Table of Contents

DIFFERENCES FROM THE BOOK OF BACK-TESTS

The Book of Trading Strategies takes a different approach from the Book of Back-tests, published in 2020. First, the focus is purely on technical strategies rather than discussing strategies from different fields as seen in the Book of Back-tests. Meaning that technical analysis is the protagonist of this book. The second particularity is that we will dive deeper into the code through fully replicable examples and how the results are calculated. Even though this book will not show any results as that is a function of many variables (namely transaction costs, risk management, risk appetite, etc.), we will still see how to evaluate performance and how to code the metrics and the equity curves so that we remain in the context of back-testing, only this time, it is up to you to come up with the results. We will also see a new part called Structured Strategies where I will present how different indicators are combined so that they form quality signals. I have chosen the ones that seem to add value but as mentioned above, the results are left for you to analyze and improve. The last thing is that the first chapter will fully introduce the concepts of Python and everything necessary to make the data analysis task possible and freely accessible.

The best way to exploit the book is to understand why it is written. The goal is to introduce many indicators, strategies, and patterns into the mind of the reader where she will later brainstorm and combine whatever is interesting so that a profitable strategy is formed. To remove the aspect of spoon-feeding, the results have been removed so that we concentrate on the core of the strategies, otherwise, we would be checking out results and basing our trading decisions on them. Nevertheless, the book will see the addition of a GitHub repository to harbor all the code seen in the book so that replicability becomes simpler.

Best of Luck!

Sofien KAABAR.

PART 1

INTRODUCTION TO PYTHON AND DATA ANALYSIS

The book will be composed of simple sections that deal with a certain type of strategy either price action, on a technical indicator, or even a timing pattern. Unlike the first edition of the first book, this edition will not present results but rather the full way leading up to the results and this is done in this part. This way, everyone can compare and tweak the expected return and risk as wanted. Undoubtedly, this will remove any bias from the strategies presented and will encourage the reader to apply the strategies herself. Part 2 will deal directly with the strategies following a specific chronological order composed of:

- Presenting the indicator or the technique in a short way, then providing the full code to calculate the required elements needed to use the strategy. In the case of a technical indicator, the code presented will simply be the indicator's code and how it can be implemented easily.
- Discussing the strategy and the conditions needed to get a signal.
- Presenting any risks or recommendations associated with using the strategy if needed.

It is very important to grasp the aim of the book as it is certainly not an encyclopedia that will give out golden eggs since the results are not provided. It must be used as an initiation to create strategies in Python as well as thinking outside the box with regards to trading. The last part of the book will present some structured strategies which are composed of two or more indicators in order to reach a signal. This should help the reader understand how to combine different elements to raise the conviction of a certain trade. The most important part of the book is the current one as it has everything needed from importing historical data to structuring data properly and defining the performance functions which are standard across all strategies. A GitHub repository under the following link: https://github.com/sofienkaabar/the-book-of-more-back-tests can be accessed to view the strategies.

1.1 IMPORTING HISTORICAL DATA USING METATRADER5

In research, we need the idea generation phase to have enough time to deliver good results and this cannot be done if we have a long process of transforming the ideas into visible results. Automating the process of importing historical data is valuable as it allows us to focus on more important non-repetitive tasks. By creating a few simple functions, we will be able to call thousands of OHLC[1] data in a matter of seconds. For this, we need to download a library and a software described below. But first, we need to download a Python interpreter such as SPYDER[2], PyCharm, or Jupyter.

1.1.1 METATRADER5 SOFTWARE & LIBRARY

One of the most famous trading platforms in the retail community is the MetaTrader5 software. It is a powerful tool that comes with its own programming language and its huge online community support. Most importantly, it offers the possibility to export its historical short-term and long-term FX data. The first thing we need to do is to simply download the software from the official website. Then, after creating the unlimited demo account, we are ready to import the library in Python that allows us to import the OHLC data from MetaTrader5. A library is a group of structured functions that can be imported into the Python interpreter from where we can call and use the ones we want. The easiest way to install the MT5 library is to go to the Python prompt on our computer and type:

```
pip install MetaTrader5
```

This should install the library in our local Python. Now, we want to import it to the Python interpreter such as SPYDER so that we can use it. Let us import all the libraries we will be using for this:

[1] Open, High, Low, and Close values are the most basic form of financial historical data.
[2] https://www.anaconda.com/

```
import datetime # Date acquiring
import pytz # Time zone management
import pandas as pd # Mostly for Data frame manipulation
import MetaTrader5 as mt5 # Importing OHLC data
import matplotlib.pyplot as plt # Plotting charts
import numpy as np # Mostly for array manipulation
```

Anything that comes after "as" is a shortcut. The plt shortcut is there so that each time we want to call a function from that library we do not have to type the full matplotlib.pyplot statement. The official documentation for the library can be found in the official website[3].

1.1.2 CREATING THE IMPORTING FUNCTIONS STEP-BY-STEP

Our aim is to have OHLC arrays of the currency pairs we choose. The first thing we can do is to select which time frame we want to import. Let us suppose that there are only two frames, the 30-minute and the hourly bars. We can therefore create variables that hold the statement to tell the MetaTrader5 library which frame we want.

```
# Choosing the 30-minute time frame
frame_M30  = mt5.TIMEFRAME_M30
# Choosing the hourly time frame
frame_H1   = mt5.TIMEFRAME_H1
```

Then, by staying in the spirit of importing variables, we can define the variable that states what date is it now. This helps the algorithm know the stopping date of the import. We can do this by the simple line of code below.

```
# Defining the variable now to give out the current date
now = datetime.datetime.now()
```

[3] https://www.mql5.com/en/docs/integration/python_metatrader5

Note that these code snippets are better used chronologically, hence, I encourage you to copy them in order and then execute them one by one so that you understand the evolution of what you are doing. The below is a function that holds whichever assets we want. Generally, I use 10 or more but for simplicity, let us consider that there are only two currency pairs: EURUSD and USDCHF.

```
def asset_list(asset_set):
  if asset_set == 1:
    assets = ['EURUSD', 'USDCHF']
  return assets
```

The below establishes a connection to MetaTrader5, applies the current date, and extracts the needed data. Notice the arguments year, month, and day. These will be filled by us to select from when we want the data to start. Note, I have inputted Europe/Paris as my time zone, you should use your time zone to get more accurate data.

```
def get_quotes(time_frame, year = 2005, month = 1, day = 1, asset = "EURUSD"):
  # Establish connection to MetaTrader 5
  if not mt5.initialize():
    print("initialize() failed, error code =", mt5.last_error())
    quit()
  timezone = pytz.timezone("Europe/Paris")
  utc_from = datetime.datetime(year, month, day, tzinfo = timezone)
  utc_to = datetime.datetime(now.year, now.month, now.day + 1, tzinfo = timezone)
  rates = mt5.copy_rates_range(asset, time_frame, utc_from, utc_to)
  rates_frame = pd.DataFrame(rates)
  return rates_frame
```

And finally, the last function we will use is the one that uses the get_quotes function and then cleans the results so that we have a nice array. We have selected data since January 2019 as shown.

```
def mass_import(asset, horizon):
 if horizon == 'M30':
   data = get_quotes(frame_M30, 2019, 1, 1, asset = assets[asset])
   data = data.iloc[:, 1:5].values
   data = data.round(decimals = 5)
 return data
```

Finally, we are done building the blocks necessary to import the data. To import EURUSD OHLC historical data, we simply use the below code line:

```
# Choosing the horizon
horizon = 'M30'
# Creating an array called EURUSD having M30 data since 2019
EURUSD = mass_import(0, horizon)
```

Now, we have a variable called EURUSD with historical OHLC data inside. The first phase is done and now, we will proceed to data analysis. The utility of coding is bigger than ever nowadays. A trader or an analyst must know at least the basics of how algorithms work and must also know some functions and the logic behind the calculations. Many analysts use indicators but do not know how they are calculated nor how they can be coded. As this book presents many indicators, we need to first understand the basics, and prepare the necessary functions to analyze the data.

1.2 BASIC SYNTAX & REQUIRED FUNCTIONS

Python is rapidly growing to be the go-to language when it comes to creating and testing strategies. It is a fast, versatile, and easy language to learn that will allow you to perform many operations. Basically, for the most part of the book, we will be using 3 known libraries to handle, manipulate, and plot our data. Let us start with the two similar libraries, numpy[4] and pandas[5] (which will be used to a much lesser extent than numpy).

