EMERITUS
INSTITUTE OF MANAGEMENT

WEEK 12

DATA ANALYSIS AND VISUALIZATION — USING PYTHON'S NUMPY FOR ANALYSIS

COLUMBIA | ENGINEERING
EXECUTIVE EDUCATION

Python Libraries Supporting Data Analysis

- ➡Numpy: supports numerical and array operations
- ➡Scipy:
- ➡Pandas: supports data manipulation and analysis
- ➡Visualization libraries: matplotlib, seaborne, bokeh, plotly, gmplot, and many others provide support for charts and graphs

Why Numpy?

- Multi-dimensional arrays:
  - Faster and more space efficient than lists
- Can incorporate C/C++/Fortran code
- Linear algebra, Fourier transforms, Random number support

Numpy Arrays

- The basic numpy data structure is an array
- An array is a sequential collection of “like” objects
- unlike python lists, numpy arrays contain objects of the same type
  - this makes indexed access faster
  - and more memory efficient
- numpy arrays are mutable
- numpy are optimized for matrix operations
  - much faster than lists
- numpy provides random number support

Numpy Arrays

numpy

Creating a numpy array

In [3]:
```
import numpy as np
ax = np.array([1,2,3,4,5])
print(x,id(x),len(x))
```

```
[1 2 3 4 5] 4371008848 5
```

Specifying the type

Useful when reading a text stream directly into a numerical array

In [5]:
```
x=['1','2','3']
xi = np.array(x,'int')
xf = np.array(x,'float')
xs = np.array(x,'str')
print(xi,xf,xs,sep='\n')
```

```
[1 2 3]
[ 1.  2.  3.]
['1' '2' '3']
```

Numpy Arrays

Specifying the type

Useful when reading a text stream directly into a numerical array

```
In [3]: x=['1','2','3']
        xi = np.array(x,'int')
        xf = np.array(x,'float')
        xs = np.array(x,'str')
        print(xi,xf,xs,sep='\n')
```

```
[1 2 3]
[ 1.  2.  3.]
['1' '2' '3']
```

Basic operations

```
In [6]: x = np.array([13,24,21.2,17.6,21.7],'float')
        print(x.sum(),x.mean(),x.std(),sep='\n')
```

```
97.5
19.5
3.84291555983
```

Pandas Arrays

Pandas

In [ ]:
```
#installing pandas libraries
!pip install pandas-datareader
!pip install --upgrade html5lib==1.0b8

#There is a bug in the latest version of html5lib so install an earlier version
#Restart kernel after installing html5lib
```

Imports

In [ ]:
```
import pandas as pd #pandas library
from pandas_datareader import data #data readers (google, html, etc.)
#The following line ensures that graphs are rendered in the notebook
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt #Plotting library
import datetime as dt #datetime for timeseries support
```

Multidimensional Arrays

Multi-dimensional arrays

```
In [7]: x=[[0,1,2,3,4,5],[10,11,12,13,14,15],[20,21,22,23,24,25]]
        ax=np.array(x,float)
        print(ax)

        [[  0.   1.   2.   3.   4.   5.]
         [ 10.  11.  12.  13.  14.  15.]
         [ 20.  21.  22.  23.  24.  25.]]
```

Indexing

```
In [8]: ax[1,3] #indexing

Out[8]: 13.0
```

Slicing

```
In [11]: ax[1:,5:]
         #ax[:,2:]

Out[11]: array([[ 15.],
                [ 25.]])
```

Reshaping

```
In [12]: print(ax.shape)
         #ax.reshape(9,2)
         #ax.reshape(10,3)

         (3, 6)
```

Initialized Matrices

Creating Initialized Matrix

```
In [18]: ax = np.arange(10)
         print(ax)
         ay = np.array([np.arange(2,10,2),np.arange(10)])
         print(ay)

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

In [19]: ax = np.ones(10)
         print(ax)

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

In [20]: ax = np.arange(10)**2
         print(ax)

         [ 0  1  4  9 16 25 36 49 64 81]

In [21]: np.identity(10)

Out[21]: array([[ 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.]])
```

Matrix Multiplication

Matrix multiplication

In [22]:
```
ax = np.arange(10)
ay = np.array([ax,ax])
#Scalar multiplication
ay*2
```

