1 Introduction to Pandas an other libraries

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

1.1 Python

In this tutorial we will see Python Data Analysis library, as well as other libraries useful for data processing.

I assume that you all know Python. A brief introduction to Python Basics can be found in this notebook from last year (ipynb, html). Here we will only review the use of list comprehension

1.1.1 List Comprehension

Recall the mathematical notation:

\[ L_1 = \{x^2 : x \in \{0 \ldots 9\}\} \]

\[ L_2 = (1, 2, 4, 8, \ldots, 2^{12}) \]

\[ M = \{x \mid x \in L_1 \text{ and } x \text{ is even}\} \]

[1]: `L1 = [x**2 for x in range(10)]`  # range(n): returns an iterator over the numbers 0, \ldots, n-1
   L2 = [2**i for i in range(13)]
   L3 = [x for x in L1 if x \% 2 == 0]
   print (L1)
   print (L2)
   print (L3)

   [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
   [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
   [0, 4, 16, 36, 64]

[2]: `[x for x in [x**2 for x in range(10)] if x \% 2 == 0]`

[2]: [0, 4, 16, 36, 64]
words = 'The quick brown fox jumps over the lazy dog'.split()
print(words)
upper = [w.upper() for w in words]
print(upper)
stuff = [[w.upper(), w.lower(), len(w)] for w in words]
print(stuff)

['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
['THE', 'QUICK', 'BROWN', 'FOX', 'JUMPS', 'OVER', 'THE', 'LAZY', 'DOG']
[['THE', 'the', 3], ['QUICK', 'quick', 5], ['BROWN', 'brown', 5], ['FOX', 'fox', 3], ['JUMPS', 'jumps', 5], ['OVER', 'over', 4], ['THE', 'the', 3], ['LAZY', 'lazy', 4], ['DOG', 'dog', 3]]
s = input('Give numbers separated by comma: ')
x = [int(n) for n in s.split(',')]
print(x)

Give numbers separated by comma: 1,2,3
[1, 2, 3]

z = [0 for i in range(10)]
print(z)

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

M = [[0 for i in range(10)] for j in range(10)]
for i in range(10): M[i][i] = 1
print(M)

[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]]
	number integers in [0,99]
import random
R = [random.choice(range(100)) for i in range(10)]
print(R)

[56, 93, 73, 24, 81, 73, 99, 44, 14, 22]

[10]: # Removing elements from a list while you iterate it can lead to problems
    L = [1,2,4,5,6,8]
    for x in L:
        if x%2 == 0:
            L.remove(x)
    print(L)

[1, 4, 5, 8]

[11]: #Another way to do this:
    L = [1,2,4,5,6,8]
    L = [x for x in L if x%2 == 1] #creates a new list
    L[:] = [x for x in L if x%2 == 1]
    print(L)

[1, 5]

[12]: L = [1,2,4,5,6,8]
    R = [y for y in L if y%2 == 0]
    for x in R: L.remove(x)
    print(L)

[1, 5]

1.2 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

The most important tool provided by Pandas is the data frame.

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

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

1.3 Getting started
1.4 Fetching, storing and retrieving your 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 shown below

```python
# For accessing web data
import pandas_datareader.data as web
from pandas import Series, DataFrame
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib as mpl
import datetime
import scipy as sp
from scipy import stats
#pd.__version__

# For presenting plots inline
%matplotlib inline

import os
os.environ['IEX_API_KEY'] = 'pk_***********

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']]

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

[16]:  stocks_data.head()

[16]:
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

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

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

stocks_data.date

AttributeError Traceback (most recent call last)
<ipython-input-17-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)
    5065 if self._info_axis._can_hold_identifiers_and_holds_name(name):
    5066     return self[name]
--> 5067 return object.__getattribute__(self, name)
    5068
    5069 def __setattr__(self, name, value):

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

Use describe to get some basic statistics for the data
1.4.1 Reading data from a .csv file

```python
stocks_data.to_csv('stocks_data.csv')
for x in open('stocks_data.csv').readlines()[0:10]:
    print(x.strip())
```

The index values are also printed in the file, together with the column name

```python
df = pd.read_csv('stocks_data.csv')
df.head()
```

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

The number of rows in the DataFrame:

```python
len(df)
```

251

Getting the attribute characteristics

```python
df.info()
```
1.4.2 Other ways to define data frames

```py
[22]: d = {'A': [1., 2., 3., 4.],
       'B': [4., 3., 2., 1.]}
    ddf = pd.DataFrame(d)

