PYTHON FOR FINANCIAL ANALYSIS

Investment Strategies Backtesting with Python

pandas

NumPy

matplotlib

Python for Financial Analysis

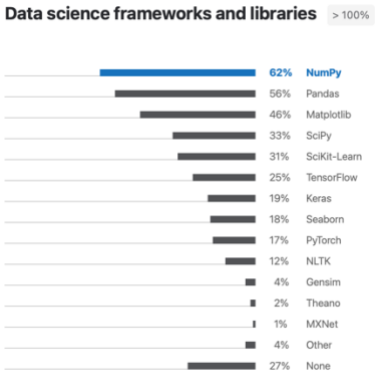
This **eBook** will teach you all you need for **backtesting investment strategies** using **Python** libraries **Pandas**, **NumPy**, and **Matplotlib** in **Jupyter Notebook**.

What will this eBook teach you?

Great question.

This **eBook** assumes that you are familiar with **Python**. If not, feel free to follow along.

The main focus of this book is to use the **Python** libraries **Pandas**, **NumPy** and **Matplotlib** for **backtesting investment strategies**.



According to the **2020 official annual Python Developers Survey** these are the most commonly used libraries by Data Scientist.

By the time you finish this **eBook**, you will have covered the following with practical code examples (**available on GitHub**):

- How to work with **financial time series data**.
- Read data from **CSV** files and directly from **API**.
- Visualize financial time series data with **Matplotlib**.
- Calculate return (**CAGR**), **maximum drawdown**, and **volatility** (backtesting performance).
- How to work with **investment strategies**.
- **Backtesting** investment strategies.

Simply, *master all you need to know for **backtesting investment strategies** with **Python***.

How to get the most out of this eBook

This book is accompanied with **Jupyter** Notebooks available from this **GitHub repository**.

Using the **Notebooks** along with the book will improve your learning journey.

There are two ways to move forward.

- Go to this **GitHub repository** and download all Notebooks and work with them in your own **Jupyter Notebook** environment (**you can download as Zip-file**).
- Use the links to **Colab** given in the **GitHub description** and work with the **Notebooks** interactively in **Colab** (no need for installation of setup).

Install your own setup?

Anaconda comes with **Jupyter Notebook** and **Python**, which is all you need. Install **Anaconda Individual Edition** and Launch **Jupyter Notebook**. See the **documentation for troubleshooting**.

Table of Contents

00 – Pandas DataFrames

In this chapter we will get acquainted with the main library when working with financial data in Python: **Pandas**.

Note Pandas is part of the Anaconda distribution and is not needed to be installed separately.

Learning objectives

- Open the project in **Jupyter Notebook**.
- How to execute the code.
- Load financial data from a CSV file.
- Learn about the main data type in **Pandas**, **DataFrame.**
- Make your first project where you calculate the **Simple Moving Average** of the Close price.

What is Pandas?

Pandas *(derived from **panel** and **data**)* contains powerful and easy-to-use tools for doing practical, real world data analysis in **Python**.

The best way to learn about new things is to relate it to something similar.

Let's try to load some data into **Pandas** and see how similar it is to an **Excel** spreadsheet.

DataFrames and Read Data

Download CSV from page and read into DataFrame

```
In [1]: import pandas as pd

In [2]: csv_file = "https://raw.githubusercontent.com/LearnPythonWithRune/PythonForFinancialAnalysis/
        aapl = pd.read_csv(csv_file)

In [3]: aapl.head()
```

Out[3]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2016-03-09	25.327499	25.395000	25.067499	25.280001	23.513155	108806800
1	2016-03-10	25.352501	25.559999	25.037500	25.292500	23.524775	134054400
2	2016-03-11	25.559999	25.570000	25.375000	25.565001	23.778234	109632800
3	2016-03-14	25.477501	25.727501	25.445000	25.629999	23.838688	100304400
4	2016-03-15	25.990000	26.295000	25.962500	26.145000	24.317701	160270800

Note The above code snippets are available directly on **GitHub**. You can either download the Notebooks or run them directly in **Colab** from the links provided (**see GitHub**).

A brief introduction to Jupyter Notebook

The above code snippets are taken directly from a Jupyter Notebook.

If you are new to Jupyter Notebook – this section is for you. Otherwise, you can skip to the next section, where we will learn about the code above.

What is Jupyter Notebook?

Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text.

Said differently.

- It works in your browser.
- You can write your **Python** code in it.
- It can execute the **Python** code for you and show the results.
- You can write text in it.
- It can create visualizations with charts.

Jupyter Notebooks are very popular among beginners and **Data Scientists**.

- It is easy to learn programming in a **Notebook**.
- The needs of a **Data Scientist** are well served in a **Notebook**.
- **Notebooks** are good to explore data and programming in an easy way.

Financial analysis can be seen as a **Data Science** task.

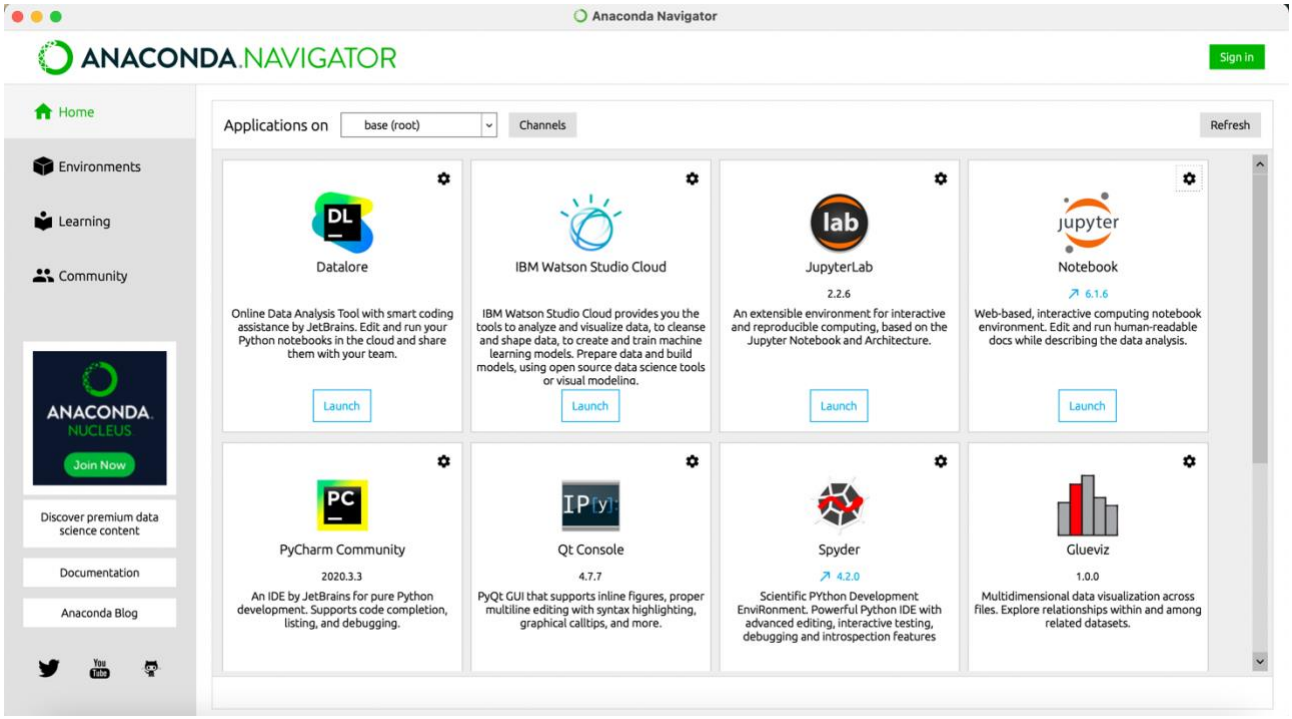
- You work with **financial data**.
- You explore the data and make calculations.
- You visualize the data.

Jupyter Notebook seems to be a perfect match for financial analysis.

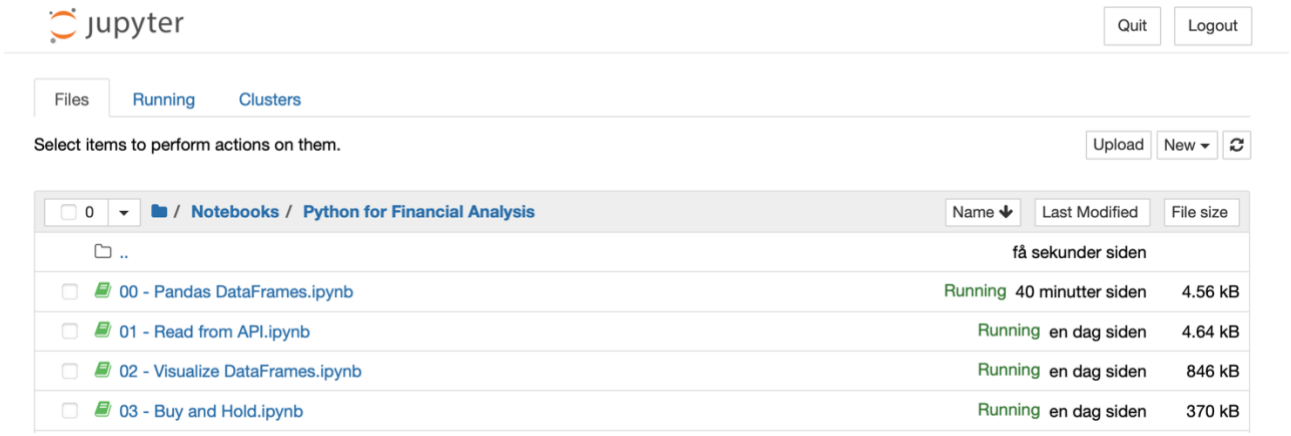
How to open the Jupyter Notebooks from this eBook

How to get started.

- Install **Anaconda Individual Edition**.
- **Download the zip file from the GitHub**.
- Unzip the content of the zip-file in a location you can find.
- Launch **Anaconda Navigator** from your main menu *(Launchpad or Windows menu).*
- Inside **Anaconda Navigator** launch **Jupyter Notebook**.



This should bring you into your main browser with the **Jupyter Notebook Dashboard** (see below).



- Navigate to the extracted files from the downloaded zip file.
- Click on the first **Notebook 00 – Pandas DataFrames.ipynb**

This will open the Notebook in a new browser tab.

How to execute the code in the Jupyter Notebook

A **Notebook** is divided into cells. A cell is a box like this one.

```
In [1]: import pandas as pd
```

The above cell contains 1 line of code. The **import** statement of the main library in this case.

A cell has been executed if it has a number, like the above **In [1]:**

If the cell hasn't been executed it would not contain a number.

```
In [ ]: import pandas as pd
```

To execute a cell can be done in various ways. The normal way is to mark the cell (can be done with mouse or keys and press enter).

```
In [ ]: import pandas as pd
```

Then press **Shift + Enter** to execute it.

Alternatively, you can press **Run** with your mouse 

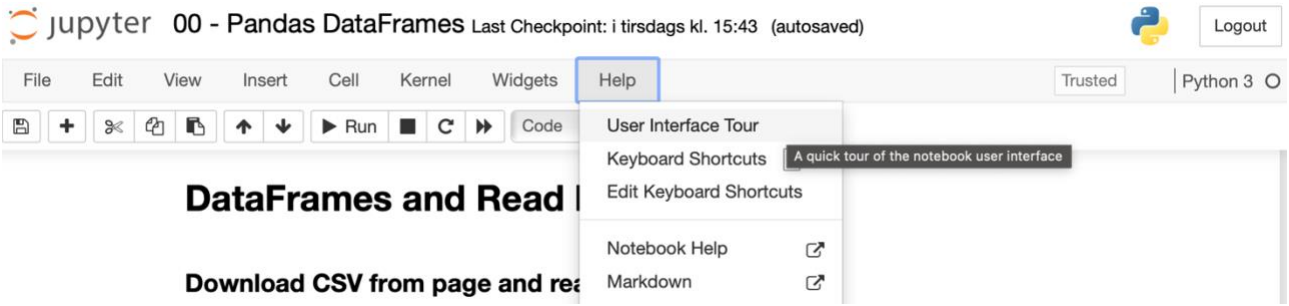
The cell will execute and when done finish with a number.

```
In [1]: import pandas as pd
```

To edit a cell, just mark it and press **enter.** Then you enter edit-mode.

This is all you need to know to follow along this **eBook**.

If you would like to learn more about the specific elements within the Notebook Editor, you can go through the user interface tour by selecting Help in the menu bar then selecting **User Interface Tour**.



The code explained

Let's get back to the code and break it down.

DataFrames and Read Data

Download CSV from page and read into DataFrame

```
In [1]: import pandas as pd
```

```
In [2]: csv_file = "https://raw.githubusercontent.com/LearnPythonWithRune/PythonForFinancialAnalysis/
        aapl = pd.read_csv(csv_file)
```

```
In [3]: aapl.head()
```

Out[3]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2016-03-09	25.327499	25.395000	25.067499	25.280001	23.513155	108806800
1	2016-03-10	25.352501	25.559999	25.037500	25.292500	23.524775	134054400
2	2016-03-11	25.559999	25.570000	25.375000	25.565001	23.778234	109632800
3	2016-03-14	25.477501	25.727501	25.445000	25.629999	23.838688	100304400
4	2016-03-15	25.990000	26.295000	25.962500	26.145000	24.317701	160270800

The first code cell, **import pandas as pd**, can look intimidating at first.

This ensures access to a library called **pandas** (the main library we use – **read more about Pandas here**). You use the **as pd** for easy access, as we see in the next cell.

The real magic happens in the next cell. We define the location to the **CSV** file we want to read (**csv_file = "..."**).

This CSV file was downloaded from **Yahoo! Finance**. It can be downloaded by finding **Apple stock** and go to **Historic Data** and use the **Download**.

What is a CSV file?

A **CSV** file is a **Comma Separated Values** file.

The one we use here looks like this (or the beginning of it).

```
1 Date,Open,High,Low,Close,Adj Close,Volume
2 2016-03-09,25.327499,25.395000,25.067499,25.280001,23.513155,108806800
3 2016-03-10,25.352501,25.559999,25.037500,25.292500,23.524775,134054400
4 2016-03-11,25.559999,25.570000,25.375000,25.565001,23.778234,109632800
5 2016-03-14,25.477501,25.727501,25.445000,25.629999,23.838688,100304400
6 2016-03-15,25.990000,26.295000,25.962500,26.145000,24.317701,160270800
7 2016-03-16,26.152500,26.577499,26.147499,26.492500,24.640911,153214000
8 2016-03-17,26.379999,26.617500,26.240000,26.450001,24.601379,137682800
```

As you can see, the first row contains the names of each column. Then each line is separated with commas for each value corresponding to the name in the first row.

Back on track

The second cell.

```
In [2]: csv_file = "https://raw.githubusercontent.com/LearnPythonWithRune/PythonForFinancialAnalysis/
        aapl = pd.read_csv(csv_file)
```

Here we define where the **CSV** file is located. Now, this is set to be the location in the repository. If you execute it locally, you can change the **cvs_file** to the following.

csv_file = "AAPL.csv"

This will load it from your local computer and not from the repository.

Note If you run the code in **Colab** you need to load it from the repository.

Finally, the following line.

```
In [3]: aapl.head()
```

Out[3]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2016-03-09	25.327499	25.395000	25.067499	25.280001	23.513155	108806800
1	2016-03-10	25.352501	25.559999	25.037500	25.292500	23.524775	134054400
2	2016-03-11	25.559999	25.570000	25.375000	25.565001	23.778234	109632800
3	2016-03-14	25.477501	25.727501	25.445000	25.629999	23.838688	100304400
4	2016-03-15	25.990000	26.295000	25.962500	26.145000	24.317701	160270800

It shows the 5 first lines of the content we loaded.

The content is referenced by the variable **aapl.** To shows the first 5 lines of content we use **aapl.head().** This calls the function **head()** on the object **aapl**.

The object **aapl** is a **DataFrame**.

As you see, the **DataFrame** looks similar to an Excel sheet.

Learn about DataFrames

A **DataFrame** as rows and columns.

Each column has a data type.

```
In [4]: aapl.dtypes
```

```
Out[4]: Date          object
        Open         float64
        High         float64
        Low          float64
        Close        float64
        Adj Close    float64
        Volume         int64
        dtype: object
```

Here we see the that column **Date** has data type **object**, while **Open, High, Low, Close, Adj Close** has data type **float64**, while **Volume** has data type **int64**.

A **DataFrame** has an **index**.

```
In [5]: aapl.index
```

```
Out[5]: RangeIndex(start=0, stop=1258, step=1)
```

Here the index is a range from 0 to 1258 with step 1.

Basically, that means the numbers 0, 1, 2, 3, …, 1257.

Get the data as we want

When working with financial data there are two things that would make the above data more convenient.

First, if the **Date** columns was interpreted as dates. As it is now, it is just a string stored with some numbers. If the **DataFrame** knows it is dates, we can do a lot of nice manipulations with the data. This will be handy later.

Second, we want to use the **Date** column as **index**. This is makes it easy to see data in a specific date time interval.

Luckily, this is straight forward to do.

```
In [6]: aapl = pd.read_csv(csv_file, index_col="Date", parse_dates=True)
```

The **index_col="Date"** sets the index to be the **Date** column. And **parse_dates=True** ensures that the dates (in this case in the Date column) are parsed as dates.

```
In [7]: aapl.head()
```

Out[7]:

Date	Open	High	Low	Close	Adj Close	Volume
2016-03-09	25.327499	25.395000	25.067499	25.280001	23.513155	108806800
2016-03-10	25.352501	25.559999	25.037500	25.292500	23.524775	134054400
2016-03-11	25.559999	25.570000	25.375000	25.565001	23.778234	109632800
2016-03-14	25.477501	25.727501	25.445000	25.629999	23.838688	100304400
2016-03-15	25.990000	26.295000	25.962500	26.145000	24.317701	160270800

As the above shows, we now have the **Date** column as the index.

```
In [8]: aapl.dtypes
```

Out[8]:

```
Open         float64
High         float64
Low          float64
Close        float64
Adj Close    float64
Volume         int64
dtype: object
```

The data types are looking similar, except we do not have the **Date** column there anymore. This is because it is the index now.

```
In [9]: aapl.index
```

Out[9]:

```
DatetimeIndex(['2016-03-09', '2016-03-10', '2016-03-11', '2016-03-14',
               '2016-03-15', '2016-03-16', '2016-03-17', '2016-03-18',
               '2016-03-21', '2016-03-22',
               ...
               '2021-02-23', '2021-02-24', '2021-02-25', '2021-02-26',
               '2021-03-01', '2021-03-02', '2021-03-03', '2021-03-04',
               '2021-03-05', '2021-03-08'],
              dtype='datetime64[ns]', name='Date', length=1258, freq=None)
```

We see that the index is now of type **DatetimeIndex**.

How to use the index of a DataFrame

The **DateFrame** has two main ways to access rows with the index.

The **loc** and **iloc**. We will explore both here.

```
In [10]: aapl.loc['2021-02-24']
Out[10]: Open          1.249400e+02
         High          1.255600e+02
         Low           1.222300e+02
         Close         1.253500e+02
         Adj Close     1.253500e+02
         Volume        1.110399e+08
         Name: 2021-02-24 00:00:00, dtype: float64
```

The **loc** can access a row in our **DataFrame** by using the *"index name"*. Here, it is a date, hence, we can write the date to access it.

Notice, that the output if formatted a bit different than when we used **head()**.

Here the data is simply put in a column like fashion. Also, notice, that it only has one data type (float64).

This is because it converts the row to a **Series** (the other main data type from the **Pandas** library).

Don't worry too much about that now.

```
In [11]: aapl.loc['2021-02-24':'2021-02-28']
Out[11]:
```

Date	Open	High	Low	Close	Adj Close	Volume
2021-02-24	124.940002	125.559998	122.230003	125.349998	125.349998	111039900
2021-02-25	124.680000	126.459999	120.540001	120.989998	120.989998	148199500
2021-02-26	122.589996	124.849998	121.199997	121.260002	121.260002	164320000

You can access a date interval by using the slicing technique above.

Notice that because we use **DatetimeIndex**, it can actually figure out what the date range is, even though we have used dates with no data (it was weekend the 27th and 28th).

```
In [12]: aapl.iloc[0]
Out[12]: Open          2.532750e+01
         High          2.539500e+01
         Low           2.506750e+01
         Close         2.528000e+01
         Adj Close     2.351315e+01
         Volume        1.088068e+08
         Name: 2016-03-09 00:00:00, dtype: float64
```

The **iloc** is an integer location. The **iloc[0]** will access the first row.

Similarly, **iloc[1]** the second row and so forth.

```
In [13]: aapl.iloc[-1]
Out[13]: Open          1.209300e+02
         High          1.210000e+02
         Low           1.162100e+02
         Close         1.163600e+02
         Adj Close     1.163600e+02
         Volume        1.539186e+08
         Name: 2021-03-08 00:00:00, dtype: float64
```

We can access the data from the back with negative indexing. Just like **Python** lists.

```
In [14]: aapl.iloc[0:4]
Out[14]:
```

Date	Open	High	Low	Close	Adj Close	Volume
2016-03-09	25.327499	25.395000	25.067499	25.280001	23.513155	108806800
2016-03-10	25.352501	25.559999	25.037500	25.292500	23.524775	134054400
2016-03-11	25.559999	25.570000	25.375000	25.565001	23.778234	109632800
2016-03-14	25.477501	25.727501	25.445000	25.629999	23.838688	100304400

Finally, we can make slicing from row 0 to 4 (where row 4 is excluded). Hence, we see the rows of index 0, 1, 2, and 3.

