17994726	1.08	medium_gain

Here is another, more "functional", way to accomplish the same thing.

Define a function that classifies rows, and **apply** it to each row.

[71]:
```
def gainrow(row):
    if row.close < row.open:
        return 'negative'
    elif (row.close - row.open) < 1:
        return 'small_gain'
    elif (row.close - row.open) < 3:
        return 'medium_gain'
    else:
        return 'large_gain'

df['test_column'] = df.apply(gainrow, axis = 1)
#axis = 0 means columns, axis =1 means rows
```

[72]:
```
df.head()
```

[72]:

date	open	high	low	close	vol	profit	gain	\
2018-01-02	177.68	181.58	177.55	181.42	18151903	3.74	large_gain	
2018-01-03	181.88	184.78	181.33	184.67	16886563	2.79	medium_gain	

```
2018-01-04  184.90  186.21  184.10  184.33  13880896   -0.57     negative
2018-01-05  185.59  186.90  184.93  186.85  13574535    1.26  medium_gain
2018-01-08  187.20  188.90  186.33  188.28  17994726    1.08  medium_gain

            test_column
date
2018-01-02   large_gain
2018-01-03  medium_gain
2018-01-04     negative
2018-01-05  medium_gain
2018-01-08  medium_gain
```

OK, point made, let's get rid of that extraneous `test_column`:

[73]:
```
df = df.drop('test_column', axis = 1)
df.head()
```

[73]:
```
              open    high     low   close       vol  profit         gain
date
2018-01-02  177.68  181.58  177.55  181.42  18151903    3.74   large_gain
2018-01-03  181.88  184.78  181.33  184.67  16886563    2.79  medium_gain
2018-01-04  184.90  186.21  184.10  184.33  13880896   -0.57     negative
2018-01-05  185.59  186.90  184.93  186.85  13574535    1.26  medium_gain
2018-01-08  187.20  188.90  186.33  188.28  17994726    1.08  medium_gain
```

1.3.2 Missing values

Data often has missing values. In Pandas these are denoted as NaN values. These may be part of our data (e.g. empty cells in an excel sheet), or they may appear as a result of a join. There are special methods for handling these values.

[74]:
```
mdf = pd.read_csv('example-missing.csv')
mdf
```

[74]:
```
     A    B    C
0  1.0    a    x
1  5.0    b  NaN
2  3.0    c    y
3  9.0  NaN    z
4  NaN    a    x
```

We can fill the values using the fillna method

[75]:
```
mdf.fillna(0)
```

[75]:
```
     A  B  C
0  1.0  a  x
1  5.0  b  0
```

```
2  3.0  c  y
3  9.0  0  z
4  0.0  a  x
```

[76]:
```
mdf.A = mdf.A.fillna(0)
mdf = mdf.fillna('')
mdf
```

[76]:
```
     A  B  C
0  1.0  a  x
1  5.0  b
2  3.0  c  y
3  9.0     z
4  0.0  a  x
```

We can drop the rows with missing values

[77]:
```
mdf = pd.read_csv('example-missing.csv')
mdf.dropna()
```

[77]:
```
     A  B  C
0  1.0  a  x
2  3.0  c  y
```

We can find those rows

[78]:
```
mdf[mdf.B.isnull()]
```

[78]:
```
     A    B  C
3  9.0  NaN  z
```

1.4 Grouping

An **extremely** powerful DataFrame method is `groupby()`.

This is entirely analagous to `GROUP BY` in SQL.

It will group the rows of a DataFrame by the values in one (or more) columns, and let you iterate through each group.

Here we will look at the average gain among the categories of gains (negative, small, medium and large) we defined above and stored in column `gain`.

[79]:
```
gain_groups = df.groupby('gain')
```

[80]:
```
type(gain_groups)
```

[80]:
```
pandas.core.groupby.generic.DataFrameGroupBy
```

Essentially, **gain_groups** behaves like a dictionary * The keys are the unique values found in the **gain** column, and * The values are DataFrames that contain only the rows having the corresponding unique values.

[85]:
```
for gain, gain_data in gain_groups:
    print(gain)
    print(gain_data.head())
    print('============================')
```

```
large_gain
              open    high     low   close       vol  profit        gain
date
2018-01-02  177.68  181.58  177.55  181.42  18151903    3.74  large_gain
2018-01-22  180.80  185.39  180.41  185.37  21059464    4.57  large_gain
2018-01-23  186.05  189.55  185.55  189.35  25678781    3.30  large_gain
2018-01-30  183.01  188.18  181.84  187.12  20858556    4.11  large_gain
2018-02-01  188.22  195.32  187.89  193.09  54211293    4.87  large_gain
============================
medium_gain
              open    high     low   close       vol  profit         gain
date
2018-01-03  181.88  184.78  181.33  184.67  16886563    2.79  medium_gain
2018-01-05  185.59  186.90  184.93  186.85  13574535    1.26  medium_gain
2018-01-08  187.20  188.90  186.33  188.28  17994726    1.08  medium_gain
2018-01-12  178.06  181.48  177.40  179.37  77551299    1.31  medium_gain
2018-01-18  178.13  180.98  177.08  179.80  23304901    1.67  medium_gain
============================
negative
              open    high     low   close       vol  profit      gain
date
2018-01-04  184.90  186.21  184.10  184.33  13880896   -0.57  negative
2018-01-09  188.70  188.80  187.10  187.87  12393057   -0.83  negative
2018-01-11  188.40  188.40  187.38  187.77   9588587   -0.63  negative
2018-01-16  181.50  181.75  178.04  178.39  36183842   -3.11  negative
2018-01-17  179.26  179.32  175.80  177.60  27992376   -1.66  negative
============================
small_gain
              open    high     low   close       vol  profit        gain
date
2018-01-10  186.94  187.89  185.63  187.84  10529894    0.90  small_gain
2018-01-19  180.85  182.37  180.17  181.29  26826540    0.44  small_gain
2018-02-20  175.77  177.95  175.11  176.01  21204921    0.24  small_gain
2018-02-22  178.70  180.21  177.41  178.99  18464192    0.29  small_gain
2018-02-26  184.58  185.66  183.22  184.93  17599703    0.35  small_gain
============================
```

We can obtain the dataframe that corresponds to a specific group by using the get_group method of the groupby object

[81]:
```
sm = gain_groups.get_group('small_gain')
sm.head()
```

[81]:

date	open	high	low	close	vol	profit	gain
2018-01-10	186.94	187.89	185.63	187.84	10529894	0.90	small_gain
2018-01-19	180.85	182.37	180.17	181.29	26826540	0.44	small_gain
2018-02-20	175.77	177.95	175.11	176.01	21204921	0.24	small_gain
2018-02-22	178.70	180.21	177.41	178.99	18464192	0.29	small_gain
2018-02-26	184.58	185.66	183.22	184.93	17599703	0.35	small_gain

[82]:
```
for gain, gain_data in df.groupby("gain"):
    print('The average closing value for the {} group is {}'.format(gain,
                                                    gain_data.close.mean()))
```

```
The average closing value for the large_gain group is 174.99081081081084
The average closing value for the medium_gain group is 174.18557692307695
The average closing value for the negative group is 169.2336363636363
The average closing value for the small_gain group is 171.69926829268292
```

[83]:
```
for gain, gain_data in df.groupby("gain"):
    print('The median volumn value for the {} group is {}'.format(gain,
                                                    gain_data.vol.median()))
```

```
The median volumn value for the large_gain group is 21059464.0
The median volumn value for the medium_gain group is 23155130.0
The median volumn value for the negative group is 22627569.0
The median volumn value for the small_gain group is 20197680.0
```

We often want to do a typical SQL-like group by, where we group by one or more attributes, and aggreagate the values of some other attributes. For example group by "gain" and take the average of the values for open, high, low, close, volume. You can also use other aggregators such as count, sum, median, max, min. Pandas is now returning a new dataframe indexed by the values if the group-by attribute(s), with columns the other attributes

[84]:
```
gdf= df[['open','low','high','close','vol','gain']].groupby('gain').mean()
type(gdf)
```

[84]:
```
pandas.core.frame.DataFrame
```

[85]:
```
gdf
```

[85]:

gain	open	low	high	close	vol
large_gain	170.459730	169.941351	175.660811	174.990811	3.034571e+07
medium_gain	172.305769	171.410962	175.321346	174.185577	2.795407e+07
negative	171.473306	168.024545	172.441322	169.233636	2.771124e+07
small_gain	171.218049	169.827317	173.070488	171.699268	2.488339e+07

If you want to remove the (hierarchical) index and have the group-by atrribute(s) to be part of the table, you can use the reset_index method

[86]:
```
#This can be used to remove the hiearchical index, if necessary
gdf = gdf.reset_index()
gdf
```

[86]:

	gain	open	low	high	close	vol
0	large_gain	170.459730	169.941351	175.660811	174.990811	3.034571e+07
1	medium_gain	172.305769	171.410962	175.321346	174.185577	2.795407e+07
2	negative	171.473306	168.024545	172.441322	169.233636	2.771124e+07
3	small_gain	171.218049	169.827317	173.070488	171.699268	2.488339e+07

[87]:
```
gdf.set_index('gain')
```

[87]:

gain	open	low	high	close	vol
large_gain	170.459730	169.941351	175.660811	174.990811	3.034571e+07
medium_gain	172.305769	171.410962	175.321346	174.185577	2.795407e+07
negative	171.473306	168.024545	172.441322	169.233636	2.771124e+07
small_gain	171.218049	169.827317	173.070488	171.699268	2.488339e+07

Another example:

[88]:
```
test = pd.DataFrame({'A':[1,2,3,4],'B':['a','b','b','a'],'C':['a','a','b','a']})
test
```

[88]:

	A	B	C
0	1	a	a
1	2	b	a
2	3	b	b
3	4	a	a

[89]:
```
gtest = test.groupby(['B','C']).mean()
gtest
```

[89]:

		A
B	C	
a	a	2.5
b	a	2.0
	b	3.0

[90]:
```
gtest = gtest.reset_index()
gtest
```

[90]:

	B	C	A
0	a	a	2.5
1	b	a	2.0

```
2  b  b  3.0
```

1.5 Joins

We can join data frames in a similar way that we can do joins in SQL

[92]:
```
data_source = 'iex'
start = datetime(2018,1,1)
end = datetime(2018,12,31)

dfb = web.DataReader('FB', data_source, start, end)
dgoog = web.DataReader('GOOG', data_source, start, end)

print(dfb.head())
print(dgoog.head())
```

```
              open    high     low   close    volume
date
2018-01-02  177.68  181.58  177.55  181.42  18151903
2018-01-03  181.88  184.78  181.33  184.67  16886563
2018-01-04  184.90  186.21  184.10  184.33  13880896
2018-01-05  185.59  186.90  184.93  186.85  13574535
2018-01-08  187.20  188.90  186.33  188.28  17994726
               open     high      low    close   volume
date
2018-01-02  1048.34  1066.94  1045.23  1065.00  1237564
2018-01-03  1064.31  1086.29  1063.21  1082.48  1430170
2018-01-04  1088.00  1093.57  1084.00  1086.40  1004605
2018-01-05  1094.00  1104.25  1092.00  1102.23  1279123
2018-01-08  1102.23  1111.27  1101.62  1106.94  1047603
```

Perform join on the date (the index value)

[93]:
```
common_dates = pd.merge(dfb,dgoog,on='date')
common_dates.head()
```

[93]:
```
            open_x  high_x   low_x  close_x  volume_x   open_y   high_y  \
date
2018-01-02  177.68  181.58  177.55   181.42  18151903  1048.34  1066.94
2018-01-03  181.88  184.78  181.33   184.67  16886563  1064.31  1086.29
2018-01-04  184.90  186.21  184.10   184.33  13880896  1088.00  1093.57
2018-01-05  185.59  186.90  184.93   186.85  13574535  1094.00  1104.25
2018-01-08  187.20  188.90  186.33   188.28  17994726  1102.23  1111.27

              low_y  close_y  volume_y
date
2018-01-02  1045.23  1065.00   1237564
2018-01-03  1063.21  1082.48   1430170
```

```
2018-01-04  1084.00  1086.40   1004605
2018-01-05  1092.00  1102.23   1279123
2018-01-08  1101.62  1106.94   1047603
```