[23]: ddf.to_csv('test.csv')
    print(open('test.csv').read())

A,B
0,1.0,4.0
1,2.0,3.0
2,3.0,2.0
3,4.0,1.0

[24]: d = [[1, 2, 3], [4, 5, 6]]
    test_df = pd.DataFrame(d)
    print(test_df)
    test_df.columns = ['A', 'B', 'C']
    test_df.index = ['one', 'two']
    test_df

   0  1  2
   0  1  2  3
   1  4  5  6

[24]:   A  B  C
    one 1  2  3
    two 4  5  6

1.5 Working with data columns

The columns or “features” in your data
df.columns

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:

df = df.rename(columns = {'volume':'vol'})

df.columns = ['date', 'open', 'high', 'low', 'close', 'vol']

df.head()

Selecting a single column from your data

df['open']

0 177.68
1 181.88
2 184.90
3 185.59
4 187.20
5 188.70
...
Another way of selecting a single column from your data

```
[28]: df.open

[28]:
0    177.68
1    181.88
2    184.90
3    185.59
4    187.20
5    188.70
6    186.94
7    188.40
8    178.06
9    181.50
10   179.26
```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>178.13</td>
</tr>
<tr>
<td>12</td>
<td>180.85</td>
</tr>
<tr>
<td>13</td>
<td>180.80</td>
</tr>
<tr>
<td>14</td>
<td>186.05</td>
</tr>
<tr>
<td>15</td>
<td>189.89</td>
</tr>
<tr>
<td>16</td>
<td>187.95</td>
</tr>
<tr>
<td>17</td>
<td>187.75</td>
</tr>
<tr>
<td>18</td>
<td>188.75</td>
</tr>
<tr>
<td>19</td>
<td>183.01</td>
</tr>
<tr>
<td>20</td>
<td>188.37</td>
</tr>
<tr>
<td>21</td>
<td>188.22</td>
</tr>
<tr>
<td>22</td>
<td>192.04</td>
</tr>
<tr>
<td>23</td>
<td>186.93</td>
</tr>
<tr>
<td>24</td>
<td>178.57</td>
</tr>
<tr>
<td>25</td>
<td>184.15</td>
</tr>
<tr>
<td>26</td>
<td>181.01</td>
</tr>
<tr>
<td>27</td>
<td>174.76</td>
</tr>
<tr>
<td>28</td>
<td>177.06</td>
</tr>
<tr>
<td>29</td>
<td>175.62</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>221</td>
<td>142.33</td>
</tr>
<tr>
<td>222</td>
<td>141.07</td>
</tr>
<tr>
<td>223</td>
<td>137.61</td>
</tr>
<tr>
<td>224</td>
<td>127.03</td>
</tr>
<tr>
<td>225</td>
<td>134.40</td>
</tr>
<tr>
<td>226</td>
<td>133.65</td>
</tr>
<tr>
<td>227</td>
<td>133.00</td>
</tr>
<tr>
<td>228</td>
<td>135.75</td>
</tr>
<tr>
<td>229</td>
<td>136.28</td>
</tr>
<tr>
<td>230</td>
<td>135.92</td>
</tr>
<tr>
<td>231</td>
<td>138.26</td>
</tr>
<tr>
<td>232</td>
<td>143.00</td>
</tr>
<tr>
<td>233</td>
<td>140.73</td>
</tr>
<tr>
<td>234</td>
<td>133.82</td>
</tr>
<tr>
<td>235</td>
<td>139.25</td>
</tr>
<tr>
<td>236</td>
<td>139.60</td>
</tr>
<tr>
<td>237</td>
<td>143.88</td>
</tr>
<tr>
<td>238</td>
<td>143.08</td>
</tr>
<tr>
<td>239</td>
<td>145.57</td>
</tr>
<tr>
<td>240</td>
<td>143.34</td>
</tr>
<tr>
<td>241</td>
<td>143.08</td>
</tr>
<tr>
<td>242</td>
<td>141.08</td>
</tr>
<tr>
<td>243</td>
<td>141.21</td>
</tr>
<tr>
<td>244</td>
<td>130.70</td>
</tr>
<tr>
<td>245</td>
<td>133.39</td>
</tr>
<tr>
<td>246</td>
<td>123.10</td>
</tr>
<tr>
<td>247</td>
<td>126.00</td>
</tr>
</tbody>
</table>
We can use the values method to obtain the values of one or more attributes. It returns a numpy array. You can transform it into a list by applying the list() operator.