Our first project – Simple Moving Average

We will learn how to calculate a **simple moving average**.

In finance, a simple moving average (SMA) is a stock indicator that is commonly used in technical analysis.

*"A simple moving average (SMA) is a calculation that takes the arithmetic mean of a given set of prices over the specific number of days in the past; for example, over the previous 15, 30, 100, or 200 days." – **Investopedia.org**.*

That is, for every day, we look at the last days prices and take the average of them.

It will be clear in a moment.

Let's calculate the SMA of the **Close** price.

Step 1

Then we need to access the **Close** prices.

```
In [15]: aapl['Close']

Out[15]: Date
         2016-03-09     25.280001
         2016-03-10     25.292500
         2016-03-11     25.565001
         2016-03-14     25.629999
         2016-03-15     26.145000
                           ...
         2021-03-02    125.120003
         2021-03-03    122.059998
         2021-03-04    120.129997
         2021-03-05    121.419998
         2021-03-08    116.360001
         Name: Close, Length: 1258, dtype: float64
```

As you see, this can be done quite easy.

Step 2

We need to understand the method **rolling(.)**, which provides a rolling window of calculation.

Let's explore it.

```
In [16]: aapl['Close'].rolling(2).mean().head()

Out[16]: Date
         2016-03-09          NaN
         2016-03-10    25.286251
         2016-03-11    25.428751
         2016-03-14    25.597500
         2016-03-15    25.887500
         Name: Close, dtype: float64
```

We apply **rolling** with argument 2, **rolling(2)**.

The **mean()** calculates the average of the rolling window.

```
In [17]: (25.292500 + 25.280001)/2

Out[17]: 25.2862505

In [18]: (25.565001 + 25.292500)/2

Out[18]: 25.4287505
```

If we take the average of the **Close** price of the 9th and 10th of March, then we get the value given for *the* **rolling(2).mean()** the 10th of March.

Similarly, for 10th and 11th of March, as the above example demonstrates.

Hence, the **rolling(2).mean()** applies the **mean()** function on the window of the last 2 rows.

```
In [19]: aapl['Close'].rolling(5).mean().head()

Out[19]: Date
         2016-03-09        NaN
         2016-03-10        NaN
         2016-03-11        NaN
         2016-03-14        NaN
         2016-03-15    25.5825
         Name: Close, dtype: float64
```

Similarly, the **rolling(5).mean()** applies the **mean()** function on the window of the last 5 rows.

Step 3

Now we can calculate the simple moving average of any days.

In this example we will calculate the simple moving average of 10 days.

Often, the simple moving average is called the moving average (**MA**). On financial pages, like **Yahoo! Finance**, they use the abbreviation **MA** for the simple moving average.

```
In [20]: aapl['MA10'] = aapl['Close'].rolling(10).mean()

In [21]: aapl.tail()
```

Out[21]:

Date	Open	High	Low	Close	Adj Close	Volume	MA10
2021-03-02	128.410004	128.720001	125.010002	125.120003	125.120003	102015300	126.279000
2021-03-03	124.809998	125.709999	121.839996	122.059998	122.059998	112430400	125.401000
2021-03-04	121.750000	123.599998	118.620003	120.129997	120.129997	177275300	124.442999
2021-03-05	120.980003	121.940002	117.570000	121.419998	121.419998	153590400	123.598000
2021-03-08	120.930000	121.000000	116.209999	116.360001	116.360001	153918600	122.634000

Now that was nice.

Notice that we added a new column to our **DataFrame** simply by specifying it.

Also, notice that we use **aapl.tail()** to get the last 5 rows of our dataset.

Summary

Now we have learnt what a **DataFrame** is. How to read data properly from financial pages like **Yahoo! Finance** into our **DataFrame**. How to access data rows based on a date index and integer value index. Also, how to slice data, i.e., showing subsets of data rows.

Finally, we made a project where we calculated the simple moving average of the **Close** price and added it to our **DataFrame**.

01 – Read from API

In the last chapter we learned about **DataFrames** and how to operate time series data from financial pages like **Yahoo! Finance**. This required a manual step where we download a **CSV** file from the page.

In this chapter we will learn how to automatically read from the **Yahoo! Finance** API using the **Pandas Datareader** library.

Learning objectives

- What is an API.
- How to read historic stock prices directly from an API.
- Specify the time interval you want prices from.
- How to read historic stock prices for multiple tickers at the same time.

What is an API

"An application programming interface (API) is a computing interface that defines interactions between multiple software or mixed hardware-software intermediaries." – **Wikipedia.org**

For our purpose, it helps us read historic stock prices directly from **Jupyter Notebook** without manually downloading a **CSV** file from a page like **Yahoo! Finance**.

How to read data from Yahoo! Finance API

To achieve that we will use another library.

Pandas Datareader, an up to date remote data access for pandas, works for multiple versions of pandas.

```
In [1]: import pandas_datareader as pdr
        import datetime as dt
```

The library is imported together with another library, the **datetime**.

The **datetime** library is a standard library used to specify dates, which we need to specify a start date and potentially an end date.

This already sound complex.

Luckily it is not. The **Pandas Datareader** will do all the hard work for us.

You only need to specify a ticker and start date.

"A stock symbol is a unique series of letters assigned to a security for trading purposes. Symbols are just a shorthand way of describing a company's stock, so there is no significant difference between those that have three letters and those that have four or five. Stock symbols are also

*known as ticker symbols." – **Investopedia.org***

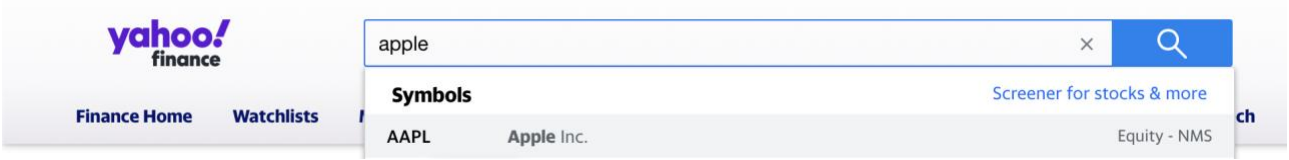
```
In [2]: start = dt.datetime(2010, 1, 1)

        aapl = pdr.get_data_yahoo("AAPL", start)
```

As you see, the start date is defined by the **dt.datetime(2010, 1, 1)**. This creates a **datetime** object representing the year 2020, month 1 (January), and day 1.

Then we call the **Pandas Datareader** with **pdr.get_data_yahoo(…)**, and specify the ticker **AAPL**, which is the ticker (stock symbol) for **Apple**.

You can find any available ticker on **Yahoo! Finance** by searching the name of the company you wish to have historic stock prices from.



The **Pandas Datareader** returns a **DataFrame** as we know it from last chapter.

```
In [3]: aapl.head()
```

Out[3]:

Date	High	Low	Open	Close	Volume	Adj Close
2010-01-04	7.660714	7.585000	7.622500	7.643214	493729600.0	6.583586
2010-01-05	7.699643	7.616071	7.664286	7.656429	601904800.0	6.594968
2010-01-06	7.686786	7.526786	7.656429	7.534643	552160000.0	6.490066
2010-01-07	7.571429	7.466071	7.562500	7.520714	477131200.0	6.478067
2010-01-08	7.571429	7.466429	7.510714	7.570714	447610800.0	6.521136

The good thing is, that it already has the **Date** as the index and it is a **DatetimeIndex**.

All ready to use.

How to specify an end date

This is probably just as you would guess.

```
In [4]: start = dt.datetime(2010, 1, 1)
        end = dt.datetime(2020, 1, 1)

        aapl = pdr.get_data_yahoo("AAPL", start, end)
```

In this case we want all the data from January 1st, **2010** until January 1st, **2020**.

We can investigate the start and end data as follows.

```
In [5]: aapl
```

Out[5]:

Date	High	Low	Open	Close	Volume	Adj Close
2010-01-04	7.660714	7.585000	7.622500	7.643214	493729600.0	6.583586
2010-01-05	7.699643	7.616071	7.664286	7.656429	601904800.0	6.594968
2010-01-06	7.686786	7.526786	7.656429	7.534643	552160000.0	6.490066
2010-01-07	7.571429	7.466071	7.562500	7.520714	477131200.0	6.478067
2010-01-08	7.571429	7.466429	7.510714	7.570714	447610800.0	6.521136
...	...	...	...	...	...	...
2019-12-24	71.222504	70.730003	71.172501	71.067497	48478800.0	70.353882
2019-12-26	72.495003	71.175003	71.205002	72.477501	93121200.0	71.749733
2019-12-27	73.492500	72.029999	72.779999	72.449997	146266000.0	71.722488
2019-12-30	73.172501	71.305000	72.364998	72.879997	144114400.0	72.148178
2019-12-31	73.419998	72.379997	72.482498	73.412498	100805600.0	72.675339

2516 rows × 6 columns

Notice, that the output of the **DataFrame** (**aapl)** is by default shortened with … in the middle, and not displaying the full set of 2,516 rows.

How to read data from multiple tickers at once

You can read historic stock prices from multiple tickers simultaneously.

```
In [6]: tickers = ['AAPL', 'MSFT', 'NFLX', 'TSLA']

        start = dt.datetime(2010, 1, 1)

        data = pdr.get_data_yahoo(tickers, start)
```

This is convenient for speed and to lower the number of requests you make to the **API**.

The data is structured as follows.

```
In [7]: data.head()
```

Out[7]:

Attributes	Adj Close				Close				High		...	Low
Symbols	AAPL	MSFT	NFLX	TSLA	AAPL	MSFT	NFLX	TSLA	AAPL	MSFT	...	NFLX
Date												
2010-01-04	6.583586	24.049969	7.640000	NaN	7.643214	30.950001	7.640000	NaN	7.660714	31.100000	...	7.565714
2010-01-05	6.594968	24.057743	7.358571	NaN	7.656429	30.959999	7.358571	NaN	7.699643	31.100000	...	7.258571
2010-01-06	6.490066	23.910097	7.617143	NaN	7.534643	30.770000	7.617143	NaN	7.686786	31.080000	...	7.197143
2010-01-07	6.478067	23.661432	7.485714	NaN	7.520714	30.450001	7.485714	NaN	7.571429	30.700001	...	7.462857
2010-01-08	6.521136	23.824627	7.614286	NaN	7.570714	30.660000	7.614286	NaN	7.571429	30.879999	...	7.465714

5 rows × 24 columns

Later we will learn how to access the data in a convenient way, when we work with data from multiple tickers.

Project – Normalize and Calculate Return of Portfolio

In this project we will learn how to normalize data and calculate the return of a portfolio using **Pandas DataFrames**.

*"In statistics and applications of statistics, normalization can have a range of meanings. In the simplest cases, normalization of ratings means adjusting values measured on different scales to a notionally common scale, often prior to averaging." – **Wikipedia.org***

As you will see in this project it is easy to normalize data with **DataFrames** and it is a great tool to master when working with financial data.

Step 1

We continue with the data we have prepared in this chapter.

The first step is to get the **Adj Close** values for all the tickers.

```
In [8]: data = data['Adj Close']
```

This simple statement takes all the **Adj Close** prices from the **DataFrame**.

```
In [9]: data.head()
```

Out[9]:

Symbols	AAPL	MSFT	NFLX	AMZN
Date				
2010-01-04	6.583586	24.049969	7.640000	133.899994
2010-01-05	6.594968	24.057743	7.358571	134.690002
2010-01-06	6.490066	23.910097	7.617143	132.250000
2010-01-07	6.478067	23.661432	7.485714	130.000000
2010-01-08	6.521136	23.824627	7.614286	133.520004

As you see the advantage of the two-layer column names, which the **Pandas Datareader** returns. It makes it easy to get the **Adj Close** from all tickers.

If you are unfamiliar with adjusted close (**Adj Close**):

*"The adjusted closing price amends a stock's closing price to reflect that stock's value after accounting for any corporate actions. It is often used when examining historical returns or doing a detailed analysis of past performance." – **Investopedia.org***

Step 2

Now we need to normalize the data.

```
In [10]: norm = data/data.iloc[0]
```

This looks too easy, right?

What happens there? Well, it takes the entries of the first row and divides it with all the rows.

The result?

```
In [11]: norm.head()
```

Out[11]:

Symbols	AAPL	MSFT	NFLX	AMZN
Date				
2010-01-04	1.000000	1.000000	1.000000	1.000000
2010-01-05	1.001729	1.000323	0.963164	1.005900
2010-01-06	0.985795	0.994184	0.997008	0.987677
2010-01-07	0.983973	0.983845	0.979806	0.970874
2010-01-08	0.990514	0.990630	0.996634	0.997162

As you see, all the start prices are adjusted to 1. This is the normalization.

Then the relative change can be seen from day **2010-01-05** and forward.

Step 3

Let's assume that our portfolio consists of 25% of each ticker.

How can we model that?

```
In [12]: portfolio = [.25, .25, .25, .25]
```

Create a list with four 0.25 values.

```
In [13]: weights = (norm*portfolio)
```

Then we multiply that list to the normalized data.

```
In [14]: weights.head()
```

Out[14]:

Symbols	AAPL	MSFT	NFLX	AMZN
Date				
2010-01-04	0.250000	0.250000	0.250000	0.250000
2010-01-05	0.250432	0.250081	0.240791	0.251475
2010-01-06	0.246449	0.248546	0.249252	0.246919
2010-01-07	0.245993	0.245961	0.244951	0.242718
2010-01-08	0.247629	0.247658	0.249159	0.249291

Then we get a weighted distribution of our portfolio. Starting with 25% for each ticker.

Step 4

If we calculate the sum of that and add that to the **DataFrame**. Notice, that we sum **axis=1,** which means along the rows.

```
In [15]: weights['Total'] = (norm*portfolio).sum(axis=1)
```

This is convenient as we will see in a moment.

```
In [16]: weights.head()
```

Out[16]:

Symbols	AAPL	MSFT	NFLX	AMZN	Total
Date					
2010-01-04	0.250000	0.250000	0.250000	0.250000	1.000000
2010-01-05	0.250432	0.250081	0.240791	0.251475	0.992779
2010-01-06	0.246449	0.248546	0.249252	0.246919	0.991166
2010-01-07	0.245993	0.245961	0.244951	0.242718	0.979624
2010-01-08	0.247629	0.247658	0.249159	0.249291	0.993735

Then we have the sum of our weighted sum of how the portfolio evolves in the **Total** column.

Step 5

Let's assume we invested 100,000 USD. This can be modeled quite easy as you see here.

```
In [17]: (weights*100000).head()
```

Out[17]:

Symbols	AAPL	MSFT	NFLX	AMZN	Total
Date					
2010-01-04	25000.000000	25000.000000	25000.000000	25000.000000	100000.000000
2010-01-05	25043.221519	25008.081459	24079.094179	25147.499735	99277.896891
2010-01-06	24644.873750	24854.603140	24925.207092	24691.935405	99116.619387
2010-01-07	24599.312801	24596.115428	24495.137728	24271.845767	97962.411723
2010-01-08	24762.857451	24765.756663	24915.857588	24929.053465	99373.525168

Our portfolio looks like this in the end (**tail()** shows the last 5 rows by default).

```
In [18]: (weights*100000).tail()
```

Out[18]:

Symbols	AAPL	MSFT	NFLX	AMZN	Total
Date					
2021-03-05	461070.923126	240748.760222	1.689758e+06	560205.395390	2.951783e+06
2021-03-08	441856.479208	236372.448174	1.614300e+06	551148.260965	2.843677e+06
2021-03-09	459817.799661	243014.867822	1.657199e+06	571854.039819	2.931886e+06
2021-03-10	455602.798046	241601.143935	1.650982e+06	570881.260634	2.919067e+06
2021-03-11	462172.150204	247957.706769	1.685275e+06	582135.934279	2.977541e+06

We have that **2021-03-11** we will have 2,977,541 USD in total.

This value can also be obtained directly as follows.

```
In [19]: (weights['Total']*100000).iloc[-1]
Out[19]: 2977540.753725041
```

Hence, this shows that if we invested 100,000 USD the opening day of 2010 in a portfolio consisting of 25% of each ticker, then we would have 2,977,540 USD on March 11th, 2021.

Summary

In this chapter we learned how to read historic stock prices directly from the **Yahoo! Finance API** using **Pandas Datareader**. How to set the start and end date. Also, how to read multiple tickers at the same time.

In the project we learned how to normalize data to calculate the return of a portfolio.

02 – Visualize DataFrames

In this chapter we will learn how to visualize historic stock prices on a chart using **Matplotlib**, as well as values calculated and added to the DataFrames.

*"Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python." – **matplotlib.org***

Learning objectives

- Learn how to make a chart of historic stock prices.
- Different modes to visualize charts in Jupyer Notebook.
- Have multiple charts on the same axis.
- Have multiple axis in one figure.

What you need to know about Matplotlib

Matplotlib is an easy to use library to create charts from **Pandas DataFrames** and **Series**.

With a few lines of code, you are up and running.

There are two different ways to use **Matplotlib**, which often make users more confused than good it.

- The object-oriented way, and
- The functional way.

By experience, I found that it is easier to understand the object-oriented way and it leads to less confusion.

It is true, that it needs a bit more code than with the functional way.

But used right, you only need to add one extra line of code and pass one argument.

This gives you more flexibility and less confusion.

Therefore, this **eBook** will only use the object-oriented way with **Matplotlib**.

Let's get started and see how easy it is.

Your first chart with Matplotlib

We need to import **Matplotlib** and set the mode.

```
In [1]: import pandas as pd
        import matplotlib.pyplot as plt
        %matplotlib notebook
```

The first line (**import pandas as pd**) is familiar and imports the **Pandas** library.

Then the second line imports the **Matplotlib** library, where we choose the **pyplot** module and want access to with the shorthand **plt**.

Finally, the **%matplotlib notebook** needs some explanation.

This sets the way **Jupyter Notebook** renders the figures from **Matplotlib**.

Many people use the **%matplotlib inline**, which for most cases is fine. The **inline** renders the figure once and inserts a static image into the notebook when the cell is executed.

While the **%matplotlib notebook** makes the figures interactive. In this case you can zoom in on details of the chart.

There are other interactive options as well, but they need to be installed separately at the time of writing.

In this eBook we will use the **notebook** mode, but feel free to use the **inline** mode.

For this demonstration purpose of **Matplotlib**, we will use the data from the **CSV** file from first chapter. This ensures you will get the same results as presented here.

```
In [2]: csv_file = "https://raw.githubusercontent.com/LearnPythonWithRune/PythonForFinancialAnalysis/
        aapl = pd.read_csv(csv_file, index_col="Date", parse_dates=True)
```

It is always good habit to check the data is as expected to avoid surprises later.

```
In [3]: aapl.head()
```

Out[3]:

Date	Open	High	Low	Close	Adj Close	Volume
2016-03-09	25.327499	25.395000	25.067499	25.280001	23.513155	108806800
2016-03-10	25.352501	25.559999	25.037500	25.292500	23.524775	134054400
2016-03-11	25.559999	25.570000	25.375000	25.565001	23.778234	109632800
2016-03-14	25.477501	25.727501	25.445000	25.629999	23.838688	100304400
2016-03-15	25.990000	26.295000	25.962500	26.145000	24.317701	160270800

This looks as expected.

Now to the exciting part. The visualization.

```
In [4]: fig, ax = plt.subplots()
        aapl['Close'].plot(ax=ax)
```

This is the object-oriented way to use **Matplotlib**. You might be familiar with rendering a chart directly by one line of code.

In this case, the advantage is small, but still it is there. If you render another chart from the same dataset, then this will be added to the existing figure. That leads to a lot of confusion. Here, we avoid that.

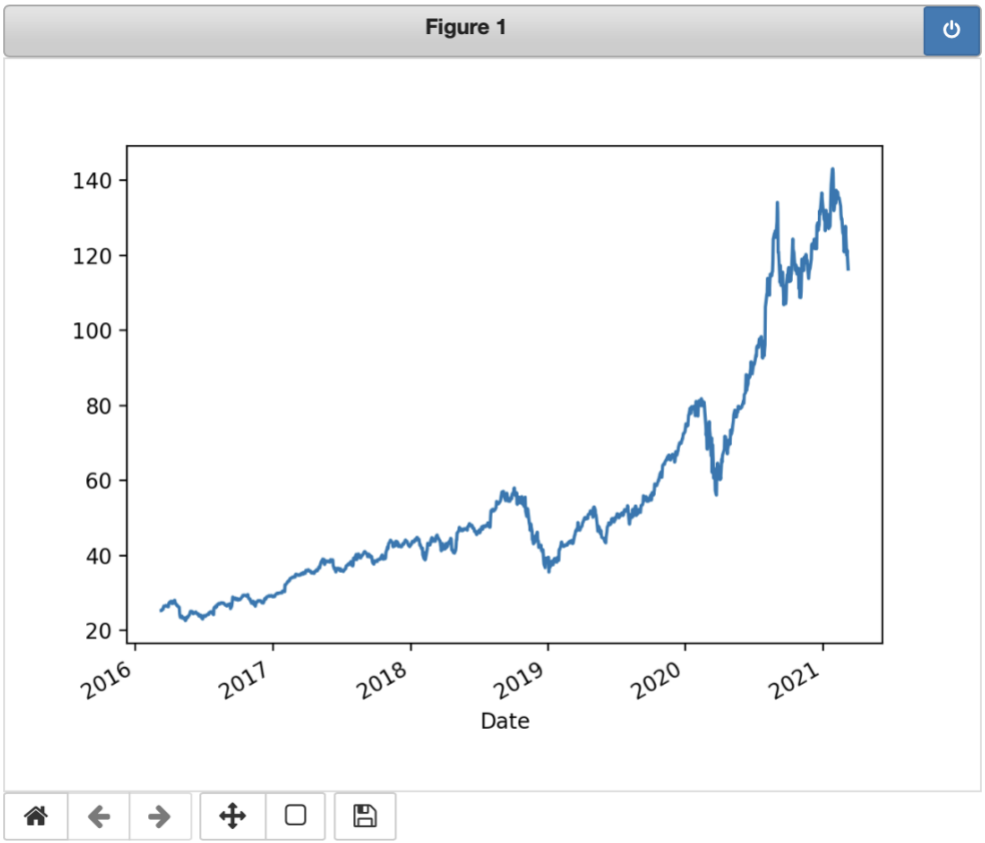
A few notes on the above code.

