5QQMN534: Algorithmic Finance

Week5: Financial Data and Pre-processing Part2

Eryk Lewinson – Python for Finance Cookbook 2020
Chapter1

Agenda

- **Converting prices to returns**
  - Introduction
  - How to do simple and log returns
  - Inflation Adjusted Returns
- **Changing Frequency**
  - Introduction
  - Monthly Volatility from Daily Returns
- **Visualising Time Series Data**
  - Plot method of pandas
  - Plotly and cufflinks
- **Identifying Outliers**
- **Investigating Stylized Facts of Asset Returns**
  - Non-Gaussian Distribution of Returns
  - Volatility Clustering
  - Absence of Autocorrelation in Returns
  - Small and decreasing Autocorrelation in squared / absolute Returns
  - Leverage Effect

Converting prices to returns1

- Asset prices are usually non-stationary, that is, their statistics, such as mean and variance (mathematical moments) change over time. This could also mean observing some trends or seasonality in the price series. By transforming the prices into returns, we attempt to make the time series stationary, which is the desired property in statistical modelling.

There are two types of returns:

- **Simple returns:** They aggregate over assets; the simple return of a portfolio is the weighted sum of the returns of the individual assets in the portfolio. Simple returns are defined as:

$$R_t = (P_t - P_{t-1})/P_{t-1} = P_t/P_{t-1} - 1$$

- **Log returns:** They aggregate over time; it is easier to understand with the help of an example—the log return for a given month is the sum of the log returns of the days within that month. Log returns are defined as:

$$r_t = log(P_t/P_{t-1}) = log(P_t) - log(P_{t-1})$$

P_t is the price of an asset in time t. In the preceding case, we do not consider dividends, which obviously impact the returns and require a small modification of the formulas.

KEY TAKEAWAYS

- In statistics, a time series analysis involves measuring how things change over time with respect to certain variables of interest.
- When a time series is stationary, it means that certain attributes of the data do not change over time.
- However, some time series are non-stationary, whereby values and associations between and among variables do vary with time.
- In finance, many processes are non-stationary, and so must be handled appropriate.

Converting prices to returns2

- The best practice while working with stock prices is to use **adjusted** values, as they account for possible corporate actions, such as stock splits.
- The difference between simple and log returns for daily/intraday data will be very small, however, the general rule is that log returns are **smaller** in value than simple returns.
- In lecture 10 you'll learn how to also convert between simple and log returns for portfolio management.
- Here, we show how to calculate both types of returns using Apple's stock prices.

How to do it: Simple and Log Returns

- Execute the following steps to download the stock prices and calculate simple/log returns.

1. Import the libraries:

```
import pandas as pd
import numpy as np
import yfinance as yf
```

2. Download the data and keep the adjusted close prices only:

```
df = yf.download('AAPL',
                 start='2000-01-01',
                 end='2010-12-31',
                 progress=False)

df = df.loc[:, ['Adj Close']]
df.rename(columns={'Adj Close':'adj_close'}, inplace=True)
```

3. Calculate the simple and log returns using the adjusted close prices:

```
df['simple_rtn'] = df.adj_close.pct_change()
df['log_rtn'] = np.log(df.adj_close/df.adj_close.shift(1))
```

The resulting DataFrame looks as follows:

Date	adj_close	simple_rtn	log_rtn
1999-12-31	3.194901	NaN	NaN
2000-01-03	3.478462	0.088754	0.085034
2000-01-04	3.185191	-0.084311	-0.088078
2000-01-05	3.231803	0.014634	0.014528
2000-01-06	2.952128	-0.086538	-0.090514

The first row will always contain a **not a number** (**NaN**) value, as there is no previous price to use for calculating the returns.

How it Works: Explanation

- In *Step 2*, we downloaded price data from Yahoo Finance, and only kept the adjusted close price for the returns calculation.
- To calculate the simple returns, we used the pct_change method of pandas Series/DataFrame, which calculates the percentage change between the current and prior element (we can specify the number of lags, but for this specific case, the default value of 1 suffices).
- To calculate the log returns, we followed the formula given in the introduction to this code. When dividing each element of the series by its lagged value, we used the shift method with a value of 1 to access the prior element.
- In the end, we took the natural logarithm of the divided values by using np.log.

There's More … Inflation Adjusted Return

- We will also discuss how to account for inflation in the returns series.
- To do so, we continue with the example used in this code.
- We first download the monthly **Consumer Price Index** (CPI) values from Quandl and calculate the percentage change (simple return) in the index.
- We can then merge the inflation data with Apple's stock returns, and account for inflation by using the following formula:

$$R_t^r = \frac{1 + R_t}{1 + \pi_t} - 1$$

Which is essentially

$$\text{Inflation Adjusted Return} = \frac{1 + \text{Return}}{1 + \text{Inflation Rate}} - 1$$

Here, R_t is a time t simple return and π_t is the inflation rate.

There's More … Inflation Adjusted Return1

1. Import libraries and authenticate:

```
import pandas as pd
import quandl

QUANDL_KEY = '{key}'
quandl.ApiConfig.api_key = QUANDL_KEY
```

Note: You will need your Quandl API key from prior class

2. Create a DataFrame with all possible dates, and left join the prices to it:

```
df_all_dates = pd.DataFrame(index=pd.date_range(start='1999-12-31',
                                                end='2010-12-31'))
df = df_all_dates.join(df[['adj_close']], how='left') \
                 .fillna(method='ffill') \
                 .asfreq('M')
```

- We used a left join, which is a type of join (used for merging DataFrames) that returns all rows from the left table and the matched rows from the right table while leaving the unmatched rows empty.
- In case the last day of the month was not a trading day, we used the last known price of that month (fillna(method='ffill')). Lastly, we selected the end-of-month rows only by applying asfreq('M').

There's More … Inflation Adjusted Return2

3. Download the inflation data from Quandl:

```
df_cpi = quandl.get(dataset='RATEINF/CPI_USA',
                    start_date='1999-12-01',
                    end_date='2018-12-31')
df_cpi.rename(columns={'Value':'cpi'}, inplace=True)
```

4. Merge the inflation data to the prices:

```
df_merged = df.join(df_cpi, how='left')
```

5. Calculate the simple returns and inflation rate:

```
df_merged['simple_rtn'] = df_merged.adj_close.pct_change()
df_merged['inflation_rate'] = df_merged.cpi.pct_change()
```

6. Adjust the returns for inflation:

```
df_merged['real_rtn'] = (df_merged.simple_rtn + 1) /
(df_merged.inflation_rate + 1) - 1
```

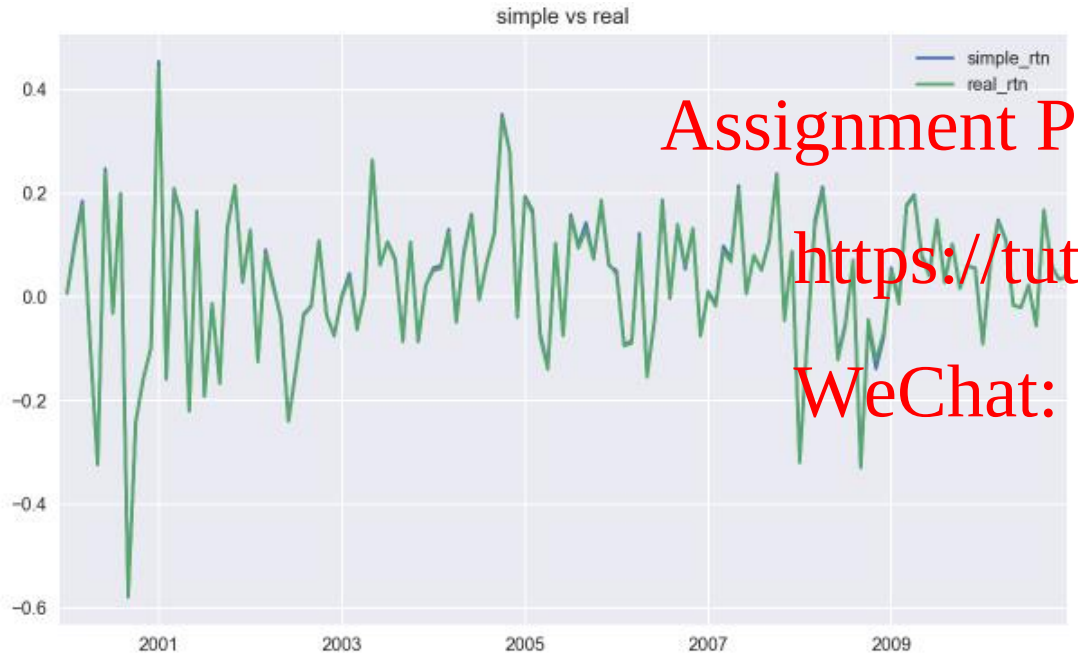
There's More … Inflation Adjusted Return2

The output looks as follows:

	adj_close	cpi	simple_rtn	inflation_rate	real_rtn
1999-12-31	3.194901	168.3	NaN	NaN	NaN
2000-01-31	3.224035	168.8	0.009119	0.002971	0.006130
2000-02-29	3.561976	169.8	0.104819	0.005924	0.098313
2000-03-31	4.220376	171.2	0.184841	0.008245	0.175152
2000-04-30	3.855247	171.3	-0.086516	0.000584	-0.087049

The DataFrame contains all the intermediate results, and the `real_rtn` column contains the inflation-adjusted returns.