Out[22]:
```
array([[ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18],
       [ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18]])
```

In [23]:
```
ay.shape
```

Out[23]:
```
(2, 10)
```

In [24]:
```
np.dot(ay,ay.reshape(10,2)) #Dot product
```

Out[24]:
```
array([[220, 265],
       [220, 265]])
```

Comparing Numpy Arrays with Lists

Comparing numpy arrays with lists

In [26]:
```
n=10
ax = np.array([np.arange(n)**2,np.arange(n)**3])
print(ax)
ay = ax.transpose()
#print(ax)
#print(ay)
np.dot(ax,ay)
```

```
[[  0   1   4   9  16  25  36  49  64  81]
 [  0   1   8  27  64 125 216 343 512 729]]
```

Out[26]:
```
array([[ 15333, 120825],
       [120825, 978405]])
```

Functionalize this

In [27]:
```
def dotproduct(n):
    ax = np.array([np.arange(n)**2,np.arange(n)**3])
    ay = ax.transpose()
    import datetime
    start = datetime.datetime.now()
    np.dot(ax,ay)
    end = datetime.datetime.now()
    return end-start

dotproduct(10)
```

Out[27]:
```
datetime.timedelta(0, 0, 18)
```

Do the same with python lists

In [29]:
```
def dot_product_lists(n):
    x = [x**2 for x in range(n)]
    y = [x**3 for x in range(n)]
    ax = [x,y]
    ay = [list(i) for i in zip(*ax)]
    import datetime
    start = datetime.datetime.now()
    [[sum(a*b for a,b in zip(X_row,Y_col)) for Y_col in zip(*ay)] for X_row in ax]
    end = datetime.datetime.now()
    return end-start

dot_product_lists(10)
```

Out[29]:
```
datetime.timedelta(0, 0, 22)
```

In [30]:
```
for n in [10,100,1000,10000]:
    numpy_result = dotproduct(n)
    list_result = dot_product_lists(n)
    print(n,numpy_result,list_result,sep='\t')
```

```
10      0:00:00.000014  0:00:00.000016
100     0:00:00.000009  0:00:00.000068
1000    0:00:00.000013  0:00:00.000632
10000   0:00:00.000129  0:00:00.012150
```

In [31]:
```
for n in [10,100,1000,10000]:
    numpy_result = dotproduct(n)
    list_result = dot_product_lists(n)
    print(n,numpy_result,list_result,sep='\t')
```

```
10      0:00:00.000016  0:00:00.000016
100     0:00:00.000008  0:00:00.000100
1000    0:00:00.000081  0:00:00.000670
10000   0:00:00.000056  0:00:00.036216
```

Comparing Numpy Arrays with Lists

Selecting elements from an np array

```
In [34]: x=[[0,1,2,3,4,5],[10,11,12,13,14,15],[20,21,22,23,24,25]]
         ax=np.array(x,float)
         np.where(ax%2==0,1,0)

Out[34]: array([[1, 0, 1, 0, 1, 0],
                [1, 0, 1, 0, 1, 0],
                [1, 0, 1, 0, 1, 0]])

In [35]: #linalg, a linear algebra module
         #functions dealing with polynomials, differentials, etc

In [36]: import scipy
         scipy.nanmean(x)

Out[36]: 12.5
```

Random number support in numpy

```
In [42]: np.random.normal(size=10)
         np.random.normal(size=(100,100))
         #np.random.exponential()
         #np.random.exponential(1.0,size=(6,3))
         #np.random.randint(-10,10,size=(9,9))

Out[42]: array([[ 1.05206221,  0.47781854,  2.89776774, ...,  0.3374305 ,
                 -0.49803948,  0.72237763],
                [ 0.60929371, -0.41048125,  0.20983578, ..., -1.640422  ,
                 -0.20119451,  1.09254855],
                [ 1.34244391,  1.09538764, -0.65353113, ...,  0.5886497 ,
                 -0.06382598,  1.50691536],
                ...,
                [-0.61623743,  0.55130456, -1.00555482, ...,  0.5921164 ,
                  0.17721396, -0.44003929],
                [ 0.92331676, -0.60830634,  0.75547553, ..., -0.65687201,
                  0.38049825,  0.69859464],
                [ 1.21479891,  0.4068382 ,  0.6057141 , ...,  0.1671013 ,
                 -0.6642757 ,  1.30283501]])
```