Compute gain and perform join on the data AND gain

[94]:
```
dfb['gain'] = dfb.apply(gainrow, axis = 1)
dgoog['gain'] = dgoog.apply(gainrow, axis = 1)
dfb['profit'] = dfb.close-dfb.open
dgoog['profit'] = dgoog.close-dgoog.open
```

[95]:
```
common_gain_dates = pd.merge(dfb, dgoog, on=['date','gain'])
common_gain_dates.head()
```

[95]:
```
            open_x  high_x   low_x  close_x  volume_x        gain  profit_x  \
date
2018-01-02  177.68  181.58  177.55   181.42  18151903  large_gain      3.74
2018-01-04  184.90  186.21  184.10   184.33  13880896    negative     -0.57
2018-01-09  188.70  188.80  187.10   187.87  12393057    negative     -0.83
2018-01-11  188.40  188.40  187.38   187.77   9588587    negative     -0.63
2018-01-16  181.50  181.75  178.04   178.39  36183842    negative     -3.11

             open_y   high_y    low_y  close_y  volume_y  profit_y
date
2018-01-02  1048.34  1066.94  1045.23  1065.00   1237564     16.66
2018-01-04  1088.00  1093.57  1084.00  1086.40   1004605     -1.60
2018-01-09  1109.40  1110.57  1101.23  1106.26    902541     -3.14
2018-01-11  1106.30  1106.53  1099.59  1105.52    978292     -0.78
2018-01-16  1132.51  1139.91  1117.83  1121.76   1575261    -10.75
```

More join examples, including left outer join

[96]:
```
left = pd.DataFrame({'key': ['foo', 'foo', 'boo'], 'lval': [1, 2,3]})
print(left)
print('\n')
right = pd.DataFrame({'key': ['foo', 'hoo'], 'rval': [4, 5]})
print(right)
print('\n')
dfm = pd.merge(left, right, on='key') #keeps only the common key 'foo'
print(dfm)
```

```
   key  lval
0  foo     1
1  foo     2
2  boo     3


   key  rval
```

```
0  foo     4
1  hoo     5
```

```
   key  lval  rval
0  foo     1     4
1  foo     2     4
```

Left outer join

[97]:
```
dfm = pd.merge(left, right, on='key', how='left') #keeps all the keys from the␣
 ↪left and puts NaN for missing values
print(dfm)
print('\n')
dfm = dfm.fillna(0) #fills the NaN values with specified value
dfm
```

```
   key  lval  rval
0  foo     1   4.0
1  foo     2   4.0
2  boo     3   NaN
```

[97]:
```
   key  lval  rval
0  foo     1   4.0
1  foo     2   4.0
2  boo     3   0.0
```

2 Plotting

The main library for plotting is **matplotlib**, which uses the Matlab plotting capabilities.

We can also use the **seaborn** library on top of that to do visually nicer plots

[104]:
```
import matplotlib.pyplot as plt #main plotting tool for python
import matplotlib as mpl

import seaborn as sns #A more fancy plotting library

#For presenting plots inline
%matplotlib inline
```

2.0.1 Simple plots

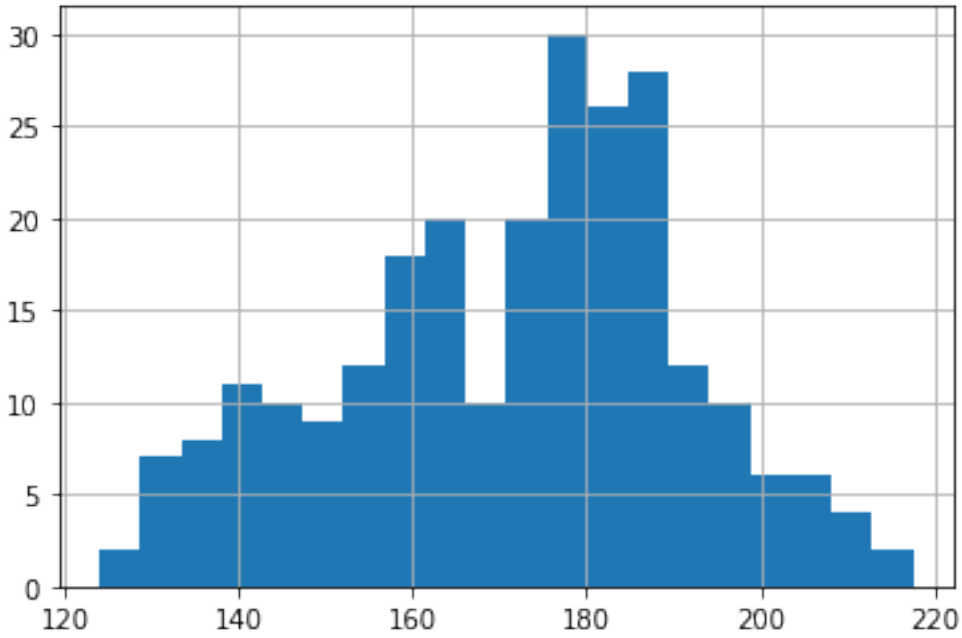
[105]:
```
df.high.plot()
df.low.plot(label='low values')
plt.legend(loc='best') #puts the ledgent in the best possible position
```

[105]: <matplotlib.legend.Legend at 0x19240699668>

2.0.2 Histograms

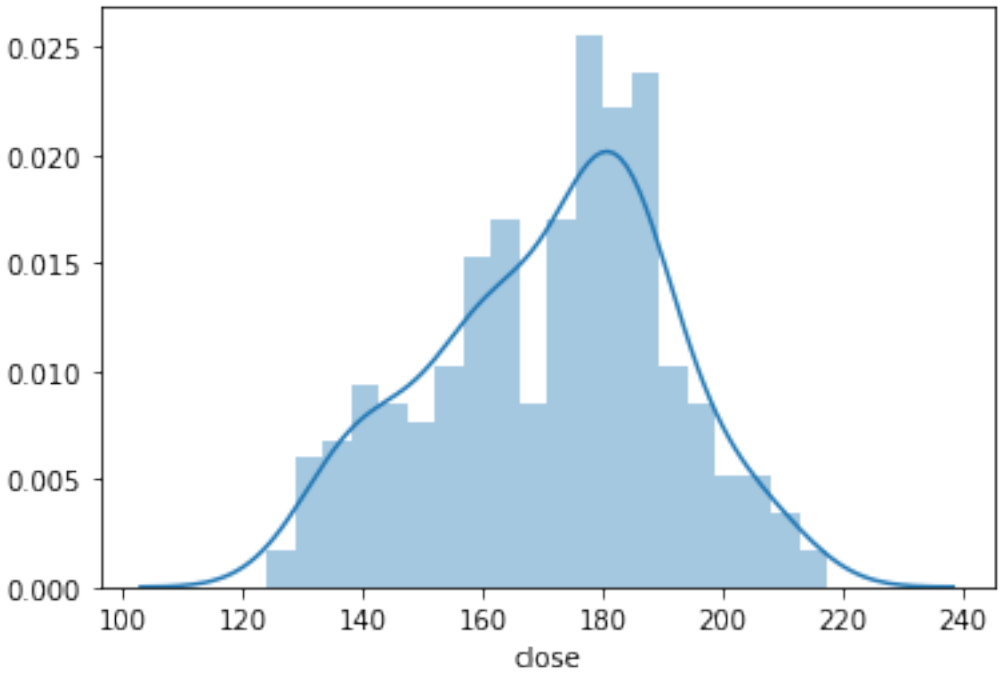
[106]:
```
df.close.hist(bins=20)
```

[106]: <matplotlib.axes._subplots.AxesSubplot at 0x192409daa90>



[107]:
```
sns.distplot(df.close,bins=20)
```

[107]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x19240a9eb00>
```



2.0.3 Plotting columns against each other

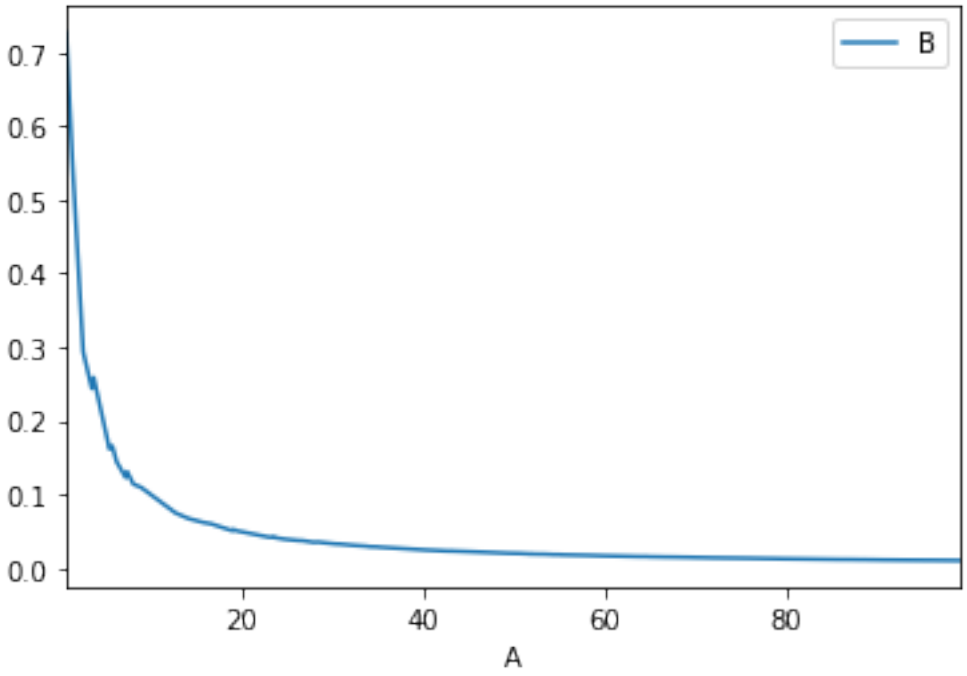
[108]:
```
dff = pd.read_excel('example-functions.xlsx')
dfs = dff.sort_values(by='A', ascending = True) #Sorting in data frames
```

Plot columns B,C,D against A

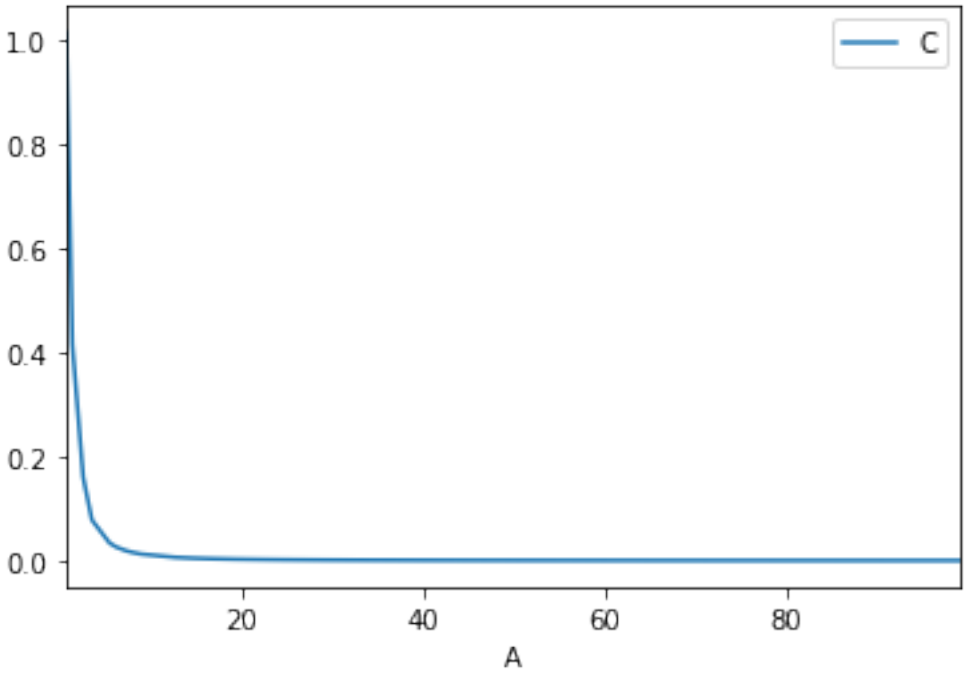
The plt.figure() command creates a new figure for each plot

[109]:
```
plt.figure();
dfs.plot(x = 'A', y = 'B');
plt.figure();
dfs.plot(x = 'A', y = 'C');
plt.figure();
dfs.plot(x = 'A', y = 'D');
```

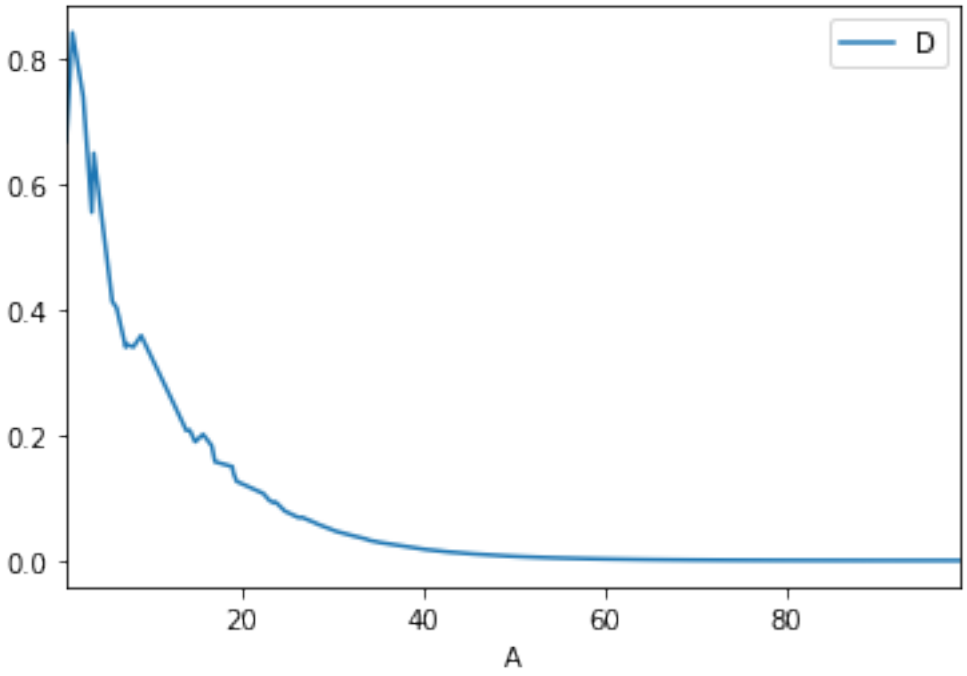
```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```