* Extra Plotting Simple vs Real Returns



Changing Frequency: Introduction

The general rule of thumb for changing frequency can be broken down into the following:

- Multiply/divide the log returns by the number of time periods.
- Multiply/divide the volatility by the square root of the number of time periods.

In this code, we present an example of how to calculate the monthly realized volatilities for Apple using daily returns and then annualize the values.

The formula for realized volatility is as follows:

$$RV = \sqrt{\sum_{i=1}^{T} r_t^2}$$

Realized volatility is frequently used for daily volatility using the intraday returns.

The steps we need to take are as follows:

- Download the data and calculate the log returns.
- Calculate the realized volatility over the months.
- Annualize the values by multiplying by , as we are converting from monthly values.

This will be practically applied in lecture 10 portfolio management as well.

How to do it: Monthly Volatility from Daily Returns1

- We should have a DataFrame called df with a single log_rtn column and timestamps as the index from the prior code.

Execute the following steps to calculate and annualize the monthly realized volatility.

1. Import the libraries:

```
import pandas as pd
```


2. Define the function for calculating the realized volatility:


```
def realized_volatility(x):
    return np.sqrt(np.sum(x**2))
```

3. Calculate the monthly realized volatility:


```
df_rv = df.groupby(pd.Grouper(freq='M')).apply(realized_volatility)
df_rv.rename(columns={'log_rtn': 'rv'}, inplace=True)
```

4. Annualize the values:

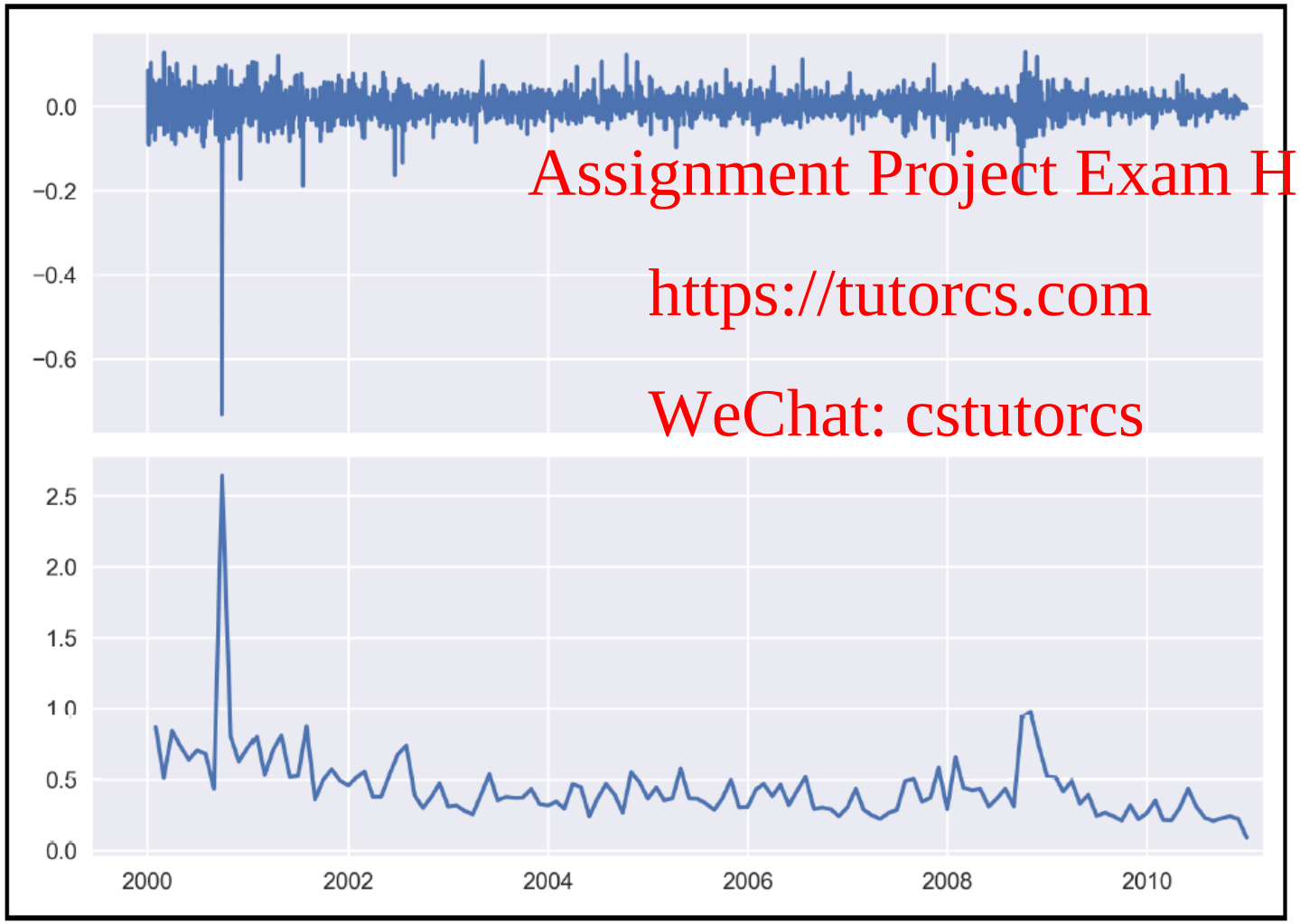
```
df_rv.rv = df_rv.rv * np.sqrt(12)
```

5. Plot the results:

```
fig, ax = plt.subplots(2, 1, sharex=True)
ax[0].plot(df)
ax[1].plot(df_rv)
```

How to do it: Monthly Volatility from Daily Returns2

- Executing the preceding code results in the following plots



Note x & y axis, titles and legends should be applied to graphs

We can see that the spikes in the realized volatility coincide with some extreme returns (which might be outliers).

How it Works: Monthly Volatility from Daily Returns

- Normally, we could use the resample method of a pandas DataFrame.
- Supposing we wanted to calculate the average monthly return, we could run df.log_rtn.resample('M').mean().
- For the resample method, we can use any built-in aggregate functions of pandas, such as mean, sum, min, and max.
- However, our case is a bit more complex, so we defined a helper function called realized_volatility, and replicated the behaviour of resample by using a combination of groupby, Grouper, and apply.
- We presented the most basic visualization of the results (please refer to the next code for information about visualizing time series).

Visualising Time Series Data

Getting ready

- For this code, we assume we already have a DataFrame called df with three columns (adj_close, simple_rtn, and log_rtn) and dates set as the index. Please refer to the notebook on the GitHub repository for details on downloading data for this code. This is provided for us in the code already.

How to do it...

- In this section, we introduce how to plot time series data.
- We start by using the default plot method of a pandas DataFrame/Series, and then present the interactive alternative offered by the combination of plotly and cufflinks.

The Plot Method of Pandas1

Execute the following code to plot Microsoft's stock prices together with the simple and log returns.

```
fig, ax = plt.subplots(3, 1, figsize=(14, 20), sharex=True)

df.adj_close.plot(ax=ax[0])
ax[0].set(title = 'MSFT time series',
          ylabel = 'Stock price ($)')
df.simple_rtn.plot(ax=ax[1])
ax[1].set(ylabel = 'Simple returns (%)')

df.log_rtn.plot(ax=ax[2])
ax[2].set(xlabel = 'Date',
          ylabel = 'Log returns (%)')
```

The Plot Method of Pandas2

- The resulting plot contains three axes. Each one of them presents a different series: raw prices, simple returns, and log returns.
- Inspecting the plot in such a setting enables us to see the periods of heightened volatility and what was happening at the same time with the price of Microsoft's stock.
- Additionally, we see how similar simple and log returns are.

