LECTURE 5 - APRIL 24, 2024

RECAP ON DATA STRUCTURES (COLLECTIONS)

- **LISTS** (NATIVE IN PYTHON)
  1. INDEXED WITH NUMBERS 0, 1, ...,
  2. THE ORDER OF ELEMENTS MATTERS
  3. HOMOGENEOUS TYPES OF ELEMENTS; REPETITIONS
  4. MODIFIABLE

```
[]: a = [50, 60, 70, 80]
     0   1   2   3   <- INDICES
```

- **TUPLES** (NATIVE IN PYTHON)
  1. INDEXED WITH NUMBERS 0, 1, ...,
  2. THE ORDER OF ELEMENTS MATTERS
  3. NON-HOMOGENEOUS TYPES OF ELEMENTS; REPETITIONS
  4. NON-MODIFIABLE

```
[]: b = ('Mark', 16, 54.11)
     0      1    2   <- INDICES
```

- **ARRAYS** (import numpy as np) → EFFICIENT
  1. INDEXED WITH NUMBERS 0, 1, ...,
  2. THE ORDER OF ELEMENTS MATTERS
  3. HOMOGENEOUS TYPES OF ELEMENTS; REPETITIONS
  4. MODIFIABLE
  5. OPTIMIZED FOR VECTOR AND MATRIX CALCULUS

```
[]: import numpy as np
    c = np.array([50, 60, 70, 80])
         0   1   2   3   <- INDICES
```

PYTHON HAS OTHER TWO NATIVE DATA STRUCTURES

- SETS (NATIVE IN PYTHON)
  1. NON-INDEXED
  2. THE ORDER OF ELEMENTS IS NOT IMPORTANT
  3. NON-HOMOGENEOUS TYPES OF ELEMENTS; NO REPETIONS
  4. NON-MODIFIABLE

```
[]: d = {50, 60, 70, 60, 60, 50}
    d
```

```
[]: e = {70, 50, 60, 50}
    e
```

```
[]: d == e
```

COMPARISON: THEY ARE EQUAL!

- DICTIONARIES (NATIVE IN PYTHON)
  1. INDEXED BY CUSTOM (POSSIBLY NON-NUMERIC) INDICES
  2. ORDER GIVEN BY THE INDICES
  3. NON-HOMOGENEOUS TYPES OF ELEMENTS; REPETITIONS
  4. MODIFIABLE

```
[]: students = {'Mark': 30, 'David': 25, 'Mary': 28}
    students
```

```
[]: # Change Mark's grade
    students['Mark'] = 28
    students
```

IMPORTANT: DICTIONARIES ARE NOT OPTIMIZED FOR BIG DATA!

INTRODUCTION TO Pandas

NumPy's ARRAYS ARE OPTIMIZED FOR HOMOGENEOUS NUMERIC DATA ACCESSED VIA INTEGER INDICES.

DATA SCIENCE REQUIRES TO DEAL WITH: MIXED DATA TYPES, CUSTOMIZED INDEXING, MISSING DATA.

Pandas IS THE MOST POPULAR LIBRARY TO DEAL WITH SUCH DATA. IT PROVIDES TWO COLLECTIONS:

- **Series**: FOR ONE-DIMENSIONAL COLLECTIONS (LIKE TIME SERIES)
- **DataFrame**: FOR TWO-DIMENSIONAL COLLECTIONS (LIKE TABULAR DATASETS)

Series → EFFICIENT

A Series IS AN ENHANCED ONE-DIMENSIONAL ARRAY.