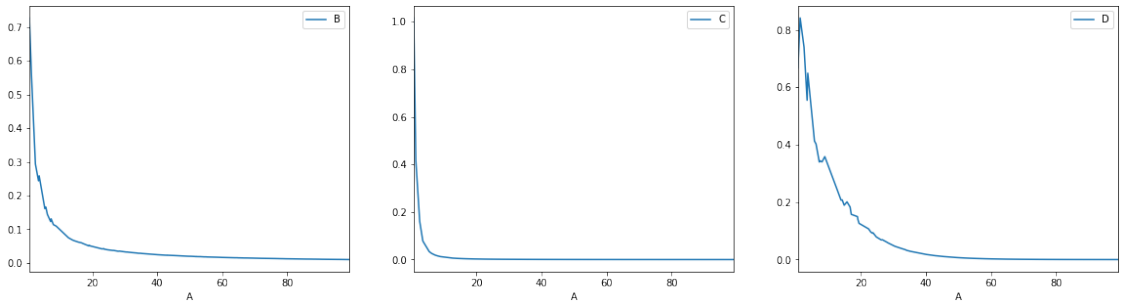



```
<Figure size 432x288 with 0 Axes>
```



Use a grid to put all the plots together

[110]:
```
#plt.figure();
fig, ax = plt.subplots(1, 3,figsize=(20,5))
dfs.plot(x = 'A', y = 'B',ax = ax[0]);
dfs.plot(x = 'A', y = 'C',ax = ax[1]);
dfs.plot(x = 'A', y = 'D',ax = ax[2]);
```

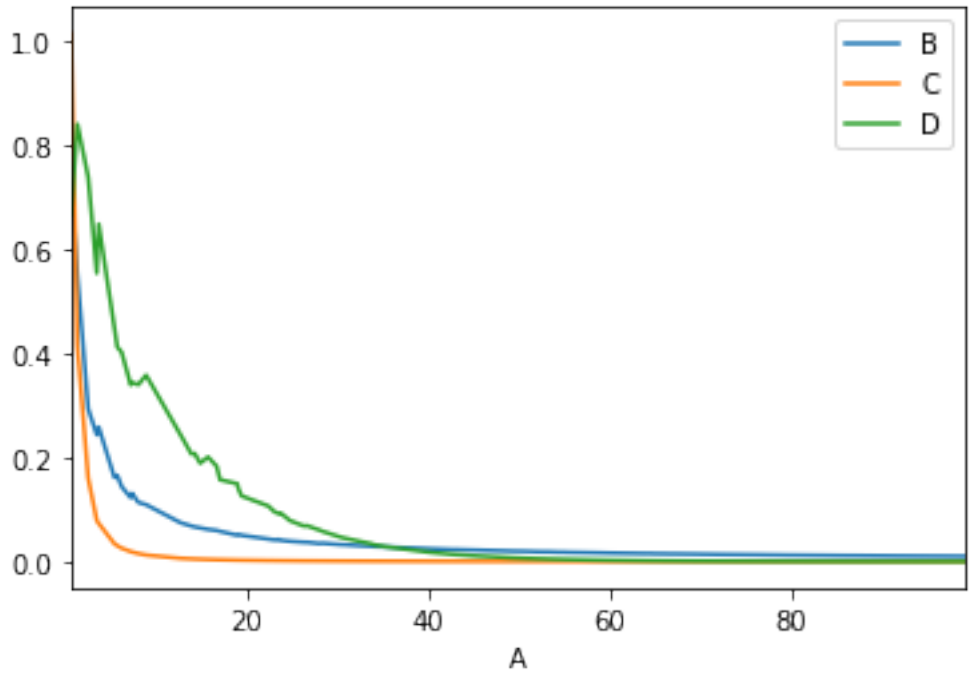


Plot all colums together against A.

Clearly they are different functions

[111]:
```
plt.figure(); dfs.plot(x = 'A', y = ['B','C','D']);
```

```
<Figure size 432x288 with 0 Axes>
```

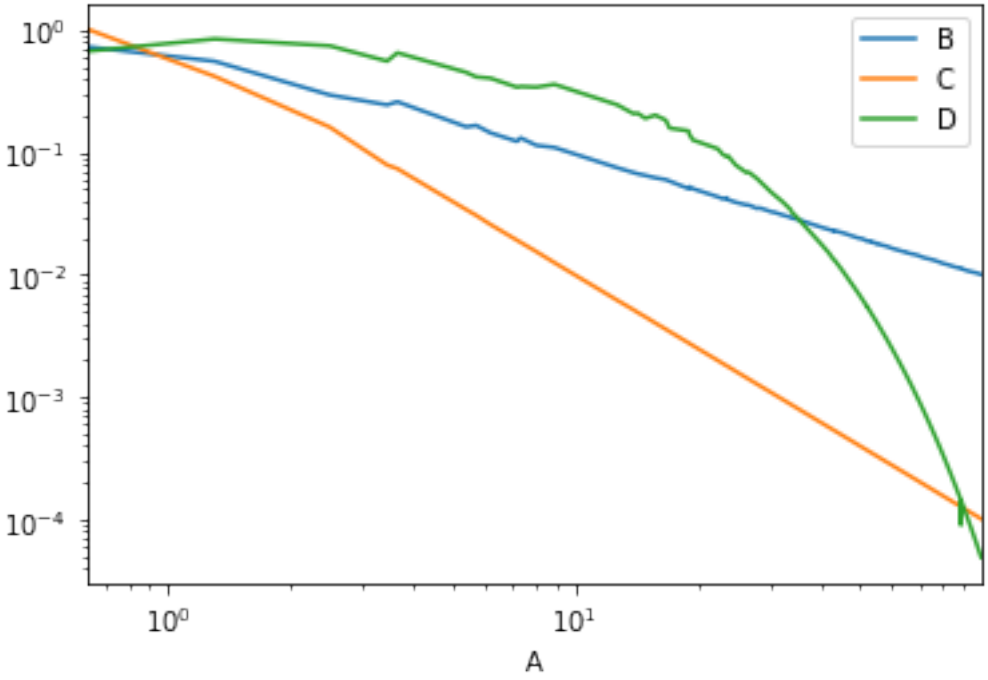


Plot all columns against A in log scale

We observe straight lines for B,C while steeper drop for D

[112]:
```
plt.figure(); dfs.plot(x = 'A', y = ['B','C','D'], loglog=True);
```

```
<Figure size 432x288 with 0 Axes>
```

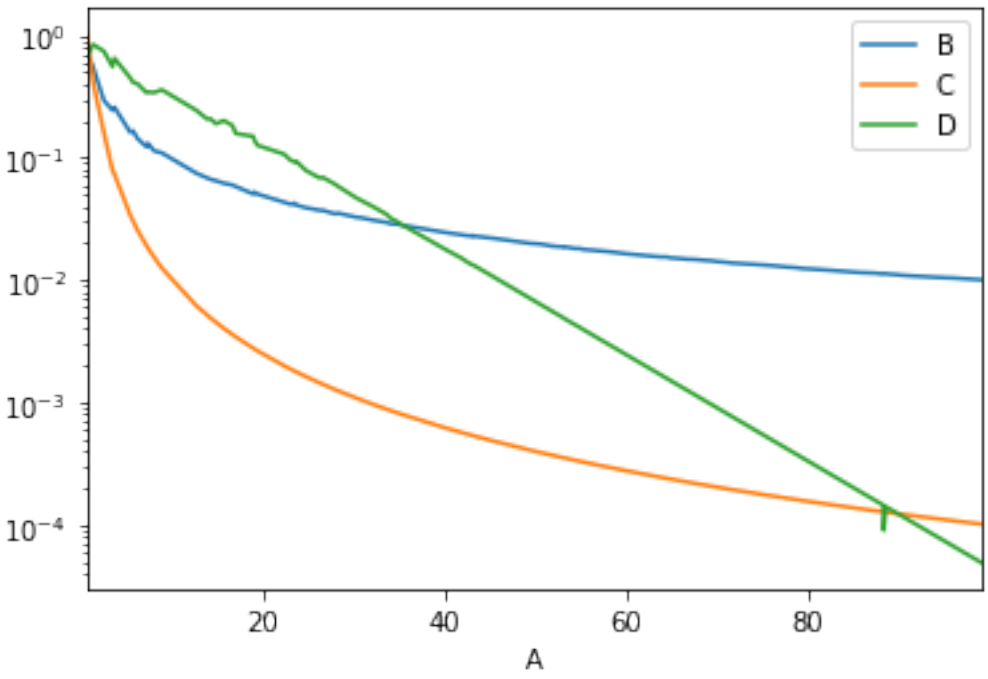


Plot with log scale only on y-axis.