Executing the preceding code results in the following plot:

Plotly and Cufflinks (Only in Jupyter Notebook)1

You will need to install these two libraries into Spyder in IPython. Then restart kernel

- pip install plotly
- pip install cufflinks
- pip install chart_studio

* Note if using latest Spyder version > 5.1 precede the install command with a !. E.g. !pip install plotly

- Cufflinks is a productivity tool for use with plotly and pandas
- Plotly is a dynamic web based interactive graphing library. Plotly allows us for in browser **dynamic** web based graphical visualising of data.

If want to run this section:

1. **Load Jupyter from Start Menu and**
2. **Open plotly_cufflinks_example.ipynb**

Libraries:

- https://pypi.org/project/cufflinks1/
- https://pypi.org/project/plotly/

Plotly and Cufflinks (Only in Jupyter Notebook)2

Execute the following code to plot Microsoft's stock prices together with the simple and log returns.

1. Import the libraries and handle the settings:

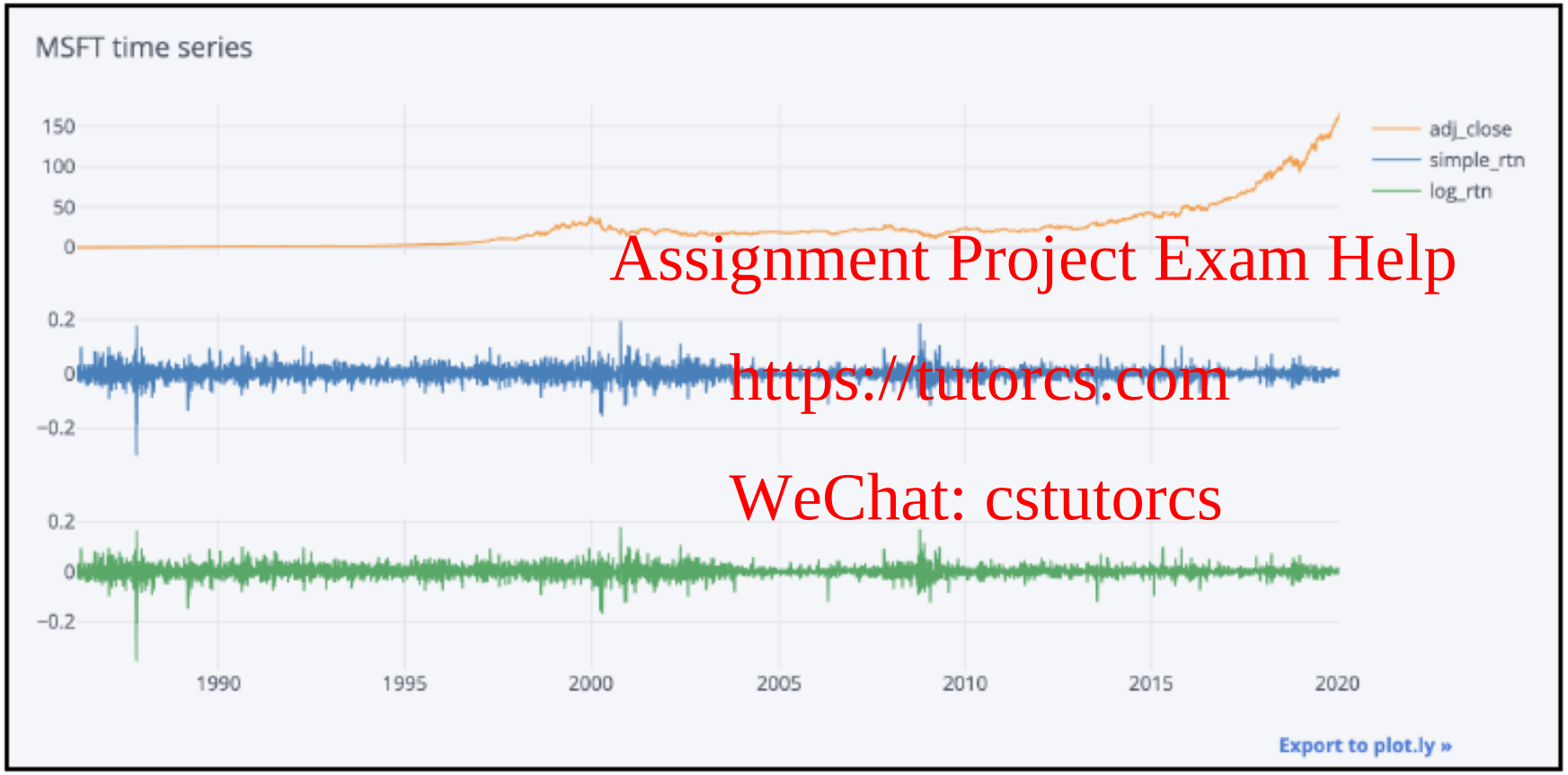
```
import cufflinks as cf
from plotly.offline import iplot, init_notebook_mode

# set up configuration (run it once)
#cf.set_config_file(world_readable=True, theme='pearl',
#                   offline=True)

init_notebook_mode()
```

2. Create the plot:

```
df.iplot(subplots=True, shape=(3,1), shared_xaxes=True,
         title='MSFT time series')
```

Plotly and Cufflinks (Only in Jupyter Notebook)3

We can observe the time series in the following plot:



- The main advantage of using plotly with cufflinks is the interactivity of the preceding chart, which is unfortunately only demonstrable in the Notebook.
- You can also export to Plotly and create a free account here.
- https://chart-studio.plotly.com/Auth/login/?action=signup#/

The Plot Method of Pandas Explanation

Our goal was to visualize all three series on the same plot (sharing the *x*-axis) to enable a quick visual comparison.

To achieve this, we had to complete the following steps:

- We created a subplot, which we then populated with individual plots. We specified that we wanted three plots vertically (by indicating `plt.subplots(3,1)`). We also specified the figure size by setting the `figsize` parameter.

- We added the individual plots using the `plot` method on a single Series (column) and specifying the axis on which we wanted to place the plot.

- We used the `set` method to specify the title and axis labels on each of the plots.

- When working in Jupyter Notebook, best practice is to run the `%matplotlib inline` magic (once per kernel) in order to display the plots directly below the code cell that has produced it.
- Additionally, if you are working on a MacBook with a Retina screen, running the extra IPython magic `%config InlineBackend.figure_format='retina'` will double the resolution of the plots we make.
- Definitely worth the extra line!

Plotly and Cufflinks Explantion

- By using cufflinks, we can use the iplot method directly on a pandas DataFrame.
- To create the previous plot, we used subplots (subplots=True), specified the shape of the figure (shape=(3,1)), indicated that the plots share the x-axis (shared_xaxes=True), and added the title (title='MSFT time series'). By default, the selected type of plot is a line chart (kind='line').
- One note about using plotly in Jupyter—in order to share a notebook with the option to view the plot (without running the script again), you should use nbviewer or render the notebook as an HTML file and share it then.
- The extra line of code cf.set_config_file(world_readable=True, theme='pearl', offline=True) sets up the configuration (such as the current theme or the offline mode) and should be used only once. It can be used again to reconfigure.

More plotting libraries

There are many more ways to create plots in Python. We list some of the libraries:

- Matplotlib (this is default plotting library we previously looked at)
- seaborn
- plotly
- plotly_express
- altair
- Plotnine

- We have decided to present the two selected for their simplicity, however, a specific use case might require using some of the previously mentioned libraries as they offer more freedom when creating the visualization.
- We should also mention that the plot method of a pandas DataFrame actually uses matplotlib for plotting, but the pandas API makes the process easier.

Identifying Outliers

- While working with any kind of data, we often encounter observations that are significantly different from the majority, that is, outliers.
- They can be a result of a wrong tick (price), something major happening on the financial markets, an error in the data processing pipeline, and so on.
- Many machine learning algorithms and statistical approaches can be influenced by outliers, leading to incorrect/biased results.
- That is why we should handle the outliers before creating any models.
- In this code, we look into detecting outliers using the 3σ approach.

Getting ready

- We continue from the *Converting prices to returns* code and have a DataFrame with Apple's stock price history and returns.

How to do it1

Execute the following steps to detect outliers using the 3σ approach, and mark them on a plot.