Random number support in numpy

```
In [41]: np.random.normal(size=10)
         #np.random.normal(size=(100,100))
         #np.random.exponential()
         #np.random.exponential(1.0,size=(6,3))
         #np.random.randint(-10,10,size=(9,9))

Out[41]: array([-0.76696598, -0.83422317,  0.16582169, -1.25107114,  0.16834798,
                 0.68083225,  0.67202279, -0.75005858, -0.37529753,  1.76749163])
```

Pandas

- Integrated data manipulation and analysis capabilities
- Integration with data visualization libraries
- Built in time-series capabilities
- Optimized for speed
- Built-in support for grabbing data from multiple sources
  - csv, xls, html tables, yahoo, google, worldbank, FRED

- Pandas organizes data into two data objects
  - Series: A one dimensional array object
  - DataFrame: A two dimensional table object

- Each column in a dataframe corresponds to a named series

- Rows in a dataframe can be indexed by a column of any datatype

Pandas: Importing

Pandas

```
In [ ]: #installing pandas libraries
        !pip install pandas-datareader
        !pip install --upgrade html5lib==1.0b8

        #There is a bug in the latest version of html5lib so install an earlier version
        #Restart kernel after installing html5lib
```

Imports

```
In [ ]: import pandas as pd #pandas library
        from pandas_datareader import data #data readers (google, html, etc.)
        #The following line ensures that graphs are rendered in the notebook
        %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt #Plotting library
        import datetime as dt #datetime for timeseries support
```

Structure of a Dataframe

The structure of a dataframe

```
In [2]: pd.DataFrame([[1,2,3],[1,2,3]],columns=['A','B','C'])
```

Out[2]:

	A	B	C
0	1	2	3
1	1	2	3

Accessing columns and rows

```
In [5]: df = pd.DataFrame([['r1','00','01','02'],['r2','10','11','12'],['r3','20','21','22']],columns=['row_label','A','B','
        df
```

Out[5]:

	row_label	A	B	C
0	r1	00	01	02
1	r2	10	11	12
2	r3	20	21	22

Structure of a Dataframe

```
In [7]: df = pd.DataFrame([['r1','00','01','02'],['r2','10','11','12'],['r3','20','21','22']],columns=['row_label','A','B','
        print(id(df))
        print(df)
        x = df.set_index('row_label',inplace=True)
        print(id(df))
        print(x)
        df
```

```
4636485392
  row_label   A   B   C
0        r1  00  01  02
1        r2  10  11  12
2        r3  20  21  22
4636485392
             A   B   C
row_label
r1          00  01  02
r2          10  11  12
r3          20  21  22
```

Out[7]:

	row_label	A	B	C
0	r1	00	01	02
1	r2	10	11	12
2	r3	20	21	22

Getting column data

```
In [9]: df['B']
```

```
Out[9]: row_label
        r1    01
        r2    11
        r3    21
        Name: B, dtype: object
```

```
In [10]: type(df['B'])
```

```
Out[10]: pandas.core.series.Series
```

```
In [12]: df['B']
```

```
Out[12]: row_label
         r1    01
         r2    11
         r3    21
         Name: B, dtype: object
```

Getting row data

```
In [13]: df.loc['r1']
```

```
Out[13]: A    00
         B    01
         C    02
         Name: r1, dtype: object
```

Structure of a Dataframe

Getting a row by row number

```
In [16]: df.iloc[0]
Out[16]: A    00
         B    01
         C    02
         Name: r1, dtype: object
```

Getting multiple columns

```
In [20]: df[['B','A']] #Note that the column identifiers are in a list
```

Out[20]:

	B	A
row_label		
r1	01	00
r2	11	10
r3	21	20

Getting a specific cell

```
In [21]: df.loc['r2','B']
Out[21]: '11'

In [23]: df.loc['r2']['B']
Out[23]: '11'
```

Slicing

```
In [24]: df.loc['r1':'r2']
```

Out[24]:

	A	B	C
row_label			
r1	00	01	02
r2	10	11	12

```
In [25]: df.loc['r1':'r2','B':'C']
```

Out[25]:

	B	C
row_label		
r1	01	02
r2	11	12

Pandas Datareader

Pandas datareader

- Access data from html tables on any web page
- Get data from google finance
- Get data from the federal reserve

```
<table>
<tr><td>c1</td></tr>
</table>
```

HTML Tables

- Pandas datareader can read a table in an html page into a dataframe
- the read_html function returns a list of all dataframes with one dataframe for each html table on the page

Example: Read the tables on the google finance page

In [27]:
```
df_list = pd.read_html('http://www.bloomberg.com/markets/currencies/major')
print(len(df_list))
```

```
1
```

The page contains only one table so the read_html function returns a list of one element

In [28]:
```
df = df_list[0]
print(df)
```

```
   Currency      Value   Change Net Change Time (EDT)  2 Day
0   EUR-USD     1.1193   0.0048      +0.43%    1:33 PM    NaN
1   USD-JPY   110.7900  -0.1400      -0.13%    1:33 PM    NaN
2   GBP-USD     1.2793   0.0035      +0.27%    1:32 PM    NaN
3   AUD-USD     0.7623   0.0044      +0.58%    1:32 PM    NaN
4   USD-CAD     1.3223  -0.0046      -0.35%    1:32 PM    NaN
5   USD-CHF     0.9737  -0.0016      -0.16%    1:33 PM    NaN
6   EUR-JPY   124.0200   0.3800      +0.31%    1:33 PM    NaN
7   EUR-GBP     0.8749   0.0012      +0.14%    1:33 PM    NaN
8   USD-HKD     7.8003  -0.0013      -0.02%    1:33 PM    NaN
9   EUR-CHF     1.0899   0.0029      +0.27%    1:32 PM    NaN
10  USD-KRW  1134.1800  10.0200      +0.89%    2:29 AM    NaN
```

Pandas Datareader

The page contains only one table so the read_html function returns a list of one element

```
In [30]: df = df_list[0]
         df
```

Out[30]:

	Currency	Value	Change	Net Change	Time (EDT)	2 Day
0	EUR-USD	1.1193	0.0048	+0.43%	1:33 PM	NaN
1	USD-JPY	110.7900	-0.1400	-0.13%	1:33 PM	NaN
2	GBP-USD	1.2793	0.0035	+0.27%	1:32 PM	NaN
3	AUD-USD	0.7623	0.0044	+0.58%	1:32 PM	NaN
4	USD-CAD	1.3223	-0.0046	-0.35%	1:32 PM	NaN
5	USD-CHF	0.9737	-0.0016	-0.16%	1:33 PM	NaN
6	EUR-JPY	124.0200	0.3800	+0.31%	1:33 PM	NaN
7	EUR-GBP	0.8749	0.0012	+0.14%	1:33 PM	NaN
8	USD-HKD	7.8003	-0.0013	-0.02%	1:33 PM	NaN
9	EUR-CHF	1.0899	0.0029	+0.27%	1:32 PM	NaN
10	USD-KRW	1134.1800	10.0200	+0.89%	2:29 AM	NaN

Note that the read_html function has automatically detected the header columns

If an index is necessary, we need to explicitly specify it

```
In [31]: df.set_index('Currency',inplace=True)
         print(df)
```

```
                Value   Change Net Change Time (EDT)  2 Day
Currency
EUR-USD        1.1193   0.0048      +0.43%    1:33 PM    NaN
USD-JPY      110.7900  -0.1400      -0.13%    1:33 PM    NaN
GBP-USD        1.2793   0.0035      +0.27%    1:32 PM    NaN
AUD-USD        0.7623   0.0044      +0.58%    1:32 PM    NaN
USD-CAD        1.3223  -0.0046      -0.35%    1:32 PM    NaN
USD-CHF        0.9737  -0.0016      -0.16%    1:33 PM    NaN
EUR-JPY      124.0200   0.3800      +0.31%    1:33 PM    NaN
EUR-GBP        0.8749   0.0012      +0.14%    1:33 PM    NaN
USD-HKD        7.8003  -0.0013      -0.02%    1:33 PM    NaN
EUR-CHF        1.0899   0.0029      +0.27%    1:32 PM    NaN
USD-KRW     1134.1800  10.0200      +0.89%    2:29 AM    NaN
```