The plot of D becomes a line, indicating that D is an exponential function of A

[113]:
```
plt.figure(); dfs.plot(x = 'A', y = ['B','C','D'], logy=True);
```

```
<Figure size 432x288 with 0 Axes>
```

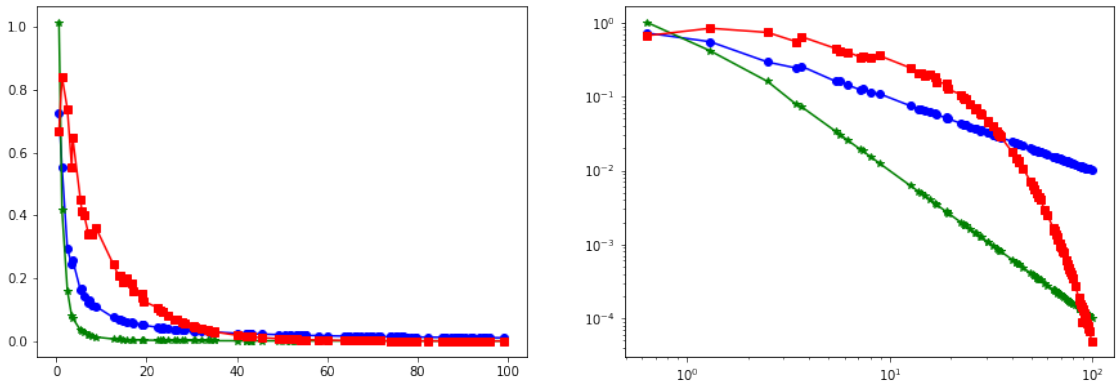


Plotting using matlab notation

Also how to put two figures in a 1x2 grid

[114]:
```
plt.figure(figsize = (15,5)) #defines the size of figure
plt.subplot(121) #plot with 1 row, 2 columns, 1st plot
plt.
 ↪plot(dfs['A'],dfs['B'],'bo-',dfs['A'],dfs['C'],'g*-',dfs['A'],dfs['D'],'rs-')
plt.subplot(122)  #plot with 1 row, 2 columns, 2nd plot
plt.
 ↪loglog(dfs['A'],dfs['B'],'bo-',dfs['A'],dfs['C'],'g*-',dfs['A'],dfs['D'],'rs-')
```

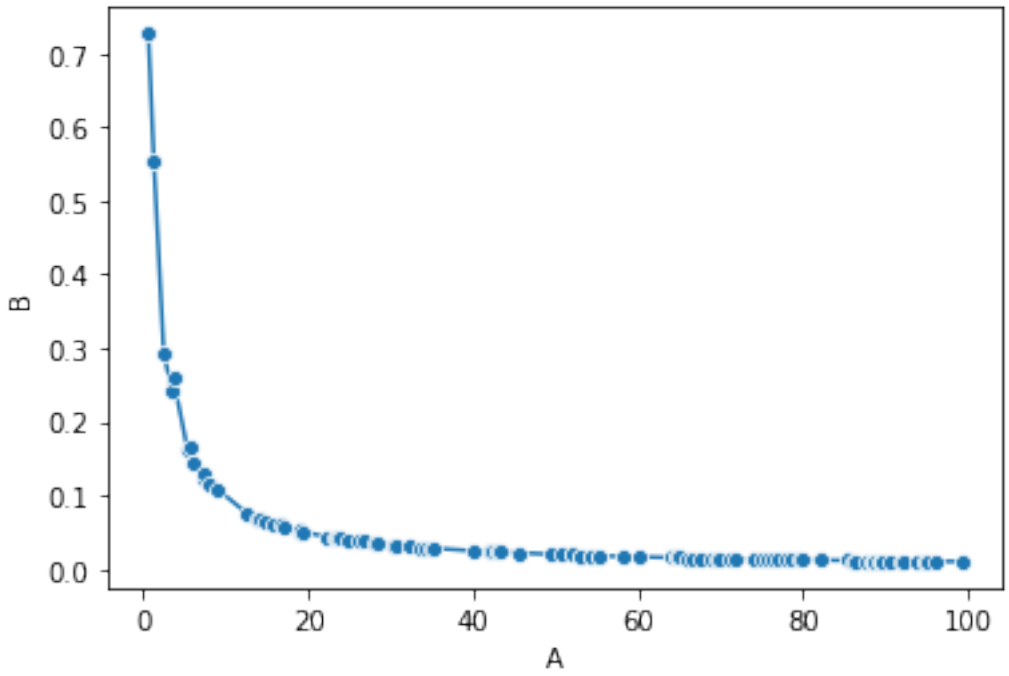
[114]:
```
[<matplotlib.lines.Line2D at 0x19240b5a400>,
 <matplotlib.lines.Line2D at 0x19240be0e48>,
 <matplotlib.lines.Line2D at 0x19240be0f98>]
```



Using seaborn

[115]:
```
sns.lineplot(x= 'A', y='B',data = dfs,marker='o')
```

[115]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x192424d5be0>
```



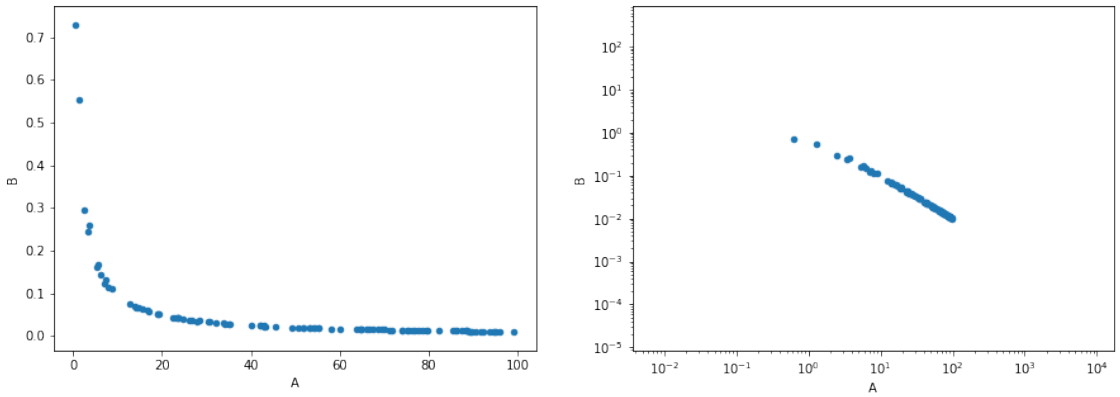
Scatter plots: Scatter plots take as imput two series X and Y and plot the points (x,y).

We will do the same plots as before as scatter plots using the dataframe functions

[116]:
```
fig, ax = plt.subplots(1, 2, figsize=(15,5))
dff.plot(kind ='scatter', x='A', y='B', ax = ax[0])
```

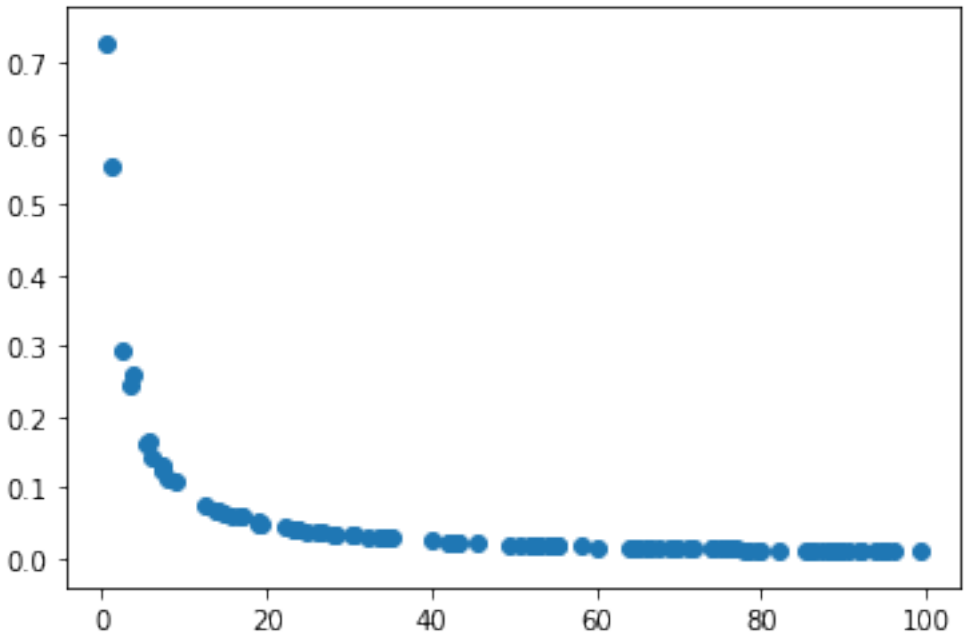
```
dff.plot(kind ='scatter', x='A', y='B', loglog = True,ax = ax[1])
```

[116]: <matplotlib.axes._subplots.AxesSubplot at 0x19242690278>



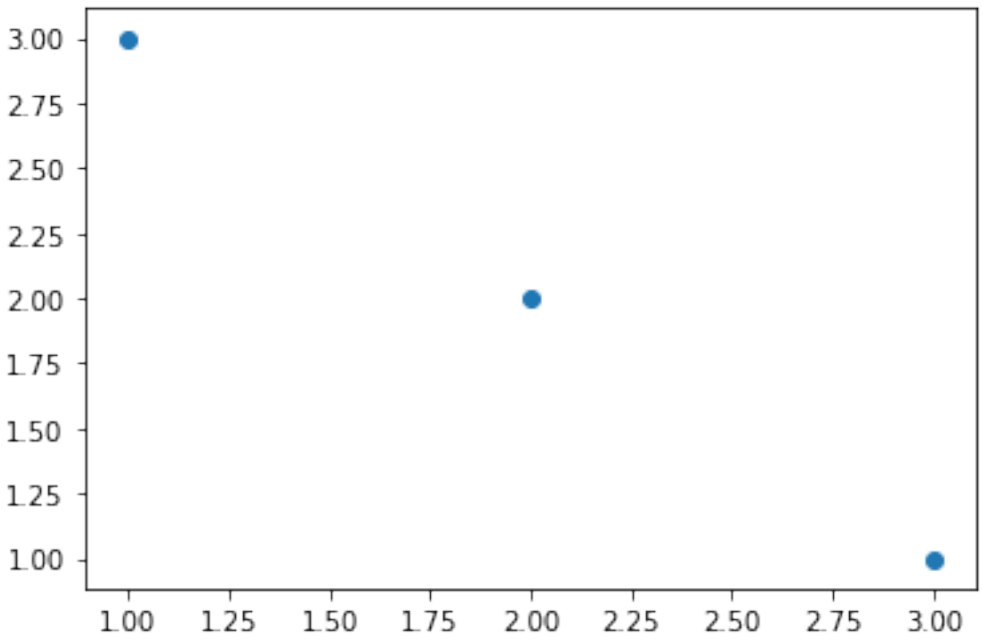
[117]:
```
plt.scatter(dff.A, dff.B)
```

[117]: <matplotlib.collections.PathCollection at 0x19240d4ee48>



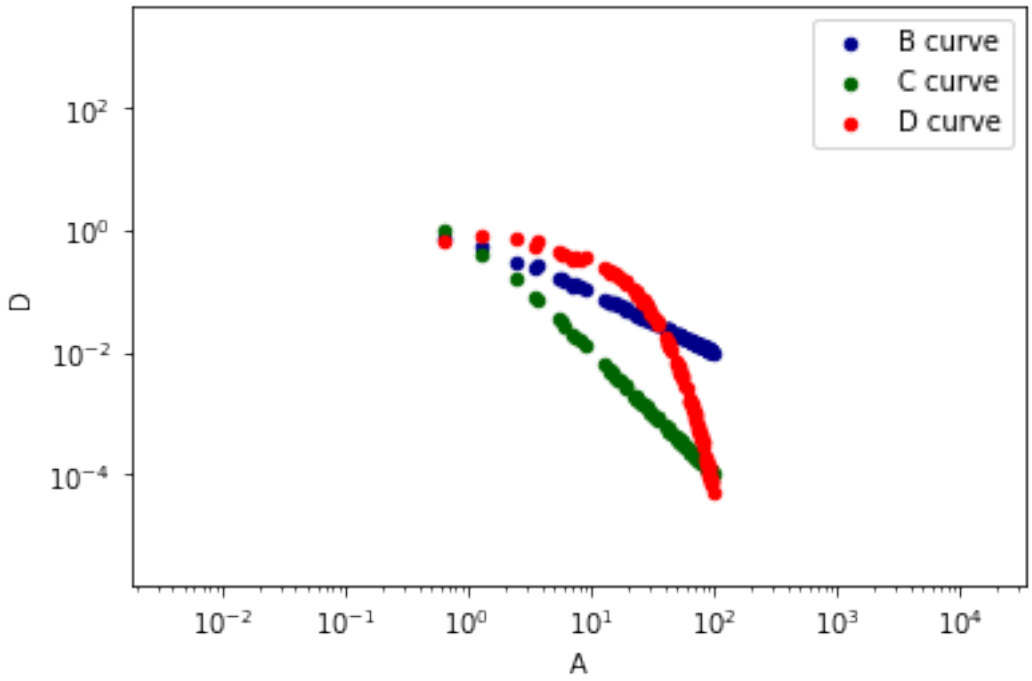
[118]:
```
plt.scatter([1,2,3],[3,2,1])
```

[118]: <matplotlib.collections.PathCollection at 0x19240da4be0>



Putting many scatter plots into the same plot

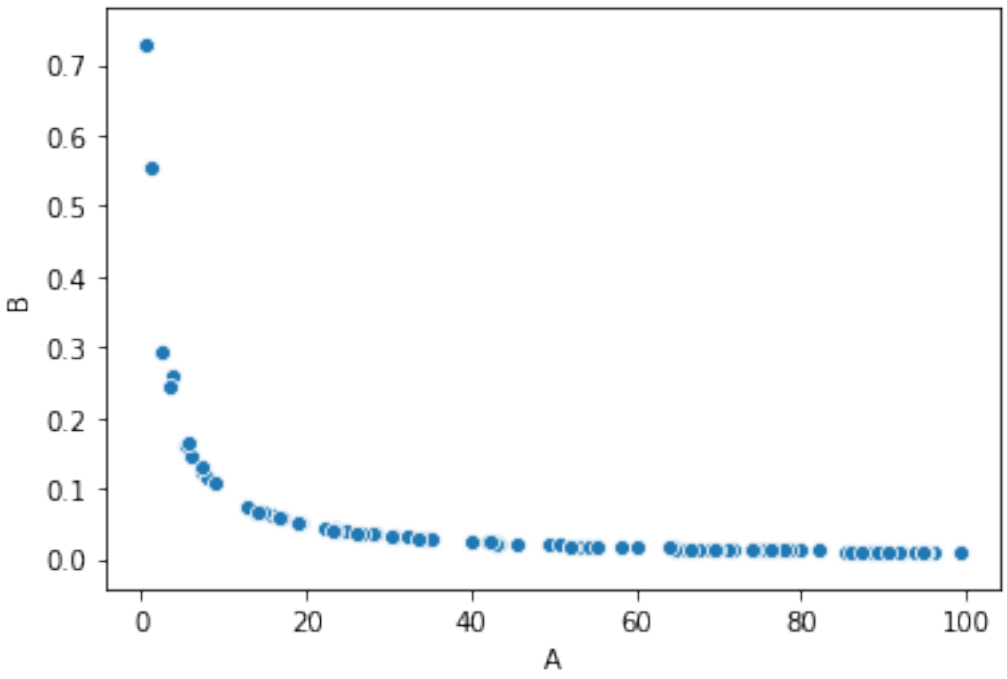
[119]:
```
t = dff.plot(kind='scatter', x='A', y='B', color='DarkBlue', label='B curve',␣
 ↪loglog=True);
dff.plot(kind='scatter', x='A', y='C',color='DarkGreen', label='C curve', ax=t,␣
 ↪loglog = True);
dff.plot(kind='scatter', x='A', y='D',color='Red', label='D curve', ax=t,␣
 ↪loglog = True);
```



Using seaborn

[120]:
```
sns.scatterplot(x='A',y='B', data = dff)
```

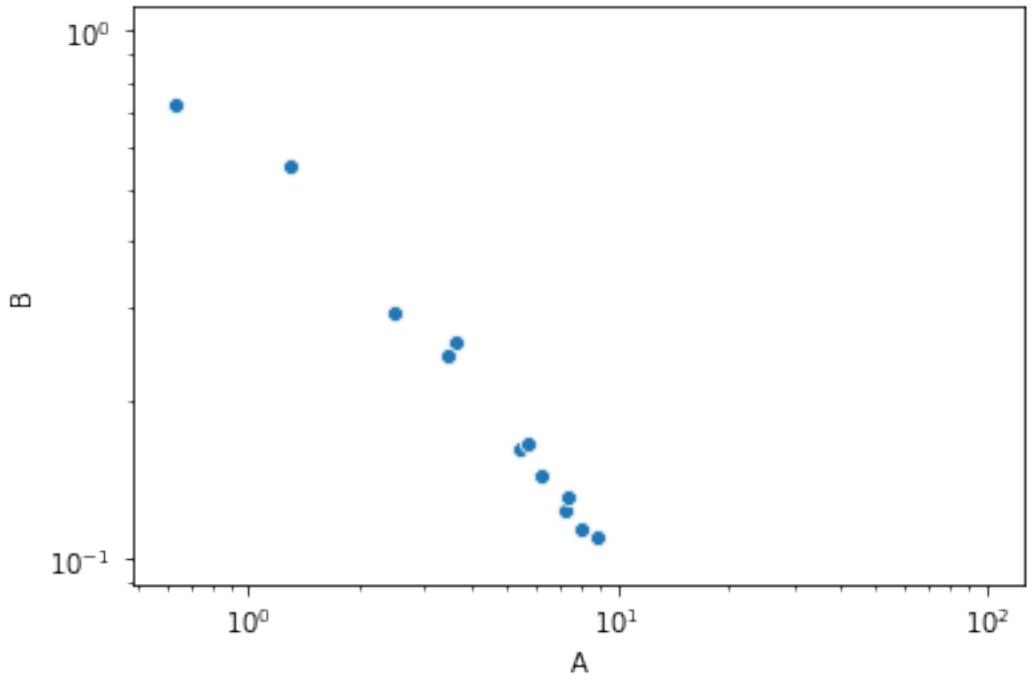
[120]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x192429a8e80>
```