[]:
```
import pandas as pd

grades = pd.Series([87, 100, 94])
grades
```

LIKE ARRAYS WE HAVE FUNCTIONS count, mean, min, max, std

[]:
```
grades.mean()
```

[]:
```
grades.std()
```

WE CAN SEE ALL DESCRIPTIVE STATISTICS USING **describe**

[]:
```
grades.describe()
```

WE CAN CUSTOMIZE INDICES LIKE A DICTIONARY

```
[]: grades = pd.Series({'Wally': 87, 'Eva': 100, 'Sam': 94})
    grades
```

```
[]: grades['Wally'] = 90
```

DataFrames → EFFICIENT

A DataFrame IS AN ENHANCED TWO-DIMENSIONAL ARRAY.

```
[]: import pandas as pd

    grades_dict = {'Wally': [87, 90, 70],
                   'Eva' : [100, 87, 90],
                   'Sam' : [94, 77, 90]}
    grades = pd.DataFrame(grades_dict)
```

THIS IS A DICTIONARY

WE CREATE A DataFrame FROM A DICTIONARY

WE GET A TABLE

ROW INDEX	Wally	Eva	Sam
0	87	100	94
1	90	87	77
2	70	77	90

```
[]: # Select a column as a Series
    grades['Wally']
```

```
[]: # Select some columns
    grades[['Wally', 'Sam']]
```

WRITE grades[['Wally']] TO EXTRACT A SUB DataFrame: CHECK THE DIFFERENCE!

WE CAN GET ALL STATISTICS OF THE TABLE WITH describe

```
[]: grades.describe()
```

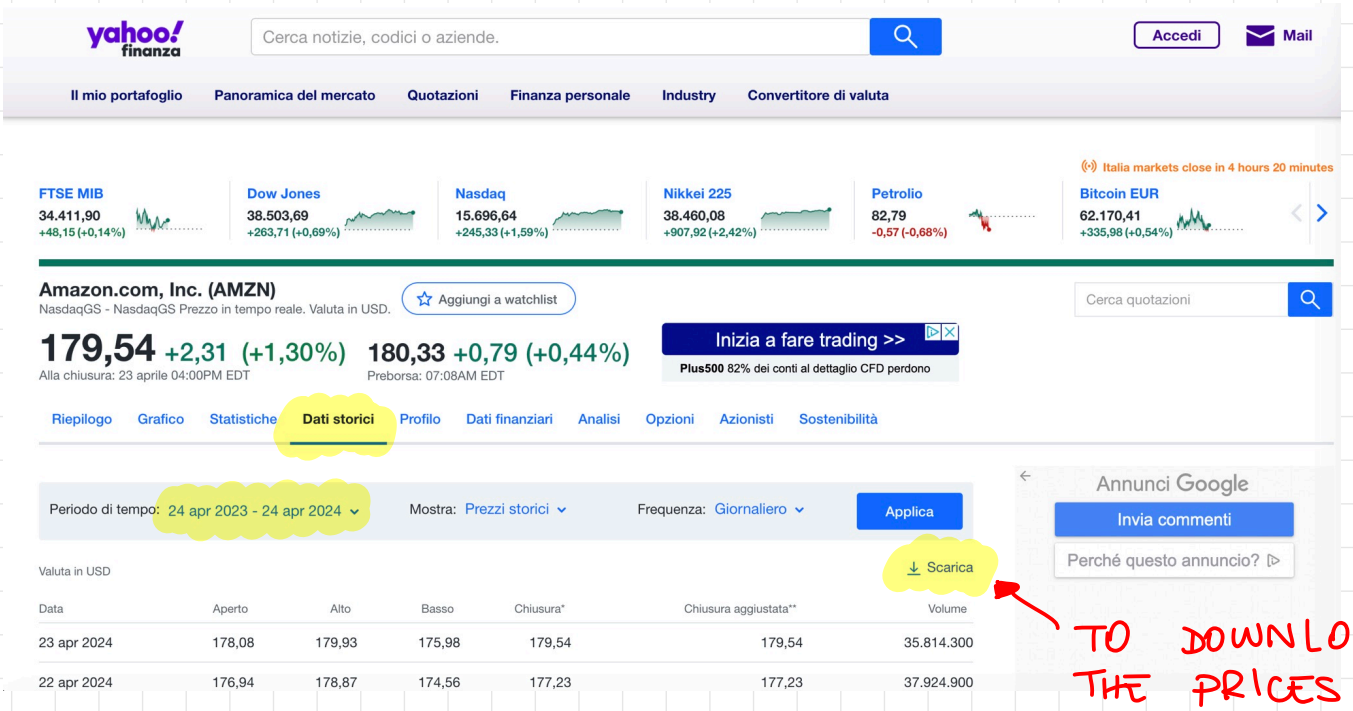
IMPORTANT: Series AND DataFrames ARE USED IN DATA SCIENCE TO DEAL WITH BIG DATA. WE LOAD DATA FROM A DATASET!