1. Calculate the rolling mean and standard deviation:

```
df_rolling = df[['simple_rtn']].rolling(window=21) \
                               .agg(['mean', 'std'])
df_rolling.columns = df_rolling.columns.droplevel()
```

2. Join the rolling metrics to the original data:

```
df_outliers = df.join(df_rolling)
```

3. Define a function for detecting outliers:

```
def indentify_outliers(row, n_sigmas=3):
    x = row['simple_rtn']
    mu = row['mean']
    sigma = row['std']
    if (x > mu + 3 * sigma) | (x < mu - 3 * sigma):
        return 1
    else:
        return 0
```

How to do it2

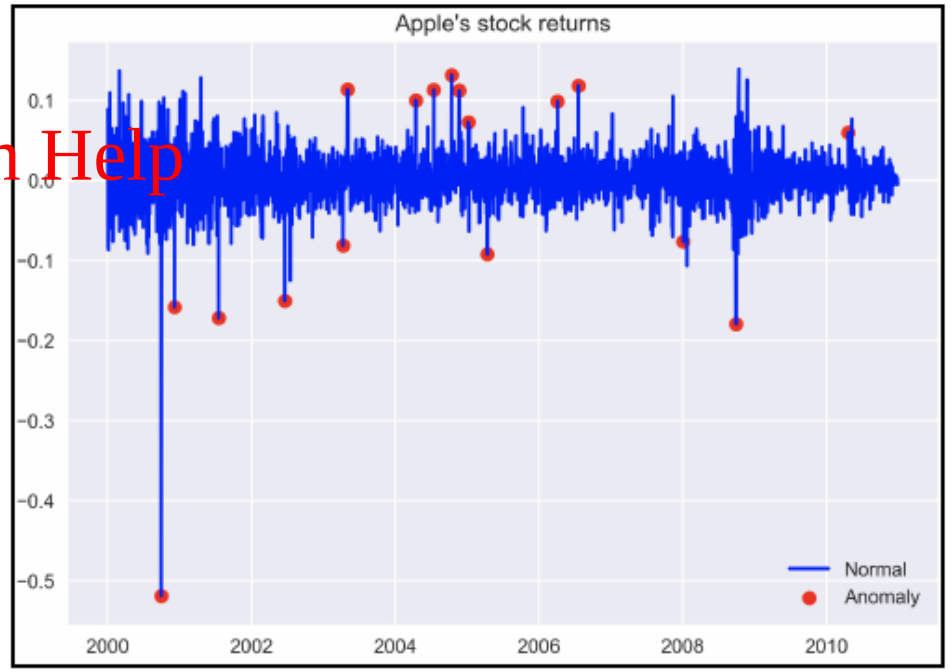
4. Identify the outliers and extract their values for later use:

```
df_outliers['outlier'] = df_outliers.apply(indentify_outliers,
                                           axis=1)
outliers = df_outliers.loc[df_outliers['outlier'] == 1,
                           ['simple_rtn']]
```

5. Plot the results:

```
fig, ax = plt.subplots()

ax.plot(df_outliers.index, df_outliers.simple_rtn,
        color='blue', label='Normal')
ax.scatter(outliers.index, outliers.simple_rtn,
           color='red', label='Anomaly')
ax.set_title("Apple's stock returns")
ax.legend(loc='lower right')
```


Executing the code results in the following plot:



- In the plot, we can observe outliers marked with a red dot.
- One thing to notice is that when there are two large returns in the vicinity, the algorithm identifies the first one as an outlier and the second one as a regular observation.
- This might be due to the fact that the first outlier enters the rolling window and affects the moving average/standard deviation.

How it works

- In the 3σ approach, for each time point, we calculated the moving average (μ) and standard deviation (σ) using the last 21 days (not including that day).
- We used 21 as this is the average number of trading days in a month, and we work with daily data.
- However, we can choose different values, and then the moving average will react faster/slower to changes.
- We can also use (exponentially) weighted moving average if we find it more meaningful in our particular case.
- The condition for a given observation x to be qualified as an outlier is $x > \mu + 3\sigma$ or $x < \mu - 3\sigma$.
- In the first step, we calculated the rolling metrics using the `rolling` method of a `pandas` DataFrame.
- We specified the window's size and the metrics we would like to calculate.
- In the second step, we joined the two DataFrames.
- In *Step 3*, we defined a function that returns 1 if the observation is considered an outlier, according to the 3σ rule (we parametrized the number of standard deviations), and 0 otherwise.
- Then, in the fourth step, we applied the function to all rows in the DataFrame using the `apply` method.
- In the last step, we visualized the returns series and marked the outliers using a red dot.
- In real-life cases, we should not only identify the outliers, but also treat them, for example, by capping them at the maximum/minimum acceptable value, replacing them by interpolated values, or by following any of the other possible approaches.

There's More …

- There are many different methods of identifying outliers in a time series, for example, using Isolation Forest, Hampel Filter, Support Vector Machines, and z-score (which is similar to the presented approach). These are beyond the topics of this lecture.

Additional link knowledge only (not compulsory)

- https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html
- https://towardsdatascience.com/outlier-detection-with-hampel-filter-85ddf523c73d?gi=aaf2b0b2fa69

- https://scikit-learn.org/stable/modules/svm.html
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.zscore.html

Investigating Stylized Facts of Asset Returns

- **Stylized facts** are statistical properties that appear to be present in many empirical asset returns (across time and markets).
- It is important to be aware of them because when we are building models that are supposed to represent asset price dynamics, the models must be able to capture/replicate these properties.
- In the following codes, we investigate the five stylized facts using an example of daily S&P 500 returns from the years 1985 to 2018.

Getting ready

- We download the S&P 500 prices from Yahoo Finance (following the approach in the *Getting data from Yahoo Finance* code) and calculate returns as in the *Converting prices to returns* code.
- We use the following code to import all the required libraries:

```
import pandas as pd
import numpy as np
import yfinance as yf
import seaborn as sns
import scipy.stats as scs
import statsmodels.api as sm
import statsmodels.tsa.api as smt
```

How to do it

- In this section, we investigate, one by one, five stylized facts in the S&P 500 series.

Fact1: Non-Gaussian Distribution of Returns1

Run the following steps to investigate the existence of this first fact by plotting the histogram of returns and a Q-Q plot.

1. Calculate the normal **Probability Density Function** (PDF) using the mean and standard deviation of the observed returns:

```
r_range = np.linspace(min(df.log_rtn), max(df.log_rtn), num=1000)
mu = df.log_rtn.mean()
sigma = df.log_rtn.std()
norm_pdf = scs.norm.pdf(r_range, loc=mu, scale=sigma)
```

2. Plot the histogram and the Q-Q plot:

```
fig, ax = plt.subplots(1, 2, figsize=(16, 8))

# histogram
sns.distplot(df.log_rtn, kde=False, norm_hist=True, ax=ax[0])
ax[0].set_title('Distribution of MSFT returns', fontsize=16)
ax[0].plot(r_range, norm_pdf, 'g', lw=2,
           label=f'N({mu:.2f}, {sigma**2:.4f})')
ax[0].legend(loc='upper left');

# Q-Q plot
qq = sm.qqplot(df.log_rtn.values, line='s', ax=ax[1])
ax[1].set_title('Q-Q plot', fontsize = 16)
```


https://numpy.org/doc/stable/reference/generated/numpy.linspace.html

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html

QQ Plots Extra Info:
https://towardsdatascience.com/q-q-plots-explained-5aa8495426c0

The probability density function for `norm` is:

$$f(x) = \frac{\exp(-x^2/2)}{\sqrt{2\pi}}$$

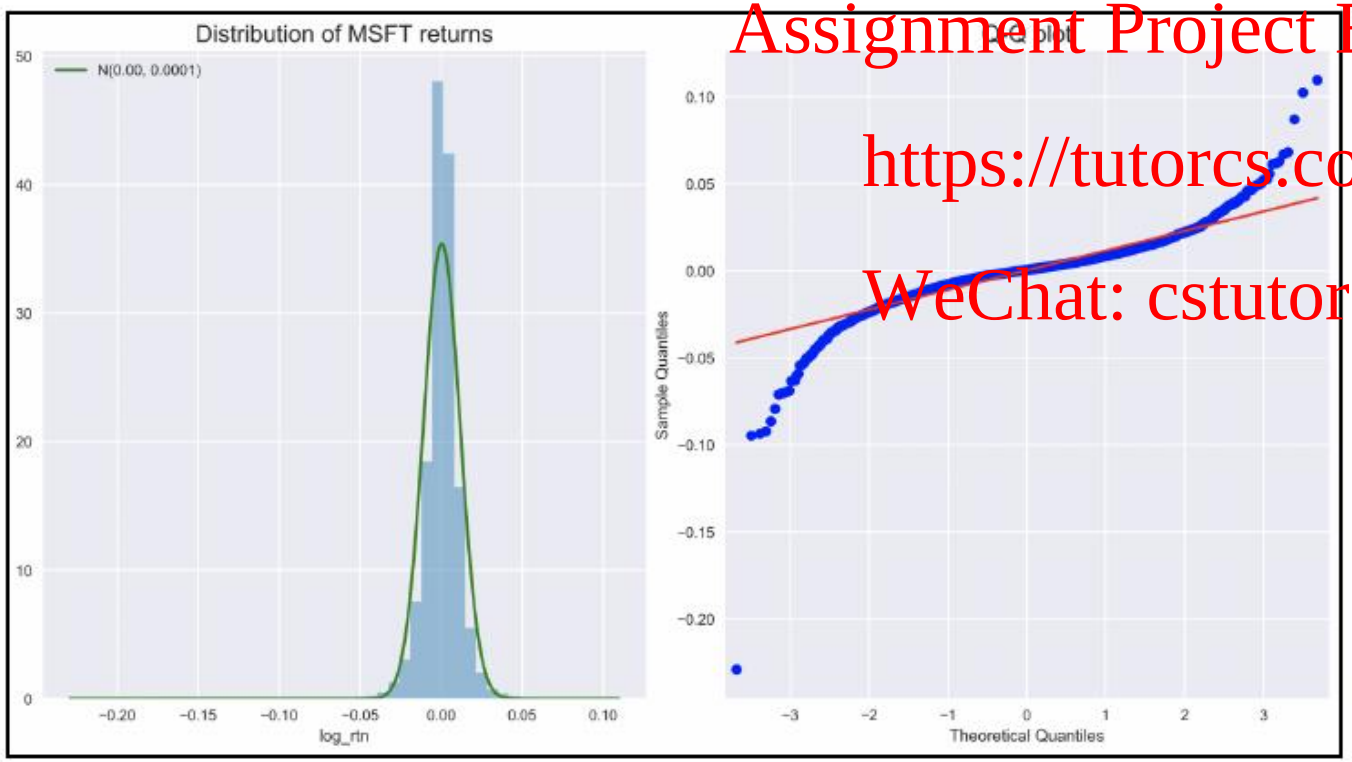
for a real number x.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `norm.pdf(x, loc, scale)` is identically equivalent to `norm.pdf(y) / scale` with `y = (x - loc) / scale`. Note that shifting the location of a distribution does not make it a "noncentral" distribution; noncentral generalizations of some distributions are available in separate classes.

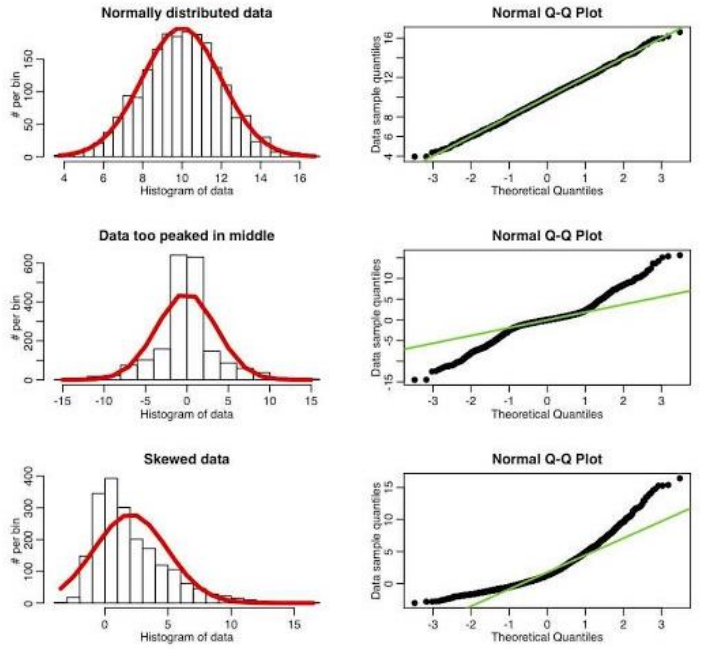
Non-Gaussian Distribution of Returns2

We can use the histogram (showing the shape of the distribution) and the Q-Q plot to assess the normality of the log returns.

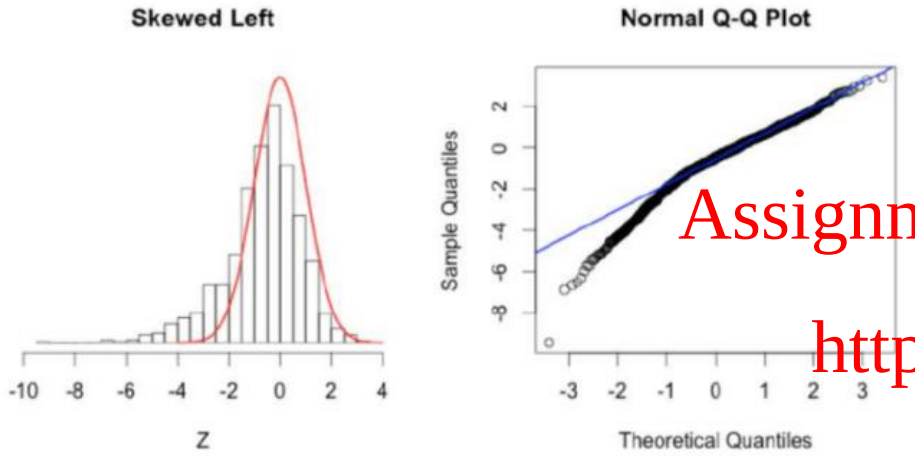
Executing the preceding code results in the following plot:



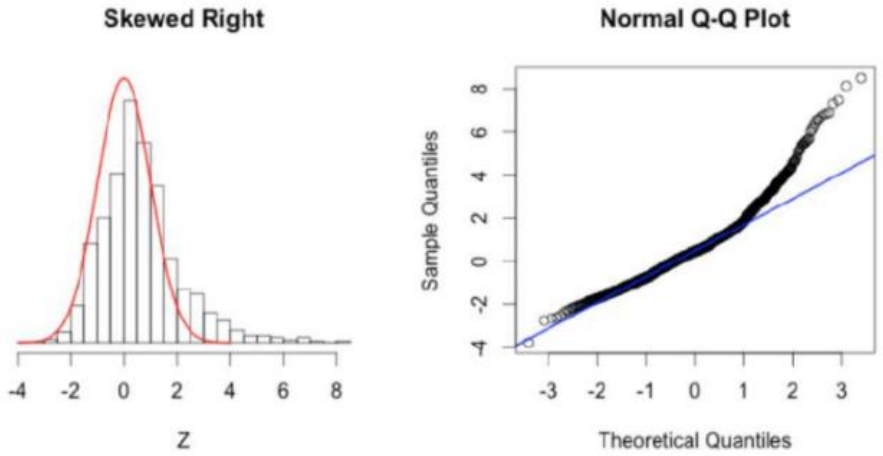
If all the points plotted on the graph perfectly lies on a straight line then we can clearly say that this distribution is Normally distribution because it is evenly aligned with the standard normal variate which is the simple concept of Q-Q plot.



Skewed QQ Plots



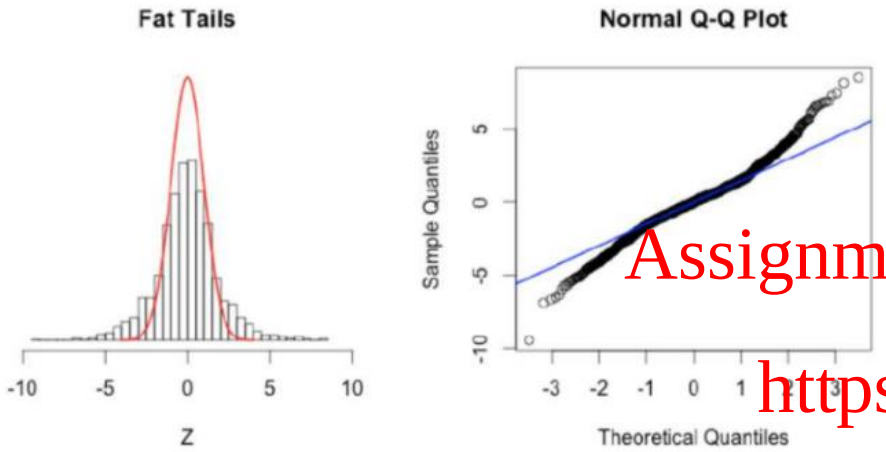
Left Skewed Q-Q plot for Normal Distribution