[4] https://docs.scipy.org/doc/numpy/user/whatisnumpy.html
[5] https://pandas.pydata.org/pandas-docs/stable/

Every python developer must imperatively know a thing or two about them. We use numpy to manipulate arrays and perform operations and other useful mathematical functions, it is the go-to package when you are dealing with time series or any other dataset as it is fast due to vectorization. As for pandas, it is also similar to numpy with some differences and is mainly for reading and slicing data frames and excel files among other functions such as calculating exponential moving averages. Of course, these are simplistic definitions, and the libraries can do much more than the described tasks.

In python, we import a library using the import statement followed by the name of the library and an optional shortcut or nickname for that library so you can call it later.

Remember, a library can be considered as a box of functions where you can just load them and use whichever ones you want. If the field of mathematics was a library, then addition and multiplication would be functions that are inside of it.

The word np is the standard numpy shortcut while pd is the standard pandas shortcut. Also, plt is the standard matplotlib shortcut.

```
# Importing the necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

After importing the libraries, we now have the necessary tools to start analyzing the time series. Generally, with financial time series, we will have what we call OHLC data. This is short for Open, High, Low, and Close. The data should be in the form of 4 columns. This will be our basic data structure. Our job in the full research process is to add columns where we place technical indicators and buy/sell orders as well as trading results.

Let us start by adding a column. Remember, we have four columns containing OHLC data and we want to add a fifth column containing zero values so that we populate it with values later. To add a new empty column, we can use the following function.

```
# The function to add a certain number of columns
def adder(Data, times):
    for i in range(1, times + 1):
        z = np.zeros((len(Data), 1), dtype = float)
        Data = np.append(Data, z, axis = 1)
    return Data
# Using the function to add 5 columns
my_data = adder(my_data, 5)
```

Sometimes, we need to remove columns when we have more than enough or when we want to clean out the data.

```
# The function to delete a certain number of columns
def deleter(Data, index, times):
    for i in range(1, times + 1):
        Data = np.delete(Data, index, axis = 1)
    return Data
# Using the function to delete 2 columns starting from column 5
my_data = deleter(my_data, 5, 2)
```

It is worth mentioning that Python starts indexing at zero, therefore column 5 is chronologically the sixth column and not the fifth column. What about rows? Sometimes, we will calculate some indicators that start from the 100th line, therefore, the first 100 lines will have no use. The next page shows a function to delete several rows from an array that is descending in order. This means that the first values are the earliest ones with the latest values below being the latest ones.

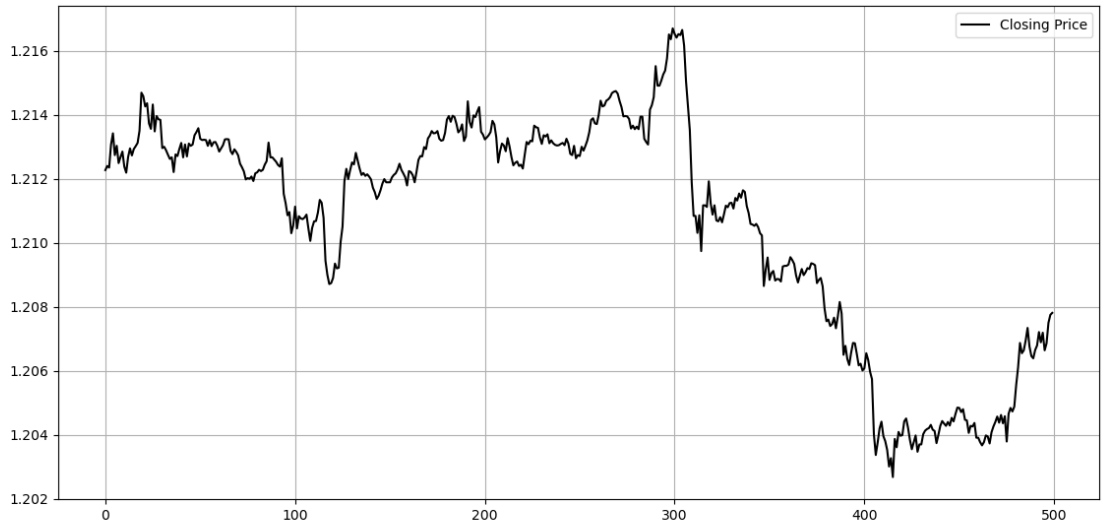
```
# The function to delete a certain number of rows
def jump(Data, jump):
   Data = Data[jump:, ]
   return Data
# Using the function to delete the first 10 rows of an array
my_data = jump(my_data, 10)
```

1.3 DATA VISUALIZATION & PLOTTING

How about we try visualizing the data now? This is very easy in Python and does not require any sophisticated code. Let us consider as usual that we have an OHLC array, and we want to visualize the latest 500 closing prices in black with a legend that describes what we are seeing. We also want to see a grid on the chart to facilitate the visual interpretation.

```
# Plotting
plt.plot(my_data[-500:, 3], color = 'black', label = 'Closing Price')
# Creating the grid
plt.grid()
# Calling the legend Function
plt.legend()
```

The above function says that we want to plot the latest 500 values (hence, the negative sign) of the fourth column (hence, the index 3 as Python starts indexing from zero). We also want the color to be black which is why the argument of color equals what we want it to be. The label is a string (text) of what we want to describe. This is what will appear on the chart in the form of a text box called later by the legend function. Finally, the grid function is self-explanatory. It is optional but it adds a visual effect to the chart.



1.4 WRITING ORDERS & CALCULATING PROFIT

We want to create trading strategies out of technical indicators or patterns. This means we have to do two things after importing the OHLC data:

- Create the technical indicators in the columns next to the OHLC data.
- Create the conditions of buy and sell orders and populate the required columns.

Let us code the most basic of technical indicators, the simple moving average. Having already four columns containing the OHLC data, we will use the adder function to add three new columns, the column where the moving average is located and then the buy column (bullish signal) followed by the sell column (bearish signal).

```
def ma(Data, lookback, what, where):
    for i in range(len(Data)):
        try:
            Data[i, where] = (Data[i - lookback + 1:i + 1, what].mean())
        except IndexError:
            pass
    return Data
```

```
# Calling a 20-period Moving Average function based on closing prices (index = 3) and to be populated on the fifth column (index = 4)
my_data = ma(my_data, 20, 3, 4)
# The signal function that populates the buy/sell columns following the crossovers
def signal(Data, closing, moving_average, buy, sell):
  for i in range(len(Data)):
    if Data[i, closing] > Data[i, moving_average and Data[i - 1, closing] < Data[i - 1, moving_average]:
      Data[i, buy] = 1
    if Data[i, closing] < Data[i, moving_average and Data[i - 1, closing] > Data[i - 1, moving_average]:
      Data[i, sell] = -1
# Using the signal function
signal(my_data, 4, 5, 6)
```

The above signal function is what will be used to populate the sixth (index = 5) and seventh (index = 6) columns. The proxy for a buy signal or order will be 1 while the proxy for a sell (short) signal or order will be -1. Note that the strategies are all long/short, meaning that we can either buy and sell higher or sell short now and buy back lower hence, having two possible ways of benefitting from the asset's direction. At the moment, we should have a data array with the following elements:

- OHLC data populating the first four columns.
- A 20-period moving average populating the fifth column.
- Buy orders with the value of 1 populating the column 6.
- Sell orders with the value of -1 populating column 7.

This is the basic strategy of when the market price surpasses its moving average, a buy signal is generated because it is implied that a new bullish trend has started. Similarly, when the market price breaks its moving average, a sell signal is generated because it is implied that a new bearish trend has started. We will discuss it more in detail in the second part of the book.

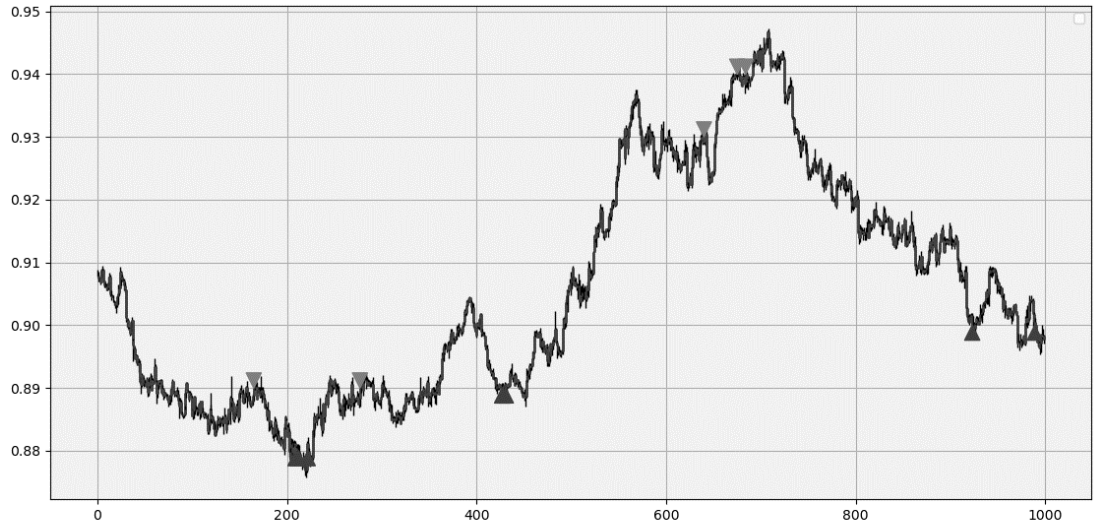
If we back-test this strategy on the hourly values of the EURUSD since 2010, we can get the following equity curve (which we will later see how we have calculated it). Note that for simplicity, I will omit the proxy for transaction costs.

As expected from a simple strategy (which is weirdly highly praised by technical analysts), the returns are disastrous. We cannot really say that a 20-period moving average crossover adds much to our trading framework. Remember, it is your duty to learn to code the strategies and back-test them using your favorite conditions.

1.5 PLOTTING SIGNAL CHARTS

A quick an easy way to plot the signals is to place upward pointing arrows whenever we have a buy signal and to place downward pointing arrows whenever we have a sell signal. We have to define the function and then apply it on the signals generated in the columns.