Now we can use .loc to extract specific currency rates

```
In [33]: df.loc['EUR-CHF','Change']
Out[33]: 0.0028999999999999998
```

Pandas: Working with Views and Copies

Chained indexing creates a copy and changes to the copy won't be reflected in the original dataframe

In [34]:
```
eur_usd = df.loc['EUR-USD']['Change'] #This is chained indexing
df.loc['EUR-USD']['Change'] = 1.0 #Here we are changing a value in a copy of the dataframe
print(eur_usd)
print(df.loc['EUR-USD']['Change']) #Neither eur_usd, nor the dataframe are changed
```

```
0.0048
0.0048
```

```
/Users/cvn-mm-pbs-001/anaconda/lib/python3.6/site-packages/ipykernel/__main__.py:2: SettingWit
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.htm
sus-copy
  from ipykernel import kernelapp as app
```

In [35]:
```
eur_usd = df.loc['EUR-USD','Change'] #eur_usd points to the value inside the dataframe
df.loc['EUR-USD','Change'] = 1.0 #Change the value in the view
print(eur_usd) #eur_usd is changed (because it points to the view)
print(df.loc['EUR-USD']['Change']) #The dataframe has been correctly updated
```

```
0.0048
1.0
```

Historical Stock Prices from Google Finance

```
In [36]: from pandas_datareader import data
         import datetime
         start=datetime.datetime(2017, 1, 1)
         end=datetime.datetime.today()


         print(start,end)


         df = data.DataReader('IBM', 'google', start, end)
```

```
2017-01-01 00:00:00 2017-06-16 13:42:46.441450
```

```
In [37]: df
```

2017-06-02	153.07	153.20	151.80	152.05	3585701
2017-06-05	151.82	152.93	151.68	152.41	3975268
2017-06-06	152.00	152.89	152.00	152.37	3797173
2017-06-07	153.05	154.20	150.80	150.98	4865712
2017-06-08	151.00	152.82	150.92	152.10	3708962
2017-06-09	152.00	154.26	151.88	154.10	4361460
2017-06-12	154.19	157.20	154.02	155.18	6471479
2017-06-13	155.44	155.48	154.15	154.25	3523529
2017-06-14	153.97	154.94	152.94	153.81	3049726
2017-06-15	153.29	154.69	153.29	154.22	4654297

114 rows × 5 columns

Datareader Documentation

Datareader documentation
http://pandas-datareader.readthedocs.io/en/latest/

Working with a timeseries data frame

- The data is organized by time with the index serving as the timeline

Creating new columns

- Add a column to a dataframe
- Base the elements of the column on some combination of data in the existing columns

Example: Number of Days that the stock closed higher than it opened
- **We'll create a new column with the header "UP"**
- **And use np.where to decide what to put in the column**

```
In [38]: df['UP']=np.where(df['Close']>df['Open'],1,0)
         df
```

2017-01-03	167.00	167.87	166.01	167.19	2934299	1
2017-01-04	167.77	169.87	167.36	169.26	3381432	1
2017-01-05	169.25	169.39	167.26	168.70	2682301	0
2017-01-06	168.69	169.92	167.52	169.53	2945536	1
2017-01-09	169.47	169.80	167.62	167.65	3189891	0
2017-01-10	167.98	168.09	165.34	165.52	4118694	0
2017-01-11	166.05	167.76	165.60	167.75	3599464	1
2017-01-12	167.77	168.01	165.56	167.95	2927973	1
2017-01-13	167.97	168.48	166.88	167.34	2875433	0
2017-01-17	166.69	168.18	166.12	167.89	3315655	1
2017-01-18	167.45	168.59	166.69	166.80	4007779	0
2017-01-19	166.96	167.45	165.80	166.81	6963386	0

Get Summary Statistics

```
In [39]: df.describe()
```

Out[39]:

	Open	High	Low	Close	Volume	UP
count	114.000000	114.000000	114.000000	114.000000	1.140000e+02	114.000000
mean	167.589474	168.500351	166.695702	167.618246	4.178610e+06	0.482456
std	10.567480	10.590303	10.481766	10.572472	2.160995e+06	0.501898
min	150.300000	151.150000	149.790000	150.370000	1.825048e+06	0.000000
25%	156.265000	157.685000	155.202500	156.042500	3.095110e+06	0.000000
50%	170.365000	171.275000	169.725000	170.620000	3.571548e+06	0.000000
75%	176.177500	177.007500	175.540000	175.890000	4.353095e+06	1.000000
max	182.000000	182.790000	180.920000	181.950000	1.928428e+07	1.000000