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

```
<table>
<thead>
<tr>
<th></th>
<th>open</th>
<th>close</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>177.68</td>
<td>181.42</td>
</tr>
<tr>
<td>1</td>
<td>181.88</td>
<td>184.67</td>
</tr>
<tr>
<td>2</td>
<td>184.90</td>
<td>184.33</td>
</tr>
<tr>
<td>3</td>
<td>185.59</td>
<td>186.85</td>
</tr>
<tr>
<td>4</td>
<td>187.20</td>
<td>188.28</td>
</tr>
</tbody>
</table>
```
1.6 Data Frame methods

A DataFrame object has many useful methods.

[33]: df.mean()

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>1.714546e+02</td>
</tr>
<tr>
<td>high</td>
<td>1.736153e+02</td>
</tr>
<tr>
<td>low</td>
<td>1.693031e+02</td>
</tr>
<tr>
<td>close</td>
<td>1.855888e+02</td>
</tr>
<tr>
<td>vol</td>
<td>2.768798e+07</td>
</tr>
<tr>
<td>dtype</td>
<td>float64</td>
</tr>
</tbody>
</table>

[34]: df.std()

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>1.968352e+01</td>
</tr>
<tr>
<td>high</td>
<td>1.942387e+01</td>
</tr>
<tr>
<td>low</td>
<td>2.007437e+01</td>
</tr>
<tr>
<td>close</td>
<td>1.997745e+01</td>
</tr>
<tr>
<td>vol</td>
<td>1.922117e+07</td>
</tr>
<tr>
<td>dtype</td>
<td>float64</td>
</tr>
</tbody>
</table>
```python
[35]: df.median()
open      174.89
          high    176.98
          low     172.83
          close   174.70
          vol     21860931.00
dtype: float64

[36]: df.open.mean()
[36]: 171.45458167330676

[37]: df.high.mean()
[37]: 173.61533864541832

1.6.1 Plotting methods

[38]: df.high.plot()
df.low.plot(label='low')
plt.legend(loc='best') #puts the legend in the best possible position
[38]: <matplotlib.legend.Legend at 0x2220bcb3a20>
```

```python
220
200
180
160
140
120

0      50      100    150    200    250
```

![Plot of high and low values](image)
1.6.2 Histograms

[39]: `df.close.hist(bins=50)`

[39]: `<matplotlib.axes._subplots.AxesSubplot at 0x2220bfba438>`

[40]: `sns.distplot(df.close,bins=50)`

[40]: `<matplotlib.axes._subplots.AxesSubplot at 0x2220c02a7b8>`
1.6.3 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()`.

```python
df.date.head()
```

```
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.

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

```
'2018-01-02'
```

```python
datetime.strptime(first_date, "%Y-%m-%d")
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 anonymous 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):

```
[51]: df.date = df.date.apply(lambda d: datetime.strptime(d, "%Y-%m-%d"))
```

```
[51]:
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]
```

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:

```
[52]: df.index = df.date
```

```
[52]:
     date open  high  low  close  volume
    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
```

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.

```
[47]: df = df.drop(['date'],axis=1) #axis = 0 means rows, axis = 1 means columns
```

```
<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: 11.8 KB
```
1.6.4 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.

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

<table>
<thead>
<tr>
<th>date</th>
<th>2018-05-07 00:00:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>177.35</td>
</tr>
<tr>
<td>high</td>
<td>179.5</td>
</tr>
<tr>
<td>low</td>
<td>177.17</td>
</tr>
<tr>
<td>close</td>
<td>177.97</td>
</tr>
<tr>
<td>volume</td>
<td>18697195</td>
</tr>
</tbody>
</table>

Name: 2018-05-07 00:00:00, dtype: object

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

```python
df.iloc[10:30]  # dataframe with rows from 10 to 30
df.iloc[0:2, [1,3]]  # dataframe with rows 0:2, and the second and fourth columns
```

<table>
<thead>
<tr>
<th>date</th>
<th>2018-01-02</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>177.68</td>
</tr>
<tr>
<td>low</td>
<td>177.55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>date</th>
<th>2018-01-03</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>181.88</td>
</tr>
<tr>
<td>low</td>
<td>181.33</td>
</tr>
</tbody>
</table>