The **plt.subplots()** returns a figure **fig** and axis **ax.** We need to parse the axis **ax** to the plot method of the **DataFrame** (the second line **.plot(ax=ax)**).

The **figure** is the entire image of one or more charts (axes). The axis is the where the chart is drawn. You can have multiple axes in one figure as we will see later.

This is all you need to know.

The **DataFrame** takes in this case the only the data from the column **Close** and renders it.




```
Out[4]: <AxesSubplot:xlabel='Date'>
```

This shows the interactive figure that the cell outputs.

You can save the figure, you can zoom in on areas, and more.

The zoom function is given by the ▢ button. Press it and mark the area you want to zoom in on.

To get back to the original figure, simply press 🏠.

You can move around the chart by pressing the ✣ and navigate in your history of changes by using the arrows back and forward.

This makes it easy to explore the chart interactively.

Multi-line chart

Let's try to add the simple moving average to our chart. This requires that we calculate them and add them to our **DataFrame**.

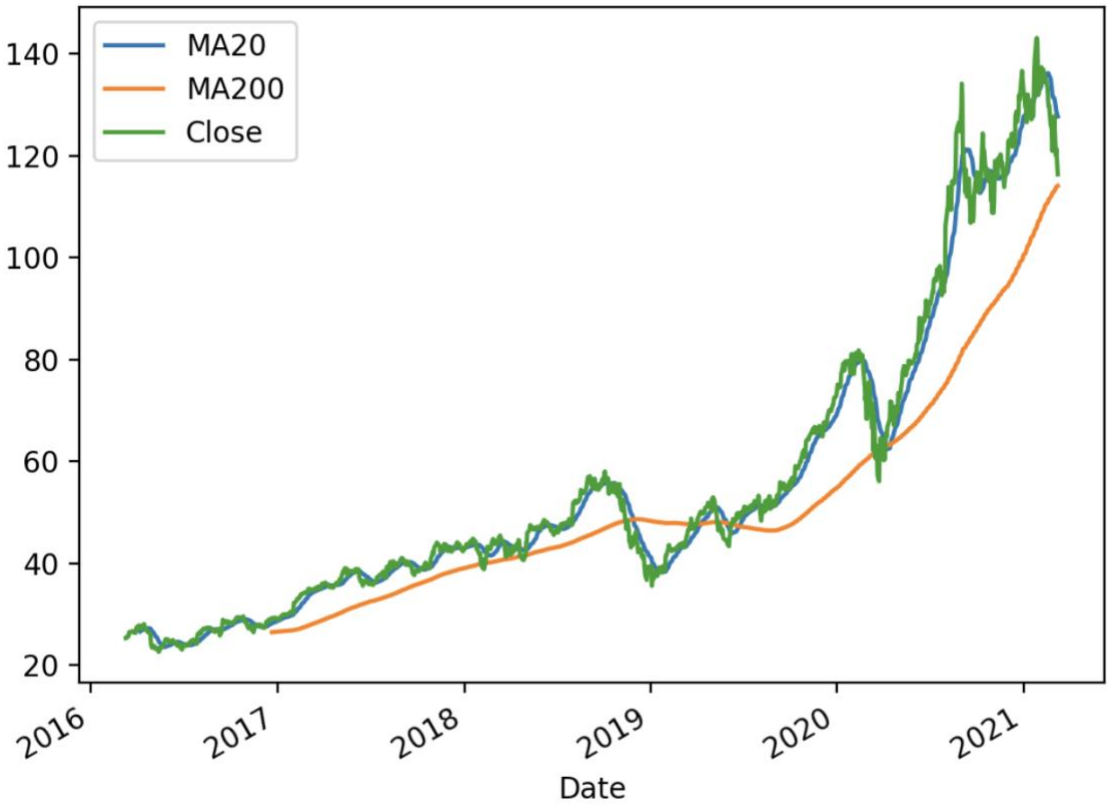
```
In [5]: aapl['MA20'] = aapl['Close'].rolling(20).mean()
        aapl['MA200'] = aapl['Close'].rolling(200).mean()
```

Then it can be visualized as follows.

```
In [6]: fig, ax = plt.subplots()
        aapl[['MA20', 'MA200', 'Close']].plot(ax=ax)
```

Notice, that the **DataFrame** (**aapl)** can take a list of columns and that list will be the columns it makes available.

This will give a chart as follows.



In your **Jupyter Notebook** it will be shown with the interactive setup. Here we only see the chart.

Notice, that the legend in the upper left corners shows the color coding of the lines on the chart.

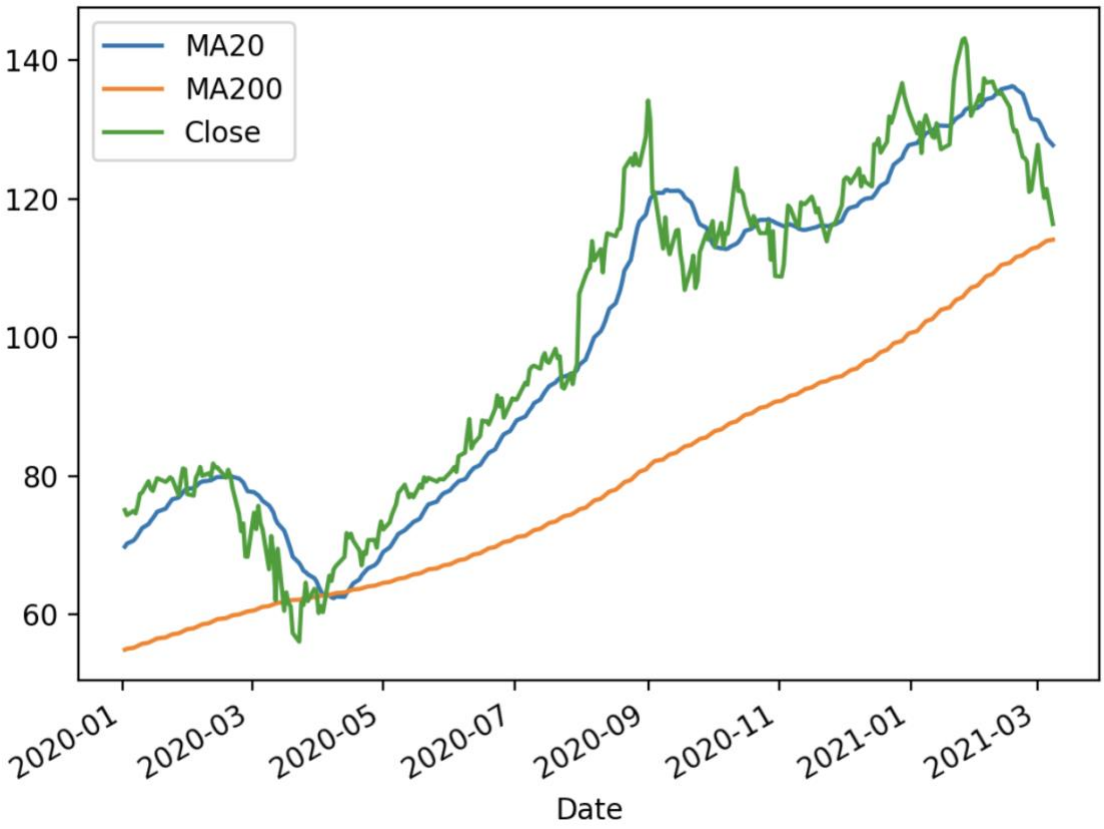
Limit the time period of the chart

While you can interactively zoom into the part of the chart that has your interest, you can also focus a specific time interval on your rendered chart.

Let's say, you are only interested in the historic stock prices from the beginning of 2020 and forward.

```
In [7]: fig, ax = plt.subplots()
        aapl[['MA20', 'MA200', 'Close']].loc['2020':].plot(ax=ax)
```

Notice how easy it is to do. Simply by adding a **.loc['2020':]** you tell the **DataFrame** to take the data from the beginning of 2020 and forward.



Multiple charts in one figure

Let's try to have multiple charts (or axes) in one figure.

```
In [8]: fig, ax = plt.subplots(2, 2)
        aapl['Open'].loc['2020':].plot(ax=ax[0, 0], c='r')
        aapl['Close'].loc['2020':].plot(ax=ax[0, 1], c='g')
        aapl['High'].loc['2020':].plot(ax=ax[1, 0], c='c')
        aapl['Low'].loc['2020':].plot(ax=ax[1, 1], c='y')
        ax[0, 0].legend()
        ax[0, 1].legend()
        ax[1, 0].legend()
        ax[1, 1].legend()
        plt.tight_layout()
```

Now, that was a lot of code.

First look at the first line. There is a slight difference **plt.subplots(2, 2).**

This will return a figure with 2-by-2 axes.

As you see on the following 4 lines, you access an axis with the variable **ax** indexed as a double list. Hence, the first axis is given by **ax[0, 0]**, the second **ax[0, 1]** and so forth.

The trained eye will also notice we added an argument **c='r'** or similar in each plot. This will set the color of the line.

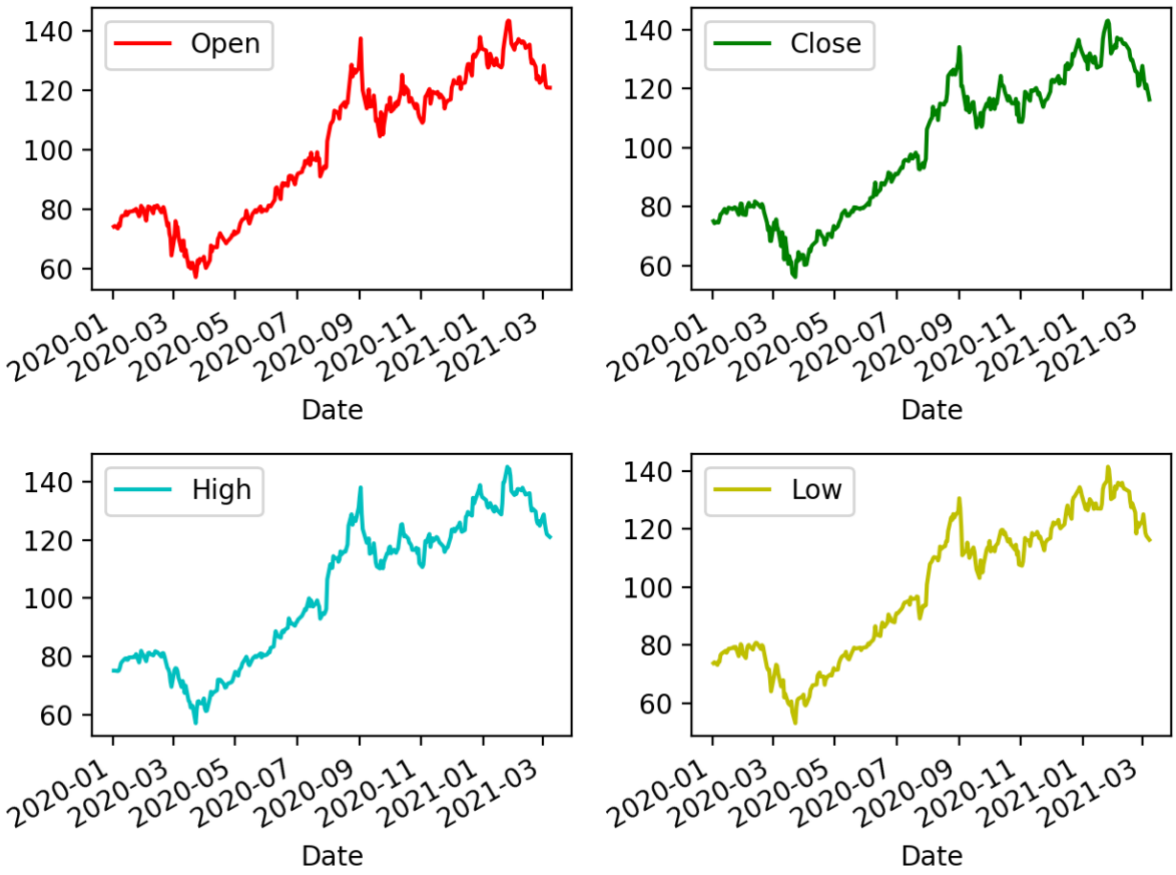
You can find color names in the **Matplotlib documentation**.

The color coding is optional and just used for demonstration purposes here.

Then we have some lines of the form **ax[0, 0].legend()**, which again are optional. They only tell the axis to have the **legend** visible.

You can also call **ax[0, 0].set_title("My title")** to set a title on your axis.

Finally, we call **plt.tight_layour().** All that does is to render the axes (charts) nice to avoid overlap of the text of the axes. You can try to comment it out and see the difference.



Project – Compare two investments

In this project we will compare two investments and visualize it. We will compare an investment in Apple to an investment in Microsoft in 2020. That is, if you invested money in Apple on the first trading day in 2020 and sold it all on the first trading day in 2021. How would that compare if you had invested it in Microsoft instead.

Well, let's figure it out together.

Step 1

First, we need to read the historic stock prices of the two investments.

We can use the **Pandas Datareader** to get the data.

First, we need to import the library and the **datetime** library.

```
In [9]: import pandas_datareader as pdr
        import datetime as dt
```

Then read the data from the **API**.

```
In [10]: tickers = ['AAPL', 'MSFT']

         start = dt.datetime(2020, 1, 1)
         end = dt.datetime(2021, 1, 1)

         data = pdr.get_data_yahoo(tickers, start, end)
```

Step 2

We will compare the investments using the **Adj Close** (adjusted close) price, as it includes dividend payout and other corporate adjustments.

```
In [11]: data = data['Adj Close']
```

Let's see if the data looks at we expect.

```
In [12]: data.head()
```

Out[12]:

Symbols	AAPL	MSFT
Date		
2020-01-02	74.333511	158.571075
2020-01-03	73.610840	156.596588
2020-01-06	74.197395	157.001373
2020-01-07	73.848442	155.569855
2020-01-08	75.036385	158.047836

Step 3

To make an easy comparison we normalize the data.

```
In [13]: norm = data/data.iloc[0]
```

And check that all is as it should be.

```
In [14]: norm.head()
```

Out[14]:

Symbols	AAPL	MSFT
Date		
2020-01-02	1.000000	1.000000
2020-01-03	0.990278	0.987548
2020-01-06	0.998169	0.990101
2020-01-07	0.993474	0.981073
2020-01-08	1.009456	0.996700

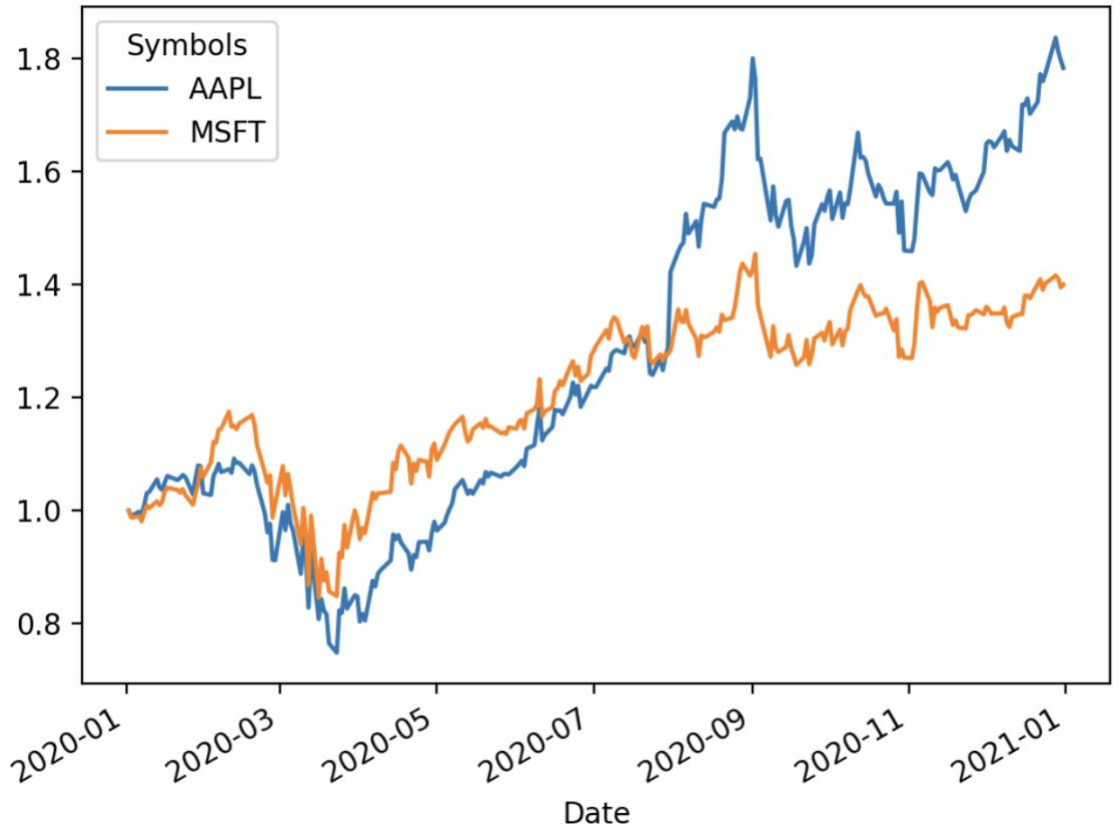
Step 4

Now to the fun stuff.

Visualization.

```
In [15]: fig, ax = plt.subplots()
         norm.plot(ax=ax)
```

Now let's take a minute and see what we can conclude from the figure.



Because we have normalized the data both Apple and Microsoft start at the same point in start of 2020.

This is convenient, because if we invested either in Apple or Microsoft, it will show the relative difference on how our investment would evolve.

Said differently, for each dollar invested in Apple and for each dollar invested in Microsoft, then we can see how it evolves over time.

Here at this stage, we are only interested in the return of our investment.

Hence, we only look at what we get at the end.

Later we will learn about **volatility** and **maximum drawdown**, which are important factors when evaluating the risk of potential investments.

But for now, let's investigate the return.

Step 5

Let's calculate the return and visualize it.

```
In [16]: aapl_rtn = norm['AAPL'].iloc[-1] - 1
         msft_rtn = norm['MSFT'].iloc[-1] - 1
```

As we have normalized data, we can get the return from the last day. We subtract 1 to get the percentage growth of the investment.

```
In [17]: aapl_rtn, msft_rtn
Out[17]: (0.7823995832102539, 0.3994286436811605)
```

Hence, if we invested in Apple, we would have gained 78.24% in 2020.