Calculate the percentage of days that the stock has closed higher than its open

```
In [40]: df['UP'].sum()/df['UP'].count()
Out[40]: 0.48245614035087719
```

Calculate percent changes

- The function pct_change computes a percent change between successive rows (times in timeseries data)
- Defaults to a single time delta
- With an argument, the time delta can be changed

```
In [42]: df['Close'].pct_change() #One timeperiod percent change
Out[42]: Date
2017-01-03       NaN
2017-01-04  0.012381
2017-01-05 -0.003309
2017-01-06  0.004920
2017-01-09 -0.011089
2017-01-10 -0.012705
2017-01-11  0.013473
2017-01-12  0.001192
2017-01-13 -0.003632
2017-01-17  0.003287
2017-01-18 -0.006492
2017-01-19  0.000060
2017-01-20  0.022421
2017-01-23  0.002814
```

```
In [43]: n=13
         df['Close'].pct_change(n) #n timeperiods percent change
Out[43]: Date
2017-01-03       NaN
2017-01-04       NaN
2017-01-05       NaN
2017-01-06       NaN
2017-01-09       NaN
2017-01-10       NaN
2017-01-11       NaN
2017-01-12       NaN
2017-01-13       NaN
2017-01-17       NaN
2017-01-18       NaN
2017-01-19       NaN
2017-01-20       NaN
2017-01-23  0.022968
2017-01-24  0.039230
2017-01-25  0.056846
2017-01-26  0.053855
```

NaN Support

NaN support

Pandas functions can ignore NaNs

```
In [44]: n=13
         df['Close'].pct_change(n).mean()

Out[44]: -0.011193964480185867
```

Calculate something on the rolling windows

Example: mean (the 21 day moving average of the 13 day percent change)

```
In [46]: n=13
         df['Close'].pct_change(n).rolling(21).mean()

Out[46]: Date
         2017-01-03         NaN
         2017-01-04         NaN
         2017-01-05         NaN
         2017-01-06         NaN
         2017-01-09         NaN
         2017-01-10         NaN
         2017-01-11         NaN
         2017-01-12         NaN
         2017-01-13         NaN
         2017-01-17         NaN
         2017-01-18         NaN
         2017-01-19         NaN
         2017-01-20         NaN
         2017-01-23         NaN
         2017-01-24         NaN
         2017-01-25         NaN
         2017-01-26         NaN
         2017-01-27         NaN
```

Rolling windows

- "rolling" function extracts rolling windows
- For example, the 21 period rolling window of the 13 period percent change

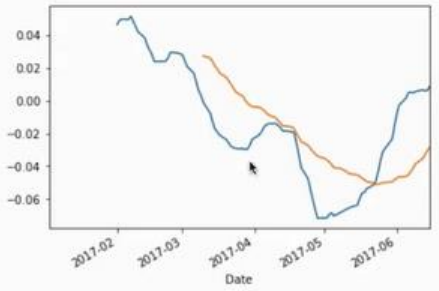
```
In [45]: df['Close'].pct_change(n).rolling(21)
Out[45]: Rolling [window=21,center=False,axis=0]
```

Calculate several moving averages and graph them

```
In [47]: ma_8 = df['Close'].pct_change(n).rolling(window=8).mean()
         ma_13= df['Close'].pct_change(n).rolling(window=13).mean()
         ma_21= df['Close'].pct_change(n).rolling(window=21).mean()
         ma_34= df['Close'].pct_change(n).rolling(window=34).mean()
         ma_55= df['Close'].pct_change(n).rolling(window=55).mean()

In [48]: ma_8.plot()
         ma_34.plot()

Out[48]: <matplotlib.axes._subplots.AxesSubplot at 0x115b1f940>
```