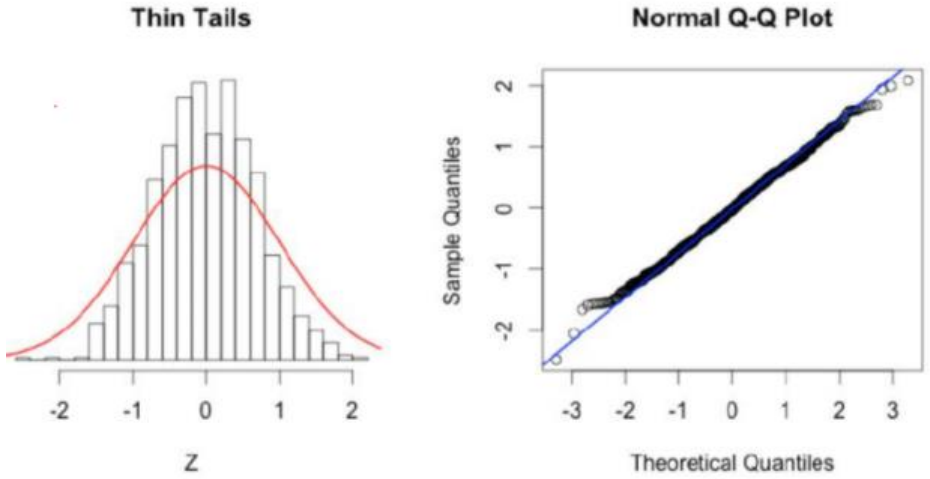
Right Skewed Q-Q plot for Normal Distribution

- If the bottom end of the Q-Q plot deviates from the straight line but the upper end is not, then we can clearly say that the distribution has a longer tail to its left or simply it is **left-skewed** (or ***negatively skewed***) but when we see the upper end of the Q-Q plot to deviate from the straight line and the lower and follows a straight line then the curve has a longer tail to its right and it is **right-skewed** (or ***positively skewed***).

Tailed QQ Plots



Fat-Tailed Q-Q plot for Normal DIstribution



Thin-Tailed Q-Q plot for Normal DIstribution

- Similarly, we can talk about the **Kurtosis** (a measure of "***Tailedness***") of the distribution by simply looking at its Q-Q plot
- The distribution with a fat tail will have both the ends of the Q-Q plot to deviate from the straight line and its center follows a straight line, whereas a thin-tailed distribution will form a Q-Q plot with a very less or negligible deviation at the ends thus making it a perfect fit for the Normal Distribution.

Non-Gaussian Distribution of Returns3

```
jb_test = scs.jarque_bera(df.log_rtn.values)

print('---------- Descriptive Statistics ----------')
print('Range of dates:', min(df.index.date), '-', max(df.index.date))
print('Number of observations:', df.shape[0])
print(f'Mean: {df.log_rtn.mean():.4f}')
print(f'Median: {df.log_rtn.median():.4f}')
print(f'Min: {df.log_rtn.min():.4f}')
print(f'Max: {df.log_rtn.max():.4f}')
print(f'Standard Deviation: {df.log_rtn.std():.4f}')
print(f'Skewness: {df.log_rtn.skew():.4f}')
print(f'Kurtosis: {df.log_rtn.kurtosis():.4f}')
print(f'Jarque-Bera statistic: {jb_test[0]:.2f} with p-value: {jb_test[1]:.2f}')
```

```
---------- Descriptive Statistics ----------
Range of dates: 1985-01-02 - 2018-12-28
Number of observations: 8569
Mean: 0.0003
Median: 0.0006
Min: -0.2290
Max: 0.1096
Standard Deviation: 0.0113
Skewness: -1.2624
Kurtosis: 28.0111
Jarque-Bera statistic: 282076.61 with p-value: 0.00
```


- By looking at the metrics such as the mean, standard deviation, skewness, and kurtosis we can infer that they deviate from what we would expect under normality.
- Additionally, the Jarque-Bera normality test gives us reason to reject the null hypothesis stating that the distribution is normal at the 99% confidence level.
- The null hypothesis for the test is that the data is normally distributed; the alternate hypothesis is that the data does not come from a normal distribution.
- In general, a large J-B value **indicates that errors are not normally distributed** and p value is 0.
- A p values of 0, it means the **null hypothesis is rejected and your test is statistically significant.**
- * Jarque, C. and Bera, A. (1980) “Efficient tests for normality, homoscedasticity and serial independence of regression residuals”, 6 Econometric Letters 255-259.
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.jarque_bera.html
- https://www.statisticshowto.com/jarque-bera-test/
- **Interpreting skewness and kurtosis:** https://www.analyticsvidhya.com/blog/2021/05/shape-of-data-skewness-and-kurtosis/

Non-Gaussian Distribution of Returns: Explanation

The name of the fact (Non-Gaussian distribution of returns) is pretty much self explanatory.

It was observed in the literature that (daily) asset returns exhibit the following:

- **Negative skewness (third moment)**: Large negative returns occur more frequently than large positive ones.
- **Excess kurtosis (fourth moment)** : Large (and small) returns occur more often than expected.

The pandas implementation of kurtosis is the one that literature refers to as excess kurtosis or Fisher's kurtosis. Using this metric, the excess kurtosis of a Gaussian distribution is 0, while the standard kurtosis is 3. This is not to be confused with the name of the stylized fact's excess kurtosis, which simply means kurtosis higher than that of normal distribution.

https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.kurtosis.html

Non-Gaussian Distribution of Returns: Histogram Distribution

Histogram of returns

- The first step of investigating this fact was to plot a histogram visualizing the distribution of returns. To do so, we used `sns.distplot` while setting `kde=False` (which does not use the Gaussian kernel density estimate) and `norm_hist=True` (this plot shows density instead of the count).
- Kde understanding here: https://scikit-learn.org/stable/modules/density.html
- To see the difference between our histogram and Gaussian distribution, we superimposed a line representing the PDF of the Gaussian distribution with the mean and standard deviation coming from the considered return series.
- First, we specified the range over which we calculated the PDF by using `np.linspace` (we set the number of points to 1,000, generally the more points the smoother the line) and then calculated the PDF using `scs.norm.pdf`. The default arguments correspond to the standard normal distribution, that is, with zero mean and unit variance.
- That is why we specified the `loc` and `scale` arguments as the sample mean and standard deviation, respectively.
- To verify the existence of the previously mentioned patterns, we should look at the following:
  - **Negative skewness**: The left tail of the distribution is longer, while the mass of the distribution is concentrated on the right side of the distribution.
  - **Excess kurtosis**: Fat-tailed and peaked distribution.

Non-Gaussian Distribution of Returns: QQ Plot

- After inspecting the histogram, we looked at the Q-Q (quantile-quantile) plot, on which we compared two distributions (theoretical and observed) by plotting their quantiles against each other.
- In our case, the theoretical distribution is Gaussian (Normal) and the observed one comes from the S&P 500 returns.
- To obtain the plot, we used the `sm.qqplot` function. If the empirical distribution is Normal, then the vast majority of the points will lie on the red line.
- However, we see that this is not the case, as points on the left side of the plot are more negative (that is, lower empirical quantiles are smaller) than expected in the case of the Gaussian distribution, as indicated by the line.
- This means that the left tail of the returns distribution is heavier than that of the Gaussian distribution. Analogical conclusions can be drawn about the right tail, which is heavier than under normality.

Non-Gaussian Distribution of Returns: Descriptive Statistics

- The last part involves looking at some statistics. We calculated them using the appropriate methods of pandas Series/DataFrames.
- We immediately see that the returns exhibit negative skewness and excess kurtosis.
- We also ran the Jarque-Bera test (scs.jarque_bera) to verify that returns do not follow a Gaussian distribution.
- With a pvalue of zero, we reject the null hypothesis that sample data has skewness and kurtosis matching those of a Gaussian distribution.

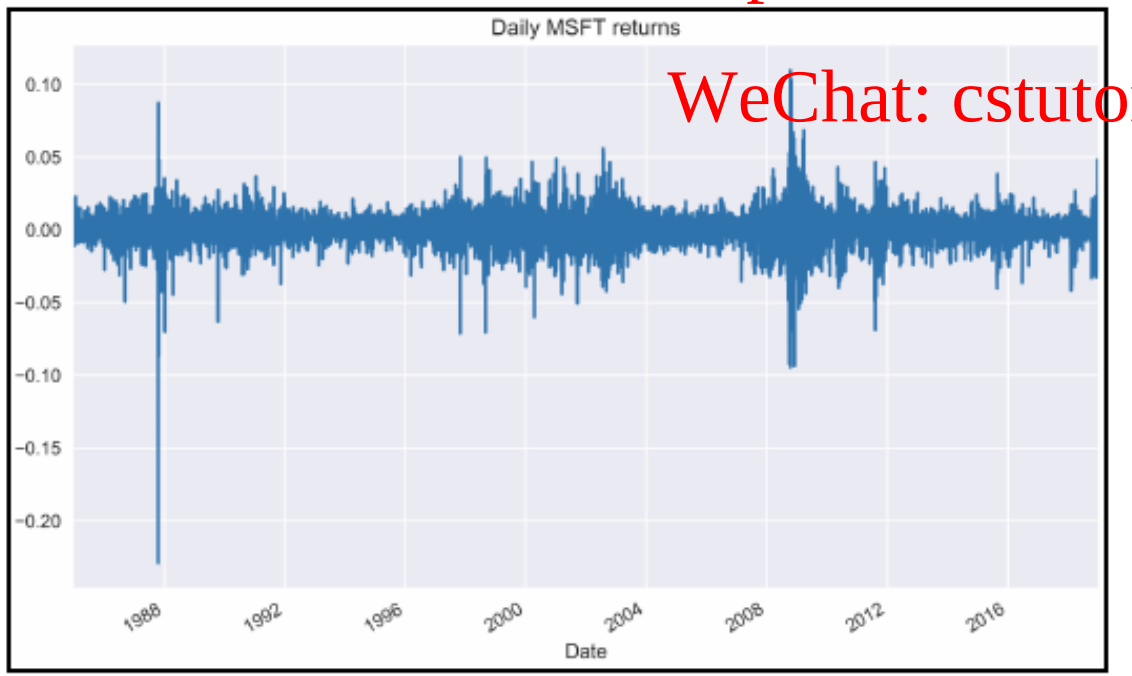
Fact 2: Volatility Clustering

Run the following code to investigate this second fact by plotting the log returns series.