LOADING DATA FROM A CSV FILE

CSV (COMMA-SEPARATED VALUES) IS A FILE FORMAT OF MANY DATASETS WE CAN FIND ON THE INTERNET. GO TO yahoo! FINANCE AND DOWNLOAD THE CSV FILES WITH HISTORICAL PRICES FOR THE LAST YEAR OF THE ASSETS:

- AMZN ← EQUITY, NO DIVIDENDS, $
- TSLA ← EQUITY, NO DIVIDENDS, $
- BTC-USD ← CRYPTOCURRENCY EXC, $
- ETH-USD ← CRYPTOCURRENCY EXC, $

SAVE THE CSV FILES IN THE SAME DIRECTORY OF THE JupiterLab NOTEBOOK.



TO DOWNLOAD THE PRICES OF THE LAST YEAR

[]: import pandas as pd

WE WANT TO CREATE TWO JOINT DataFrames FOR EQUITY AND CRYPTO ASSETS.

[]:
```
# Create a DataFrame for AMZN
amzn = pd.read_csv('AMZN.csv')
amzn
```

[]:
```
amzn.describe()
```

[]:
```
# Create a DataFrame for TSLA
tsla = pd.read_csv('TSLA.csv')
tsla
```

[]:
```
tsla.describe()
```

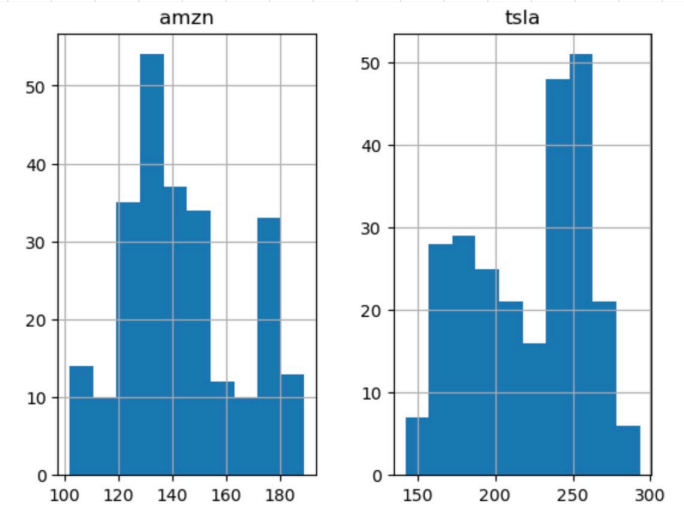
[]:
```
# Create a joint DataFrame of equity prices
amzn = amzn[['Date', 'Close']].set_index('Date')
tsla = tsla[['Date', 'Close']].set_index('Date')
joint_equity = pd.merge(amzn, tsla,
                        left_index=True,
                        right_index=True)

# Rename the columns
joint_equity.columns = ['amzn', 'tsla']
joint_equity
```

TO TAKE ONLY COMMON DATES

```
[]: # Save the joint DataFrame as CSV
     joint_equity.to_csv('joint_equity.csv')
```

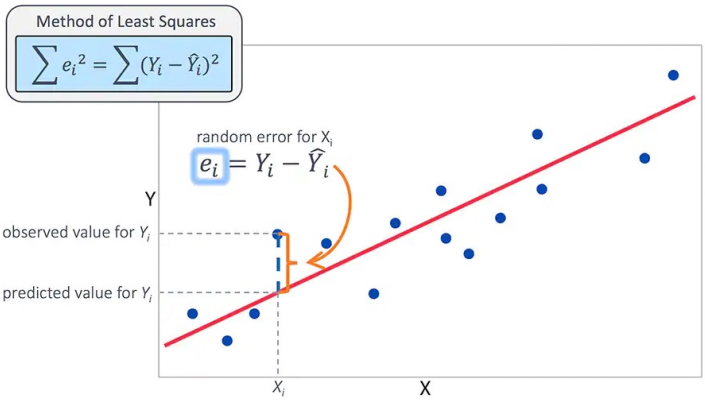
```
[]: # Show the histograms
     joint_equity.hist()
```