2 To iterate over the rows, use .iterrows()

```python
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:

```python
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:
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

2.1 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.

tmp_high = df.high > 170
tmp_high.head()

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.

sum(tmp_high)

149

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

df[tmp_high].head()

date       open    high    low    close    volume
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

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

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

date       open    high    low    close    volume
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
2018-01-05  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
2018-01-10  2018-01-10  186.94  187.89  185.63  187.84  10529894

very_positive_days = df[df.close-df.open > 5]
very_positive_days.head()
2.2 Creating new columns

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

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

```
<table>
<thead>
<tr>
<th>date</th>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
<th>volume</th>
<th>profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-02</td>
<td>177.68</td>
<td>181.58</td>
<td>177.55</td>
<td>181.42</td>
<td>18151903</td>
<td>3.74</td>
</tr>
<tr>
<td>2018-01-03</td>
<td>181.88</td>
<td>184.78</td>
<td>181.33</td>
<td>184.67</td>
<td>16886563</td>
<td>2.79</td>
</tr>
<tr>
<td>2018-01-04</td>
<td>184.90</td>
<td>186.21</td>
<td>184.10</td>
<td>184.33</td>
<td>13880896</td>
<td>-0.57</td>
</tr>
<tr>
<td>2018-01-05</td>
<td>185.59</td>
<td>186.90</td>
<td>184.93</td>
<td>186.85</td>
<td>13574535</td>
<td>1.26</td>
</tr>
<tr>
<td>2018-01-08</td>
<td>187.20</td>
<td>188.90</td>
<td>186.33</td>
<td>188.28</td>
<td>17994726</td>
<td>1.08</td>
</tr>
</tbody>
</table>
```

```python
df.profit[df.profit>0].describe()
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
```

```python
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()
```

```
<table>
<thead>
<tr>
<th>date</th>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
<th>volume</th>
<th>profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-02</td>
<td>177.68</td>
<td>181.58</td>
<td>177.55</td>
<td>181.42</td>
<td>18151903</td>
<td>3.74</td>
</tr>
<tr>
<td>2018-01-03</td>
<td>181.88</td>
<td>184.78</td>
<td>181.33</td>
<td>184.67</td>
<td>16886563</td>
<td>2.79</td>
</tr>
</tbody>
</table>
```
Here is another, more “functional”, way to accomplish the same thing. Define a function that classifies rows, and apply it to each row.

```python
# [66]:
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
```

```
# [67]:
df.head()
```

```
# [67]:

<table>
<thead>
<tr>
<th>date</th>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
<th>volume</th>
<th>profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-02</td>
<td>177.68</td>
<td>181.58</td>
<td>177.55</td>
<td>181.42</td>
<td>18151903</td>
<td>3.74</td>
</tr>
<tr>
<td>2018-01-03</td>
<td>181.88</td>
<td>184.78</td>
<td>181.33</td>
<td>184.67</td>
<td>16886563</td>
<td>2.79</td>
</tr>
<tr>
<td>2018-01-04</td>
<td>184.90</td>
<td>186.21</td>
<td>184.10</td>
<td>184.33</td>
<td>13880896</td>
<td>-0.57</td>
</tr>
<tr>
<td>2018-01-05</td>
<td>185.59</td>
<td>186.90</td>
<td>184.93</td>
<td>186.85</td>
<td>13574535</td>
<td>1.26</td>
</tr>
<tr>
<td>2018-01-08</td>
<td>187.20</td>
<td>188.90</td>
<td>186.33</td>
<td>188.28</td>
<td>17994726</td>
<td>1.08</td>
</tr>
</tbody>
</table>

# [68]:
def = df.drop('test_column', axis = 1)
df.head()```
2.3 Grouping

An extremely powerful DataFrame method is `groupby()`.
This is entirely analogous 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.

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

[70]: type(gain_groups)
```

```
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.

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