In log-log scale (for some reason it seems to throw away small values)

[121]:
```
splot = sns.scatterplot(x='A',y='B', data = dff)
#splot.set(xscale="log", yscale="log")
splot.loglog()
```

[121]: []



[122]:
```
gain_groups = df.groupby('gain')
```

2.0.4 Statistical Significance

Recall the dataframe we obtained when grouping by gain

[123]:
```
gdf
```

[123]:

	gain	open	low	high	close	vol
0	large_gain	170.459730	169.941351	175.660811	174.990811	3.034571e+07
1	medium_gain	172.305769	171.410962	175.321346	174.185577	2.795407e+07
2	negative	171.473306	168.024545	172.441322	169.233636	2.771124e+07
3	small_gain	171.218049	169.827317	173.070488	171.699268	2.488339e+07

We see that there are differences in the volume of trading depending on the gain. But are these

differences statistically significant? We can test that using the Student t-test. The Student t-test will give us a value for the differnece between the means in units of standard error, and a p-value that says how important this difference is. Usually we require the p-value to be less than 0.05 (or 0.01 if we want to be more strict). Note that for the test we will need to use all the values in the group.

To compute the t-test we will use the **SciPy** library, a Python library for scientific computing.

[124]:
```
import scipy as sp #library for scientific computations
from scipy import stats #The statistics part of the library
```

The t-test value is:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

where $\bar{x}_i$ is the mean value of the i dataset, σ_i^2 is the variance, and n_i is the size.

[125]:
```
#Test statistical significance of the difference in the mean volume numbers

sm = gain_groups.get_group('small_gain').vol
lg = gain_groups.get_group('large_gain').vol
med = gain_groups.get_group('medium_gain').vol
neg = gain_groups.get_group('negative').vol
print(stats.ttest_ind(sm,neg,equal_var = False))
print(stats.ttest_ind(sm,med, equal_var = False))
print(stats.ttest_ind(sm,lg, equal_var = False))
print(stats.ttest_ind(neg,med,equal_var = False))
print(stats.ttest_ind(neg,lg,equal_var = False))
print(stats.ttest_ind(med,lg, equal_var = False))
```

```
Ttest_indResult(statistic=-0.7956394985081949, pvalue=0.429417750163685)
Ttest_indResult(statistic=-0.6701399815165451, pvalue=0.5044832095805987)
Ttest_indResult(statistic=-1.2311419812548245, pvalue=0.22206628199791936)
Ttest_indResult(statistic=-0.06722743349643102, pvalue=0.9465813743143181)
Ttest_indResult(statistic=-0.7690284467674665, pvalue=0.44515731685000515)
Ttest_indResult(statistic=-0.5334654665318221, pvalue=0.5950877691078409)
```

We can compute the standard error of the mean using the stats.sem method of scipy, which can also be called from the data frame

[126]:
```
print(sm.sem())
print(neg.sem())
print(stats.sem(med))
print(stats.sem(lg))
```

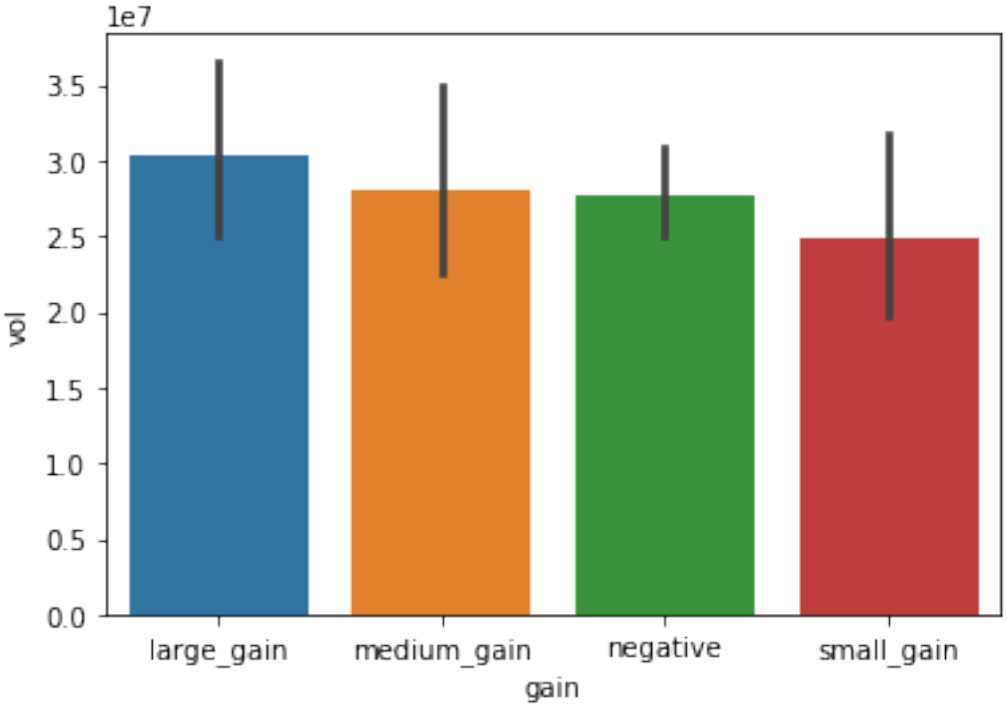
```
3207950.267667195
1530132.8120272094
```

```
3271861.2395884297
3064988.17806777
```

We can also visualize the mean and the standard error in a bar-plot, using the barplot function of seaborn. Note that we need to apply this to the original data. The averaging is done automatically.