USE .plot() TO PLOT THE TIME SERIES AND .corr() TO COMPUTE THE CORRELATION



LET US COMPUTE THE LINEAR REGRESSION WITH

```
x = joint_equity['amzn']
y = joint_equity['tsla']
```



WE FIND $\hat{a}$ AND $\hat{b}$ MINIMIZING LEAST SQUARES.

REGRESSION LINE: $y = \hat{a}x + \hat{b}$

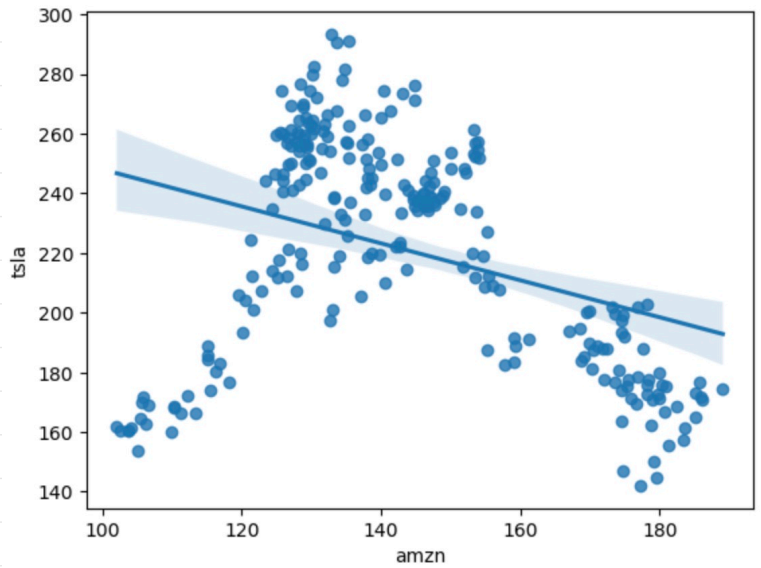
$\hat{a}$: SLOPE

$\hat{b}$: INTERCEPT

[]:
```
# Compute the linear regression
from scipy import stats

linear_regression = stats.linregress(
                        x = joint_equity['amzn'],
                        y = joint_equity['tsla'])

a = linear_regression.slope
b = linear_regression.intercept
```

[]:
```
# Linear prediction of a price
x = 300          <- AMZN PRICE
y = a*x + b
y                <- PREDICTED TSLA PRICE
```

[]:
```
# Plot the regression line using Seaborn library
import seaborn as sn

axes = sn.regplot(x = joint_equity['amzn'],
                  y = joint_equity['tsla'])
```



WE REPEAT THE ANALYSIS FOR THE CRYPTOCURRENCIES:

[]:
```
# Create a DataFrame for BTC-USD
btc = pd.read_csv('BTC-USD.csv')
btc
```

[]:
```
btc.describe()
```

[]:
```
# Create a DataFrame for ETH-USD
eth = pd.read_csv('ETH-USD.csv')
eth
```