```
<table>
<thead>
<tr>
<th>date</th>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
<th>volume</th>
<th>profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-02</td>
<td>177.68</td>
<td>181.58</td>
<td>177.55</td>
<td>181.42</td>
<td>18151903</td>
<td>3.74</td>
</tr>
<tr>
<td>2018-01-22</td>
<td>180.80</td>
<td>185.39</td>
<td>180.41</td>
<td>185.37</td>
<td>21059464</td>
<td>4.57</td>
</tr>
<tr>
<td>2018-01-23</td>
<td>186.05</td>
<td>189.55</td>
<td>185.55</td>
<td>189.35</td>
<td>25678781</td>
<td>3.30</td>
</tr>
<tr>
<td>2018-01-30</td>
<td>183.01</td>
<td>188.18</td>
<td>181.84</td>
<td>187.12</td>
<td>20858556</td>
<td>4.11</td>
</tr>
<tr>
<td>2018-02-01</td>
<td>188.22</td>
<td>195.32</td>
<td>187.89</td>
<td>193.09</td>
<td>54211293</td>
<td>4.87</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>date</th>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
<th>vol</th>
<th>profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-02</td>
<td>large_gain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2018-01-03</td>
<td>medium_gain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2018-01-04</td>
<td>negative</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2018-01-05</td>
<td>medium_gain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2018-01-08</td>
<td>medium_gain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

large_gain

```
<table>
<thead>
<tr>
<th>date</th>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
<th>vol</th>
<th>profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-02</td>
<td>177.68</td>
<td>181.58</td>
<td>177.55</td>
<td>181.42</td>
<td>18151903</td>
<td>3.74</td>
</tr>
<tr>
<td>2018-01-22</td>
<td>180.80</td>
<td>185.39</td>
<td>180.41</td>
<td>185.37</td>
<td>21059464</td>
<td>4.57</td>
</tr>
<tr>
<td>2018-01-23</td>
<td>186.05</td>
<td>189.55</td>
<td>185.55</td>
<td>189.35</td>
<td>25678781</td>
<td>3.30</td>
</tr>
<tr>
<td>2018-01-30</td>
<td>183.01</td>
<td>188.18</td>
<td>181.84</td>
<td>187.12</td>
<td>20858556</td>
<td>4.11</td>
</tr>
<tr>
<td>2018-02-01</td>
<td>188.22</td>
<td>195.32</td>
<td>187.89</td>
<td>193.09</td>
<td>54211293</td>
<td>4.87</td>
</tr>
</tbody>
</table>
```
date
2018-01-02  large_gain
2018-01-22  large_gain
2018-01-23  large_gain
2018-01-30  large_gain
2018-02-01  large_gain

medium_gain

<table>
<thead>
<tr>
<th>date</th>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
<th>vol</th>
<th>profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-03</td>
<td>181.88</td>
<td>184.78</td>
<td>181.33</td>
<td>184.67</td>
<td>16886563</td>
<td>2.79</td>
</tr>
<tr>
<td>2018-01-05</td>
<td>185.59</td>
<td>186.90</td>
<td>184.93</td>
<td>186.85</td>
<td>13574535</td>
<td>1.26</td>
</tr>
<tr>
<td>2018-01-08</td>
<td>187.20</td>
<td>188.90</td>
<td>186.33</td>
<td>188.28</td>
<td>17994726</td>
<td>1.08</td>
</tr>
<tr>
<td>2018-01-12</td>
<td>178.06</td>
<td>181.48</td>
<td>177.40</td>
<td>179.37</td>
<td>77551299</td>
<td>1.31</td>
</tr>
<tr>
<td>2018-01-18</td>
<td>178.13</td>
<td>180.98</td>
<td>177.08</td>
<td>179.80</td>
<td>23304901</td>
<td>1.67</td>
</tr>
</tbody>
</table>

gain

date
2018-01-03  medium_gain
2018-01-05  medium_gain
2018-01-08  medium_gain
2018-01-12  medium_gain
2018-01-18  medium_gain

effective

date
2018-01-04  negative
2018-01-09  negative
2018-01-11  negative
2018-01-16  negative
2018-01-17  negative

gain

date
2018-01-04  negative
2018-01-09  negative
2018-01-11  negative
2018-01-16  negative
2018-01-17  negative