[127]:
```
sns.barplot(x='gain',y='vol', data = df)
```

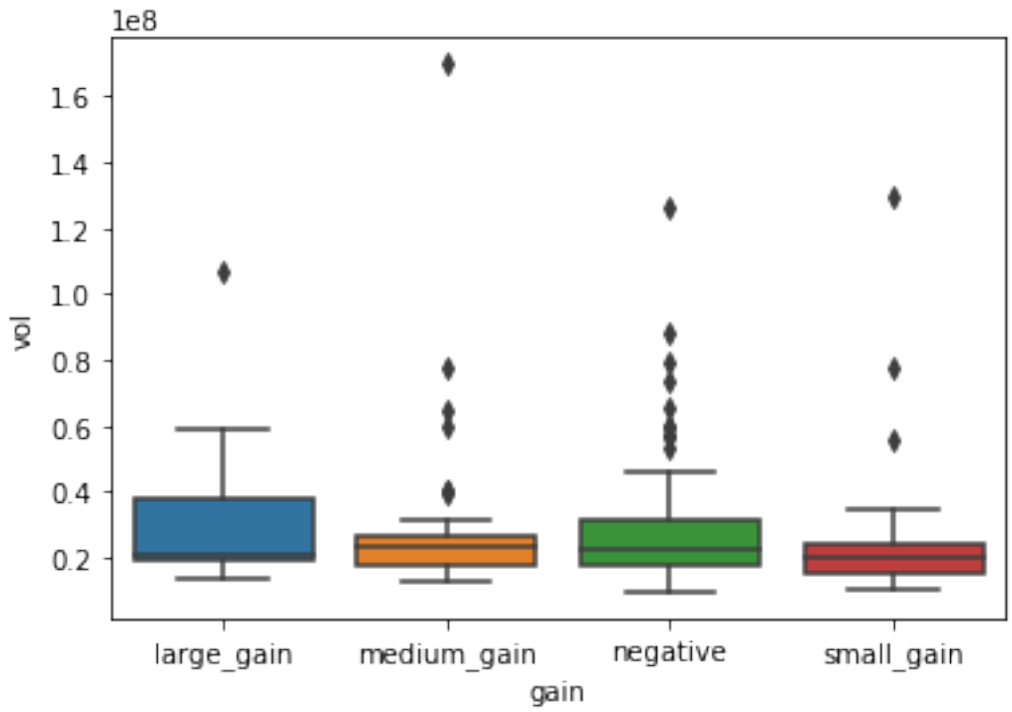
[127]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x192429c0748>
```



We can also visualize the distribution using a **box-plot**. In the box plot, the box shows the quartiles of the dataset (the part between the higher 25% and lower 25%), while the whiskers extend to show the rest of the distribution, except for points that are determined to be "outliers". The line shows the median.

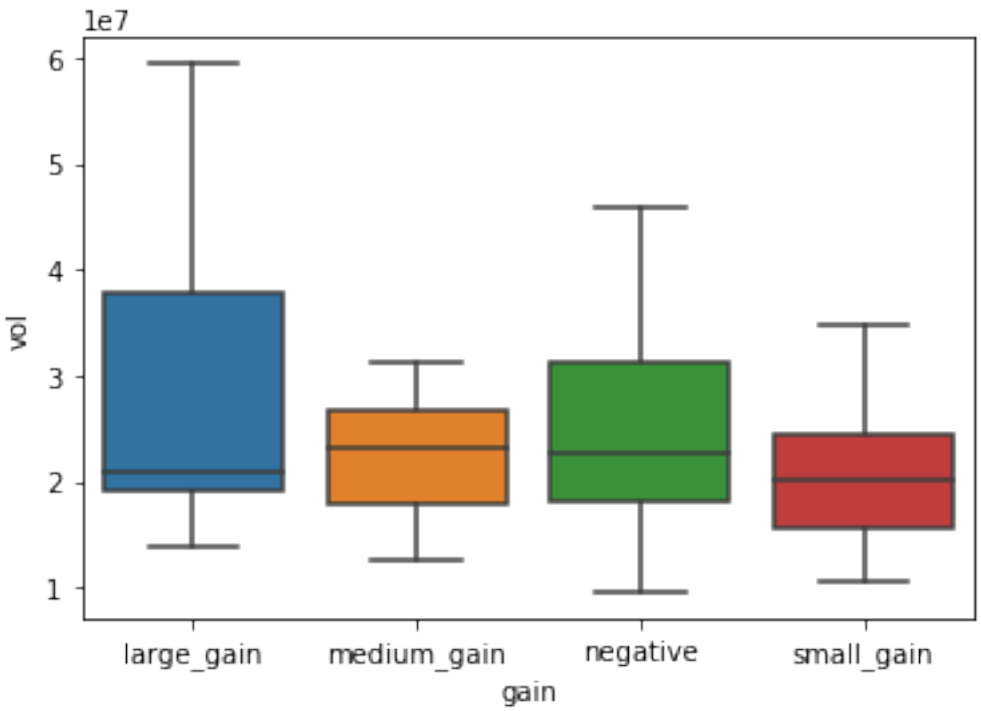
[128]:
```
sns.boxplot(x='gain',y='vol', data = df)
```

[128]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x19241121f60>
```



[129]:
```
#Removing outliers
sns.boxplot(x='gain',y='vol', data = df, showfliers = False)
```

[129]: <matplotlib.axes._subplots.AxesSubplot at 0x192429ef0f0>



Plot the average volume over the different months

[136]:
```
fbdf = fbdf.reset_index()
```

[137]:
```
fbdf.date
```

[137]:
```
0      2018-01-02
1      2018-01-03
2      2018-01-04
3      2018-01-05
4      2018-01-08
          ...
246    2018-12-24
247    2018-12-26
248    2018-12-27
249    2018-12-28
250    2018-12-31
Name: date, Length: 251, dtype: datetime64[ns]
```

[138]:
```
def get_month(row):
    return row.date.month

fbdf['month'] = fbdf.apply(get_month,axis = 1)
fbdf
```

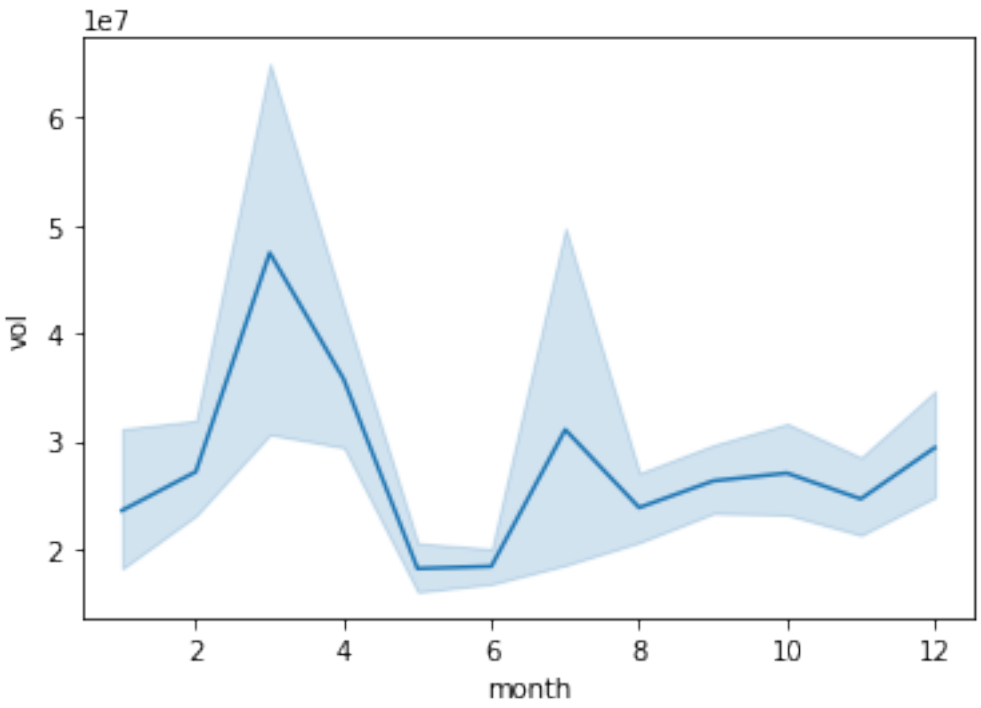
[138]:

	index	date	open	high	low	close	vol	profit	gain	month	positive_profit
0	0	2018-01-02	177.68	181.58	177.55	181.42	18151903	3.74	large_gain	1	True
1	1	2018-01-03	181.88	184.78	181.33	184.67	16886563	2.79	medium_gain	1	True
2	2	2018-01-04	184.90	186.21	184.10	184.33	13880896	-0.57	negative	1	False
3	3	2018-01-05	185.59	186.90	184.93	186.85	13574535	1.26	medium_gain	1	True
4	4	2018-01-08	187.20	188.90	186.33	188.28	17994726	1.08	medium_gain	1	True
..	…	…	…	…	…	…	…	…	…	…	…
246	246	2018-12-24	123.10	129.74	123.02	124.06	22066002	0.96	small_gain	12	True
247	247	2018-12-26	126.00	134.24	125.89	134.18	39723370	8.18	large_gain	12	True
248	248	2018-12-27	132.44	134.99	129.67	134.52	31202509	2.08	medium_gain	12	True
249	249	2018-12-28	135.34	135.92	132.20	133.20	22627569	-2.14	negative	12	False
250	250	2018-12-31	134.45	134.64	129.95	131.09	24625308	-3.36	negative	12	False

[251 rows x 11 columns]

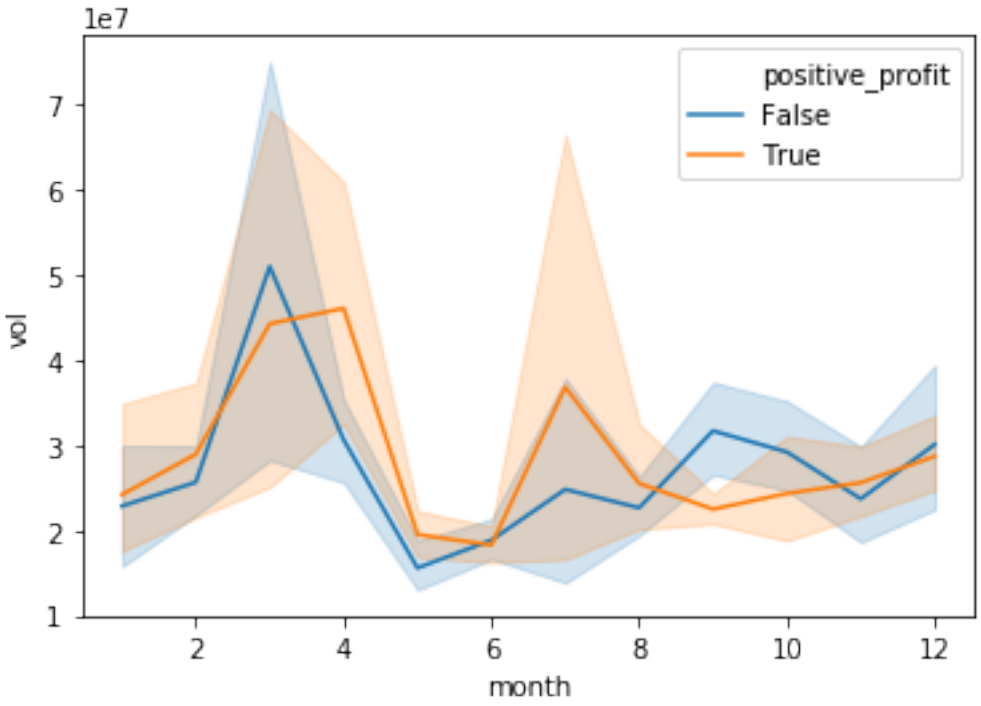
[139]:
```
sns.lineplot(x='month', y = 'vol', data = fbdf)
```