```
def ohlc_plot(Data, window, name):
    Chosen = Data[-window:, ]
    for i in range(len(Chosen)):
        plt.vlines(x = i, ymin = Chosen[i, 2], ymax = Chosen[i, 1], color = 'black', linewidth = 1)
        if Chosen[i, 3] > Chosen[i, 0]:
            color_chosen = 'blue'
            plt.vlines(x = i, ymin = Chosen[i, 0], ymax = Chosen[i, 3], color = color_chosen, linewidth = 3)
        if Chosen[i, 3] < Chosen[i, 0]:
            color_chosen = 'brown'
            plt.vlines(x = i, ymin = Chosen[i, 3], ymax = Chosen[i, 0], color = color_chosen, linewidth = 3)
        if Chosen[i, 3] == Chosen[i, 0]:
            color_chosen = 'black'
            plt.vlines(x = i, ymin = Chosen[i, 3], ymax = Chosen[i, 0], color = color_chosen, linewidth = 3)
def signal_chart_ohlc_color(Data, name, onwhat, what_bull, what_bear, window = 1000):
    Plottable = Data[-window:, ]
    fig, ax = plt.subplots(figsize = (10, 5))
    ohlc_plot(Data, window, '')
    for i in range(len(Plottable)):
        if Plottable[i, what_bull] == 1:
            x = i
            y = Plottable[i, onwhat]
            ax.annotate(' ', xy = (x, y),
              arrowprops = dict(width = 9, headlength = 11, headwidth = 11, facecolor = 'green', color = 'green'))
        elif Plottable[i, what_bear] == -1:
            x = i
            y = Plottable[i, onwhat]
            ax.annotate(' ', xy = (x, y),
              arrowprops = dict(width = 9, headlength = -11, headwidth = -11, facecolor = 'red', color = 'red'))
    ax.set_facecolor((0.95, 0.95, 0.95))
    plt.legend()
```



The above plot shows signals generated using the function. Unfortunately, to make the book accessible to more people by lowering the price, I had to write it in black and white. However, the colors will be visible when you run them in Python.

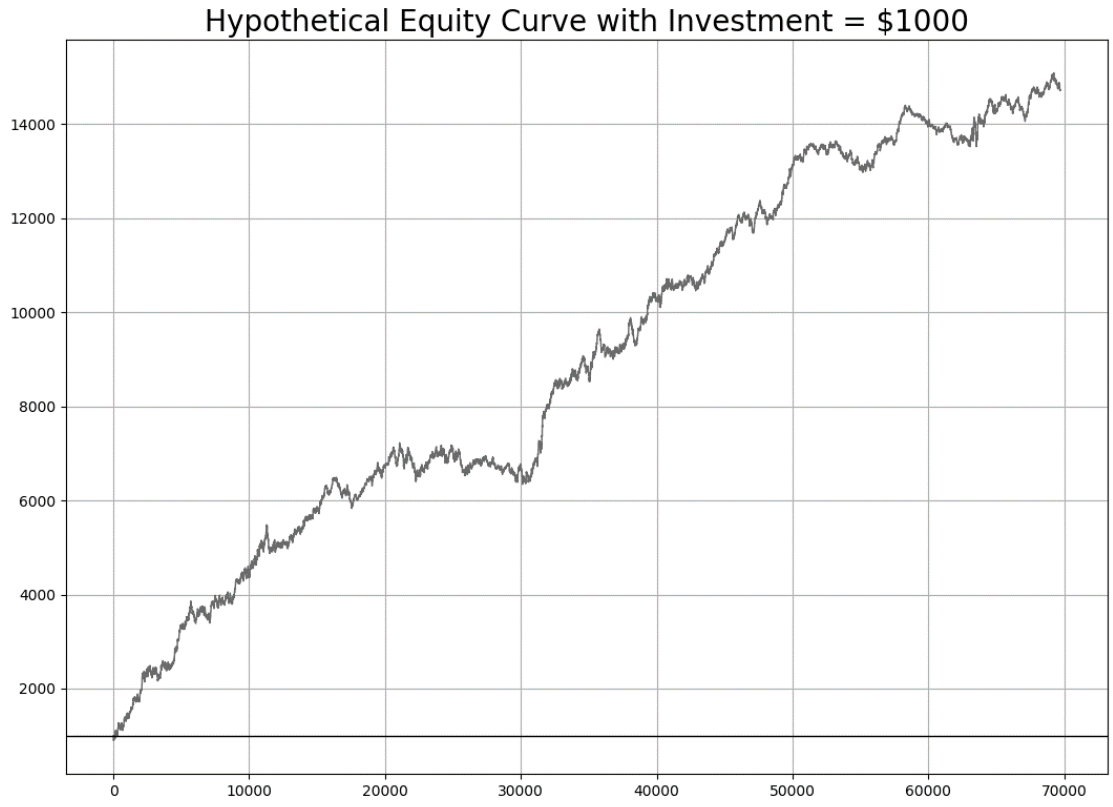
1.6 PERFORMANCE EVALUATION METRICS & PLOTTING EQUITY CURVES

The quickest and probably the preferred visual performance interpretation is the Equity Curve. It is simply the evolution of the balance which invests in the strategy. In other words, it is the cumulative profits and losses when added to the initial investment.

The first step into building the Equity Curve is to calculate the profits and losses from the individual trades we are taking. For simplicity reasons, we can consider buying and selling at the closing prices. This means that when we get the signal from the indicator on close, we initiate the trade on the close until getting another signal where we exit and initiate the new trade. The code to be defined for the profit/loss columns is as follows:

```
def holding(Data, buy, sell, buy_return, sell_return):
for i in range(len(Data)):
    try:
      if Data[i, buy] == 1:
        for a in range(i + 1, i + 1000):
          if Data[a, buy] != 0 or Data[a, sell] != 0:
            Data[a, buy_return] = (Data[a, 3] - Data[i, 3])
              break
            else:
              continue
      elif Data[i, sell] == -1:
        for a in range(i + 1, i + 1000):
          if Data[a, buy] != 0 or Data[a, sell] != 0:
            Data[a, sell_return] = (Data[i, 3] - Data[a, 3])
              break
            else:
              continue
    except IndexError:
      pass
# Using the function
holding(my_data, 6, 7, 8, 9)
```

This will give us columns 8 and 9 populated with the gross profit and loss results from the trades taken at column 6 (buy orders) and at column 7 (short orders). Now, we have to transform them into cumulative numbers so that we calculate the Equity Curve. To do that, we use the indexer function.



The above shows a hypothetical equity curve on the EURUSD. The code is used to generate the curve. Note that the indexer function nets the returns using the estimated transaction cost, hence, the equity curve above is theoretically net of fees. You can define the transaction costs as a variable before. Also, the lot variable is the quantity taken for the trades. At the moment, we can assume that we will be taking equal quantity in all the trades. The investment variable is the starting balance.