small_gain

<table>
<thead>
<tr>
<th>date</th>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
<th>vol</th>
<th>profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-10</td>
<td>186.94</td>
<td>187.89</td>
<td>185.63</td>
<td>187.84</td>
<td>10529894</td>
<td>0.90</td>
</tr>
<tr>
<td>2018-01-19</td>
<td>180.85</td>
<td>182.37</td>
<td>180.17</td>
<td>181.29</td>
<td>26826540</td>
<td>0.44</td>
</tr>
<tr>
<td>2018-02-20</td>
<td>175.77</td>
<td>177.95</td>
<td>175.11</td>
<td>176.01</td>
<td>21204921</td>
<td>0.24</td>
</tr>
<tr>
<td>2018-02-22</td>
<td>178.70</td>
<td>180.21</td>
<td>177.41</td>
<td>178.99</td>
<td>18464192</td>
<td>0.29</td>
</tr>
</tbody>
</table>
We can obtain the dataframe that corresponds to a specific group by using the get_group method of the groupby object

```python
sm = gain_groups.get_group('small_gain')
sm.head()
```

```plaintext
<table>
<thead>
<tr>
<th>date</th>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
<th>vol</th>
<th>profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-10</td>
<td>186.94</td>
<td>187.89</td>
<td>185.63</td>
<td>187.84</td>
<td>10529894</td>
<td>0.90</td>
</tr>
<tr>
<td>2018-01-19</td>
<td>180.85</td>
<td>182.37</td>
<td>180.17</td>
<td>181.29</td>
<td>26826540</td>
<td>0.44</td>
</tr>
<tr>
<td>2018-02-20</td>
<td>175.77</td>
<td>177.95</td>
<td>175.11</td>
<td>176.01</td>
<td>21204921</td>
<td>0.24</td>
</tr>
<tr>
<td>2018-02-22</td>
<td>178.70</td>
<td>180.21</td>
<td>177.41</td>
<td>178.99</td>
<td>18464192</td>
<td>0.29</td>
</tr>
<tr>
<td>2018-02-26</td>
<td>184.58</td>
<td>185.66</td>
<td>183.22</td>
<td>184.93</td>
<td>17599703</td>
<td>0.35</td>
</tr>
</tbody>
</table>
```

The average closing value for the large_gain group is 174.99081081081084
The average closing value for the medium_gain group is 174.1857692307695
The average closing value for the negative group is 169.2336363636363
The average closing value for the small_gain group is 171.69926829268292

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

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 volume value for the negative group is 22627569.0
The median volume 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 aggregate 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

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

```
pandas.core.frame.DataFrame
```

```
#This can be used to remove the hierarchical index, if necessary
```
```python
[82]: gdf
```
```
<table>
<thead>
<tr>
<th>gain</th>
<th>open</th>
<th>low</th>
<th>high</th>
<th>close</th>
<th>vol</th>
</tr>
</thead>
<tbody>
<tr>
<td>large_gain</td>
<td>170.459730</td>
<td>169.941351</td>
<td>175.660811</td>
<td>174.990811</td>
<td>3.034571e+07</td>
</tr>
<tr>
<td>medium_gain</td>
<td>172.305769</td>
<td>171.410962</td>
<td>175.321346</td>
<td>174.185577</td>
<td>2.795407e+07</td>
</tr>
<tr>
<td>negative</td>
<td>171.473140</td>
<td>168.024545</td>
<td>172.441322</td>
<td>169.233636</td>
<td>2.771124e+07</td>
</tr>
<tr>
<td>small_gain</td>
<td>171.218049</td>
<td>169.827317</td>
<td>173.070488</td>
<td>171.699268</td>
<td>2.488339e+07</td>
</tr>
</tbody>
</table>
```

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 difference 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.

The t-test value is:

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

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

```python
[89]: #Test statistical significance of the difference in the mean volume numbers
```
```python
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))
```
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:

```python
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.

```python
sns.barplot(x='gain', y='vol', data = df)
```

![Bar plot showing mean and standard error for different gain categories](image)
#Removing outliers

sns.boxplot(x='gain', y='vol', data=df, showfliers=False)

<matplotlib.axes._subplots.AxesSubplot at 0x2220c29e9b0>

[92]: 

sns.boxplot(x='gain', y='vol', data=df, showfliers=False)

[92]: <matplotlib.axes._subplots.AxesSubplot at 0x2220c2811d0>

[93]: 

#Removing outliers

sns.boxplot(x='gain', y='vol', data=df, showfliers=False)

[93]: <matplotlib.axes._subplots.AxesSubplot at 0x2220c2de9b0>
Plot the average volume over the different months

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

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

[95]: sns.lineplot(x='month', y = 'vol', data = df)

[95]: <matplotlib.axes._subplots.AxesSubplot at 0x2220c404b00>
```
df['positive_profit'] = (df.profit>0)
sns.lineplot(x='month', y='vol', hue='positive_profit', data = df)
2.4 Joins