Linear Regression with Pandas

Example: TAN is the ticker for a solar ETF. FSLR, RGSE, and SCTY are tickers of companies that build or lease solar panels. Each has a different business model. We'll use pandas to study the risk reward tradeoff between the 4 investments and also see how correlated they are

```
In [49]: import datetime
         import pandas_datareader as data
         start = datetime.datetime(2015,7,1)
         end = datetime.datetime(2016,6,1)
         solar_df = data.DataReader(['FSLR', 'TAN','RGSE','SCTY'],'google', start=start,end=end)['Close']
```

```
In [50]: solar_df
```

Out[50]:

Date	FSLR	RGSE	SCTY	TAN
2015-07-01	46.04	1128.00	52.40	38.84
2015-07-02	45.17	1200.00	52.27	38.55
2015-07-06	44.19	1008.00	51.75	36.37
2015-07-07	45.12	984.00	53.21	36.10
2015-07-08	43.27	852.00	51.48	33.67
2015-07-09	43.65	876.00	51.98	35.23
2015-07-10	44.03	942.00	53.00	36.49
2015-07-13	46.01	978.00	53.39	37.37
2015-07-14	45.81	906.00	54.26	37.86

Let's calculate returns (the 1 day percent change)

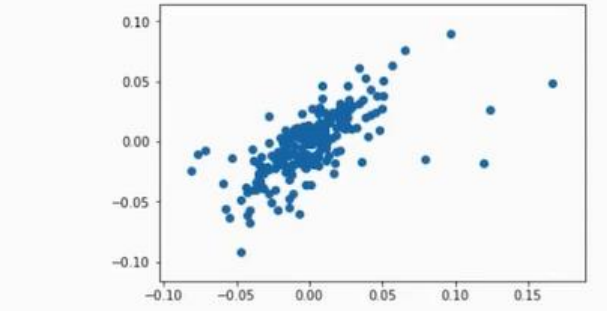
```
In [51]: rets = solar_df.pct_change()
         print(rets)
```

```
                FSLR      RGSE      SCTY       TAN
Date
2015-07-01       NaN       NaN       NaN       NaN
2015-07-02 -0.018897  0.063830 -0.002481 -0.007467
2015-07-06 -0.021696 -0.160000 -0.009948 -0.056550
2015-07-07  0.021045 -0.023810  0.028213 -0.007424
2015-07-08 -0.041002 -0.134146 -0.032513 -0.067313
2015-07-09  0.008782  0.028169  0.009713  0.046332
2015-07-10  0.008706  0
2015-07-13  0.044969  0
2015-07-14 -0.004347 -0
2015-07-15 -0.028815 -0
2015-07-16  0.006069  0
```

Let's visualize the relationship between each stock and the ETF

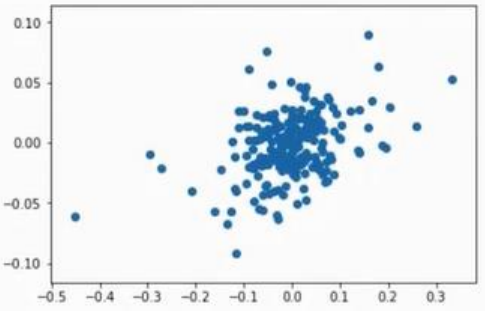
```
In [52]: import matplotlib.pyplot as plt
         plt.scatter(rets.FSLR,rets.TAN)
```

```
Out[52]: <matplotlib.collections.PathCollection at 0x115186710>
```

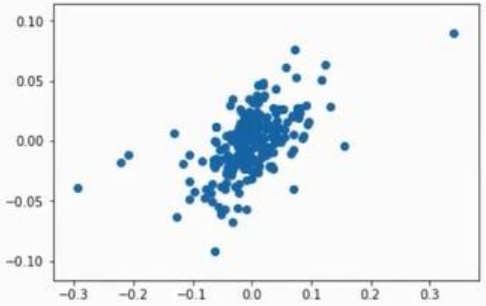


Linear Regression with Pandas

```
In [53]: plt.scatter(rets.RGSE,rets.TAN)
Out[53]: <matplotlib.collections.PathCollection at 0x115d335f8>
```




```
In [54]: plt.scatter(rets.SCTY,rets.TAN)
Out[54]: <matplotlib.collections.PathCollection at 0x119049048>
```



The correlation matrix

```
In [55]: solar_corr = rets.corr()
         print(solar_corr)
```