[139]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1924280e208>
```



[140]:
```
fbdf['positive_profit'] = (fbdf.profit>0)
sns.lineplot(x='month', y = 'vol', hue='positive_profit', data = fbdf)
```

[140]: <matplotlib.axes._subplots.AxesSubplot at 0x192428aeeb8>



2.1 Comparing multiple stocks

As a last task, we will use the experience we obtained so far – and learn some new things – in order to compare the performance of different stocks we obtained from Yahoo finance.

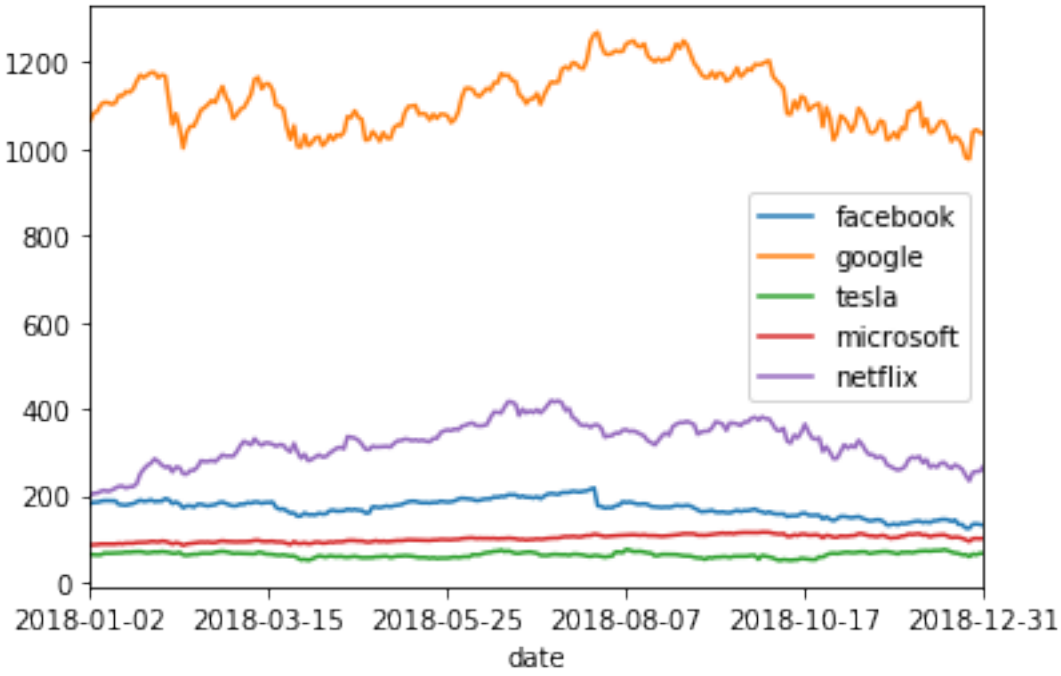
[141]:
```
stocks = ['FB','GOOG','TSLA', 'MSFT','NFLX']
attr = 'close'
dfmany = web.DataReader(stocks,
                    data_source,
                    start=datetime(2018, 1, 1),
                    end=datetime(2018, 12, 31))[attr]
dfmany.head()
```

[141]:

Symbols	FB	GOOG	TSLA	MSFT	NFLX
date					
2018-01-02	181.42	1065.00	64.11	85.95	201.07
2018-01-03	184.67	1082.48	63.45	86.35	205.05
2018-01-04	184.33	1086.40	62.92	87.11	205.63
2018-01-05	186.85	1102.23	63.32	88.19	209.99
2018-01-08	188.28	1106.94	67.28	88.28	212.05

[142]:
```
dfmany.FB.plot(label = 'facebook')
dfmany.GOOG.plot(label = 'google')
dfmany.TSLA.plot(label = 'tesla')
```

```
dfmany.MSFT.plot(label = 'microsoft')
dfmany.NFLX.plot(label = 'netflix')
_ = plt.legend(loc='best')
```



Next, we will calculate returns over a period of length T, defined as:

$$r(t) = \frac{f(t) - f(t-T)}{f(t)}$$

The returns can be computed with a simple DataFrame method **`pct_change()`**. Note that for the first T timesteps, this value is not defined (of course):

[143]:
```
rets = dfmany.pct_change(30)
rets.iloc[25:35]
```

[143]:

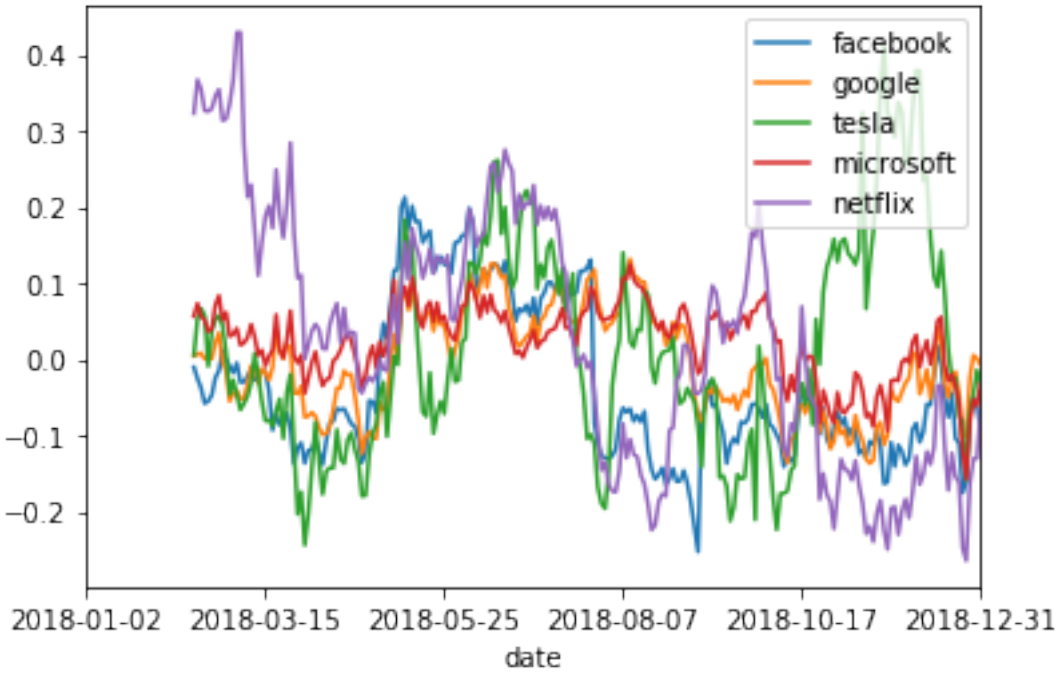
Symbols	FB	GOOG	TSLA	MSFT	NFLX
date					
2018-02-07	NaN	NaN	NaN	NaN	NaN
2018-02-08	NaN	NaN	NaN	NaN	NaN
2018-02-09	NaN	NaN	NaN	NaN	NaN
2018-02-12	NaN	NaN	NaN	NaN	NaN
2018-02-13	NaN	NaN	NaN	NaN	NaN
2018-02-14	-0.010473	0.004413	0.005459	0.056545	0.322922
2018-02-15	-0.025505	0.006504	0.052955	0.073075	0.366837
2018-02-16	-0.037813	0.007732	0.066434	0.056136	0.354472