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

```python
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)

```python
common_dates = pd.merge(dfb,dgoog,on='date')
common_dates.head()  
```

```python
open_x high_x low_x close_x volume_x open_y high_y low_y close_y volume_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

open_x high_x low_x close_x volume_x open_y high_y low_y close_y volume_y
---
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
```

```
Compute gain and perform join on the data AND gain

\[
\text{dfb['gain']} = \text{dfb.apply(gainrow, axis = 1)} \\
\text{dgoog['gain']} = \text{dgoog.apply(gainrow, axis = 1)} \\
\text{dfb['profit']} = \text{dfb.close - dfb.open} \\
\text{dgoog['profit']} = \text{dgoog.close - dgoog.open}
\]

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

<table>
<thead>
<tr>
<th>open_x</th>
<th>high_x</th>
<th>low_x</th>
<th>close_x</th>
<th>volume_x</th>
<th>gain</th>
<th>profit_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-02</td>
<td>177.68</td>
<td>181.58</td>
<td>177.55</td>
<td>181.42</td>
<td>18151903</td>
<td>large_gain 3.74</td>
</tr>
<tr>
<td>2018-01-04</td>
<td>184.90</td>
<td>186.21</td>
<td>184.10</td>
<td>184.33</td>
<td>13880896</td>
<td>negative -0.57</td>
</tr>
<tr>
<td>2018-01-09</td>
<td>188.70</td>
<td>188.80</td>
<td>187.10</td>
<td>187.87</td>
<td>12393057</td>
<td>negative -0.83</td>
</tr>
<tr>
<td>2018-01-11</td>
<td>184.40</td>
<td>184.40</td>
<td>187.38</td>
<td>187.77</td>
<td>9588587</td>
<td>negative -0.63</td>
</tr>
<tr>
<td>2018-01-16</td>
<td>181.50</td>
<td>181.75</td>
<td>178.04</td>
<td>178.39</td>
<td>36183842</td>
<td>negative -3.11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>open_y</th>
<th>high_y</th>
<th>low_y</th>
<th>close_y</th>
<th>volume_y</th>
<th>profit_y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-02</td>
<td>1048.34</td>
<td>1066.94</td>
<td>1045.23</td>
<td>1065.00</td>
<td>1237564</td>
</tr>
<tr>
<td>2018-01-04</td>
<td>1088.00</td>
<td>1093.57</td>
<td>1084.00</td>
<td>1086.40</td>
<td>1004605</td>
</tr>
<tr>
<td>2018-01-09</td>
<td>1109.40</td>
<td>1110.57</td>
<td>1101.23</td>
<td>1106.26</td>
<td>902541</td>
</tr>
<tr>
<td>2018-01-11</td>
<td>1106.30</td>
<td>1106.53</td>
<td>1099.59</td>
<td>1105.52</td>
<td>978292</td>
</tr>
<tr>
<td>2018-01-16</td>
<td>1132.51</td>
<td>1139.91</td>
<td>1117.83</td>
<td>1121.76</td>
<td>1575261</td>
</tr>
</tbody>
</table>

More join examples, including left outer join

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

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

dfm

<table>
<thead>
<tr>
<th>key</th>
<th>lval</th>
<th>rval</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>1</td>
<td>4.0</td>
</tr>
<tr>
<td>foo</td>
<td>2</td>
<td>4.0</td>
</tr>
<tr>
<td>boo</td>
<td>3</td>
<td>NaN</td>
</tr>
</tbody>
</table>

2.5 Other Pandas Classes

A DataFrame is essentially an annotated 2-D array. Pandas also has annotated versions of 1-D and 3-D arrays. A 1-D array in Pandas is called a **Series**. A 3-D array in Pandas is called a **Panel**. To use these, read the documentation!

2.6 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.