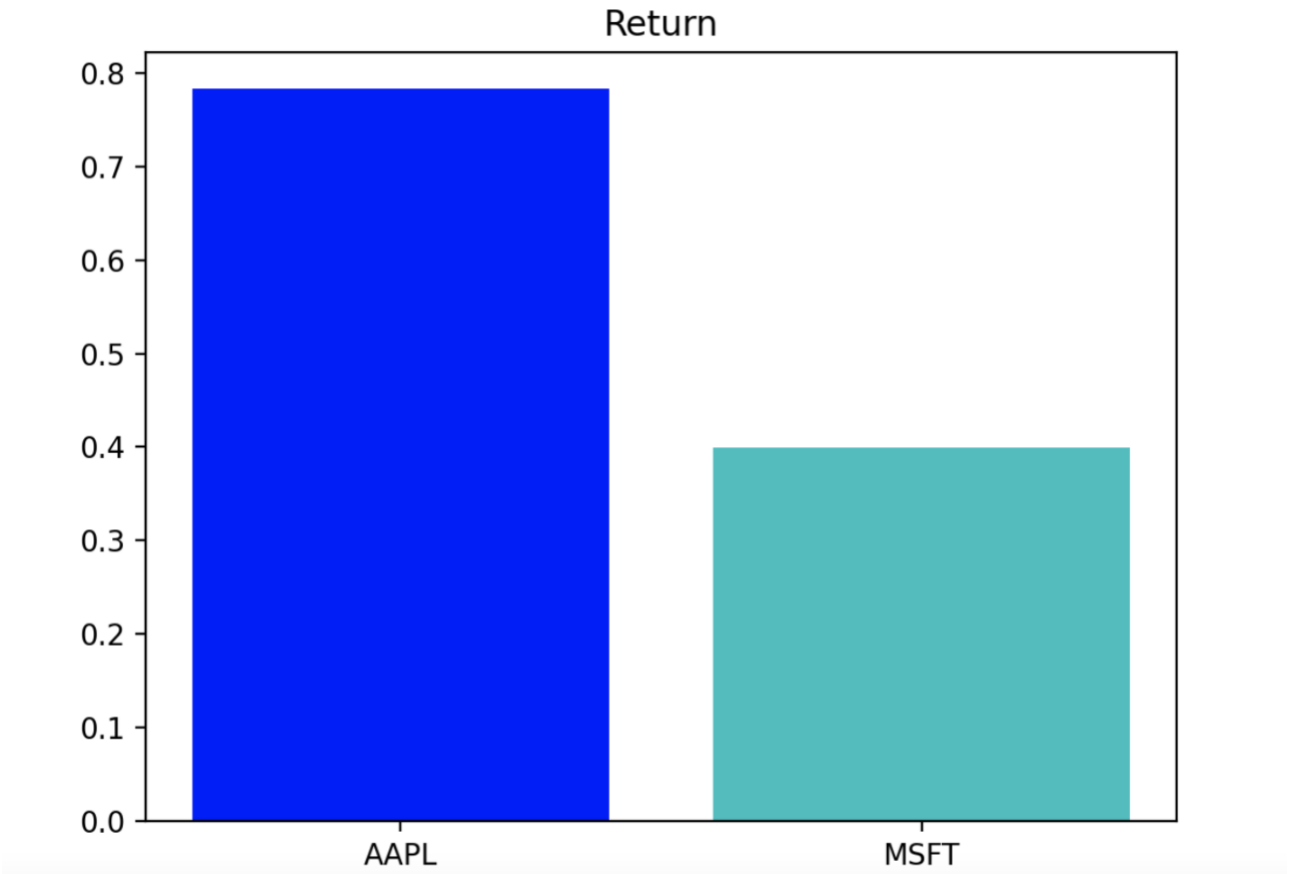
While, if we invested in Microsoft, we would have gained 39.94% in 2020.

Both good investments.

This can be visualized on a bar-chart as follows.

```
In [18]: fig, ax = plt.subplots()
         ax.bar(['AAPL', 'MSFT'], [aapl_rtn, msft_rtn], color=['b', 'c'])
         ax.set_title("Return")
```

Which results in this.



Summary

In this chapter we have learned how to use **Matplotlib** to visualize historic stock prices. How to use it with multiple lines in one chart or multiple axes in one figure. Also, how to create bar charts, which we will use later in this **eBook**.

03 – Buy and Hold

Warren Buffett advice to both stock market novices and rich investors to buy and hold a low-cost fund that tracks the S&P 500 index.

The argument is that, in the long run, you will outperform the majority of most money managers.

The S&P 500 has given an average return of 10-11% per year since it was created in 1926. And an average of 8% per year from 1956 to 2018 (**source**).

So why bother do anything else?

We will explore that in this chapter.

Learning objectives

- Evaluate the Buy-and-hold strategy Warren Buffett proposes
- How to calculate the **CAGR** – the Compound Annual Growth Rate – and what it is
- How to calculate the **Maximum Drawdown**
- Learn about the "*Lost decade*"

How to approach this

It is very tempting, right?

One of the smartest, if not the smartest, investment gurus' advices you to invest in a low-cost fund that tracks the S&P 500 index. It will give you an expected 8% of return per year in the long run.

The long run.

What is the long run?

Well, we will figure that out in this chapter.

This chapter will also give the foundation of how to evaluate an investment strategy.

But let's get started and learn along the way.

Read the data

The ticker for the S&P 500 is **^GSPC.** This is the index. We will later explore a low-cost fund that tracks it. But for now, we need to explore the nature of Buy-and-hold, and what the long run means.

First step first. Import the needed libraries.

In [1]:
```
import pandas_datareader as pdr
import datetime as dt
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib notebook
```

Nothing new here.

Read the data from the API.

```
In [2]: start = dt.datetime(1999, 1, 1)

        sp500 = pdr.get_data_yahoo("^GSPC", start)
```

We will read from beginning of 1999, and you will know why later in this chapter.

```
In [3]: sp500.head()
```

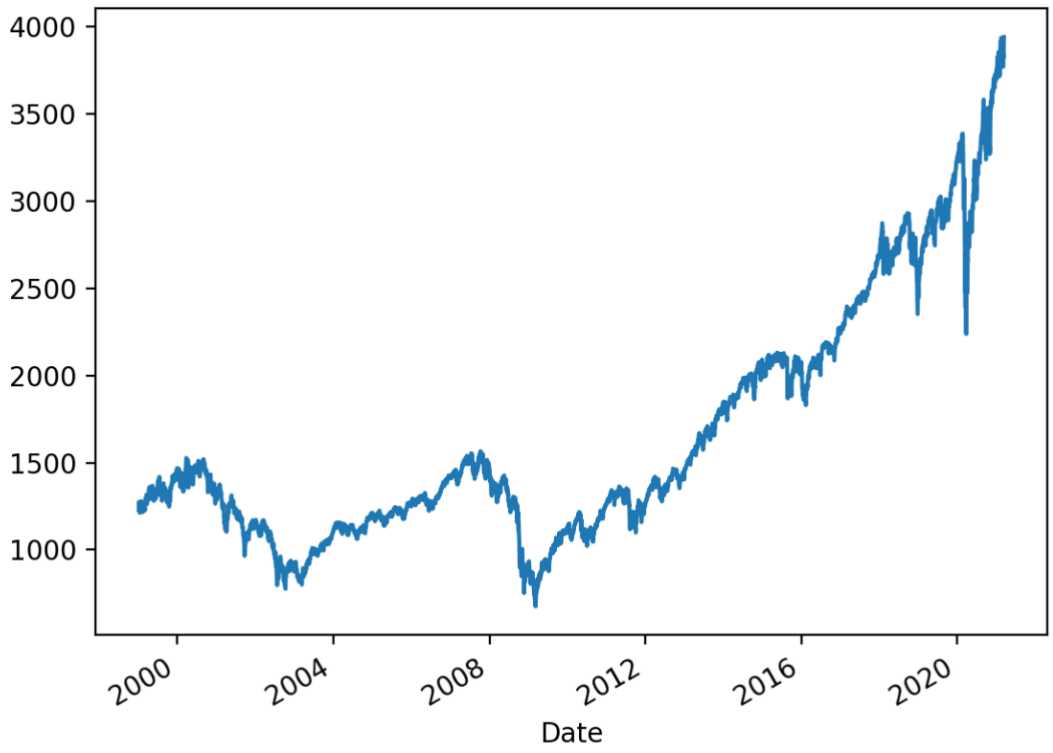
Out[3]:

Date	High	Low	Open	Close	Volume	Adj Close
1999-01-04	1248.810059	1219.099976	1229.229980	1228.099976	877000000	1228.099976
1999-01-05	1246.109985	1228.099976	1228.099976	1244.780029	775000000	1244.780029
1999-01-06	1272.500000	1244.780029	1244.780029	1272.339966	986900000	1272.339966
1999-01-07	1272.339966	1257.680054	1272.339966	1269.729980	863000000	1269.729980
1999-01-08	1278.239990	1261.819946	1269.729980	1275.089966	937800000	1275.089966

This gives data from 1999 and forward to current day.

And let's visualize it. It makes it easier to get an impression of the data that way.

```
In [4]: fig, ax = plt.subplots()
        sp500['Close'].plot(ax=ax)
```



As you see. This looks good. Nice growth in the second half.

Second half?

What about the first?

The lost decade

The years 1999 to 2009 are considered the *"lost decade"* of S&P 500.

Why?

```
In [5]: sp500['2009'].head()
```

Out[5]:

Date	High	Low	Open	Close	Volume	Adj Close
2009-01-02	934.729980	899.349976	902.989990	931.799988	4048270000	931.799988
2009-01-05	936.630005	919.530029	929.169983	927.450012	5413910000	927.450012
2009-01-06	943.849976	927.280029	931.169983	934.700012	5392620000	934.700012
2009-01-07	927.450012	902.369995	927.450012	906.650024	4704940000	906.650024
2009-01-08	910.000000	896.809998	905.729980	909.729980	4991550000	909.729980

The index starts at 931.80 in 2009.

```
In [6]: sp500['1999'].head()
```

Out[6]:

Date	High	Low	Open	Close	Volume	Adj Close
1999-01-04	1248.810059	1219.099976	1229.229980	1228.099976	877000000	1228.099976
1999-01-05	1246.109985	1228.099976	1228.099976	1244.780029	775000000	1244.780029
1999-01-06	1272.500000	1244.780029	1244.780029	1272.339966	986900000	1272.339966
1999-01-07	1272.339966	1257.680054	1272.339966	1269.729980	863000000	1269.729980
1999-01-08	1278.239990	1261.819946	1269.729980	1275.089966	937800000	1275.089966

And in 1999 it started at 1228.10 USD.

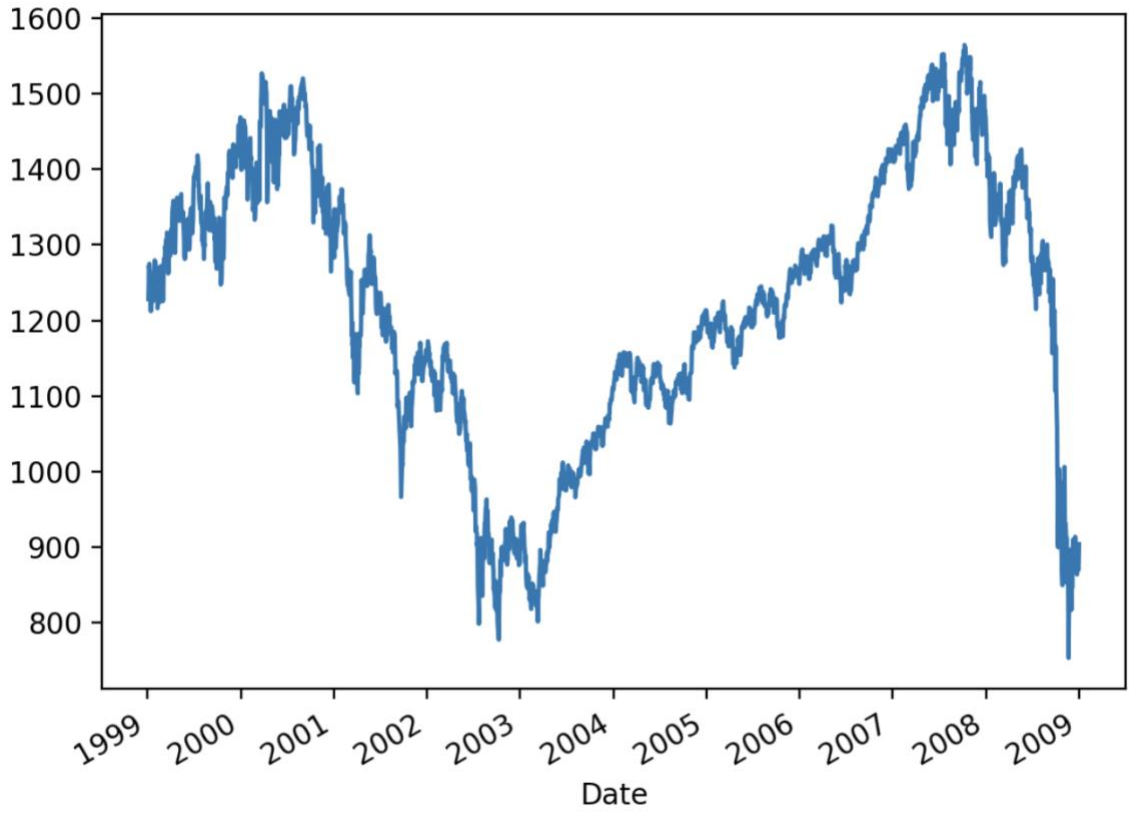
Wait a minute?

1228.10 in 1999 and 931.80 in 2009.

10 years and it goes down?

Let's see this.

```
In [7]: fig, ax = plt.subplots()
        sp500['Close'].loc[:'2009-01-01'].plot()
```



Yup. It doesn't look good.

10 years with no growth.

What is the key learning. The long run means at least more than 10 years.

Hence, if you have decades before you need your 8% return per year, then an investment in a fund tracking the S&P 500 is a great idea.

Otherwise, you might end up after 10 years and see your money being less worth than the day you invested them.

The CAGR vs AAGR

Let's formalize a few measures when it comes to evaluating an investment.

We talked about the return in the last section.

What we really should be talking about is the **Compound Annual Growth Rate (CAGR)**.

CAGR measures the yearly return of an investment.

The formula.

$$\left(\frac{end\ value}{start\ value}\right)^{1/n} - 1$$

Where n is the number of years.

The CAGR calculates the value you are interested in.

Some financial statements use the **Average Annual Growth Rate (AAGR)**. This is misleading.

$$\frac{Gr_0 + Gr_1 + \ldots + Gr_{n-1}}{n}$$

This is the sum of the growth rates over the period of *n* years.

Let's have an example.

Assume you have 100 USD and you invest them.

Over four years your return is: 20%, -50%, 50%, and 30%

Let's calculate the return after 4 years.

$$100 \times (1 + 0.2) \times (1 - 0.5) \times (1 + 0.5) \times (1 + 0.3) = 117$$

Hence, the 100 USD will be 117 USD after 4 years.

Now, let's calculate the CAGR and AAGR.

The CAGR.

$$\left(\frac{end\ value}{start\ value}\right)^{1/n} - 1 = \left(\frac{117}{110}\right)^{1/4} - 1 = 0.0400314334861216$$

The AAGR

$$\frac{Gr_0 + Gr_1 + \ldots + Gr_{n-1}}{n} = \frac{0.2 - 0.5 + 0.5 + 0.3}{4} = 0.125$$

So why care about CAGR.

CAGR says we get an annual return of 4% and the AAGR says 12.5%.

The AAGR looks better right?

But CAGR calculates the return we are looking for.

$$100 \times (1 + 0.04) \times (1 + 0.04) \times (1 + 0.04) \times (1 + 0.04) = 117$$

Note *There is some rounding in the above calculations which are not accounted for.*

What does the AAGR tell us?

For me? It is only to make the percentage look better than it is.

Calculate the CAGR for S&P 500

Now we know how to calculate the CAGR on paper, how do we do it with **Pandas DataFrames**.

Let's do that for the period of 1999 to 2009.

We will take the 10 years of data beginning from 1999 and ending last trading day in 2008.

```
In [8]: data = sp500['Close'].loc['1999':'2008']
```

The total return can be calculated as follows.

```
In [9]: total_return = data.iloc[-1]/data.iloc[0]
```

Giving the following.

```
In [10]: total_return
Out[10]: 0.7354857242538836
```

This means that we lost money. We lost 1 – 0.735 = 0.265 or 26.5%.

The CAGR can be calculated as follows.

```
In [11]: (data.iloc[-1]/data.iloc[0])**(1/10) - 1
Out[11]: -0.030255277567665995
```

That is a loss of 3% per year.

If I in 1999 invested all my money in a low-cost fund tracking the S&P 500, I would lose 3% per year or 26.5% in total loss over the next 10 years.

Maximum Drawdown

*"A maximum drawdown (MDD) is the maximum observed loss from a peak to a trough of a portfolio, before a new peak is attained. Maximum drawdown is an indicator of downside risk over a specified time period." – **Investopedia.org***

In this first stage we will calculate the **maximum drawdown** over the full period of 10 years. Later we will be calculated it as a rolling value of a period of 1 year.

For the specific study here, it will not make any difference.

How do we get the maximum and minimum value cumulated over the period?

That is actually what we are looking for.

```
In [12]: rolling_max = data.cummax()
         daily_drawdown = data/rolling_max - 1
```

The **rolling_max** will at all times keep the maximum value seen so far.

The **daily_drawdown** keeps the current maximum **drawdown**.

```
In [13]: max_drawdown = daily_drawdown.cummin().iloc[-1]
```

```
In [14]: max_drawdown
Out[14]: -0.5192537515864695
```

Remember, the **DataFrame** data keeps the 10 years we are investigating.

Hence, the maximum drawdown is 51.9%.

When investing you are looking for increasing the **CAGR**, while keeping the **maximum drawdown** low.

Why? Because the **maximum drawdown** tells you how much money you can lose, based on historic values.

In the next chapter we will learn about **volatility** of a stock.

The full picture will be to maximize the return (**CAGR**) and minimize the **maximum drawdown** and **volatility** of the stock.

Project – Calculate the CAGR and Maximum Drawdown of S&P 500 from 2010 to 2020

In this project we will make the same calculations for the 10-year period form 2010 to 2020.

Step 1

First, we need the data for that period.

```
In [15]: data = sp500['Close'].loc['2010':'2019']
```

Step 2

Then we calculate the **CAGR**.

```
In [16]: (data.iloc[-1]/data.iloc[0])**(1/10) - 1
Out[16]: 0.11047332749089067
```

That looks better. A 11% annual return.

Step 3

Finally, we calculate the **maximum drawdown**.

```
In [17]: rolling_max = data.cummax()
         daily_drawdown = data/rolling_max - 1
         max_drawdown = daily_drawdown.cummin().iloc[-1]
         max_drawdown
Out[17]: -0.19778210435681998
```

That is a **maximum drawdown** on 19.8%.

Summary

In this chapter we have started our journey to learn more about investing. We have explored the advice of Buy-and-Hold a low-cost fund tracking the S&P 500. This led us to discover that 10 years might pass without any earnings, like the *"lost decade"* 1999 to 2009.

We have formalized the return calculations **CAGR**, which is a primary interest when investing. To learn about the risk side of an investment, we introduced the concept of **maximum drawdown**. The **maximum drawdown** tells us how much money we might lose.

In the next chapter we will learn about **volatility** of a stock, which is the second risk measure we use when evaluating an investment strategy.

04 – Volatility

In this chapter we will learn about the **volatility** of a stock.

"Volatility is a statistical measure of the dispersion of returns for a given security or market index. In most cases, the higher the volatility, the riskier the security. Volatility is often measured as either the standard deviation or variance between returns from that same security or market index." – ***Investopedia.org***

A volatile stock is considered risky. It is common to assume that **volatility** is a good measure for risk in investment.

Learning objectives

- Different measures of **volatility**.
- Learn about **log return** – needed for the calculations.
- How to calculate the **volatility** of an investment.
- A way to visualize the **volatility** for better understanding.

Volatility – not just one definition

If you read the quote from **Investopedia.org** above, you would already notice something.

"*Volatility is often measured as either the standard deviation or variance between returns…"*

Already hinting at two different ways to calculate the **volatility**.

Also, if you continue reading on **Investopedia.org**, you will notice that they talk about how the **Average True Range (ATR)** can be used as a measure for volatility (see the video).

Further down, they talk about the **Beta** and more.

Luckily, one measure of **volatility** is enough for our purpose.

This is just to make you aware of the different ways to calculate **volatility**.

Log returns

What about this log?

Why bother about the log?

What is the log?

All good questions.

The answer. They make our calculations easier and faster.

It all boils down to two things about the log returns.

$$\log(a \times b) = \log(a) + \log(b)$$

And

$$a \times b = \exp(log(a \times b)) = \exp(\log(a) + \log(b))$$

That is the magic.

It can save us from expensive multiplications. Don't worry if it still confuses you.

Let's take an example before we jump to **Jupyter Notebook**.

Consider the following investment, which returns 20%, 15%, 10%, and 30%.

This can be calculated as a total return of.

$$(1 + 0.2) \times (1 + 0.15) \times (1 + 0.1) \times (1 + 0.3) = 1.9734$$

That is a return of 97.34%.

Now let's calculate it differently.

$$\log(1 + 0.2) + \log(1 + 0.15) + \log(1 + 0.1) + \log(1 + 0.3) = 0.6797579434409292$$

And

$$\exp(0.6797579434409292) = 1.9734$$

It is correct.

But isn't it just more complex?

Yes, you are right. At first sight it looks like just adding complexities.

Stay with me here, because it becomes handy later.

Let's first jump to **Jupyter Notebook** and see how it can be used.

Log returns calculations with Pandas and NumPy

NumPy you ask?

NumPy is the fundamental package for scientific computing with **Python**. Actually, **Pandas** is built upon **NumPy**.

The reason we need it, is to access some methods in it, which are not available directly on the **DataFrame** objects.

Luckily, **Pandas DataFrames** integrate well with **NumPy** as you will see in a moment.

```
In [1]: import pandas_datareader as pdr
        import datetime as dt
        import pandas as pd
        import numpy as np
```

Importing the libraries. The new one is **NumPy**.

```
In [2]: start = dt.datetime(1999, 1, 1)
        end = dt.datetime(2008, 12, 31)

        data = pdr.get_data_yahoo("^GSPC", start, end)
```

Reading the data from the *"lost decade"*.

Now to the interesting part. We need to calculate the **log returns** of the prices.

What we use is the daily change. This can be accessed by taking the current price divided by the previous price.

If we take the logarithm of that, we get the **daily log return** (called the **Log returns** in below).

```
In [3]: data['Log returns'] = np.log(data['Adj Close']/data['Adj Close'].shift())
```

The **shift()** shifts the data one place forward. Hence, the rows are shifted one day forward.

Here the **Adj Close** price from the current day is divided from the previous day. This is done for all days.

Then we have taken the **np.log**, which applies the logarithm on all rows.