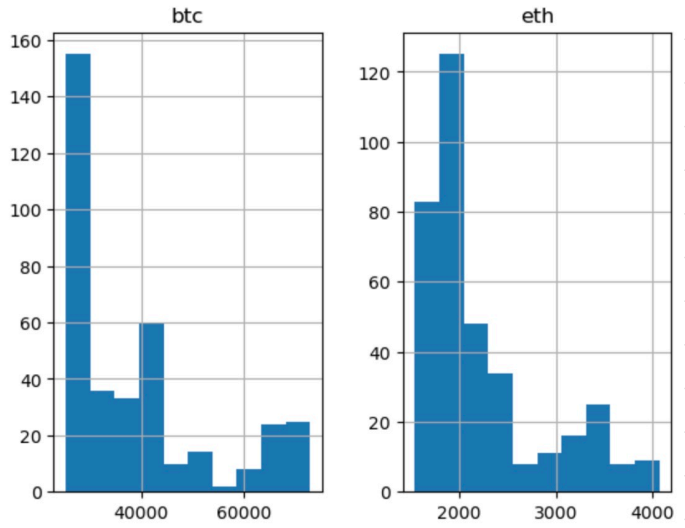
[]:
```
eth.describe()
```

[]:
```
# Create a joint DataFrame of crypto prices
btc = btc[['Date', 'Close']].set_index('Date')
eth = eth[['Date', 'Close']].set_index('Date')
joint_crypto = pd.merge(btc, eth,
                        left_index=True,
                        right_index=True)

# Rename the columns
joint_crypto.columns = ['btc', 'eth']
joint_crypto
```

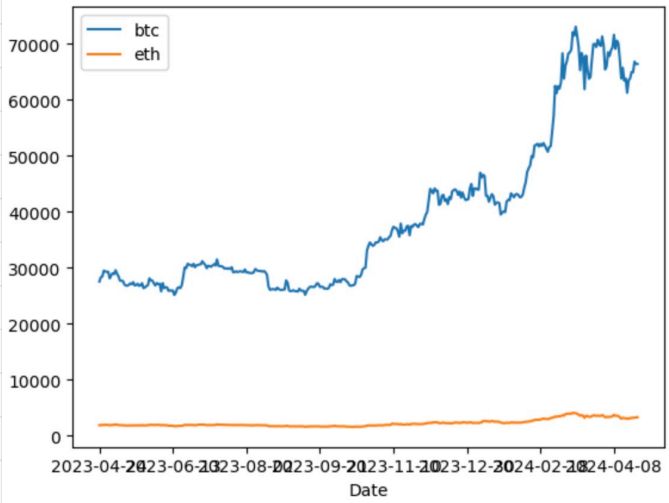
TO TAKE ONLY COMMON DATES

[]:
```
# Save the joint DataFrame as CSV
joint_crypto.to_csv('joint_crypto.csv')
```

[]:
```
# Show the histograms
joint_crypto.hist()
```



[]:
```
# Show the time series plots
joint_crypto.plot()
```



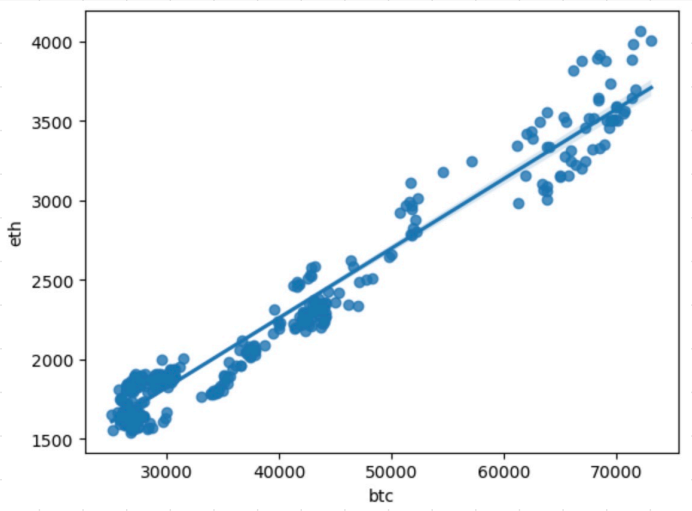
THE MAGNITUDE OF BTC PRICES IS MUCH GREATER

[]:
```
# Compute the correlation coefficient
joint_crypto.corr()
```
← $\rho = \frac{Cov(X,Y)}{\sigma_X, \sigma_Y}$

Seaborn ALLOWS TO DIRECTLY PLOT THE REGRESSION LINE THROUGH `regplot`: NO NEED TO COMPUTE THE REGRESSION LINE EXPLICITLY.