```
2018-02-20 -0.058014  0.000209  0.057328  0.051366  0.326492
2018-02-21 -0.055078  0.003975 -0.009215  0.036362  0.325348
```

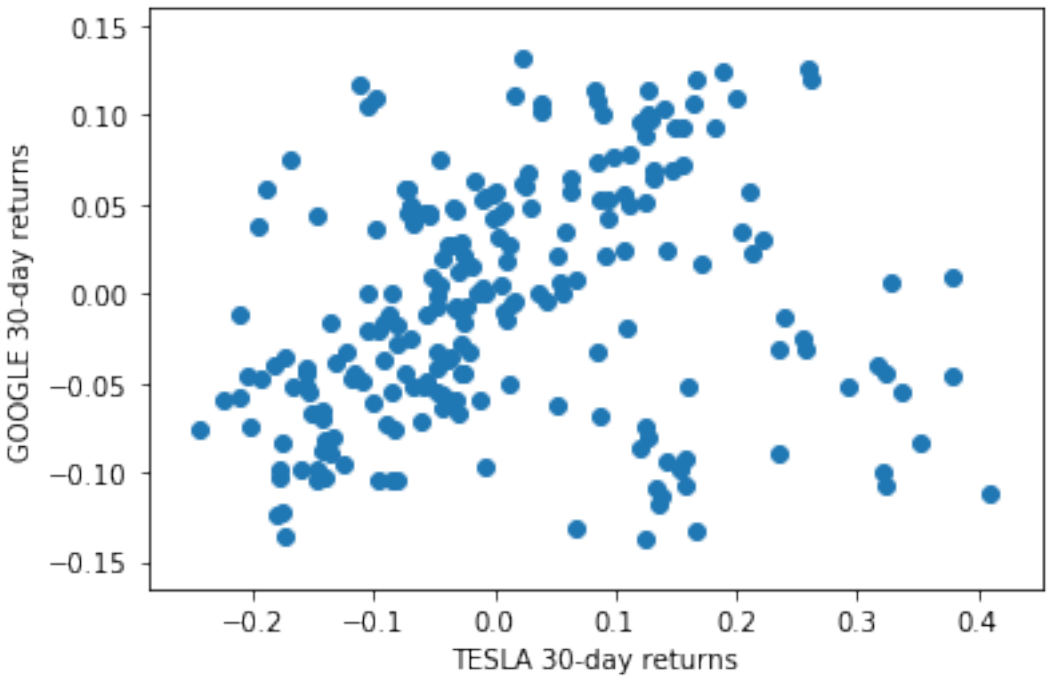
Now we'll plot the timeseries of the returns of the different stocks.

Notice that the `NaN` values are gracefully dropped by the plotting function.

[161]:
```
rets.FB.plot(label = 'facebook')
rets.GOOG.plot(label = 'google')
rets.TSLA.plot(label = 'tesla')
rets.MSFT.plot(label = 'microsoft')
rets.NFLX.plot(label = 'netflix')
_ = plt.legend(loc='best')
```



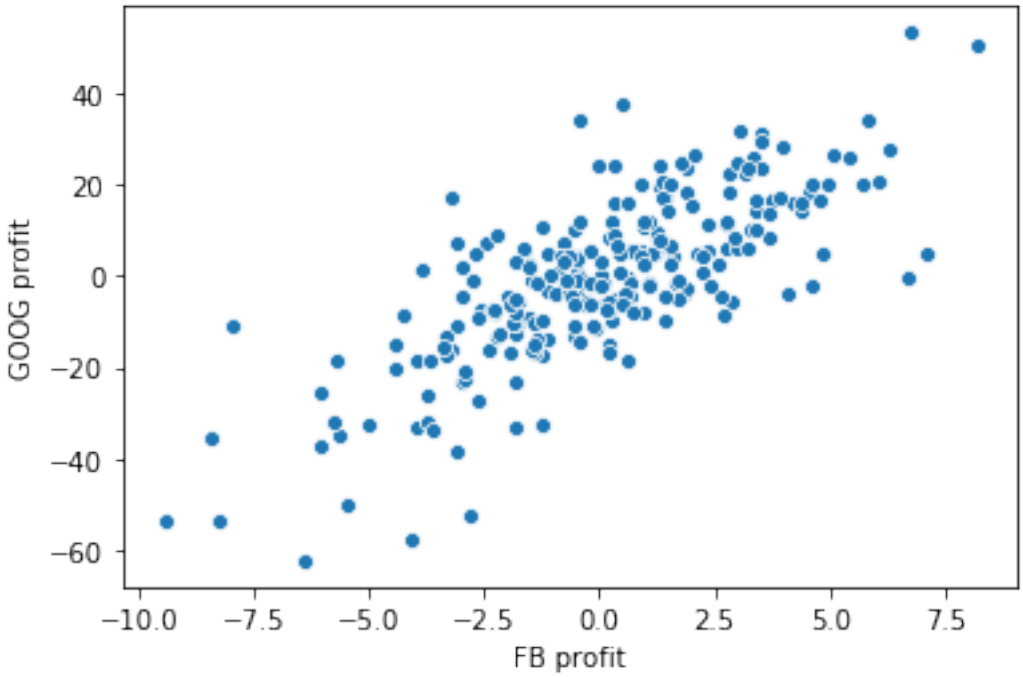
[144]:
```
plt.scatter(rets.TSLA, rets.GOOG)
plt.xlabel('TESLA 30-day returns')
_ = plt.ylabel('GOOGLE 30-day returns')
```



We can also use the seaborn library for doing the scatterplot. Note that this method returns an object which we can use to set different parameters of the plot. In the example below we use it to set the x and y labels of the plot. Read online for more options.

[145]:
```
#Also using seaborn
fig = sns.scatterplot(dfb.profit, dgoog.profit)
fig.set_xlabel('FB profit')
fig.set_ylabel('GOOG profit')
```

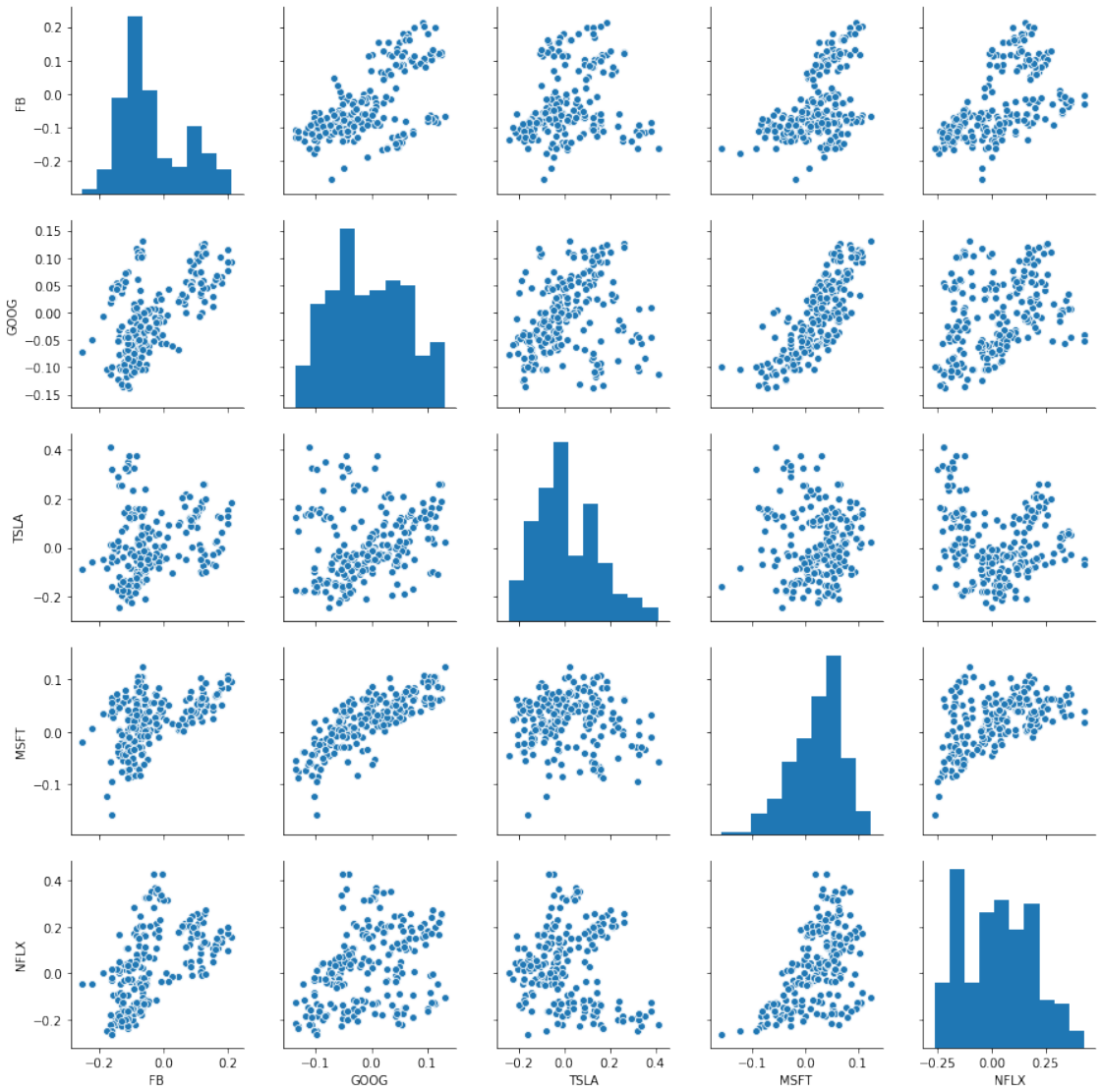
[145]:
```
Text(0, 0.5, 'GOOG profit')
```



Get all pairwise correlations in a single plot

[146]:
```
sns.pairplot(rets.iloc[30:])
```

[146]:
```
<seaborn.axisgrid.PairGrid at 0x19242e72128>
```



There appears to be some (fairly strong) correlation between the movement of TSLA and YELP stocks. Let's measure this.

The correlation coefficient between variables X and Y is defined as follows:

$$\text{Corr}(X, Y) = \frac{E\left[(X - \mu_X)(Y - \mu_Y)\right]}{\sigma_X \sigma_Y}$$

Pandas provides a DataFrame method to compute the correlation coefficient of all pairs of columns: `corr()`.

[147]:
```
rets.corr()
```

[147]:
Symbols	FB	GOOG	TSLA	MSFT	NFLX
Symbols					
FB	1.000000	0.598776	0.226645	0.470696	0.546997
GOOG	0.598776	1.000000	0.210414	0.790085	0.348008
TSLA	0.226645	0.210414	1.000000	-0.041969	-0.120794
MSFT	0.470696	0.790085	-0.041969	1.000000	0.489569
NFLX	0.546997	0.348008	-0.120794	0.489569	1.000000

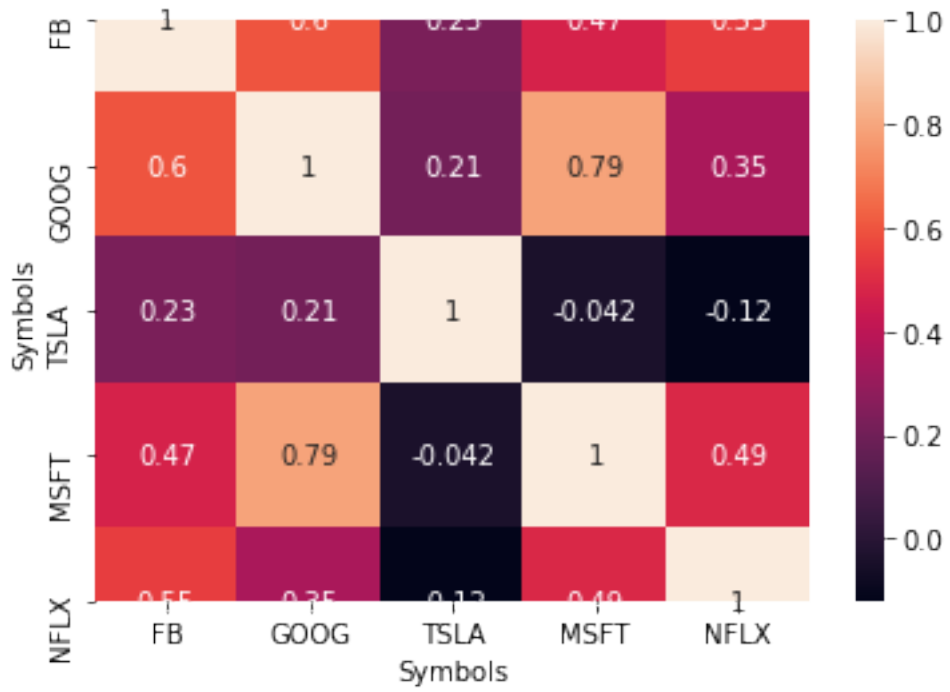
[148]:
```
rets.corr(method='spearman')
```

[148]:
Symbols	FB	GOOG	TSLA	MSFT	NFLX
Symbols					
FB	1.000000	0.540949	0.271626	0.457852	0.641344
GOOG	0.540949	1.000000	0.288171	0.803731	0.382466
TSLA	0.271626	0.288171	1.000000	0.042268	-0.066012
MSFT	0.457852	0.803731	0.042268	1.000000	0.456912
NFLX	0.641344	0.382466	-0.066012	0.456912	1.000000

It takes a bit of time to examine that table and draw conclusions.

To speed that process up it helps to visualize the table using a heatmap.

[149]:
```
_ = sns.heatmap(rets.corr(), annot=True)
```



Use the scipy.stats library to obtain the p-values for the pearson and spearman rank correlations

[150]:
```
print(stats.pearsonr(rets.iloc[30:].NFLX, rets.iloc[30:].TSLA))
print(stats.spearmanr(rets.iloc[30:].NFLX, rets.iloc[30:].TSLA))
print(stats.pearsonr(rets.iloc[30:].GOOG, rets.iloc[30:].FB))
print(stats.spearmanr(rets.iloc[30:].GOOG, rets.iloc[30:].FB))
```

```
(-0.12079364118016642, 0.07311519342514292)
SpearmanrResult(correlation=-0.06601220718867777, pvalue=0.3286469530126206)
(0.5987760976044885, 6.856639483414064e-23)
SpearmanrResult(correlation=0.5409485585956174, pvalue=3.388893335195231e-18)
```

[151]:
```
print(stats.pearsonr(dfb.profit, dgoog.profit))
print(stats.spearmanr(dfb.profit, dgoog.profit))
```

```
(0.7502980828890071, 1.1838784594493575e-46)
SpearmanrResult(correlation=0.7189927028730208, pvalue=3.177135649196623e-41)
```

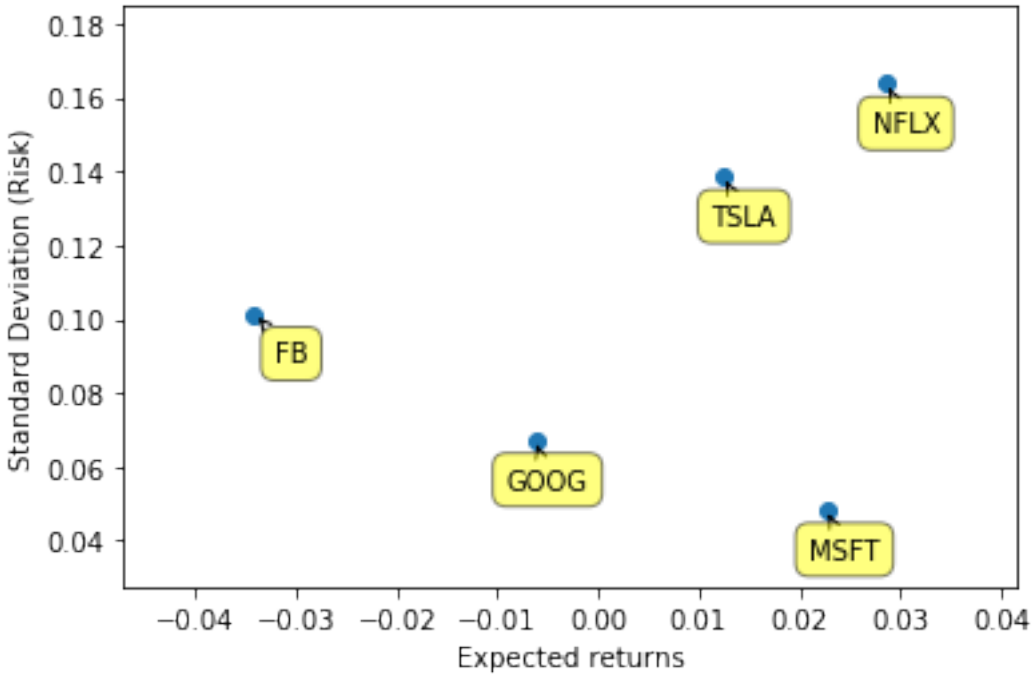
Finally, it is important to know that the plotting performed by Pandas is just a layer on top of `matplotlib` (i.e., the `plt` package).

So Panda's plots can (and should) be replaced or improved by using additional functions from `matplotlib`.

For example, suppose we want to know both the returns as well as the standard deviation of the returns of a stock (i.e., its risk).

Here is visualization of the result of such an analysis, and we construct the plot using only functions from `matplotlib`.

[152]:
```
_ = plt.scatter(rets.mean(), rets.std())
plt.xlabel('Expected returns')
plt.ylabel('Standard Deviation (Risk)')
for label, x, y in zip(rets.columns, rets.mean(), rets.std()):
    plt.annotate(
        label,
        xy = (x, y), xytext = (20, -20),
        textcoords = 'offset points', ha = 'right', va = 'bottom',
        bbox = dict(boxstyle = 'round,pad=0.5', fc = 'yellow', alpha = 0.5),
        arrowprops = dict(arrowstyle = '->', connectionstyle = 'arc3,rad=0'))
```



To understand what these functions are doing, (especially the `annotate` function), you will need to consult the online documentation for matplotlib. Just use Google to find it.