Now let's see how this connects to the return as we know how to calculate.

```
In [4]: data['Adj Close'].iloc[-1]/data['Adj Close'].iloc[0]
Out[4]: 0.7354857242538836
```

The sum of all the log returns is.

```
In [5]: data['Log returns'].sum()
Out[5]: -0.3072241487090497
```

If we apply the exponential function on that value we get the same return.

```
In [6]: np.exp(data['Log returns'].sum())
Out[6]: 0.7354857242538831
```

Wow. Check that out. We can add the daily log returns together and get the same return.

This still seems a bit counterintuitive to do.

Later when we shift our investment from one stock to another, it comes in handy to just add the daily log returns together to find the full return.

That makes it really powerful.

It makes it easy to calculate the return of investment strategies, which have different investments portfolios from day to day.

If it is not entirely clear now? Don't worry, we'll get there later and you will see the advantage of using log returns (daily log returns).

Normalized data and log returns

Let's clarify the mystery between **normalized data** and **log returns**.

We keep the data in a column called **Normalize**.

```
In [7]: data['Normalize'] = data['Adj Close']/data['Adj Close'].iloc[0]
```

Then we calculate the exponential value of the cumulated sum of the log return.

```
In [8]: data['Exp sum'] = data['Log returns'].cumsum().apply(np.exp)
```

Okay, we introduced two new things here.

First, the **cumsum()** calculates a cumulated sum of the column it is applied on. Here, we apply it to the log returns. Then we apply **apply(np.exp)** on that column. As we know **np.exp** takes the exponential function of the value.

```
In [9]: data[['Normalize', 'Exp sum']].tail()
```

Out[9]:

Date	Normalize	Exp sum
2008-12-24	0.706905	0.706905
2008-12-26	0.710691	0.710691
2008-12-29	0.707939	0.707939
2008-12-30	0.725218	0.725218
2008-12-31	0.735486	0.735486

As we see. This is exactly the same value.

Surprised?

Not if you think about it.

In the last one we actually specifically calculated the last row. Investigate that and see for yourself.

All we done here is to do it for each day in a cumulated way.

Volatility calculation

This is all we have been waiting for, right?

```
In [10]: volatility = data['Log returns'].std()*(252**0.5)

In [11]: volatility
Out[11]: 0.212783217027898
```

What?

The annualized standard deviation is used as **volatility** measure.

Let's break it down.

The **data['Log returns'].std()** returns the daily standard deviation.

To annualize it we need to multiply by the square root of the number of days.

There are calculated on average 252 trading days. Hence**, 252**0.5**, is the square root of 252 and we get the annualized standard deviation, which we use as **volatility** measure.

How to interpret the result?

It is a comparison. The lower the better.

As you will see later, we will use the S&P 500 index as our basis for the **return**, **maximum drawdown** and **volatility**. If figures are worse than the S&P 500 in all aspects, we gain nothing.

The goal is to maximize the **return (CAGR)**, and minimize the **maximum drawdown** and **volatility**.

Project – Visualize the volatility of the S&P 500 index

In this project we will visualize the **volatility** of the S&P 500 index. This will help us understand what **volatility** means.

Step 1

We need to import **Matplotlib** to visualize it.

```
In [12]: import matplotlib.pyplot as plt
         %matplotlib notebook
```

Step 2

Convert the **volatility** into a string.

```
In [13]: str_vol = str(round(volatility, 3)*100)
```

This is just needed to make the diagram nice with titles.

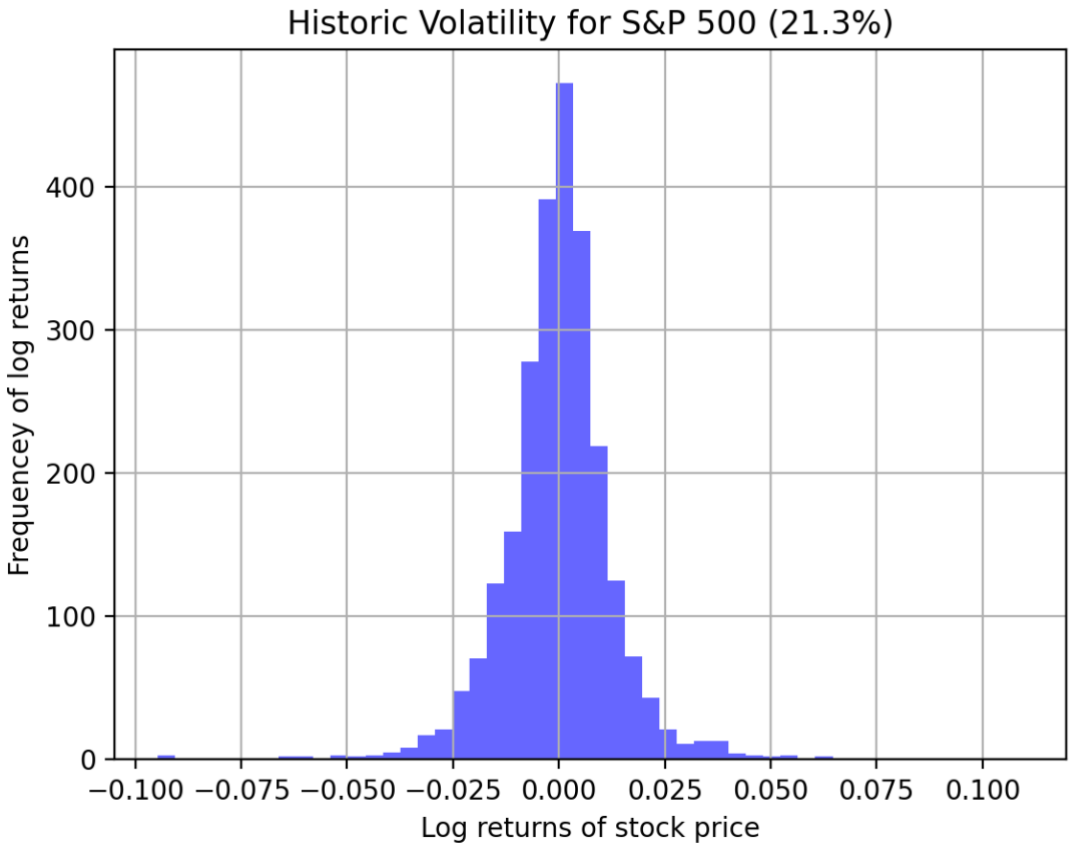
Step 3

The actual plot.

```
In [14]: fig, ax = plt.subplots()
         data['Log returns'].hist(ax=ax, bins=50, alpha=0.6, color='b')
         ax.set_xlabel("Log returns of stock price")
         ax.set_ylabel("Frequencey of log returns")
         ax.set_title("Historic Volatility for S&P 500 (" + str_vol +"%)")
```

We use the histogram to make it (**hist**) and use 50 **bins**, an **alpha** of 0.6 (this makes it a bit transparent, not strictly needed), and the **color** blue.

We also set labels (**set_xlabel** and **set_ylabel**) and titles (**set_title**).



The **volatility** says something about how these **bins** are spread and how close the majority of data is to 0.0.

Remember that we are looking at the daily log return here. The higher the daily log return, the more volatile it is. Hence, if we have a log of values that are far from 0.0, then it is volatile. If all values are close to 0.0, it is not as volatile.

Summary

In this chapter we learned about two important concepts. The **log return** and **volatility**. First, the **log return** is a great way to calculate return of. It will be handy, when we calculate returns of changing investments. Second, the **volatility** of an investment is a measure of risk.

We now have the tools to evaluate an investment strategy. The **return** (**CAGR**), which should be optimized, the **maximum drawdown** and **volatility**, which should be minimized.

05 – Correlation

In this chapter we will explore an important concept when talking about investment strategies.

*"Correlation, in the finance and investment industries, is a statistic that measures the degree to which two securities move in relation to each other. Correlations are used in advanced portfolio management, computed as the correlation coefficient, which has a value that must fall between -1.0 and +1.0." – **Investopedia.org***

Our goal is to find ways to minimize the **maximum drawdown** and **volatility**.

A great way is to find investments with negative correlation.

Why?

Because when the price of one goes down, the other is expected to go up.

Learning objectives

- Understand negative and positive **correlation**.
- How **negative correlation** can minimize the **maximum drawdown** and **volatility**.
- How to calculate the **correlation**.
- Visualize negative correlated investments.

What correlation tells us

The correlation tells us how stock prices move in relation to each other. That is, given a stock A and a stock B, how are the prices related to each other.

The correlation between two stock prices is a value between -1.0 and 1.0 (both inclusive).

A perfect positive correlation means that the correlation coefficient is exactly 1. This implies that as the price of stock A goes up, the price of stock B moves up in lockstep. Similarly, if the price of stock A goes down, the price of stock B moves down in lockstep.

A perfect negative correlation means that two stocks move in opposite directions, while a zero correlation implies no linear relationship at all.

Normally, the correlation is not perfect nor no relationship at all.

It is important to notice that a positive correlation does not promise that when stock A goes up, so does stock B. It is simply expected to happen on average.

The same holds for negative correlation.

Why do we care about correlation?

We are tying to minimize the **maximum drawdown** and **volatility** of an investment.

If our initial investment portfolio has a large **maximum drawdown**, then we can minimize that if we add investments with negative correlation to our portfolio.

The best way to understand this is by example.

Let's get started in our **Jupyter Notebook**.

```
In [1]: import pandas_datareader as pdr
        import datetime as dt
        import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib notebook
```

We start by importing all the libraries we know.

We will explore the **SPY** and **TLT** tickers.

SPY is a SPDR S&P 500 trust is an exchange-traded fund (ETF), which trades on the NYSE Arca.

*"An exchange traded fund (ETF) is a type of security that tracks an index, sector, commodity, or other asset, but which can be purchased or sold on a stock exchange the same as a regular stock." – **Investopedia.org***

*"Spider (SPDR) is a short form name for a Standard & Poor's depository receipt, an exchange-traded fund (ETF) managed by State Street Global Advisors that tracks the Standard & Poor's 500 index (S&P 500)." – **Investopedia.org***

SPY is exactly what we are looking for – a low cost fund that tracks the S&P 500.

TLT is the iShares 20+ Year Treasury Bond ETF seeks to track the investment results of an index composed of U.S. Treasury bonds with remaining maturities greater than twenty years.

Let's investigate them and see how they correlate.

```
In [2]: tickers = ['SPY', 'TLT']

        start = dt.datetime(2008, 1, 1)
        end = dt.datetime(2017, 12, 31)

        data = pdr.get_data_yahoo(tickers, start, end)
```

We will explore them over the 10 years from 2008 to 2018 (last day of December 2017).

As we know, we use the **Adj Close**.

```
In [3]: data = data['Adj Close']
```

Then calculate the log return.

```
In [4]: log_returns = np.log(data/data.shift())
```

The big question is how do we calculate the correlation?

Luckily, **Pandas DataFrames** provide it with a single call to the method **corr()**.

```
In [5]: log_returns.corr()
```

Out[5]:

Symbols	SPY	TLT
Symbols		
SPY	1.000000	-0.445481
TLT	-0.445481	1.000000

As we see, **SPY** has perfect positive correlation with **SPY.**

That is of course obvious, as it is the same prices.

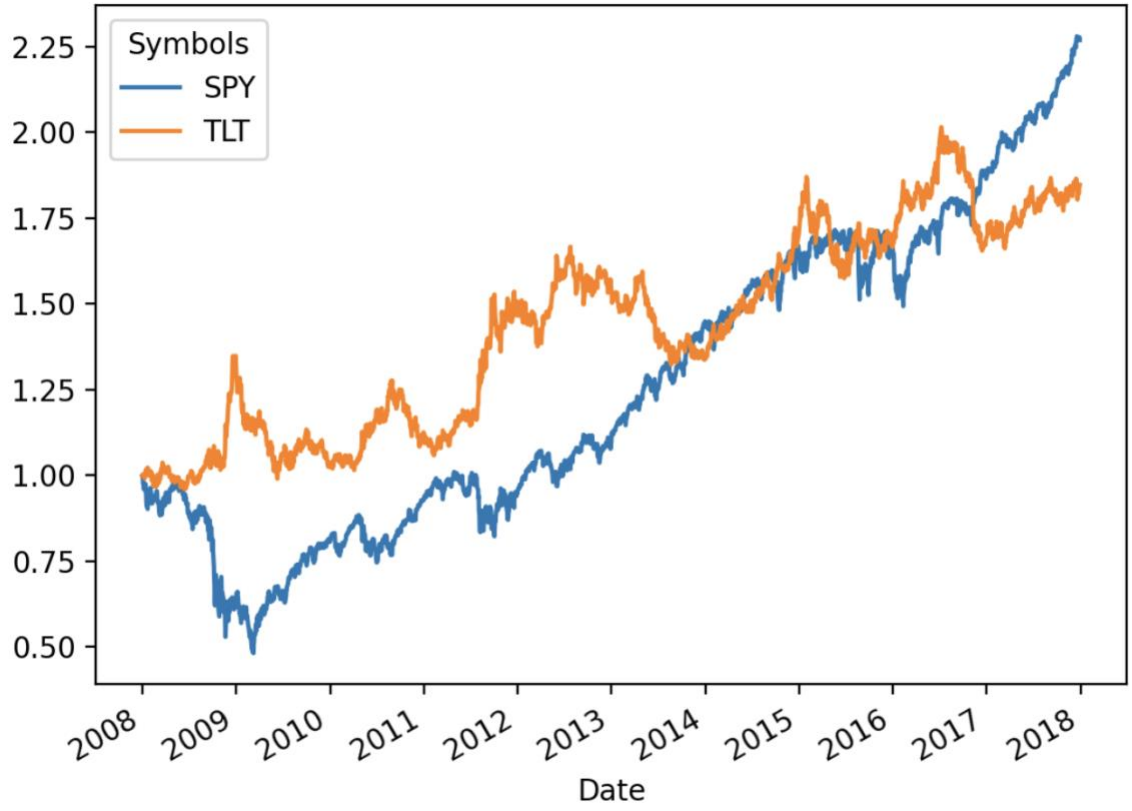
But notice the correlation between **SPY** and **TLT**.

It is negative. This is what we are looking for.

Let's try to visualize that. But we want to do it in a normalized way, as the following does.

```
In [6]: fig, ax = plt.subplots()
        (data/data.iloc[0]).plot(ax=ax)
```

Which results in this.



Now this is interesting.

We see that when the price of **SPY** goes down during the second half of 2008, the price of **TLT** goes up.

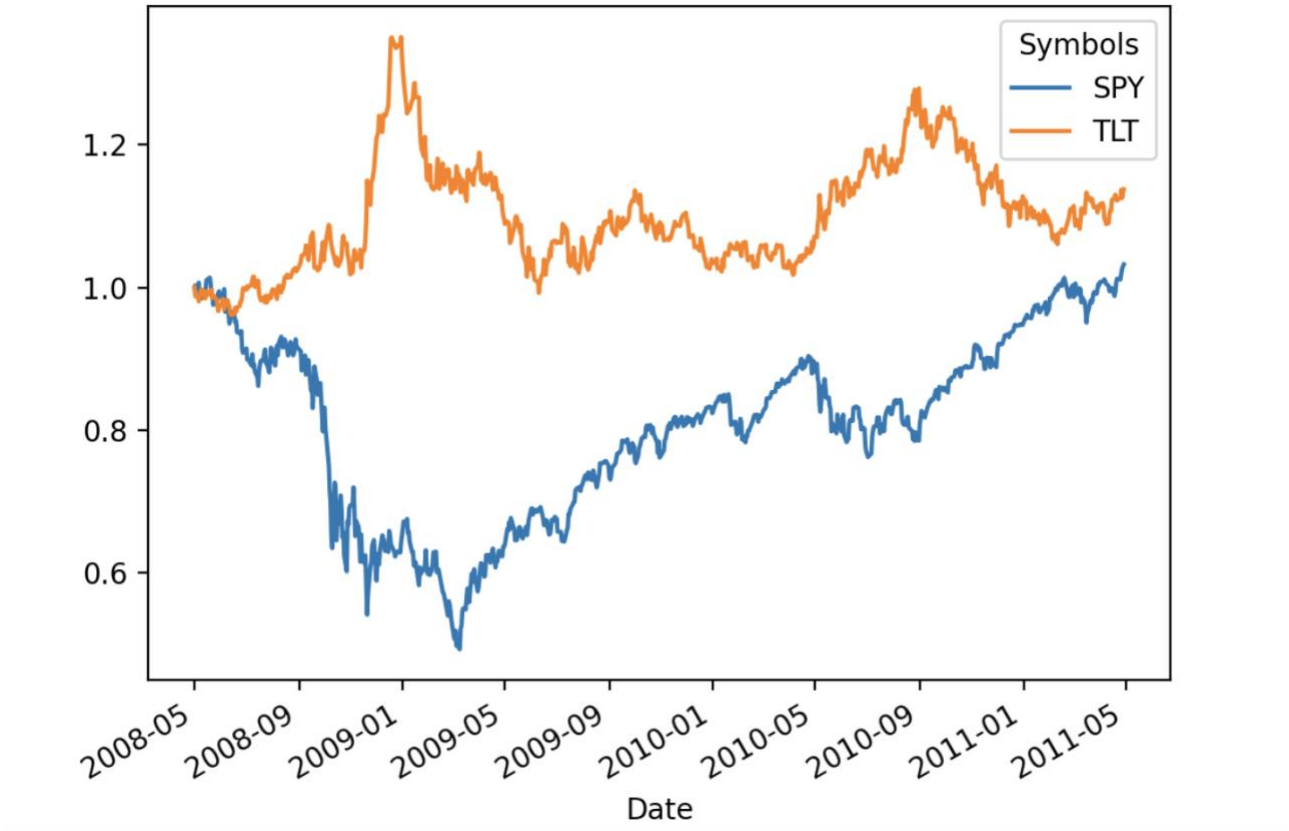
This looks pretty good.

For the fun of it, let's zoom in on May 2008 until May 2011.

```
In [7]: data_set = data.loc['2008-05':'2011-04']

        fig, ax = plt.subplots()
        (data_set/data_set.iloc[0]).plot(ax=ax)
```

This gives the following.



This period looks almost too good to be true. While the **SPY** goes down 2008 and recovers in 2011, the **TLT** goes almost opposite.

*"A hedge is an investment that is made with the intention of reducing the risk of adverse price movements in an asset." – **Investopedia.org***

In the next chapter we will look at how to make a portfolio based on **SPY** and **TLT**.

Project – Calculate the return (CAGR), maximal drawdown and volatility of SPY and TLT

In this project we will explore how **SPY** and **TLT** looks like individually from an investment perspective. In next chapter we will explore how a combination of **SPY** and **TLT** looks like.

Step 1

Let's start by calculate the return (CAGR).

```
In [8]: cagr_spy = (data['SPY'].iloc[-1]/data['SPY'].iloc[0])**(1/10) - 1
        cagr_tlt = (data['TLT'].iloc[-1]/data['TLT'].iloc[0])**(1/10) - 1

In [9]: cagr_spy, cagr_tlt

Out[9]: (0.08540695503495566, 0.06332282408649692)
```

This shows that the annual return of **SPY** is 8.5% and 6.3% for **TLT**.

Step 2

To calculate the maximum drawdown, we first define a function.

```
In [10]: def max_drawdown(data):
             rolling_max = data.cummax()
             daily_drawdown = data/rolling_max - 1
             max_drawdown = daily_drawdown.cummin().iloc[-1]
             return max_drawdown
```

Then we can call the function.

```
In [11]: max_drawdown(data['SPY']), max_drawdown(data['TLT'])
Out[11]: (-0.5187374993952358, -0.26585442818790594)
```

This shows that the maximum drawdown is 51.8% for **SPY** and 26.6% for **TLT**.

Step 3

We calculate the volatility of the log returns.

```
In [12]: log_returns.std()*(252**0.5)
Out[12]: Symbols
         SPY    0.203319
         TLT    0.151634
         dtype: float64
```

This shows that the volatility of **SPY** is 20.3% and 15.2% for **TLT**.

In the next chapter we will see how a combination of **SPY** and **TLT** affects the values.

Summary

In this chapter we have explored **correlation**. If two stocks are **negatively correlated**, we expect that if the price goes up of one the price goes down of the other.

We found that **SPY** and **TLT** are negatively correlated.

06 – A Simple Portfolio

In the last chapter we realized that **SPY** and **TLT** are negatively correlated. Here we will explore if a simple portfolio of a 50-50 percentage split of **SPY** and **TLT** will improve our investment in terms of return (**CAGR**), **maximum drawdown**, and **volatility**.

The project in this chapter will investigate how an annual rebalancing will affect the result.

Learning objectives

- Calculate the return (**CAGR**), **maximum drawdown**, and **volatility** of a portfolio
- Compare the result to **SPY**.
- Investigate the portfolio visually.
- Make a rebalancing approach and compare it.

The simple portfolio

We have identified that **SPY** and **TLT** are negatively correlated.

The calculations from last chapter showed that during 2008 to 2018 we have the following figures.

	CAGR	Maximum drawdown	Volatility
SPY	8.54%	51.9%	20.3%
TLT	6.33%	26.6%	15.2%

The **maximum drawdown** of **SPY** is scary, losing 51.9% is not what we want to experience.

This makes the period of 2008 interesting. How can we avoid a maximum drawdown like that?

In this chapter we will explore if a 50-50 percent split between **SPY** and **TLT** would improve on the above figures.

First, we will make a simple Buy-and-hold approach with the split. Then in the project we will make an annual rebalance of the split. This helps to avoid a that the split drifts far away from the initial 50-50 percentage split.