```
def indexer(Data, expected_cost, lot, investment):
    # Charting portfolio evolution
    indexer = Data[:, 8:10]
    # Creating a combined array for long and short returns
    z = np.zeros((len(Data), 1), dtype = float)
    indexer = np.append(indexer, z, axis = 1)
    # Combining Returns
    for i in range(len(indexer)):
        try:
          if indexer[i, 0] != 0:
            indexer[i, 2] = indexer[i, 0] - (expected_cost / lot)
          if indexer[i, 1] != 0:
            indexer[i, 2] = indexer[i, 1] - (expected_cost / lot)
        except IndexError:
            pass
    # Switching to monetary values
    indexer[:, 2] = indexer[:, 2] * lot
    # Creating a portfolio balance array
    indexer = np.append(indexer, z, axis = 1)
    indexer[:, 3] = investment
    # Adding returns to the balance
    for i in range(len(indexer)):
        indexer[i, 3] = indexer[i - 1, 3] + (indexer[i, 2])
    indexer = np.array(indexer)
    return np.array(indexer)
# Using the function for a 0.1 lot strategy on $10,000 investment
expected_cost = 0.5 * (lot / 10000) # 0.5 pip spread
investment = 10000
lot = 10000
equity_curve = indexer(my_data, expected_cost, lot, investment)
```

```
plt.plot(equity_curve[:, 3], linewidth = 1, label = 'EURUSD)
plt.grid()
plt.legend()
plt.axhline(y = investment, color = 'black', linewidth = 1)
plt.title('Hypothetical Strategy', fontsize = 20)
```

The focus now switches to performance metrics where they are described below followed by the code to calculate them right after executing the indexer function.

- **The Hit Ratio**

We will quickly present the main ratios and metrics before presenting a full performance function that outputs them all together. Hence, the below discussions are mainly informational, if you are interested by the code, you can find it at the end. The hit ratio is extremely easy to use. It is simply the number of winning trades over the number of the trades taken in total. For example, if we have 1359 trades over the course of 5 years and we have been profitable in 711 of them, then our hit ratio (accuracy) is 711/1359 = 52.31%.

$$Hit\ Ratio = \frac{Number\ of\ winning\ trades}{Number\ of\ winning\ trades + Number\ of\ losing\ trades}$$

- **The Net Profit**

The net profit is simply the last value in the Equity Curve net of fees minus the initial balance. It is simply the added value on the amount we have invested in the beginning. If we start with $1,000 and finish with $1,800, then our net profit is $800.

- **Average Gain & Average Loss**

A quick glance on the average gain across the trades and the average loss can help us manage our risk better. For example, if our average gain is $1.20 and our average loss is $4.02, then we know that something is not right as we are risking way too much money for way too little gain. We have to adjust our risk-reward ratio when confronted with this issue.

- **Profit Factor**

This is a relatively quick and straightforward method to compute the profitability of the strategy. It is calculated as the total gross profit over the total gross loss in absolute values, hence, the interpretation of the profit factor (also referred to as profitability index in the jargon of corporate finance) is how much profit is generated per $1 of loss. Profitable strategies must have a profit factor greater than 1.00.

$$Profit\ factor = \frac{Gross\ profit}{|Gross\ loss|}$$

- **Expectancy**

Expectancy is a flexible measure that is composed of the average win/loss and the hit ratio. It provides the expected profit or loss on a dollar figure weighted by the hit ratio. The win rate is what we refer to as the hit ratio in the below formula, and through that, the loss ratio is 1 — hit ratio. This is an amazing metric introduced by Laurent Bernut a few years ago. All the credits go to him, an extraordinary market practitioner.

$$Expectancy = (Average\ win\ x\ Hit\ ratio) - (Average\ loss\ x\ (1 - Hit\ ratio))$$

- **Putting It All Together**

Now, we are ready to have the above metrics shown at the same time. After calculating the indexer function, we can use the performance function to give us the metrics:

```
def performance(indexer, Data, name):
    # Profitability index
    indexer = np.delete(indexer, 0, axis = 1)
    indexer = np.delete(indexer, 0, axis = 1)
    profits = []
    losses  = []
    np.count_nonzero(Data[:, 7])
    np.count_nonzero(Data[:, 8])
```

```
for i in range(len(indexer)):
    if indexer[i, 0] > 0:
        value    = indexer[i, 0]
        profits  = np.append(profits, value)
    if indexer[i, 0] < 0:
        value    = indexer[i, 0]
        losses   = np.append(losses, value)
# Hit ratio calculation
hit_ratio = round((len(profits) / (len(profits) + len(losses))) * 100, 2)
realized_risk_reward = round(abs(profits.mean() / losses.mean()), 2)
# Expected and total profits / losses
expected_profits = np.mean(profits)
expected_losses  = np.abs(np.mean(losses))
total_profits    = round(np.sum(profits), 3)
total_losses     = round(np.abs(np.sum(losses)), 3)
# Expectancy
expectancy    = round((expected_profits * (hit_ratio / 100)) \
            - (expected_losses * (1 - (hit_ratio / 100))), 2)
# Largest Win and Largest Loss
largest_win = round(max(profits), 2)
largest_loss = round(min(losses), 2)
# Total Return
indexer = Data[:, 10:12]
# Creating a combined array for long and short returns
z = np.zeros((len(Data), 1), dtype = float)
indexer = np.append(indexer, z, axis = 1)
```

```
# Combining Returns
for i in range(len(indexer)):
   try:
    if indexer[i, 0] != 0:
      indexer[i, 2] = indexer[i, 0] - (expected_cost / lot)
    if indexer[i, 1] != 0:
      indexer[i, 2] = indexer[i, 1] - (expected_cost / lot)
   except IndexError:
     pass
# Switching to monetary values
indexer[:, 2] = indexer[:, 2] * lot
# Creating a portfolio balance array
indexer = np.append(indexer, z, axis = 1)
indexer[:, 3] = investment
# Adding returns to the balance
for i in range(len(indexer)):
   indexer[i, 3] = indexer[i - 1, 3] + (indexer[i, 2])
indexer = np.array(indexer)
total_return = (indexer[-1, 3] / indexer[0, 3]) - 1
total_return = total_return * 100
print('-----------Performance-----------', name)
print('Hit ratio      = ', hit_ratio, '%')
print('Net profit     = ', '$', round(indexer[-1, 3] - indexer[0, 3], 2))
print('Expectancy     = ', '$', expectancy, 'per trade')
print('Profit factor  = ' , round(total_profits / total_losses, 2))
print('Total Return   = ', round(total_return, 2), '%')
print('')
print('Average Gain   = ', '$', round((expected_profits), 2), 'per trade')
print('Average Loss   = ', '$', round((expected_losses * -1), 2), 'per trade')
print('Largest Gain   = ', '$', largest_win)
```

```
    print('Largest Loss    = ', '$', largest_loss)
    print('')
    print('Realized RR     = ', realized_risk_reward)
    print('Minimum        =', '$', round(min(indexer[:, 3]), 2))
    print('Maximum         =', '$', round(max(indexer[:, 3]), 2))
    print('Trades         =', len(profits) + len(losses))
# Using the function
performance(equity_curve, my_data, 'EURUSD)
```

This should give us something like the below:

```
-----------Performance----------- EURUSD
Hit ratio      =  42.28 %
Net profit     =  $ 1209.4
Expectancy     =  $ 0.33 per trade
Profit factor  =  1.01
Total Return   =  120.94 %
Average Gain   =  $ 56.95 per trade
Average Loss   =  $ -41.14 per trade
Largest Gain   =  $ 347.5
Largest Loss   =  $ -311.6
Realized RR    =  1.38
Minimum        = $ -1957.6
Maximum        = $ 4004.2
Trades         = 3697
```

1.7 PLOTTING DOUBLE PANELS

Double panels are used to plot the price action alongside an indicator that transforms its price. For instance, when we use s charting software, we can apply the 14-period RSI on the market price and view it. This will give us two panels. The first (usually on top)

is for the market price while the second (usually on the bottom) is for the indicator. We will consider calculating a normal indicator called the Stochastic Oscillator defined below and then plot it alongside the EURUSD values using the indicator_plot_double function.

The Stochastic Oscillator is calculated using a technique called normalization which allows us to trap the values between 0 and 1 (or 0 and 100 if we wish to multiply by 100). The concept revolves around subtracting the minimum value in a certain lookback period from the current value and dividing by the maximum value in the same lookback period minus the minimum value (the same in the nominator).

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

The Stochastic Oscillator seeks to find oversold and overbought zones by incorporating the highs and lows using the normalization formula as shown below:

$$\%K = \left(\frac{Close - Low}{High - Low}\right) x\ 100$$

An overbought level is an area where the market is perceived to be extremely bullish and is bound to consolidate. An oversold level is an area where the market is perceived to be extremely bearish and is bound to bounce. Hence, the Stochastic Oscillator is a contrarian indicator that seeks to signal reactions of extreme movements.

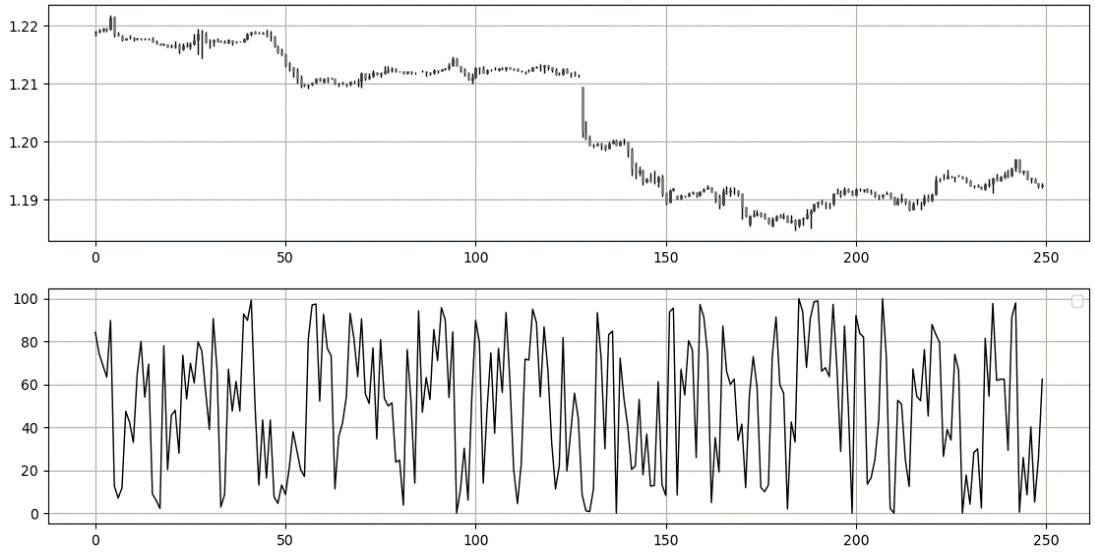
```
def stochastic(Data, lookback, close, where, genre = 'High-Low'):
  # Adding a column
  Data = adder(Data, 1)
  if genre == 'High-Low':
    for i in range(len(Data)):
      try:
        Data[i, where] = (Data[i, close] - min(Data[i - lookback + 1:i + 1, 2])) / (max(Data[i - lookback + 1:i + 1, 1]) - min(Data[i - lookback + 1:i + 1, 2]))
      except ValueError:
        pass
    Data[:, where] = Data[:, where] * 100
    Data = jump(Data, lookback)
  if genre == 'Normalization':
    for i in range(len(Data)):
      try:
        Data[i, where] = (Data[i, close] - min(Data[i - lookback + 1:i + 1, close])) / (max(Data[i - lookback + 1:i + 1, close]) - min(Data[i - lookback + 1:i + 1, close]))
      except ValueError:
        pass
    Data[:, where] = Data[:, where] * 100
    Data = jump(Data, lookback)
  return Data
```

The above code has two components, High-Low and Normalization. The former is the Stochastic Oscillator as it uses the highs and lows in the function and the latter is the normalized value which traps the market prices between 0 and 100 thus ensuring a standardized and stationary version. Now, we need to calculate the Stochastic Oscillator on the OHLC data:

```
my_data = stochastic(my_data, 14, 3, 4, genre = 'High-Low')
```

The next code is the function to plot a double panel plot.

```
def indicator_plot_double(Data, opening, high, low, close, second_panel, window = 250):
    fig, ax = plt.subplots(2, figsize = (10, 5))
    Chosen = Data[-window:, ]
    for i in range(len(Chosen)):
        ax[0].vlines(x = i, ymin = Chosen[i, low], ymax = Chosen[i, high], color = 'black', linewidth = 1)
        if Chosen[i, close] > Chosen[i, opening]:
            color_chosen = 'green'
            ax[0].vlines(x = i, ymin = Chosen[i, opening], ymax = Chosen[i, close], color = color_chosen,
linewidth = 2)
        if Chosen[i, close] < Chosen[i, opening]:
            color_chosen = 'red'
            ax[0].vlines(x = i, ymin = Chosen[i, close], ymax = Chosen[i, opening], color = color_chosen,
linewidth = 2)
        if Chosen[i, close] == Chosen[i, opening]:
            color_chosen = 'black'
            ax[0].vlines(x = i, ymin = Chosen[i, close], ymax = Chosen[i, opening], color = color_chosen,
linewidth = 2)
    ax[0].grid()
    ax[1].plot(Data[-window:, second_panel], color = 'black', linewidth = 1)
    ax[1].grid()
    ax[1].legend()
```



PART 2

TREND-FOLLOWING STRATEGIES

A trend occurs when there is an aggregate direction in the market. We tend to talk about bull markets when the general tendency is upward sloping. This means that the best action to do is to buy and hold or buy the dips in anticipation that the dip is a temporary opportunity for a better price before continuing higher. An example of a bull market is the S&P500 since 2009. Trends are usually visual from a simple graph. Some analysts choose to put more objective conditions to determine a trend such as moving averages and indicators like the Average Directional Index. Trend-following strategies are the ones that seek to detect the trend at its early stage and to ride it as long as possible before it ends. They are more concerned with time in the market than timing the market. This particularity introduces some inherent lag in trend-following strategies because they depend on the movement of price action to confirm the trend, and therefore, the best trend-following strategies are the one with the least false positives and the ones who enter and exit close to the extremes. This part will introduce indicators and systems that are used to follow the trend, namely moving averages and other price transformations. The code and the signal functions are provided, and the full code can be found in the GitHub [6]repository.

[6] https://github.com/sofienkaabar/the-book-of-more-back-tests

SIMPLE MOVING AVERAGE CROSS

"The simplest and most basic trading strategy, I wonder why."

Moving averages help us confirm and ride the trend. They are the most known technical indicator, and this is because of their simplicity and their proven track record of adding value to the analyses. We can use them to find support and resistance levels, stops and targets, and to understand the underlying trend. This versatility makes them an indispensable tool in our trading arsenal. As the name suggests, this is your plain simple mean that is used everywhere in statistics and basically any other part in our lives. It is simply the total values of the observations divided by the number of observations. Mathematically speaking, it can be written down as:

$$SMA = \frac{A_1 + A_2 + \cdots + A_n}{n}$$

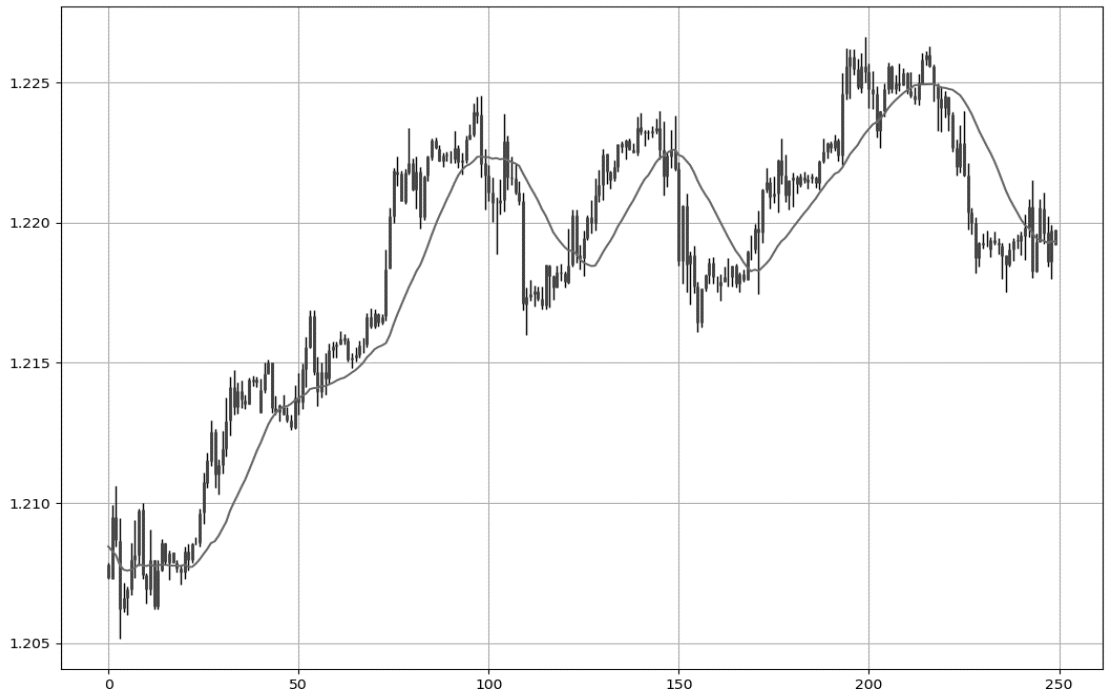
The conditions required for the moving average cross:

- For a bullish (Long) signal, the market price must close above the current moving average while the previous market price closing below the previous moving average.
- For a bearish (Short) signal, the market price must close below the current moving average while the previous market price closing above the previous moving average.

The full Python code to calculate a simple moving average and the signal function can be as follows. Note that it is important to have an OHLC data array and not a data frame.