1. Visualize the log returns series:

```
df.log_rtn.plot(figsize=(15,4), title='Daily MSFT returns')
```


Executing the code results in the following plot:



We can observe clear clusters of volatility—periods of higher positive and negative returns.

- The first thing we should be aware of when investigating stylized facts is the volatility clustering—periods of high returns alternating with periods of low returns, suggesting that volatility is not constant.
- To quickly investigate this fact, we plot the returns using the plot method of a pandas DataFrame.

Fact3: Absence of Autocorrelation in Returns

Investigate this third fact about the absence of autocorrelation in returns.

1. Define the parameters for creating the autocorrelation plots:

```
N_LAGS = 50
SIGNIFICANCE_LEVEL = 0.05
```

2. Run the following code to create the **autocorrelation function** (ACF) plot of log returns:

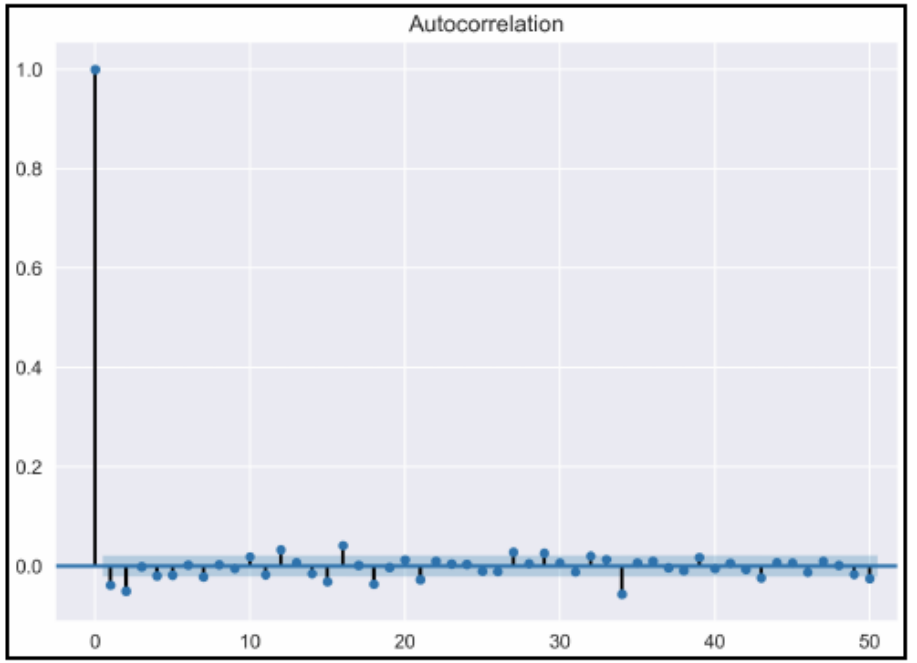
```
acf = smt.graphics.plot_acf(df.log_rtn,
                            lags=N_LAGS,
                            alpha=SIGNIFICANCE_LEVEL)
```

Executing the preceding code results in the following plot:

KEY TAKEAWAYS

- Autocorrelation represents the degree of similarity between a given time series and a lagged version of itself over successive time intervals.
- Autocorrelation measures the relationship between a variable's current value and its past values.
- An autocorrelation of +1 represents a perfect positive correlation, while an autocorrelation of negative 1 represents a perfect negative correlation.
- Technical analysts can use autocorrelation to measure how much influence past prices for a security have on its future price.

- Only a few values lie **outside** the confidence interval (we do **not** look at lag 0) and can be considered statistically significant.
- We can assume that we have verified that there is **no** autocorrelation in the log returns series.
- https://www.statsmodels.org/dev/generated/statsmodels.graphics.tsaplots.plot_acf.html
- https://corporatefinanceinstitute.com/resources/knowledge/other/autocorrelation/
- https://www.investopedia.com/terms/a/autocorrelation.asp
- https://medium.com/analytics-vidhya/5-steps-to-get-an-understanding-on-correlation-auto-correlation-and-partial-auto-correlation-e71f6b4bba81



Absence of Autocorrelation in Returns: Explanation

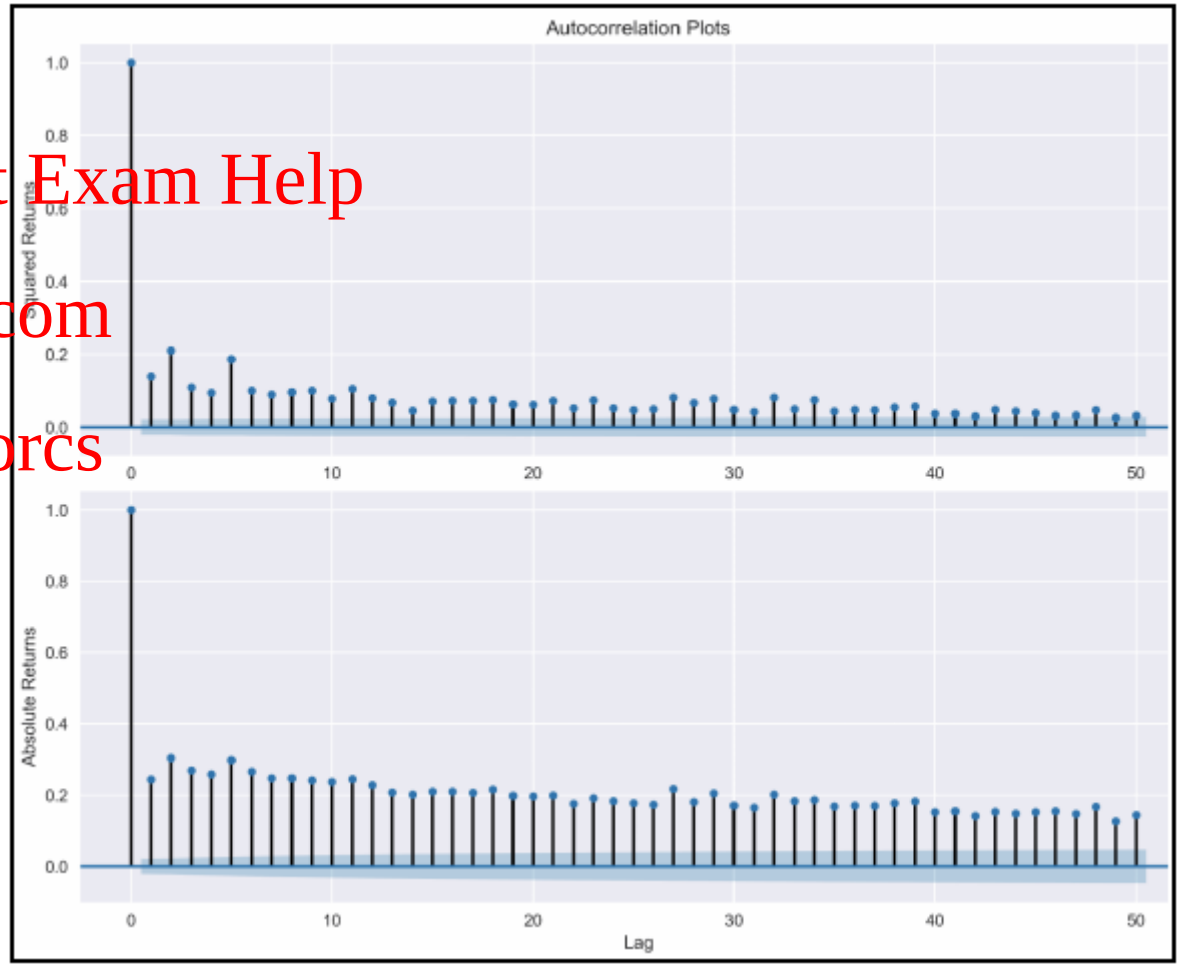
- Autocorrelation (also known as **serial correlation**) measures how similar is a given time series to the lagged version of itself, over successive time intervals.
- To investigate whether there is significant autocorrelation in returns, we created the autocorrelation plot using `plot_acf` from the statsmodels library.
- We inspected 50 lags and used the default `alpha=0.05`, which means that we also plotted the 95% confidence interval.
- Values **outside** of this interval can be considered statistically significant.