Say, if we start with a 50-50 percentage split of **SPY** and **TLT**. Imagine **SPY** drops 50% and **TLT** raises 50%. Then the split is 25-75 percentage between **SPY** and **TLT**.

50-50 split without rebalance

To make this evaluation we need to import some libraries.

```
In [1]: import pandas_datareader as pdr
        import datetime as dt
        import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib notebook
```

And read the data from the **API**.

```
In [2]: tickers = ['SPY', 'TLT']

        start = dt.datetime(2008, 1, 1)
        end = dt.datetime(2017, 12, 31)

        data = pdr.get_data_yahoo(tickers, start, end)
```

Keep the **Adj Close**.

```
In [3]: data = data['Adj Close']
```

Make the portfolio.

```
In [4]: portfolio = [.5, .5]
```

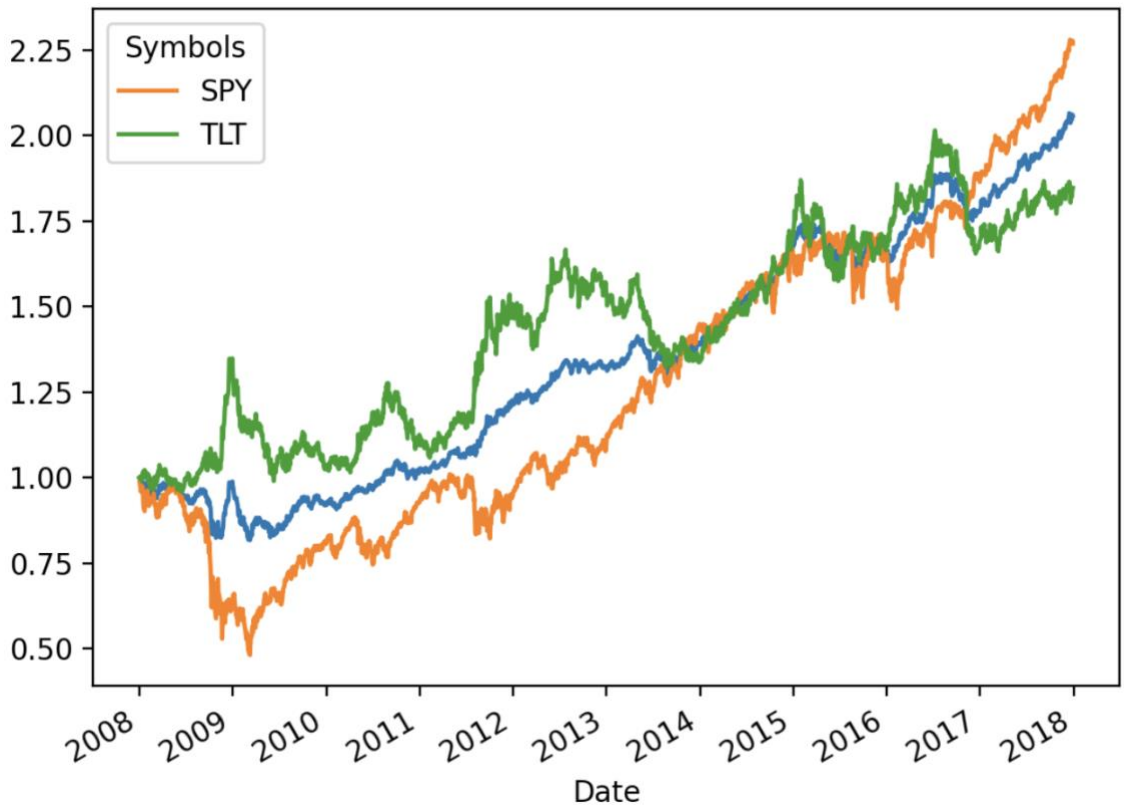
Let's visualize the portfolio with the **SPY** and **TLT**.

```
In [5]: fig, ax = plt.subplots()

        ((data/data.iloc[0])*portfolio).sum(axis=1).plot(ax=ax)
        (data/data.iloc[0]).plot(ax=ax)
```

Notice the **.sum(axis=1)**, it sums along the rows. Hence, it takes the values of the portfolios and sums each row up. This results in the normalized values of the portfolio.

Resulting in this figure.



The blue line is our portfolio.

It is interesting to see that we avoid the big drawdown SPY encounters during 2008 and 2009.

This could look like we actually benefit from this approach.

Let's calculate the figures.

To do that, we will create a series with the strategy and one with log returns.

```
In [6]: strategy = ((data/data.iloc[0])*portfolio).sum(axis=1)

In [7]: log_returns = np.log(strategy/strategy.shift())
```

Then we can calculate the **CAGR**.

```
In [8]: (strategy.iloc[-1]/strategy.iloc[0])**(1/10) - 1
Out[8]: 0.07487474582586873
```

This is a bit lower than **SPY**, which has a return of 8.54%.

The **maximum drawdown**.

```
In [9]: rolling_max = strategy.cummax()
        daily_drawdown = strategy/rolling_max - 1
        max_drawdown = daily_drawdown.cummin().iloc[-1]
        max_drawdown
Out[9]: -0.18279716207319707
```

Now that is a great improvement. The **SPY** had 51.9% and here we get 18.3%.

Finally, the **volatility**.

```
In [10]: log_returns.std()*(252**0.5)
Out[10]: 0.08850024414427328
```

Again, **SPY** had 20.3% and here we have 8.85%.

What we have gained for the price of a lower return is safety. A great improvement on **maximum drawdown**, which says we could lose 18.3% comparing to 51.9%. The **volatility** is also greatly improved by this strategy.

Can this strategy be improved?

Well, we talked about it. There is a problem with this strategy. The split of 50-50 percent will drift away over the years.

Hence, in the project of this chapter we will make a yearly rebalance and see how it affects the numbers.

Project – An annual rebalance

We will stay with **SPY** and **TLT**, but now we will rebalance the portfolio annually.

Step 1

Here we will make a rebalance the end of the year.

Let's look at the code first and see what it does.

```
In [11]: concat = []
         for year in range(2008, 2018):
             rebalance = (data.loc[str(year)]/data.loc[str(year)].iloc[0]*portfolio).sum(axis=1)
             if year > 2008:
                 rebalance = rebalance*concat[-1].iloc[-1]
             concat.append(rebalance)


         strategy = pd.concat(concat)
```

On a top level we iterate over the years and make a 50-50 percentage split of the portfolio.

The only thing we need to adjust for is the return from previous year. This is the if statement (as we should not do it the first year).

Hence after the for-loop we have a list **concat** where we have the yearly rebalanced portfolio adjusted from previous year return.

The **pd.concat(.)** simply concatenates the list **concat** into one Series.

Now that was nice.

Step 2

We can do as we did above to calculate the **CAGR**.

```
In [12]: (strategy.iloc[-1]/strategy.iloc[0])**(1/10) - 1
Out[12]: 0.08511674426186389
```

Step 3

The **maximum drawdown**.

```
In [13]: rolling_max = strategy.cummax()
         daily_drawdown = strategy/rolling_max - 1
         max_drawdown = daily_drawdown.cummin().iloc[-1]
         max_drawdown
Out[13]: -0.2028153660210089
```

Step 4

The **volatility**.

```
In [14]: (np.log(strategy/strategy.shift())).std()*(252**0.5)
Out[14]: 0.08893946618859903
```

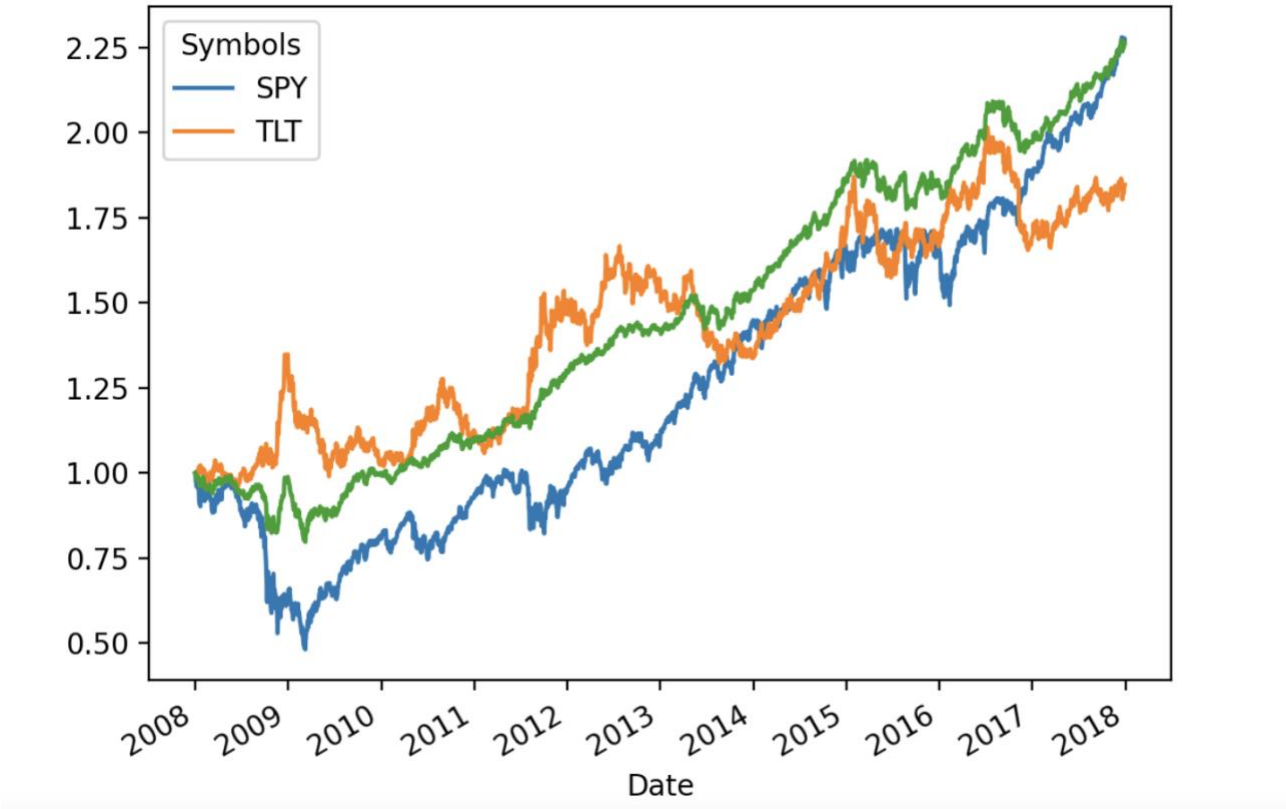
Step 5

Visualize it.

```
In [15]: fig, ax = plt.subplots()

         (data/data.iloc[0]).plot(ax=ax)
         strategy.plot(ax=ax)
```

Resulting in this chart.



The green line represents our rebalancing approach.

We see a difference already. At the far end of 2018 we end up at the same point as **SPY** (or fairly close).

Did we lose our advantage?

Let's compare it here.

	CAGR	Maximum drawdown	Volatility
SPY	8.54%	51.9%	20.3%
TLT	6.33%	26.6%	15.2%
SPY-TLT split	7.49%	18.3%	8.85%
SPY-TLT rebalance	8.51%	20.3%	8.89%

Now that looks good.

We basically have the same return as **SPY** and keep the **maximum drawdown** low as well as **volatility**.

A word of warning and how to avoid it

Backtesting is based on historic data and cannot predict the future. The above strategy was presented to me and calculated over the time period of 2008 to 2018, as we have done here.

The reason is most likely that we are interested in periods where the market crashed. We want to test up against that.

Most strategies are presented on paper and have to be recalculated manually.

Therefore, we often think that the presented strategies have been tested in all possible scenarios. This is often a false assumption.

This **eBook** was written such that you can test (backtesting) the same strategies until you feel comfortable with them.

Let's run the above strategy over the 10 years starting from 2011 and ending in 2020 to see how it performs.

	CAGR	Maximum drawdown	Volatility
SPY	13.6%	33.7%	17.3%
TLT	8.2%	20.7%	14.5%
SPY-TLT split	11.2%	18.2%	8.66%
SPY-TLT rebalance	11.0%	16.0%	8.00%

It paints a different picture.

The strategies (**SPY-TLT split** and **SPY-TLT rebalance**) did not return (**CAGR**) as well as **SPY** alone.

It is a decent return of 11%. It still greatly improves the **maximum drawdown** and **volatility** over the **SPY**, which lowers the risk of your investment.

Summary

In this chapter we explored our first strategy with a 50-50 percent split between **SPY** and **TLT.** With backtesting we saw that this strategy keeps a decent return (**CAGR**), while lowering the risk significantly (**maximum drawdown** and **volatility**). This can be further improved if we add an annual rebalancing of the portfolio.

Finally, we looked at the importance to test (backtesting) presented strategies yourself. Often strategies are presented under the best circumstances and do not show the full picture.

07 – The Asset/Hedge Split

In the last chapter we used the 50-50 percentage split between **SPY** and **TLT**. The asset is **SPY**, and the more we allocate to **SPY**, the higher return. **TLT** is the hedge and provides safety, while sacrificing return.

What ratio should we use?

It turns out that most financial advisors suggest a 60-40 percentage split. 60 percent to the asset and 40 percent to the hedge.

In this chapter we will explore how different splits perform for our portfolio of **SPY** and **TLT**. It will not conclude the 60-40 split as the best, but remember, we are looking at a concrete example and the advice is more general.

Learning objectives

- Evaluate different splits (ratios) of an asset-hedge.
- Calculate it specifically for **SPY** and **TLT**
- Visualize the figures of the different **SPY** and **TLT** splits.
- Calculate and visualize it for **Sharpe Ratio**.

The calculations

The calculations in this section can be done for any type of asset/hedge split. We will continue our journey with the **SPY** and **TLT** over the 10 years period of 2008 and 2018.

```
In [1]: import pandas_datareader as pdr
        import datetime as dt
        import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib notebook
```

Let's read the data.

```
In [2]: tickers = ['SPY', 'TLT']

        start = dt.datetime(2008, 1, 1)
        end = dt.datetime(2017, 12, 31)

        data = pdr.get_data_yahoo(tickers, start, end)

        data = data['Adj Close']
```

As we need to evaluate different splits and are interested in the same calculations we will create a function which returns the results.

The calculations are the same we have done in previous chapters.

```
In [3]: def evaluate_split(data, split):
            portfolio = [split, 1. - split]

            eval_set = ((data/data.iloc[0])*portfolio).sum(axis=1)

            cagr = (eval_set.iloc[-1]/eval_set.iloc[0])**(1/10) - 1

            rolling_max = eval_set.cummax()
            daily_drawdown = eval_set/rolling_max - 1
            drawdown = daily_drawdown.cummin().iloc[-1]

            log_returns = np.log(eval_set/eval_set.shift())
            volatility = log_returns.std()*(252**.5)

            return cagr, drawdown, volatility
```

The function takes the data and split. The split is a value assigning the ratio to the asset.

Notice, that we do not make the rebalancing here. That is a great exercise you can do on your own.

To see the 50-50 split simple call the function as follows.

```
In [4]: evaluate_split(data, .5)
Out[4]: (0.07487474582586873, -0.18279716207319707, 0.08850024414427328)
```

What we want here, is to calculate these 3 values for all possible splits.

This can be done as follows.

```
In [5]: x = np.arange(0, 1.01, .05)

        df = pd.DataFrame(x)

        res = df.apply(lambda x: evaluate_split(data, x), axis=1)

        df['CAGR'] = res.str[0]
        df['Drawdown'] = res.str[1]
        df['Volatility'] = res.str[2]

        df.set_index(0, inplace=True)
```

We create an **arange** (a **NumPy** range) from 0 to 1 (inclusive) stepping 0.05.

From that we create a **DataFrame** to keep all the data.

The apply calls the **lambda** function, which applies our function from last step. Then we map out the result to our **DataFrame**. Finally, we set the index to be the column with the split values.

You can see the full dataset with the following command (we only show the first few lines here).

```
In [6]: df
```

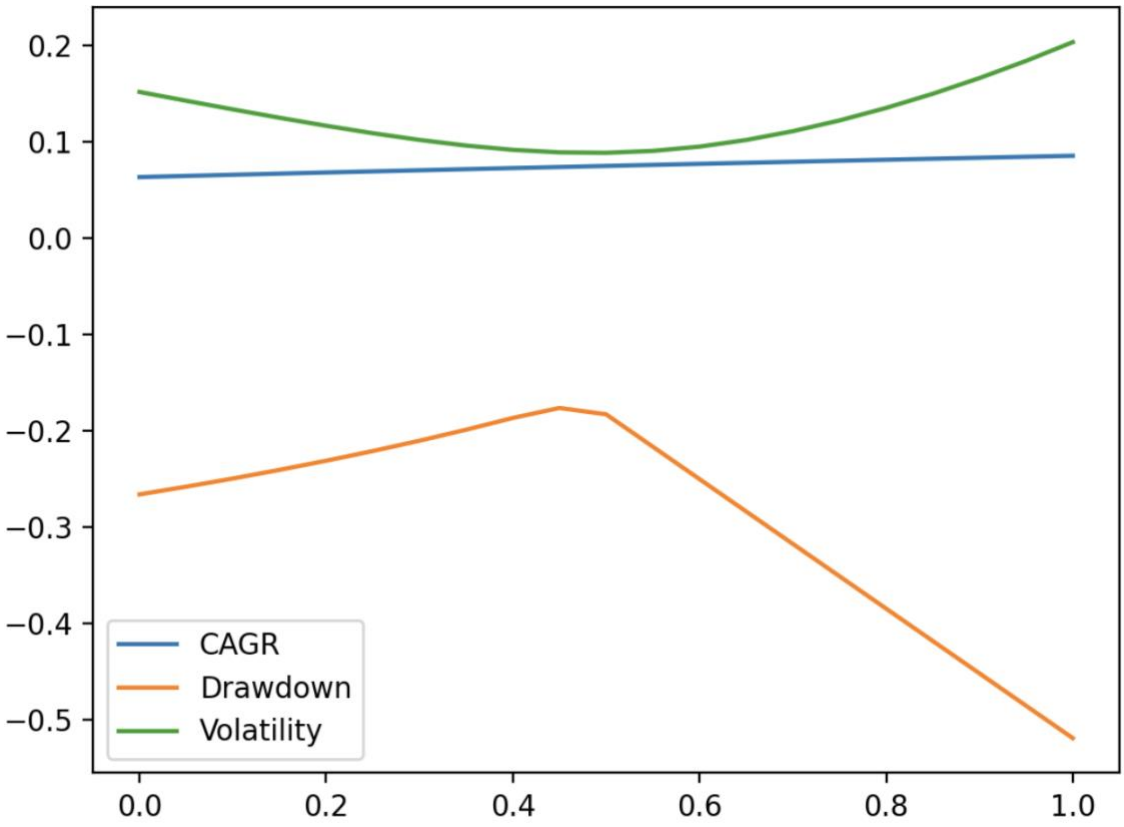
Out[6]:

0	CAGR	Drawdown	Volatility
0.00	0.063323	-0.265854	0.151634
0.05	0.064530	-0.257845	0.142502
0.10	0.065725	-0.249384	0.133543
0.15	0.066908	-0.240432	0.124849
0.20	0.068079	-0.230946	0.116548

Then we can visualize the result as follows.

```
In [7]: fig, ax = plt.subplots()
        df.plot(ax=ax)
```

Resulting in this chart.



This is a great way to get an impression of the different splits.

The x-axis shows the split. In the far left we have split 0.0 (0% to **SPY** and 100% to **TLT**) and in the far right we have the 1.0 split (100% to **SPY** and 0% to **TLT**).

We see the return (**CAGR**) increases as the split favors **SPY**. This is exactly as stated in the introduction.

We see the lowest **volatility** is somewhere in the middle. Also, the optimal **maximum drawdown** (it is negative) is somewhere in the middle.

This means, that in this specific evaluation the optimal split is somewhere in the middle if only considering **maximum drawdown** and **volatility**.

More precisely with code.

```
In [8]: df.iloc[df['CAGR'].argmax()]
Out[8]: CAGR          0.085407
        Drawdown     -0.518737
        Volatility    0.203319
        Name: 1.0, dtype: float64
```

The highest return (**CAGR**) is with 100% to **SPY** and 0% to **TLT**.

```
In [9]: df.iloc[df['Volatility'].argmin()]
Out[9]: CAGR          0.074875
        Drawdown     -0.182797
        Volatility    0.088500
        Name: 0.5, dtype: float64
```

The lowest **volatility** is at a 50-50 percent split.

```
In [10]: df.iloc[df['Drawdown'].argmax()]
Out[10]: CAGR          0.073769
         Drawdown     -0.176199
         Volatility    0.088954
         Name: 0.45, dtype: float64
```

To get the optimal **maximal drawdown** we should have a 45-55 percentage split to **SPY-TLT**.

We can get a small window of the numbers here.

```
In [11]: df.loc[.5:.7]
```

Out[11]:

	CAGR	Drawdown	Volatility
0			
0.50	0.074875	-0.182797	0.088500
0.55	0.075971	-0.216391	0.090457
0.60	0.077057	-0.249985	0.094910
0.65	0.078133	-0.283579	0.101787

What can we conclude?

That the optimal split for this specific evaluation is somewhere in the middle. This is actually great news. We see that it confirms the financial advisors' suggestion. Also, we can see that **maximal drawdown** and **volatility** are optimal around there.