```
def ma(Data, lookback, close, where):
    Data = adder(Data, 1)
    for i in range(len(Data)):
        try:
            Data[i, where] = (Data[i - lookback + 1:i + 1, close].mean())
        except IndexError:
            pass
    # Cleaning
    Data = jump(Data, lookback)
    return Data

# Using the function
lookback = 50
my_data = ma(my_data, lookback, 3, 4)
```

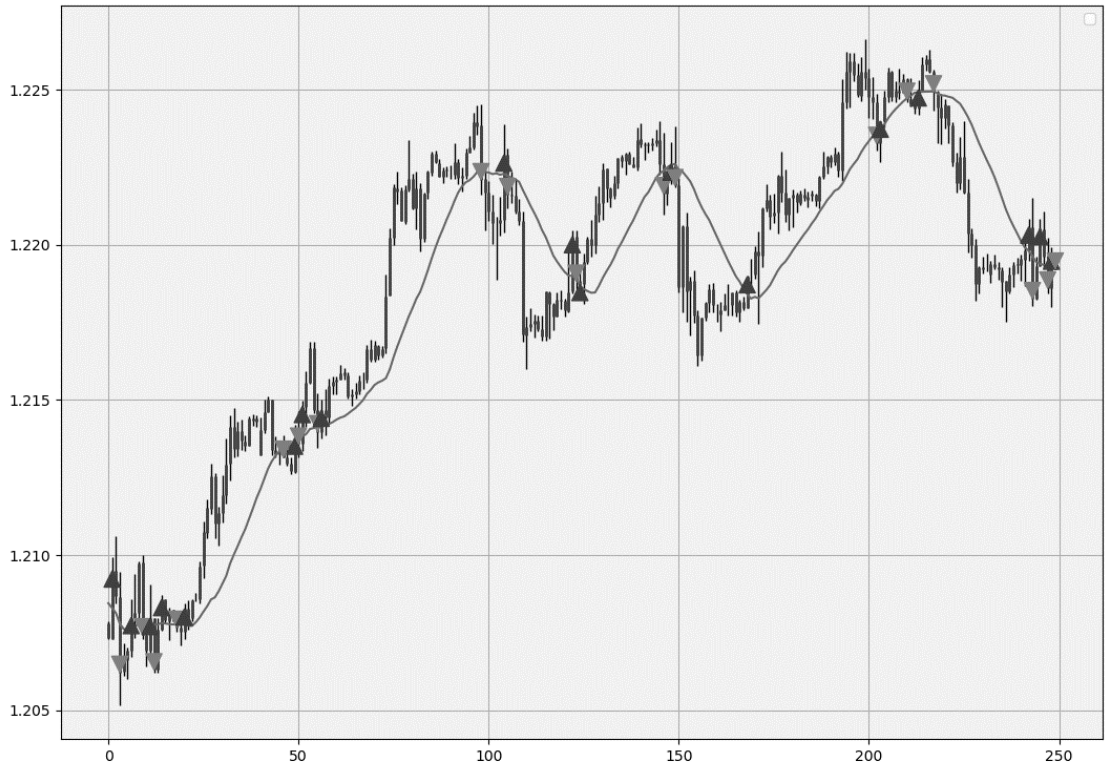
The above snippet shows how to calculate a 50-period simple moving average on the OHLC array of historical data we have recently imported to the Python interpreter.



Now, we are set to create the signal function that allows us to take a look at the signals.

```
def signal(Data, close, ma_column, buy_col, sell_col):
    Data = adder(Data, 2)
    for i in range(len(Data)):
        # Scanning for Bullish signals
        if Data[i, close] > Data[i, ma_column] and Data[i - 1, close] < Data[i - 1, ma_column]:
            Data[i, buy_col] = 1
        # Scanning for Bearish signals
        elif Data[i, close] < Data[i, ma_column] and Data[i - 1, close] > Data[i - 1, ma_column]:
            Data[i, sell_col] = -1
    return Data
my_data = signal(my_data, 3, 4, 5, 6)
```

The above function will populate the buy and sell columns when their respective conditions are met. The below is a sample of the signal chart created using the above conditions.



The rest of the coding work will be based on evaluating the performance metrics and any optimization attempt. We can use the following guide to lead us towards properly evaluating the strategy. For the remainder of the strategies presented in the book, you can follow this guideline, or you can simply copy the code from the GitHub repository.

```
# Calling the indicator's function and leaving column 4 empty in case we want to add another indicator
my_data = ma(my_data, lookback, 3, 5)
# Calling the signal function to input 1's as buy signals in column 6 and -1's as short sell signals in column 7
my_data = signal(my_data, 5, 6, 7)
# Calling the profit and loss function to calculate the differences between the closes of the exits from the entries
holding(my_data, 6, 7, 8, 9)
# Calling the equity curve function to output an array of the balance's evolution through time
lot = 10000
expected_cost    = 0.2 * (lot / 10000)
my_data_eq = equity_curve(my_data, expected_cost, lot, investment)
# Calling the performance function to output the metrics
performance(my_data_eq, my_data, '')
# Plotting the equity curve
plt.plot(my_data_eq [:, 3], linewidth = 1, label ='')
```

The above code is a sort of framework that you can follow. The full replicable version can be found in the GitHub page. In the meantime, I highly encourage you to do it yourself and even try to improve the code presented here due to it being pretty basic and user-friendly. By time, you will be able to understand every line of code you write or read. Python is a language that forces you to fall in love with it.

SIMPLE MOVING AVERAGE DISTANCE

"What goes around comes around and what goes up must go down, just like Bitcoin, only less violently."

The main idea of this technique is that when the market deviates from its moving average, it is bound to revert to it, this is known as mean-reversion. One technique to trade mean-reversion is to use the distance of the market price from its moving average and calculate normalized barriers where we can say that historically, the distance gets smaller when it reaches a certain point, thus achieving the point of mean-reversion.

The conditions required for the moving average distance trade:

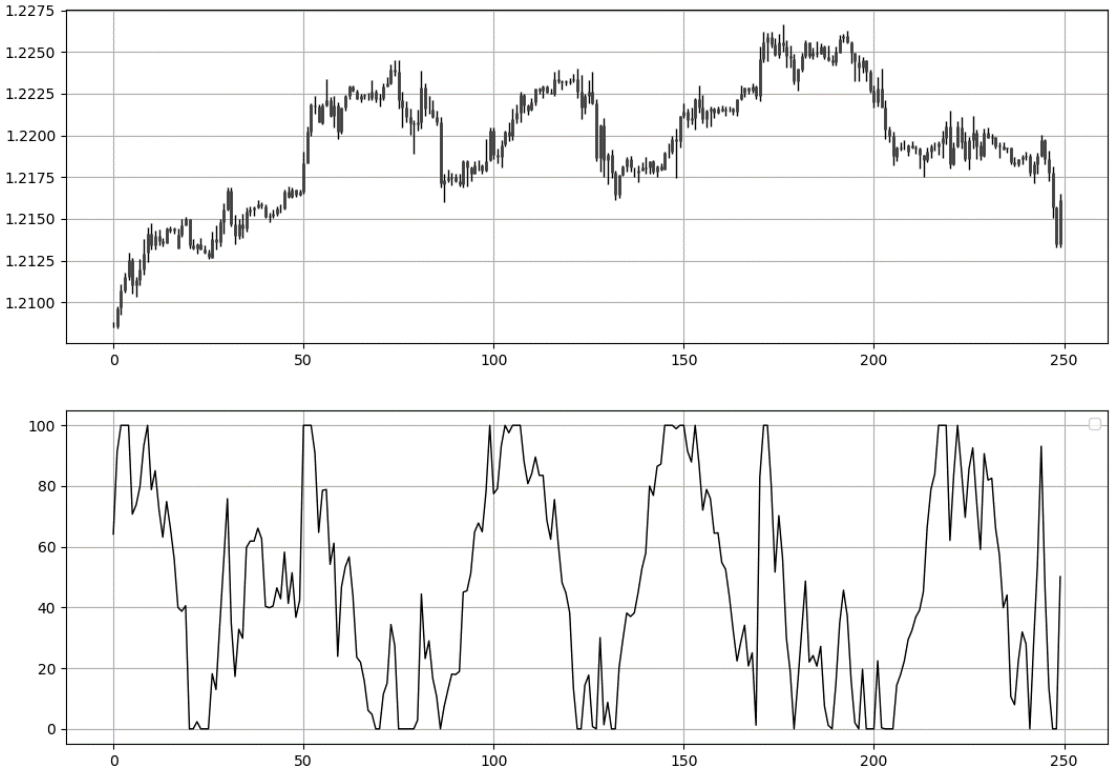
- For a bullish (Long) signal, the market price must close at the normalized lower barrier which represents the absolute negative distance from the current moving average.
- For a bearish (Short) signal, the market price must close at the normalized upper barrier which represents the absolute positive distance from the current moving average.

As we have already seen how to calculate a simple moving average, we can now calculate the distance which is the difference between the market price and the current moving average. In two lines, we can get the desired results:

```
my_data = adder(my_data, 2)
my_data[:, 5] = my_data[:, 3] - my_data[:, 4]
```

The above code will create two new columns where the first one will be populated by the distance calculation discussed above and the second one will be the normalized value of the distance. In the first part of the book, we have seen what normalization is.

```
normalization_period = 20
my_data = stochastic(my_data, normalization_period, 5, 6, genre = 'Normalization')
```



The above plot shows the hourly values of the EURUSD in the first panel with the 20-period normalized distance from its 20-period simple moving average. In layman's terms, whenever the reading in the second panel equals 100, it means that the current distance is the highest (on the positive side) since 20 periods ago. If it keeps increasing, then the reading will remain at 100. Similarly, whenever the reading in the second panel equals 0, it means that the current distance is the highest (on the negative side) since 20 periods ago. If it keeps increasing, then the reading will remain at 0. It becomes clear that now we should have the following trading conditions:

- For a bullish (Long) signal, the distance must be 0 in anticipation of mean-reversion.
- For a bearish (Short) signal, the distance must be 100 in anticipation of mean-reversion.

```
def signal(Data, normalized_distance, buy, sell):
    Data = adder(Data, 2)
    for i in range(len(Data)):
        if Data[i, normalization_distance] == 0.0000 and Data[i - 1, buy] == 0 and \
            Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0:
            Data[i, buy] = 1
        elif Data[i, normalization_distance] == 100.0000 and Data[i - 1, sell] == 0 and \
            Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0:
            Data[i, sell] = -1
    return Data
my_data = signal(my_data, 6, 7, 8)
```

The extra conditions in the signal function are there to avoid successive trades due to the indicator spending more than one time period at zero or at 100. This is important so that the result do not get biased when extreme trending conditions are occurring. An even better alternative is to not take into account any signals where we already have an open position in that direction. This way, we make sure we do not close a long trade to open another long trade.

SIMPLE DOUBLE MOVING AVERAGE CROSS

"A strategy known to crucify market participants."

The main idea is that when two moving averages cross, a new trend may be emerging. When a short-term moving average crosses a long-term moving average, it can be a signal that the regime is switching. The conditions required for the moving average cross:

- For a bullish (Long) signal, the short-term moving average must surpass the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be below the long-term moving average.
- For a bearish (Short) signal, the short-term moving average must break the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be above the long-term moving average.

The below snippet shows how to calculate a 50-period simple moving average and a 200-period simple moving average on the OHLC array of historical data we have recently imported to the Python interpreter.

```
# Using the function
short_term_ma = 50
long_term_ma  = 200
my_data = ma(my_data, short_term_ma, 3, 4)
my_data = ma(my_data, long_term_ma,  3, 5)
```

The full Python code to get signals from the moving average cross can be as follows:

```
def signal(Data, short_term_ma, long_term_ma, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, short_term_ma] > Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] < Data[i - 1, long_term_ma]:
            Data[i, buy] = 1
        elif Data[i, short_term_ma] < Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] > Data[i - 1, long_term_ma]:
            Data[i, sell] = -1
    return Data
```

The signals are quite clear. Whenever the short-term moving average is above the long-term moving average while the previous short-term moving is below the long-term moving average, then a bullish cross has occurred, and we have a buy signal. This is also called a Golden Cross. Similarly, whenever the short-term moving average is below the long-term moving average while the previous short-term moving is above the long-term moving average, then a bearish cross has occurred, and we have a short sell signal. This is also called a Death Cross. I recommend you heavily back-test and optimize this strategy as it is not as wonderful as people claim it is. Also, in ranging markets, the number of false signals you will get are enough to get you out of the game. In known trending markets such as long-term equity returns, this strategy seems to deliver on its promises, however with short-term trading on ranging assets, it is not an outperformer, not even a performer for that matter.

SIMPLE TRIPLE MOVING AVERAGE CROSS

"Why complicate life with a double cross when you could do a triple cross and complicate it even more?"

The main idea is that when three moving averages cross, a new trend may be emerging. When a short-term moving average crosses a medium-term and a long-term moving average, it can be a signal that the regime is switching. The conditions required for the triple moving average cross:

- For a bullish (Long) signal, the short-term moving average must surpass the medium-term moving average and the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be below the medium-term moving average.
- For a bearish (Short) signal, the short-term moving average must break the medium-term moving average and the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be above the medium moving average.

The below snippet shows how to calculate a 50-period simple moving average, a 100-period simple moving average, and a 200-period simple moving average on the OHLC array of historical data we have recently imported to the Python interpreter.

```
# Using the function
short_term_ma    = 50
medium_term_ma   = 100
long_term_ma     = 200
my_data = ma(my_data, short_term_ma,   3, 4)
my_data = ma(my_data, medium_term_ma,  3, 5)
my_data = ma(my_data, long_term_ma,    3, 6)
```

The full Python code to get signals from the moving average cross can be as follows:

```
def signal(Data, short_term_ma, medium_term_ma, long_term_ma, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, short_term_ma] > Data[i, medium_term_ma] and Data[i, short_term_ma] > Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] < Data[i - 1, medium_term_ma]:
            Data[i, buy] = 1
        elif Data[i, short_term_ma] < Data[i, medium_term_ma] and Data[i, short_term_ma] < Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] > Data[i - 1, medium_term_ma]:
            Data[i, sell] = -1
    return Data
```

EXPONENTIAL MOVING AVERAGE CROSS

"Some say this is an improved version of the simple moving averages, what do you think?"

As the name suggests, this is an exponential mean that places more weight on the more recent observations. It is calculated following an exponential smoothing formula as can be shown in the below mathematical equation:

$$EMA_{\text{Today}} = \left(\text{Value}_{\text{Today}} * \left(\frac{\text{Smoothing}}{1+\text{Days}}\right)\right) + EMA_{\text{Yesterday}} * \left(1 - \left(\frac{\text{Smoothing}}{1+\text{Days}}\right)\right)$$

The conditions required for the moving average cross:

- For a bullish (Long) signal, the market price must close above the current moving average while the previous market price closing below the previous moving average.
- For a bearish (Short) signal, the market price must close below the current moving average while the previous market price closing above the previous moving average.

The Python code to calculate an exponential moving average and the signal function can be as follows. Note that it is important to have an OHLC data array and not a data frame. The code requires the definition of the simple moving average first because the first value of the exponential moving average is actually a simple moving average.