[]: # Plot the regression line using Seaborn library

```
axes = sn.regplot(x = joint_crypto['btc'],
                  y = joint_crypto['eth'])
```



LOADING DATA FROM yahoo! FINANCE

UP TO NOW WE SAW HOW TO DOWLOAD A TIME SERIES OF PRICES FROM yahoo! FINANCE AS A CSV FILE AND HOW TO LOAD SUCH DATASET IN A panda's DataFrame.

THE CLASS yfinance ALLOWS TO DIRECTLY DOWLOAD DATA FROM yahoo! FINANCE THROUGH ITS API (APPLICATION PROGRAMMING INTERFACE): AN ACTIVE INTERNET CONNECTION IS NEEDED.

WHILE pandas IS USUALLY PRESENT IN A PYTHON DISTRIBUTION LIKE ANACONDA, yfinance NEEDS TO BE INSTALLED. WE CAN DO IT IN THE ANACONDA'S COMMAND LINE USING THE pip TOOL:

pip install yfinance

pip ← PACKAGE MANAGER

WE WANT TO LOAD THE DAILY PRICES FOR THE LAST YEAR OF THE STOCK AMZN

[]:
```
# Import the libraries
import yfinance as yf
import pandas as pd
import numpy as np
```

[]:
```
# Create an object to access AMZN stock
stock = yf.Ticker('AMZN')
```

```
[]: # Load AMZN's historical data
   hist_data = stock.history(period='1y')
```

IT IS ALREADY INDEXED BY Date

1d, 5d, 1mo, 3mo, 6mo, 1y, 2y, 5y, 10y, ytd, max

```
[]: # Extract closing prices
   prices = hist_data['Close']
```

IN THIS WAY WE EXTRACT A Series. TO EXTRACT A DataFrame WE WRITE hist_data[['Close']]

PROBLEM: WE WANT TO ESTIMATE "VOLATILITY" FROM HISTORICAL DATA.

DENOTE

S_0 : VALUE OF THE STOCK TODAY

S_T : VALUE OF THE STOCK AT TIME T (IN YEARS)

A COMMON ASSUMPTION IN FINANCIAL MARKETS IS THAT THE LOG-RETURN OVER $[0,T]$ IS NORMAL

$$\ln \frac{S_T}{S_0} \sim N\left(\left(\mu - \frac{\sigma^2}{2}\right)T, \sigma^2 T\right)$$

REMARK: THIS PARAMETRIZATION DEPENDS ON THE MEANING OF μ IN THE BLACK-SCHOLES MODEL

σ IS SAID "VOLATILITY" AND MEASURES UNCERTAINTY ON FUTURE LOG-RETURNS.

TO ESTIMATE σ FROM HISTORICAL DATA, CONSIDER:

- $n+1$: NUMBER OF STOCK PRICE OBSERVATIONS
- S_i : STOCK PRICE AT THE END OF iTH DAY

① COMPUTE THE LOG-RETURNS

$$u_i = \ln\left(\frac{S_i}{S_{i-1}}\right) \qquad i = 1, \dots, n$$

② COMPUTE THE MEAN OF u_i's

$$\bar{u} = \frac{1}{n} \sum_{i=1}^{n} u_i$$

③ COMPUTE THE SAMPLE STANDARD DEVIATION

$$\hat{\sigma}_{day} = \sqrt{\frac{1}{n-1} \cdot \sum_{i=1}^{n} (u_i - \bar{u})^2}$$

$\hat{\sigma}_{day}$ IS AN UNBIASED ESTIMATOR ON A DAILY BASIS

④ COMPUTE THE HISTORICAL VOLATILY PER ANNUM

$$\hat{\sigma}_{year} = \sqrt{252} \cdot \hat{\sigma}_{day}$$

TRADING DAYS IN A YEAR: USUALLY 250 OR 252

[]:
```
# Compute the log-returns
log_rets = np.log(prices / prices.shift(1))
log_rets
```

[]:
```
# Delete the first NaN row
log_rets = log_rets.dropna()
log_rets
```

[]:
```
# Compute the historical volatility
import statistics as stat

vol_day = stat.stdev(log_rets)
vol_year = vol_day * np.sqrt(252)
print('Historical volatility of AMZN =', vol_year)
```

SAMPLE STANDARD DEVIATION