Project – Sharpe Ratio

A common way to evaluate a portfolio is with **Sharpe Ratio**.

*"The Sharpe ratio was developed by Nobel laureate William F. Sharpe and is used to help investors understand the return of an investment compared to its risk." – **Investopedia.org***

The formula is as follows.

$$Sharpe\ ratio = \frac{Return\ of\ portfolio - Riskfree\ rate}{standard\ deviation}$$

The goal is to optimize the Sharpe ratio.

We already have the return of the portfolio and the standard deviation.

For the risk-free rate the U.S. Treasury rate is often used. Sometimes, even 0 is used, as the rate has been very low.

For the example we will use the 10 years U.S treasury yield. **You can find updated values here**.

Here we will use 1.65%.

One thing to notice, the **Sharpe Ratio** does not take the **maximum drawdown** into account.

Step 1

Calculate the Sharpe Ratio.

```
In [12]: sr = (df['CAGR'] - 0.0165)/df['Volatility']
```

Notice, we do it simultaneously for all possible splits.

Step 2

Find the optimal split.

```
In [13]: sr.argmax()
Out[13]: 10
```

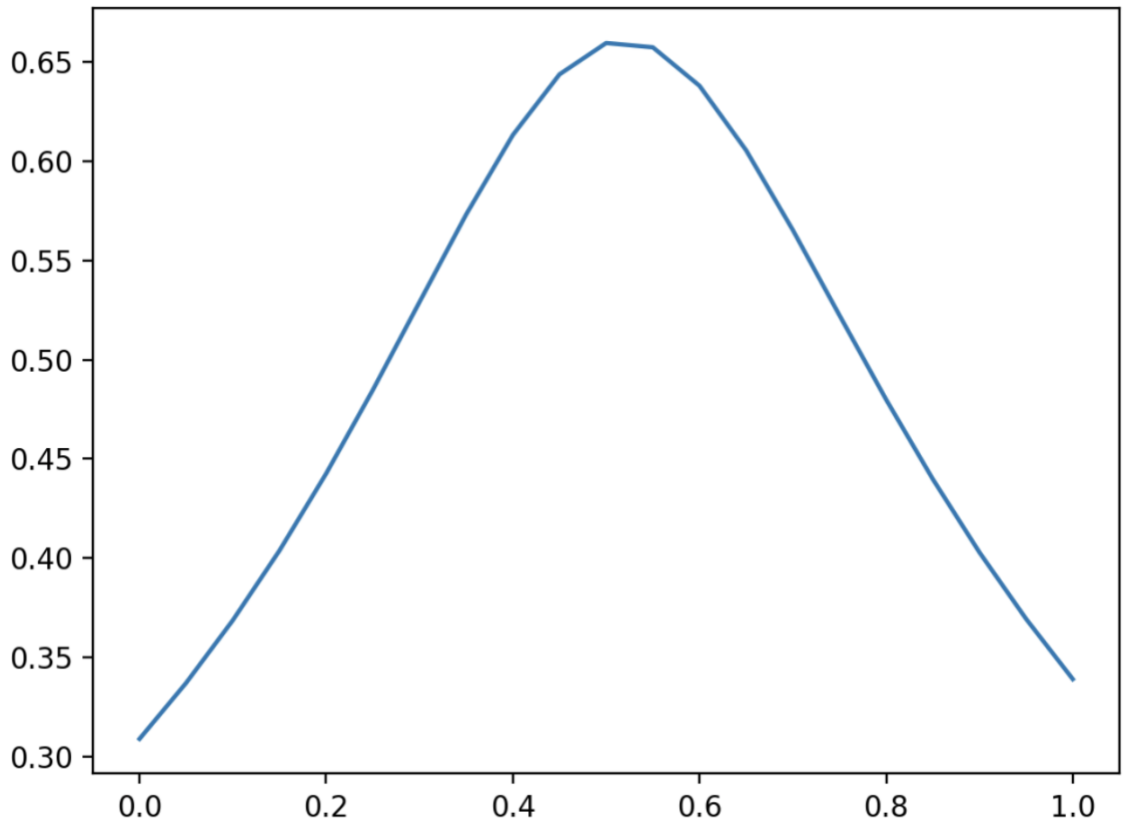
Hence, this is the 50-50 percent split (we see it in next step).

Step 3

Visualize and show the value of the optimal split.

```
In [14]: fig, ax = plt.subplots()
         sr.plot(ax=ax)
```

Resulting in the following figure.



This shows the chart of the Sharpe Ratio according to the different splits. The x-axis is showing the same as previous.

```
In [15]: df.iloc[10]

Out[15]: CAGR          0.074875
         Drawdown     -0.182797
         Volatility    0.088500
         Name: 0.5, dtype: float64
```

We see the optimal values above (the index from step 2).

What can we conclude?

Well, the theory of **Sharpe Ratio** also confirms that a split somewhere in the middle seems to be optimal. Remember, **Sharpe Ratio** tries to optimize the return in regards to risk.

Summary

In this chapter we have explored if our findings concur with the financial advisors of a 60-40 percentage split for asset-hedge. On the concreate calculations we found a split in the middle to be optimal. This means that the minimal risk is somewhere in the middle.

To give a precise general split would require more calculations and is outside the scope of this **eBook**. That said, our findings are not contradicting the general advice.

08 – Backtesting an Active Strategy

In this chapter we will evaluate a active strategy and perform backtesting on it.

The strategy will use the simple moving average (**MA**) of 200 days. When the current market price of **SPY** is above **MA** we go long (buy and hold **SPY**), when the current market price of **SPY** is below **MA**, we short and buy and hold **TLT**.

Learning objectives

- How to evaluate an active strategy as above.
- Perform backtesting for an active strategy.
- Create a visual evaluation from the backtesting.
- Drawdown lesson from 2008 in backtesting

Getting started with an active strategy

The strategy explained

Let's first be sure we understand the strategy.

On a high level, we either have all our holding in **SPY** or in **TLT**. This means, at all times during the strategy we will either have 100% in **SPY** or 100% in **TLT**.

The trigger for a change is based on the current **SPY** price and 200 days moving average of the price.

If the current **SPY** price is above the 200 days moving average, we will have 100% in **SPY**.

On the other hand, if the current **SPY** price is below the 200 days moving average, we will have 100% in **TLT**.

How can we model that?

Getting started

As usual, we need some libraries.

In [1]:
```
import pandas_datareader as pdr
import datetime as dt
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook
```

And read the data from the **API**.

In [2]:
```
tickers = ['SPY', 'TLT']

start = dt.datetime(2007, 1, 1)
end = dt.datetime(2021, 1, 1)

data = pdr.get_data_yahoo(tickers, start, end)

data = data['Adj Close']
```

We have a longer time span than usual, this is to be able to make different backtesting frames.

The signal-line

How do we model the strategy?

We can create a signal-line.

```
In [3]: ma = data['SPY'].rolling(200).mean()

        signal_line = data['SPY'] - ma
        signal_line = signal_line.apply(np.sign)
```

We first calculate the moving average (**ma**), then we have the signal line to be the difference between the price of **SPY** and **ma**. This will make **signal_line** positive when we should have our holdings in **SPY** and negative when we should have our holding in **TLT**.

The **apply(np.sign)** simply transform the sign to either 1 or -1. Hence, if 1, we hold **SPY**, and if -1, we hold **TLT**.

Log returns of the strategy

Then we calculate the log returns.

```
In [4]: log_return = np.log(data/data.shift())
```

This is needed to calculate the return of our strategy.

```
In [5]: rtn = signal_line.clip(lower=0).shift(1)*log_return['SPY']
        rtn = rtn - (signal_line.clip(upper=0).shift())*log_return['TLT']
```

Now that looked too easy to be true.

Let's break it down.

The **signal_line.clip(lower=0)** takes all the positive values (all the 1's) of **signal_line**. First leave out the shift(1), we'll get back to it.

Hence, the we take all the 1's from **signal_line** and get the log returns of **SPY**. This is all the returns we have from holding **SPY**.

Why the **shift(1)**?

As we know, when current price shifts above the moving average, we cannot react on that day. The calculations are done on the closing prices. Hence, we can first react first thing next morning. Hence, this is our reaction time.

Is it 100% accurate? No, because we cannot be ensured the price at opening the day after. But this is the best we can do in our calculations.

The second line.

```
In [5]: rtn = signal_line.clip(lower=0).shift(1)*log_return['SPY']
        rtn = rtn - (signal_line.clip(upper=0).shift())*log_return['TLT']
```

As you see, it is similar and still a bit different. It takes and subtracts all the values from **TLT**. This is needed as the **signal_line** here is negative and we want to add the log return of **TLT**.

Notice, that we also use **clip(upper=0)** to get all the negative signal values (all the -1's).

Now we have the log return of the strategy.

The cumulative return of the strategy

To calculate the cumulative return of the strategy you can use the **cumsum()** and **apply(np.exp)**.

```
In [6]: rtn.loc['2008':].cumsum().apply(np.exp)
```

```
Out[6]: Date
        2008-01-02    1.014403
        2008-01-03    1.013005
        2008-01-04    1.013219
        2008-01-07    1.017626
        2008-01-08    1.016444
                        ...
        2020-12-24    3.486034
        2020-12-28    3.515982
        2020-12-29    3.509275
        2020-12-30    3.514282
        2020-12-31    3.532137
        Length: 3274, dtype: float64
```

This sums the cumulative of the log returns and applies the exponential function on it. This will calculate the normalized return of the strategy.

Similarly, you can do the same for the **SPY**.

```
In [7]: log_return['SPY'].loc['2008':].cumsum().apply(np.exp)
```

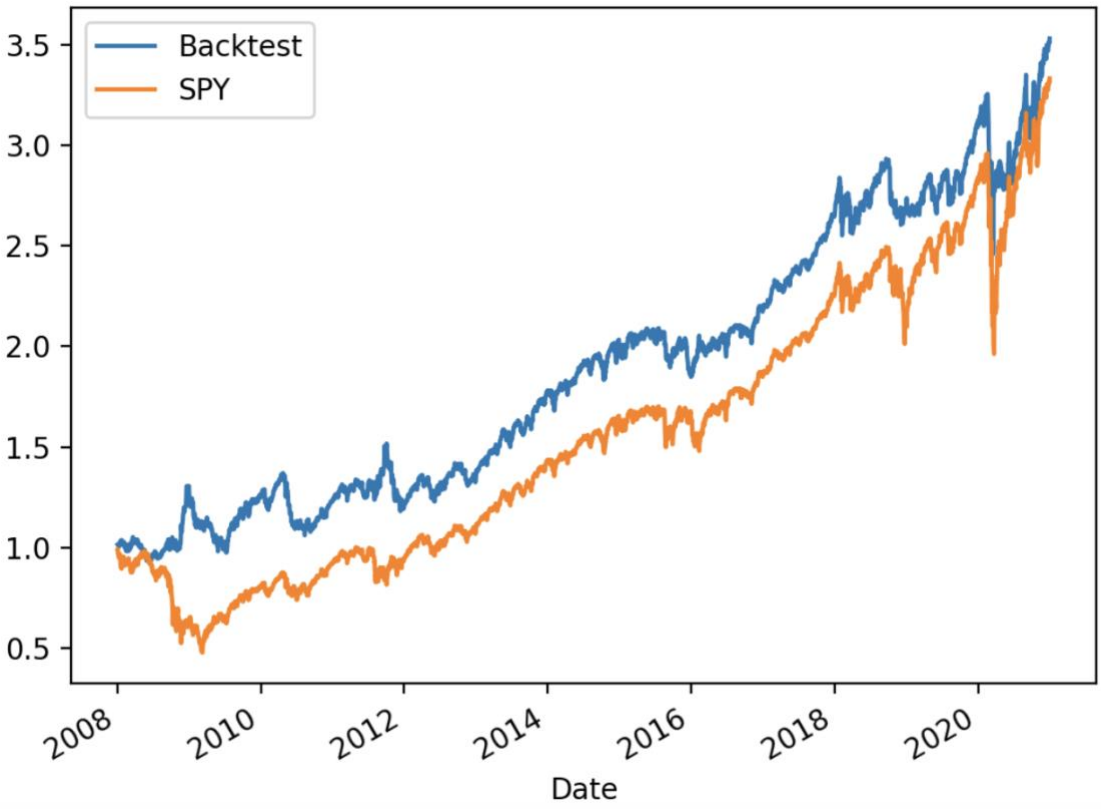
```
Out[7]: Date
        2008-01-02    0.991245
        2008-01-03    0.990767
        2008-01-04    0.966486
        2008-01-07    0.965666
        2008-01-08    0.950072
                        ...
        2020-12-24    3.290065
        2020-12-28    3.318329
        2020-12-29    3.311998
        2020-12-30    3.316724
        2020-12-31    3.333576
        Name: SPY, Length: 3274, dtype: float64
```

This comparison can be visualized with the following code.

```
In [8]: fig, ax = plt.subplots()

        rtn.loc['2008':].cumsum().apply(np.exp).plot(ax=ax, label='Backtest')
        log_return['SPY'].loc['2008':].cumsum().apply(np.exp).plot(ax=ax)
        ax.legend()
```

Resulting in this figure.



Hence, this looks like our strategy performs better than **SPY**.

Let's make a project and learn more about it.

Project – Backtesting different periods and visualize results

In this project we will create helper functions to achieve this. First a function that calculates the **CAGR**, **maximal drawdown**, and **volatility** for a given period. Then we use that function to calculate what we are looking for. Finally, visualize the result in a nice way.

Step 1

First, we will create a function, which calculates the **CAGR**, **maximal drawdown**, and **volatility** for a given time period.

```
In [9]: def calculate(log_return, start, end):
            years = int(end) - int(start) + 1

            data = log_return.loc[start:end]

            cagr = np.exp(data.sum())**(1/years) - 1

            norm = data.cumsum().apply(np.exp)

            rolling_max = norm.rolling(252).max()
            daily_drawdown = norm/rolling_max - 1
            drawdown = daily_drawdown.cummin().iloc[-1]

            volatility = data.std()*(252**.5)

            return cagr, drawdown, volatility
```

Step 2

Let's call the function for with SPY for the period 2008 to the end of 2017.

```
In [10]: calculate(log_return['SPY'], '2008', '2017')
Out[10]: (0.08445296710992145, -0.5148146413395673, 0.2032989015047215)
```

These figures look familiar.

How does our strategy perform?

```
In [11]: calculate(rtn, '2008', '2017')
Out[11]: (0.10213224463153447, -0.2530085058434649, 0.14464287771507248)
```

Now this looks pretty good. A better return 10.2% **CAGR**, 25.3% **maximum drawdown**, and 14.5% **volatility**.

Step 3

Now for the big step.

How do we visualize this in a good way?

Let's jump into it and try.

```
In [12]: def visualize(backtest, spy, start, end):
             def x_titles(spy_val, bt_val):
                 spy_str = str(round(spy_val*100, 1))
                 bt_str = str(round(bt_val*100, 1))
                 return ['SPY\n' + spy_str + '%', 'Backtest\n' + bt_str + '%']

             spy_cagr, spy_drawdown, spy_vol = calculate(spy, start, end)
             bt_cagr, bt_drawdown, bt_vol = calculate(backtest, start, end)

             fig, ax = plt.subplots(2, 2)

             spy.loc[start:end].cumsum().apply(np.exp).plot(ax=ax[0, 0])
             backtest.loc[start:end].cumsum().apply(np.exp).plot(ax=ax[0, 0], label='Backtest', c='c')
             ax[0, 0].legend()

             x = x_titles(spy_cagr, bt_cagr)
             ax[0, 1].bar(x, [spy_cagr, bt_cagr], color=['b', 'c'])
             ax[0, 1].set_title("CAGR")

             x = x_titles(spy_drawdown, bt_drawdown)
             ax[1, 0].bar(x, [spy_drawdown, bt_drawdown], color=['b', 'c'])
             ax[1, 0].set_title("Drawdown")

             x = x_titles(spy_vol, bt_vol)
             ax[1, 1].bar(x, [spy_vol, bt_vol], color=['b', 'c'])
             ax[1, 1].set_title("Volatility")

             plt.tight_layout()
```

That looks intimidating at first.

But let's break it down.

The function has a helper function inside (**x_titles()**), which creates the *x* titles in our diagrams.

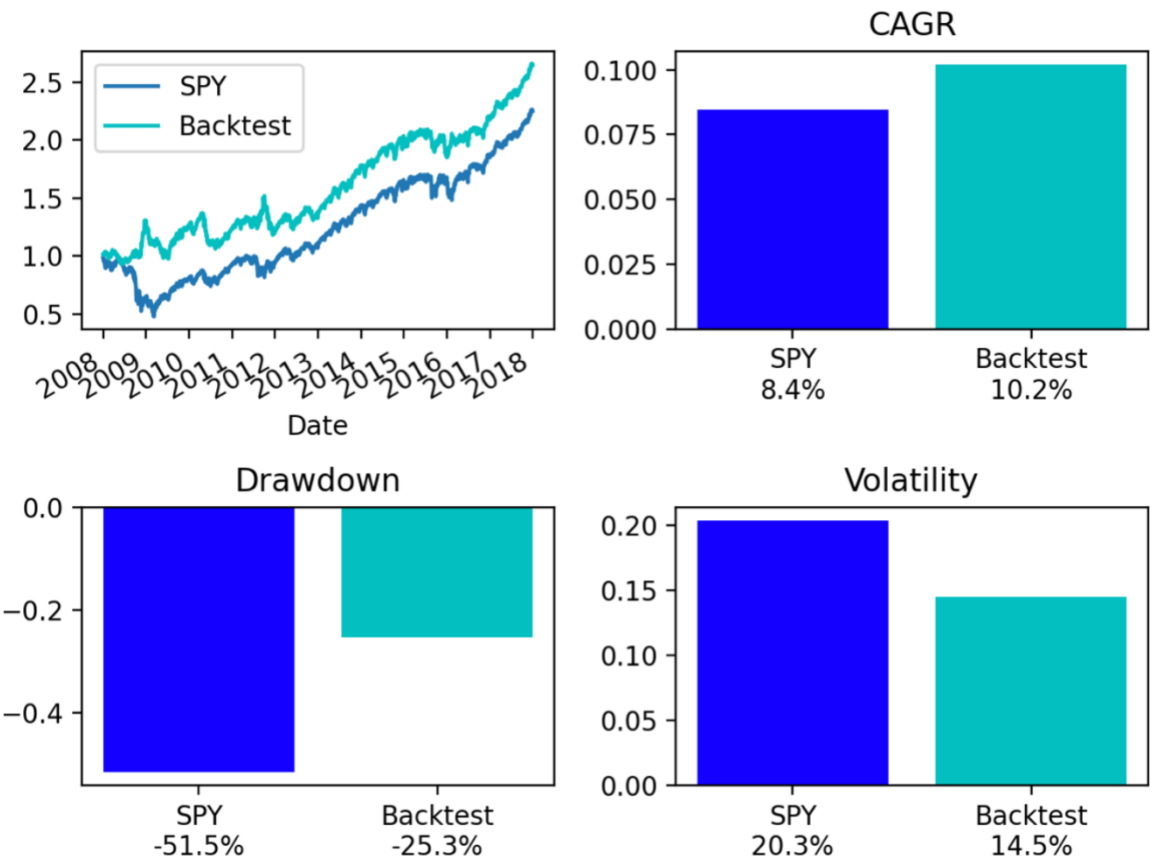
Then it calculates the values for **SPY** and our **Backtest**.

The rest of the code is creating the figure from **Matplotlib**. We create 4 axes in our figure. One for the overall performance, one for **CAGR**, one for **maximum drawdown**, and one for **volatility**.

Let's try it.

```
In [13]: visualize(rtn, log_return['SPY'], '2008', '2017')
```

Which results in the following chart.



This looks good, right?

Our strategy outperforms the SPY in all aspects. Nothing to discuss, there is no reason not to go all in with this strategy.

Or is there?

The learning from 2008 backtesting

The year 2008 was interesting for many reasons.

- Cyprus and Malta adopted the Euro.
- Iran launched a rocket to space.
- A NASA spacecraft becomes the first to land on northern polar region of Mars.

To name a few things.

But also, the S&P 500 dropped over 50%.

That was a big loss and it took long to recover. Many investors lost a great deal of money. Image you had your life savings of 1,000,000 USD in the start of 2008 and ready to retire. Later that year you would most like have less than 500,000 USD.

That hurts.

Hence, it is no great surprise that backtesting strategies over 2008 has been of great interest. How do avoid such a big loss?

This is the **maximum drawdown** factor at play. We want to minimize it to lower the risk.

What this **eBook** should teach you is to make backtesting on your own and not trust others findings.