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

<table>
<thead>
<tr>
<th>Symbols</th>
<th>FB</th>
<th>GOOG</th>
<th>MSFT</th>
<th>NFLX</th>
<th>TSLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2018-01-02</td>
<td>181.42</td>
<td>1065.00</td>
<td>85.95</td>
<td>201.07</td>
<td>320.53</td>
</tr>
<tr>
<td>2018-01-03</td>
<td>184.67</td>
<td>1082.48</td>
<td>86.35</td>
<td>205.05</td>
<td>317.25</td>
</tr>
<tr>
<td>2018-01-04</td>
<td>184.33</td>
<td>1086.40</td>
<td>87.11</td>
<td>205.63</td>
<td>314.62</td>
</tr>
<tr>
<td>2018-01-05</td>
<td>186.85</td>
<td>1102.23</td>
<td>88.19</td>
<td>209.99</td>
<td>316.58</td>
</tr>
<tr>
<td>2018-01-08</td>
<td>188.28</td>
<td>1106.94</td>
<td>88.28</td>
<td>212.05</td>
<td>336.41</td>
</tr>
</tbody>
</table>

```python
df.FB.plot(label = 'facebook')
df.GOOG.plot(label = 'google')
df.TSLA.plot(label = 'tesla')
df.MSFT.plot(label = 'microsoft')
```
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):

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

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.

```python
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')
```

```python
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.

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

[110]: Text(0, 0.5, 'GOOG profit')
Get all pairwise correlations in a single plot

```python
[116] sns.pairplot(rets.iloc[30:]
```

```python
[116] <seaborn.axisgrid.PairGrid at 0x2220e236d68>
```
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:

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

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

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

<table>
<thead>
<tr>
<th>Symbols</th>
<th>FB</th>
<th>GOOG</th>
<th>MSFT</th>
<th>NFLX</th>
<th>TSLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>FB</td>
<td>1.000000</td>
<td>0.598776</td>
<td>0.470696</td>
<td>0.546997</td>
<td>0.226680</td>
</tr>
<tr>
<td>GOOG</td>
<td>0.598776</td>
<td>1.000000</td>
<td>0.790085</td>
<td>0.348008</td>
<td>0.210441</td>
</tr>
</tbody>
</table>
It takes a bit of time to examine that table and draw conclusions. To speed that process up it helps to visualize the table. We will learn more about visualization later, but for now this is a simple example.

```
rets.corr(method='spearman')
```

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

```
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))
```
(0.5987760976044885, 6.856639483413375e-23)
SpearmanResult(correlation=0.5409485585956174, pvalue=3.388893335195231e-18)

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.

```python
_ = 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.

### 2.7 More plotting

```python
[125]: df = pd.read_csv('distributions_short.csv',
                        names=list('ABCD'))
dfs = df.sort_values(by='A', ascending=True) #Sorting in data frames

Plot column B against A
The plt.figure() command creates a new figure for each plot

[135]: plt.figure(); dfs.plot(x='A', y='B');

<Figure size 432x288 with 0 Axes>
Plot both columns B and C against A. Clearly they are different functions.

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

<Figure size 432x288 with 0 Axes>
Plot column B against A in log-log scale.
We observe a line. So B is a polynomial function of A

```python
plt.figure(); dfs.plot(x='A', y='B', loglog=True);
```

<Figure size 432x288 with 0 Axes>
Plot both columns B and C against A in log scale

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

<Figure size 432x288 with 0 Axes>
Plot B and C against A, with log scale only on y-axis.
The plot of C becomes a line, indicating that C is an exponential function of A

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

Plotting using matlab notation
Also how to put two figures in a 1x2 grid

```
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*-')
plt.subplot(122)  # plot with 1 row, 2 columns, 2nd plot
plt.loglog(dfs['A'],dfs['B'],'bo-',dfs['A'],dfs['C'],'g*-')
```

```
[<matplotlib.lines.Line2D at 0x2221476d780>,
 <matplotlib.lines.Line2D at 0x2221478af60>]
```
Using seaborn

```python
sns.lineplot(x='A', y='B', data=dfs, marker='o')
```

The same plots as scatter plots using the dataframe functions

```python
fig, ax = plt.subplots(1, 2, figsize=(15,5))
df.plot(kind='scatter', x='A', y='B', ax=ax[0])
df.plot(kind='scatter', x='A', y='B', loglog=True, ax=ax[1])
```

44
plt.scatter(df.A, df.B)

Putting many scatter plots into the same plot

t = df.plot(kind='scatter', x='A', y='B', color='DarkBlue', label='B curve', loglog=True);
df.plot(kind='scatter', x='A', y='C', color='DarkGreen', label='C curve', ax=t, loglog=True);
Using seaborn

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

[156]: `<matplotlib.axes._subplots.AxesSubplot at 0x22213595cc0>`
In log-log scale (for some reason it seems to throw away small values)

```python
splot = sns.scatterplot(x='A', y='B', data=df)
splot.set(xscale="log", yscale="log")
```

[None, None]