Fact 4: Small and decreasing Autocorrelation in Squared & Absolute Returns

Investigate this fourth fact by creating the ACF plots of squared and absolute returns.

1. Create the ACF plots:

```
fig, ax = plt.subplots(2, 1, figsize=(12, 10))

smt.graphics.plot_acf(df.log_rtn ** 2, lags=N_LAGS,
                      alpha=SIGNIFICANCE_LEVEL, ax = ax[0])

ax[0].set(title='Autocorrelation Plots',
          ylabel='Squared Returns')

smt.graphics.plot_acf(np.abs(df.log_rtn), lags=N_LAGS,
                      alpha=SIGNIFICANCE_LEVEL, ax = ax[1])
ax[1].set(ylabel='Absolute Returns',
          xlabel='Lag')
```

We can observe the small and decreasing values of autocorrelation for the squared and absolute returns, which are in line with the fourth stylized fact.

To verify this fact, we also used the plot_acf function from the statsmodels library; however, this time we applied it to the squared and absolute returns.

Fact5: Leverage Effect1

For the fifth fact, run the following steps to investigate the existence of the leverage effect.

1. Calculate volatility measures as rolling standard deviations:

```
df['moving_std_252'] = df[['log_rtn']].rolling(window=252).std()
df['moving_std_21'] = df[['log_rtn']].rolling(window=21).std()
```

2. Plot all the series for comparison:

```
fig, ax = plt.subplots(3, 1, figsize=(18, 15),
                       sharex=True)

df.adj_close.plot(ax=ax[0])
ax[0].set(title='MSFT time series',
          ylabel='Stock price ($)')

df.log_rtn.plot(ax=ax[1])
ax[1].set(ylabel='Log returns (%)')

df.moving_std_252.plot(ax=ax[2], color='r',
                       label='Moving Volatility 252d')
df.moving_std_21.plot(ax=ax[2], color='g',
                      label='Moving Volatility 21d')
ax[2].set(ylabel='Moving Volatility',
          xlabel='Date')
ax[2].legend()
```

Leverage Effect2

We can now investigate the leverage effect by visually comparing the price series to the (rolling) volatility metric:

This fact states that most measures of an asset's volatility are negatively correlated with its returns, and we can indeed observe a pattern of **increased volatility** when the **prices** go **down** and **decreased volatility** when **prices** are **rising**.

- This fact states that most measures of asset volatility are **negatively correlated** with their **returns**.
- To investigate it, we used the moving standard deviation (calculated using the rolling method of a pandas DataFrame) as a measure of historical volatility.
- We used windows of 21 and 252 days, which correspond to one month and one year of trading data.

There's More …. Leverage Effect Continued1

- We present another method of investigating the leverage effect (fact 5).
- To do so, we use the VIX (CBOE Volatility Index), which is a popular metric of the stock market's expectation regarding volatility.
- The measure is implied by option prices on the S&P 500 index.
- We take the following steps:

1. Download and preprocess the prices of the S&P 500 and VIX:

```
df = yf.download(['^GSPC', '^VIX'],
                 start='1985-01-01',
                 end='2018-12-31',
                 progress=False)
df = df[['Adj Close']]
df.columns = df.columns.droplevel(0)
df = df.rename(columns={'^GSPC': 'sp500', '^VIX': 'vix'})
```

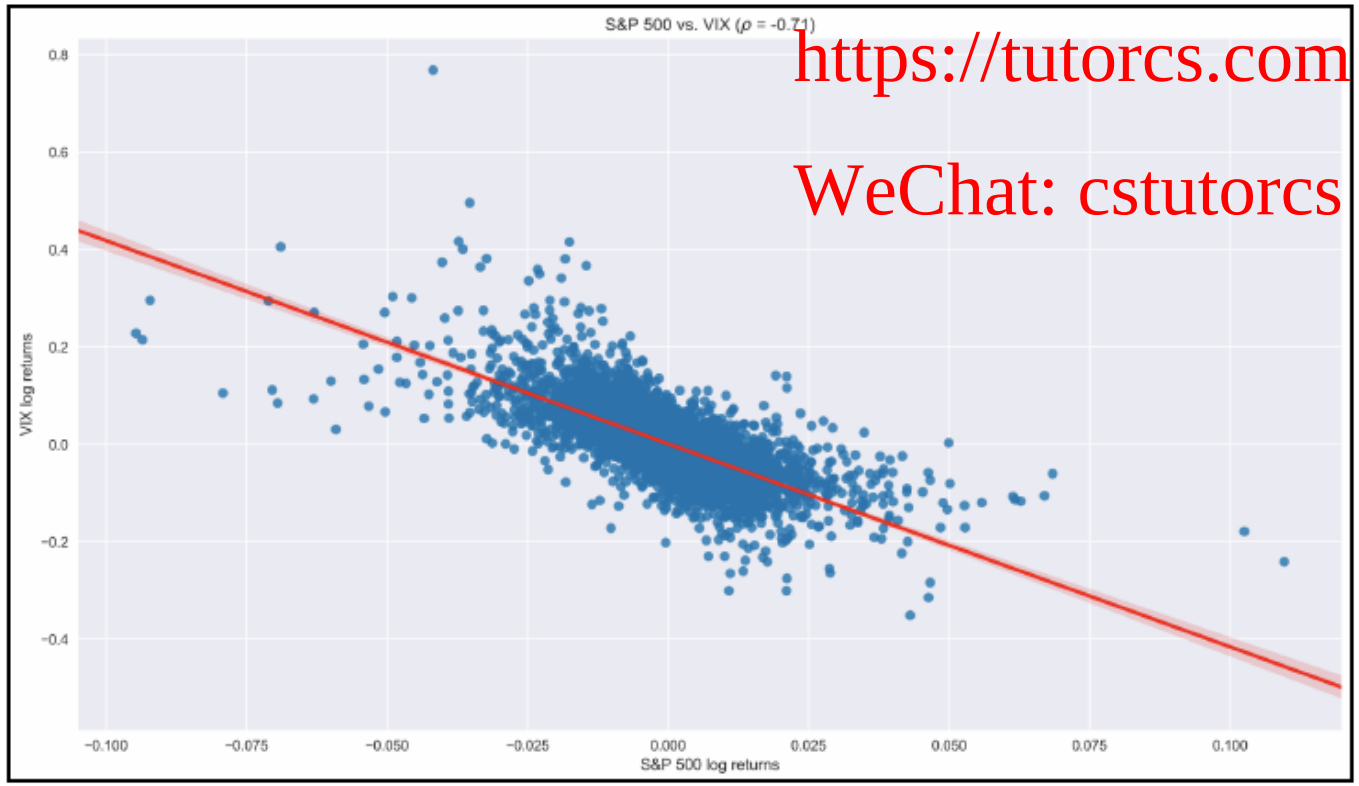
2. Calculate the log returns (we can just as well use percentage change-simple returns):

```
df['log_rtn'] = np.log(df.sp500 / df.sp500.shift(1))
df['vol_rtn'] = np.log(df.vix / df.vix.shift(1))
df.dropna(how='any', axis=0, inplace=True)
```

There's More …. Leverage Effect Continued2

3. Plot a scatterplot with the returns on the axes and fit a regression line to identify the trend:

```
corr_coeff = df.log_rtn.corr(df.vol_rtn)

ax = sns.regplot(x='log_rtn', y='vol_rtn', data=df,
                 line_kws={'color': 'red'})
ax.set(title=f'S&P 500 vs. VIX ($\\rho$ = {corr_coeff:.2f})',
       ylabel='VIX log returns',
       xlabel='S&P 500 log returns')
```

We additionally calculated the correlation coefficient between the two series and included it in the title:

We can see that both the negative slope of the regression line and a strong negative correlation between the two series confirm the existence of the leverage effect in the return series.

For more information, refer to the following:

- Cont, R. (2001). Empirical properties of asset returns: stylized facts and statistical issues.
- https://seaborn.pydata.org/generated/seaborn.regplot.html