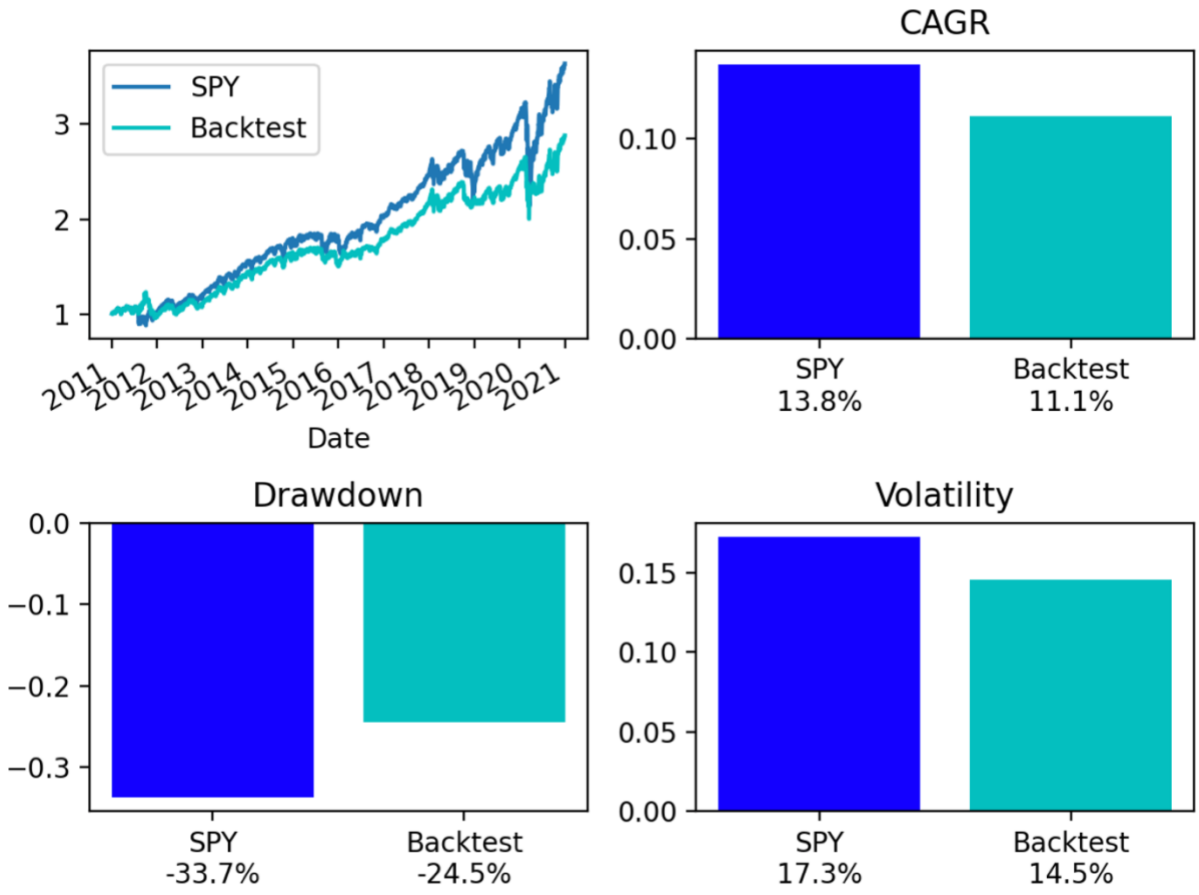
When a backtesting performs good over 2008 it actually has a great advantage.

Why?

Let's just look at the backtesting we just did and redo it over another period (yes, we made that easy to do).

```
In [14]: visualize(rtn, log_return['SPY'], '2011', '2020')
```

Let's see how 2011 until the end of 2020 looks like.



Not bad. But not as good as 2008 to and 10 years forward.

Well, there is one thing to understand.

Let's simplify it a bit.

Assume 2008 ended with 50% loss in **SPY**.

If your strategy broke even in 2008.

Then your strategy has a 100% advantage.

What?

Yes, the **SPY** needs to grow 100% again before it is on the level of your strategy.

100% growth is a lot and a great advantage.

Still confused. Let's break it down.

If you invested 100 USD in **SPY** in 2008, in the end you would have 50 USD.

If you invested 100 USD in your strategy in 2008, in the end you would have 100 USD.

Hence, in 2009 your strategy has a 100% advantage.

$$\frac{100}{50} - 1 = 1$$

That means, the **SPY** needs to grow 100% to break even with your strategy.

If your strategy grows 50% in 2009, you will have 150 USD.

Then **SPY** needs to grow 200% in 2009 to break even to 150 USD.

You see, this is a huge advantage for your strategy.

Hence, if your strategy can just break even from the big loss in 2008, it doesn't need to perform perfect afterwards as it has a great advantage over the **SPY**.

Summary

In this chapter we have explored our first active strategy. We learned a great way to calculate the return of a strategy. Using this return to make the actual backtesting. Additionally, we created a nice way to visualize a backtesting with a **SPY** comparison.

Finally, we had a discussion on how the strategies beating the 2008 market collapse has a great advantage in the following time.

09 – Backtesting Another Strategy

The strategy we will explore in this chapter comes from the book **THE 12% SOLUTION** by David Allan Carter.

We will not implement the full end solution, but pretty much the whole thing. The reason for not implementing the full solution is to keep the last details of the strategy in his book.

The exploration of his strategy starts in 2008. We will do the same here and extend it afterwards.

Learning objectives

- Understand the 12% solution.
- Show how we can implement the 12% solution.
- Perform backtesting for of the 12% solution
- Conclude if the picture changes if we change backtesting time frame

The 12% solution described

Here we will only explore a version of the 12% solution. There is a more involved solution in the book.

The book builds up a strategy step by step in a nice and easy way. Here we will only explore the solution, which should return 12% on average over 10 years.

Simply explained.

There is a 60-40 percent assets-hedge split.

The hedge is kept in **TLT** and the assets in one of the following **SPY**, **IWM**, **MDY**, **QQQ,** or cash.

In the end of every month you can change your holding between one of them. The decision is based on a 3-month lookback of the best performing choice (including the 0-return of cash).

Implementing the 12% solution

This is a monthly strategy. We only make changes on a monthly basis and based on a 3-month lookback. Hence, we only need monthly data for our backtesting.

That sounds like something new and something we like to try.

Luckily, the **Yahoo! Finance API** can help us there.

But first, we need to import some libraries.

In [1]:
```
import pandas_datareader as pdr
import datetime as dt
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook
```

Then we read the monthly data.

```
In [2]: tickers = ['SPY', 'IWM', 'MDY', 'QQQ', 'TLT']

        start = dt.datetime(2007, 1, 1)
        end = dt.datetime(2021, 1, 1)

        data = pdr.get_data_yahoo(tickers, start, end, interval='m')

        data = data['Adj Close']
```

Notice the **interval='m'**, which will read monthly data instead of daily data.

When we create our strategy, we should also have the option to hold cash. This can be modeled by adding a column with no growth.

```
In [3]: data['Short'] = 1.0
        data = data[['SPY', 'IWM', 'MDY', 'QQQ', 'Short', 'TLT']]
```

Here we call the column **Short** and assign the full column to the value 1. This will give a 0 in return and no growth if we keep money in it.

For convenience, we reorder the columns on the second line.

As usual, we calculate the log returns.

```
In [4]: log_returns = np.log(data/data.shift())
```

Now we need to find the return of the 12% solution.

We need to figure out which assets retuned the most in the last 3 months.

```
In [5]: strat = log_returns[['SPY', 'IWM', 'MDY', 'QQQ', 'Short']].copy()
        rolling_sum = strat.copy()

        for ticker in ['SPY', 'IWM', 'MDY', 'QQQ', 'Short']:
            rolling_sum[ticker] = rolling_sum[ticker].rolling(3).sum()
```

We simply calculate the rolling sum over the last 3 months for all of them. Notice, we have excluded **TLT** from our calculations, as it will only have a 40% holding and not change.

This makes us ready to calculate the monthly return.

```
In [6]: rtn = strat[rolling_sum.apply(lambda x: x == rolling_sum.max(axis=1)).shift()].sum(axis=1)*.6
        rtn = rtn + log_returns['TLT']*.4
```

Yes, that is it. On the first line we simply take the maximum of the **rolling_sum**, which contains the last 3 months return. We shift it, as we can first make the decision in the end of the month (we still do not know how to look into the future).

We multiply it by 0.6 and add the log return of **TLT** multiplied by 0.4 to make the split.

Let's evaluate this.

Project – Backtesting the 12% solution

This is basically adjusting last chapters code to monthly based data.

Step 1

Adjust the calculation function to use monthly data.

```
In [7]: def calculate(log_return, start, end):
            years = int(end) - int(start) + 1

            data = log_return.loc[start:end]

            cagr = np.exp(data.sum())**(1/years) - 1

            norm = data.cumsum().apply(np.exp)

            rolling_max = norm.cummax()
            monthly_drawdown = norm/rolling_max - 1
            drawdown = monthly_drawdown.cummin().iloc[-1]

            volatility = data.std()*(12**.5)

            return cagr, drawdown, volatility
```

We make sure the drawdown is looking at the full picture and the volatility is now over the last 12 months and not 252 trading days.

Step 2

Copy the code of the visualization from last chapter.

```
In [8]: def visualize(backtest, spy, start, end):
            def x_titles(spy_val, bt_val):
                spy_str = str(round(spy_val*100, 1))
                bt_str = str(round(bt_val*100, 1))
                return ['SPY\n' + spy_str + '%', 'Backtest\n' + bt_str + '%']

            spy_cagr, spy_drawdown, spy_vol = calculate(spy, start, end)
            bt_cagr, bt_drawdown, bt_vol = calculate(backtest, start, end)

            fig, ax = plt.subplots(2, 2)

            spy.loc[start:end].cumsum().apply(np.exp).plot(ax=ax[0, 0])
            backtest.loc[start:end].cumsum().apply(np.exp).plot(ax=ax[0, 0], label='Backtest', c='c')
            ax[0, 0].legend()
            ax[0, 0].set_xticks([start, end])

            x = x_titles(spy_cagr, bt_cagr)
            ax[0, 1].bar(x, [spy_cagr, bt_cagr], color=['b', 'c'])
            ax[0, 1].set_title("CAGR")

            x = x_titles(spy_drawdown, bt_drawdown)
            ax[1, 0].bar(x, [spy_drawdown, bt_drawdown], color=['b', 'c'])
            ax[1, 0].set_title("Drawdown")

            x = x_titles(spy_vol, bt_vol)
            ax[1, 1].bar(x, [spy_vol, bt_vol], color=['b', 'c'])
            ax[1, 1].set_title("Volatility")

            plt.tight_layout()
```

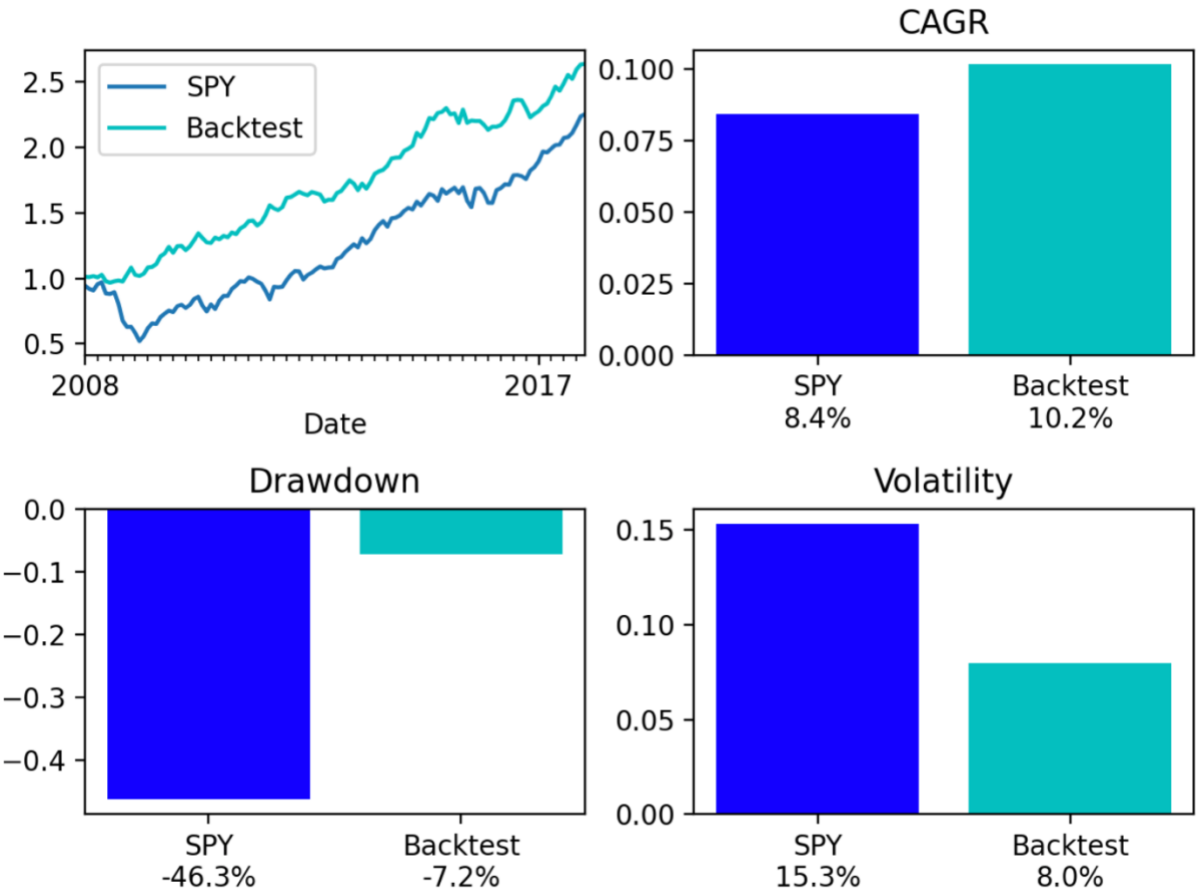
Yes, this code does not need adjustment.

Step 3

Let's do some backtesting. First for the period 2008 and 10 years forward.

```
In [9]: visualize(rtn, log_returns['SPY'], '2008', '2017')
```

Resulting in the following.



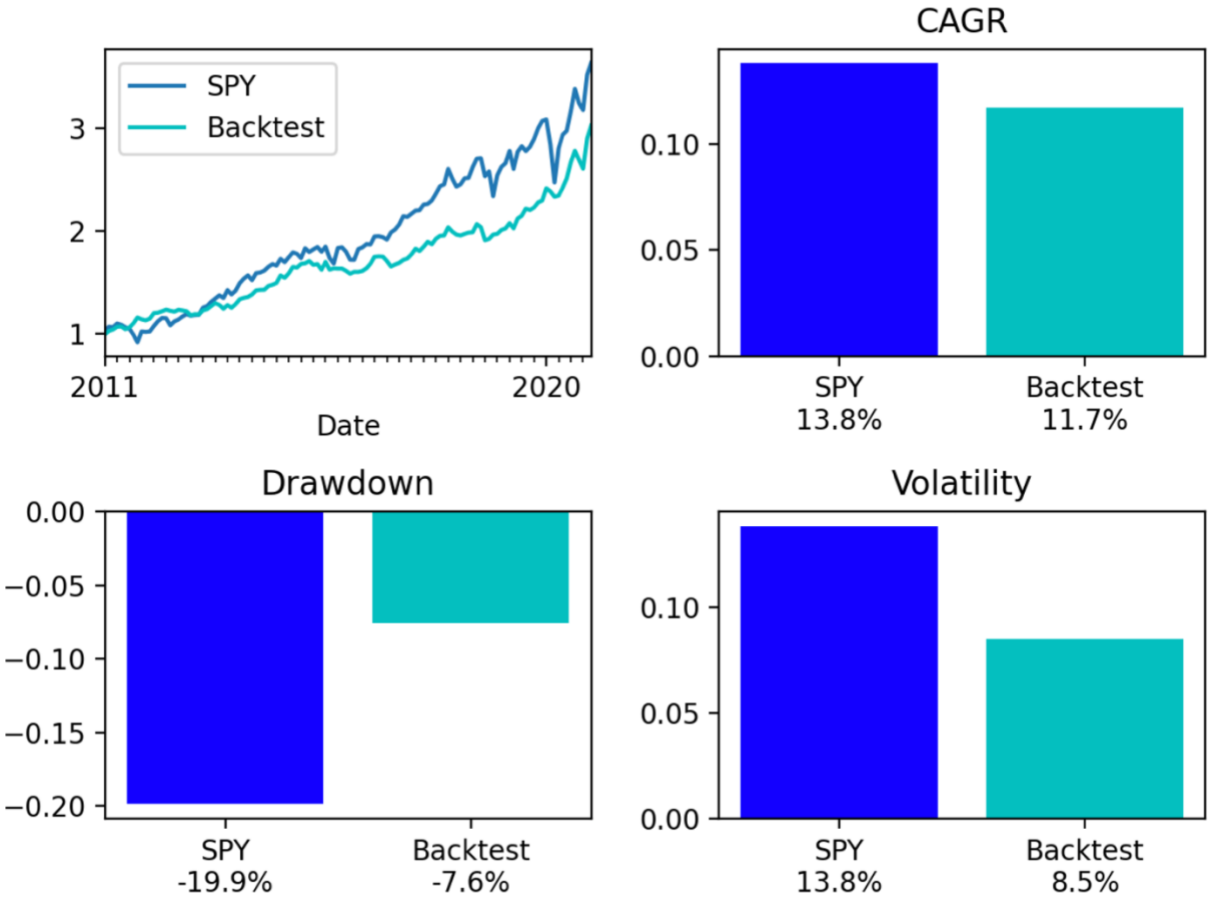
Well, on paper it looks pretty solid here. A return of 10.2%, a low **maximum drawdown** (7.2%) and low **volatility** (8.0%).

Not the fully 12% the solution is named after, but we have left out some details, like rebalancing which actually can add some.

Let's try it for 2011 and 10 years forward.

```
In [10]: visualize(rtn, log_returns['SPY'], '2011', '2020')
```

This results in the following.



Still good. A return of 11.7%, though during this 10-year period the **SPY** returned 13.8% (calculated on a monthly basis). Low drawdown and volatility.

Remember the comments from last chapter about the evaluating over 2008. It gives a great advantage to beat 2008.

Summary

In this chapter we have explored a version of the **12% solution** (not the final version in the book). We have found it is not by default outperforming the return of the last 10 years, but still keeping the risk in form of **maximum drawdown** and **volatility** lower than the **SPY**.

Next Steps – Free Video Courses

The purpose of this **eBook** is to make you able to make your own backtesting. This way you do not depend on other people's impressive figures from a strategy. You can calculate them yourself and verify them over other periods.

My experience shows that most simple and impressive looking strategies are not as impressive when you do the backtesting in a broader spectrum.

Most backtesting shared with the public only shows the best angle of an investment strategy.

Free Online video courses

Do you want to learn more about Python for financial analysis?

…and for FREE?

Then I have created two free courses with financial analysis.

Python for Finance: Risk and Return

Learn **Python for Finance** with Risk and Return with **Pandas** and **NumPy** (Python libraries) in this 2.5 hour free 8-lessons online course.

The 8 lessons will get you started with **technical analysis** for Risk and Return using **Python** with **Pandas** and **NumPy**.

The 8 lessons

- Introduction to **Pandas** and **NumPy** – Portfolios and Returns
- **Risk** and **Volatility** of a stock – **Average True Range**
- **Risk** and **Return** – **Sharpe Ratio**
- **Monte Carlo Simulation** – Optimize portfolio with Risk and Return
- **Correlation** – How to balance portfolio with Correlation
- **Linear Regression** – how X causes Y

- **Beta** – a measure of a stock's volatility in relation to the overall market.
- **CAPM** – Relationship between systematic risk and expected return

Read more on the course page and see the video lectures.

The code is available on **GitHub**.

Financial Data Analysis with Python

Learn **Python for Financial Data Analysis** with **Pandas** (Python library) in this 2 hour free 8-lessons online course.

The 8 lessons will get you started with **technical analysis** using **Python** and **Pandas**.

The 8 lessons

- Get to know **Pandas** with Python – how to get historical stock price data.
- Learn about **Series** from **Pandas** – how to make calculations with the data.
- Learn about **DataFrames** from **Pandas** – add, remove and enrich the data.
- Start visualize data with **Matplotlib** – the best way to understand price data.
- Read data from **APIs** – read data directly from pages like **Yahoo! Finance** the right way.
- Calculate the **Volatility** and **Moving Average** of a stock.
- Technical indicators: **MACD** and **Stochastic Oscillator** – easy with **Pandas**.
- Export it all into **Excel** – in multiple sheets with color formatted cells and charts.

Read more on the course page and see the video lectures.

The Code is available on **GitHub**.

A 21-hour course Python for Finance

Did you know that the No.1 killer of investment return is emotion?

Investors should not let fear or greed control their decisions.

How do you get your emotions out of your investment decisions?

A simple way is to perform objective financial analysis and automate it with Python!

Why?

- Performing financial analysis makes your decisions objective - you are not buying companies that your analysis did not recommend.
- Automating them with Python ensures that you do not compromise because you get tired of analyzing.
- Finally, it ensures that you get all the calculation done correctly in the same way.

Does this sound interesting?

- Do you want to learn how to use Python for financial analysis?
- Find stocks to invest in and evaluate whether they are underpriced or overvalued?
- Buy and sell at the right time?

This course will teach you how to use Python to automate the process of financial analysis on multiple companies simultaneously and evaluate how much they are worth (the intrinsic value).

You will get started in the financial investment world and use data science on financial data.

Read more on the course page.

Feedback

Any feedback and suggestions are welcome.

You can contact me on

- **Twitter: @PythonWithRune**
- **Facebook: @learnpythonwithrune**
- **Email: learnpythonwithrune@gmail.com**

Disclaimer

The content of this eBook and any associated resources are provided for educational purposes only. You assume all risks and costs associated with any trading you choose to take.

Version history

Version 0.9.2.1 – updates (2021.03.20)
Version 0.9.2 – updates (2021.03.15)
Version 0.9.1 – updates (2021.03.14)
Version 0.9 – first release (2021.03.13)