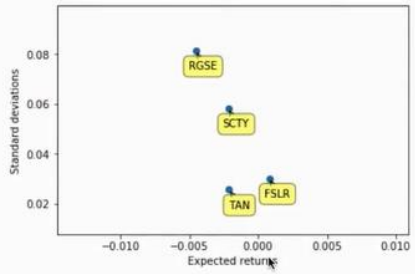
	FSLR	RGSE	SCTY	TAN
FSLR	1.000000	0.249923	0.272612	0.670114
RGSE	0.249923	1.000000	0.236604	0.389566
SCTY	0.272612	0.236604	1.000000	0.559854
TAN	0.670114	0.389566	0.559854	1.000000

Basic Risk Analysis

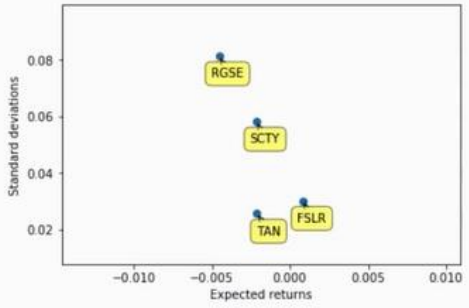
```
In [58]: rets.std()
```

```
Out[58]: FSLR    0.030188
         RGSE    0.081405
         SCTY    0.058234
         TAN     0.025696
         dtype: float64
```

```
In [56]: plt.scatter(rets.mean(), rets.std())
         plt.xlabel('Expected returns')
         plt.ylabel('Standard deviations')
         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'))
         plt.show()
```




```
In [62]: plt.scatter(rets.mean(), rets.std())
         plt.xlabel('Expected returns')
         plt.ylabel('Standard deviations')
         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'))
         plt.show()
```



Steps for Regression

- Construct y (dependent variable series)
- Construct matrix (dataframe) of X (independent variable series)
- Add intercept
- Model the regression
- Get the result

The statsmodel library contains various regression packages. We'll use the OLS (ordinary Least Squares) model

```
In [63]: import numpy as np
         import statsmodels.api as sm
         X=solar_df[['FSLR','RGSE','SCTY']]
         X = sm.add_constant(X)
         y=solar_df['TAN']
         model = sm.OLS(y,X,missing='drop')
         result = model.fit()
         print(result.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    TAN   R-squared:                       0.851
Model:                            OLS   Adj. R-squared:                  0.849
Method:                 Least Squares   F-statistic:                     435.1
Date:                Fri, 16 Jun 2017   Prob (F-statistic):           4.89e-94
Time:                        14:08:31   Log-Likelihood:                -464.33
No. Observations:                 232   AIC:                             936.7
Df Residuals:                     228   BIC:                             950.4
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
const         15.2915      1.180     12.956      0.000        12.966     17.617
FSLR           0.0087      0.017      0.521      0.603        -0.024      0.041
RGSE           0.0073      0.001      9.684      0.000         0.006      0.009
SCTY           0.2156      0.015     14.177      0.000         0.186      0.246
==============================================================================
Omnibus:                        4.153   Durbin-Watson:                   0.109
Prob(Omnibus):                  0.125   Jarque-Bera (JB):                3.754
Skew:                           0.239   Prob(JB):                        0.153
Kurtosis:                       2.600   Cond. No.                     5.88e+03
==============================================================================
```

Plotting Lines with Actual y Values

```
In [64]: fig, ax = plt.subplots(figsize=(8,6))
         ax.plot(y)
         ax.plot(result.fittedvalues)

Out[64]: [<matplotlib.lines.Line2D at 0x11bdeac18>]
```

const	15.2915	1.180	12.956	0.000	12.966	17.617
FSLR	0.0087	0.017	0.521	0.603	-0.024	0.041
RGSE	0.0073	0.001	9.684	0.000	0.006	0.009
SCTY	0.2156	0.015	14.177	0.000	0.186	0.246

Omnibus:	4.153	Durbin-Watson:	0.109
Prob(Omnibus):	0.125	Jarque-Bera (JB):	3.754
Skew:	0.239	Prob(JB):	0.153
Kurtosis:	2.600	Cond. No.	5.88e+03

www.emeritus.org