```
def ema(Data, alpha, lookback, what, where):
    alpha = alpha / (lookback + 1.0)
    beta  = 1 - alpha
    # First value is a simple SMA
    Data = ma(Data, lookback, what, where)
    # Calculating first EMA
    Data[lookback + 1, where] = (Data[lookback + 1, what] * alpha) + (Data[lookback,
where] * beta)
    # Calculating the rest of EMA
    for i in range(lookback + 2, len(Data)):
        try:
            Data[i, where] = (Data[i, what] * alpha) + (Data[i - 1, where] * beta)
        except IndexError:
            pass
    return Data
```

The signal function to calculate the exponential cross is as follows:

```
def signal(Data, close, ma_column, buy_col, sell_col):
    Data = adder(Data, 2)
    for i in range(len(Data)):
        # Scanning for Bullish signals
        if Data[i, close] > Data[i, ma_column] and Data[i - 1, close] < Data[i - 1, ma_column]:
            Data[i, buy_col] = 1
        # Scanning for Bearish signals
        elif Data[i, close] < Data[i, ma_column] and Data[i - 1, close] > Data[i - 1, ma_column]:
            Data[i, sell_col] = -1
    return Data
my_data = signal(my_data, 3, 4, 5, 6)
```

The function will output 1 in the column labelled as buy whenever the current market close is above the moving average and -1 in the column labelled as sell whenever the market close is below the moving average while imposing another important condition which requires the previous market close to be below the previous moving average, thus ensuring the cross.

The above signal chart shows a 200-period exponential moving average cross on the values of the EURCHF. We can see that whenever there is a close above or below the moving average line, a new signal is generated until another cross is made.

EXPONENTIAL MOVING AVERAGE DISTANCE

"Let there be profits."

As discussed previously, the distance strategy capitalizes on the concept of mean-reversion in an attempt to target normality.

The conditions required for the moving average distance trade:

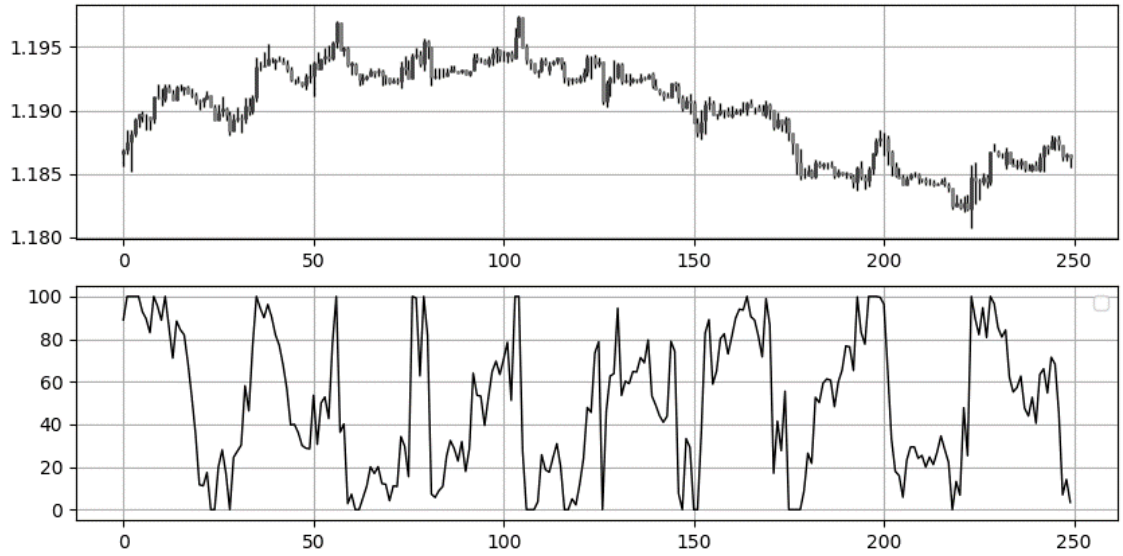
- For a bullish (Long) signal, the market price must close at the normalized lower barrier which represents the absolute negative distance from the current moving average.
- For a bearish (Short) signal, the market price must close at the normalized upper barrier which represents the absolute positive distance from the current moving average.

As we have already seen how to calculate an exponential moving average, we can now calculate the distance which is simply the difference between the market price and the current moving average. In two lines, we can get the desired results:

```
my_data = adder(my_data, 2)
my_data[:, 5] = my_data[:, 3] - my_data[:, 4]
```

The above code will create two new columns where the first one will be populated by the distance calculation discussed above and the second one will be the normalized value of the distance. In the first part of the book, we have seen what normalization is.

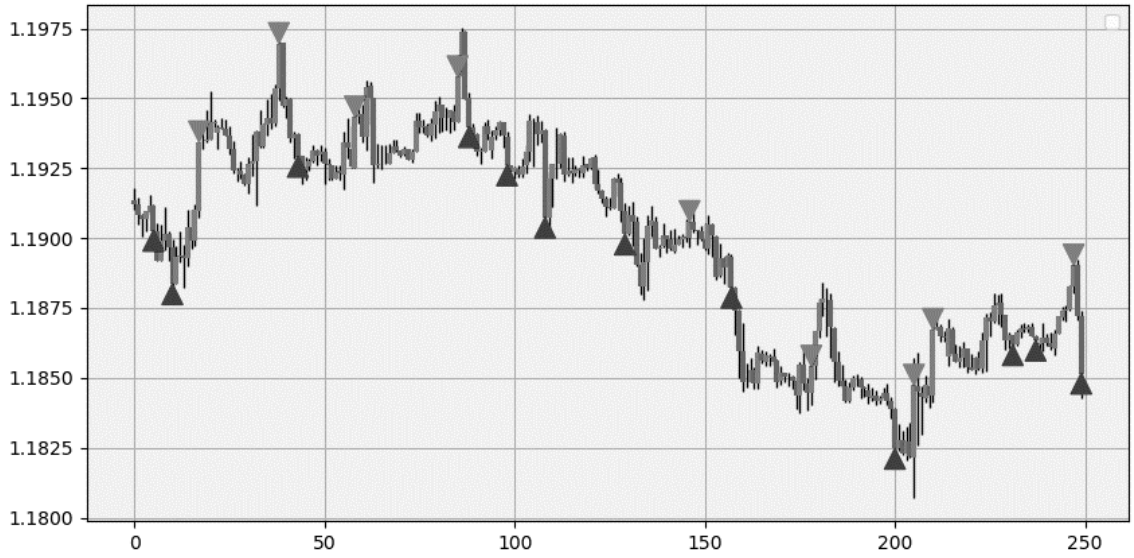
```
normalization_period = 20
my_data = stochastic(my_data, normalization_period, 5, 6, genre = 'Normalization')
```



The above plot shows the hourly values of the EURUSD in the first panel with the 20-period normalized distance from its 20-period exponential moving average. In layman's terms, whenever the reading in the second panel equals 100, it means that the current distance is the highest (on the positive side) since 20 periods ago. If it keeps increasing, then the reading will remain at 100. Similarly, whenever the reading in the second panel equals 0, it means that the current distance is the highest (on the negative side) since 20 periods ago. If it keeps increasing, then the reading will remain at 0. It becomes clear that now we should have the following trading conditions:

- For a bullish (Long) signal, the distance must be 0 in anticipation of mean-reversion.
- For a bearish (Short) signal, the distance must be 100 in anticipation of mean-reversion.

```
def signal(Data, normalized_distance, buy, sell):
    Data = adder(Data, 2)
    for i in range(len(Data)):
        if Data[i, normalization_distance] == 0.0000 and Data[i - 1, buy] == 0 and \
            Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0:
            Data[i, buy] = 1
        elif Data[i, normalization_distance] == 100.0000 and Data[i - 1, sell] == 0 and \
            Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0:
            Data[i, sell] = -1
    return Data
my_data = signal(my_data, 6, 7, 8)
```



The signals above are the result of the signal function we have created. They seem to have some strength but only back-tests can validate them. Also, when we back-test strategies, we have to run them across all covered assets and time frames. Now, an important question arises here. How are the distance strategies part of the trend-following part? It is true that they are mostly contrarian strategies, but we can actually use them in trend-following strategies by combining them with long-term moving averages. Also, I wanted to keep moving averages together in one part.

EXPONENTIAL DOUBLE MOVING AVERAGE CROSS

"Double crossing? Yes."

The main idea is that when two moving averages cross, a new trend may be emerging. When a short-term moving average crosses a long-term moving average, it can be a signal that the regime is switching. The conditions required for the moving average cross:

- For a bullish (Long) signal, the short-term moving average must surpass the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be below the long-term moving average.
- For a bearish (Short) signal, the short-term moving average must break the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be above the long-term moving average.

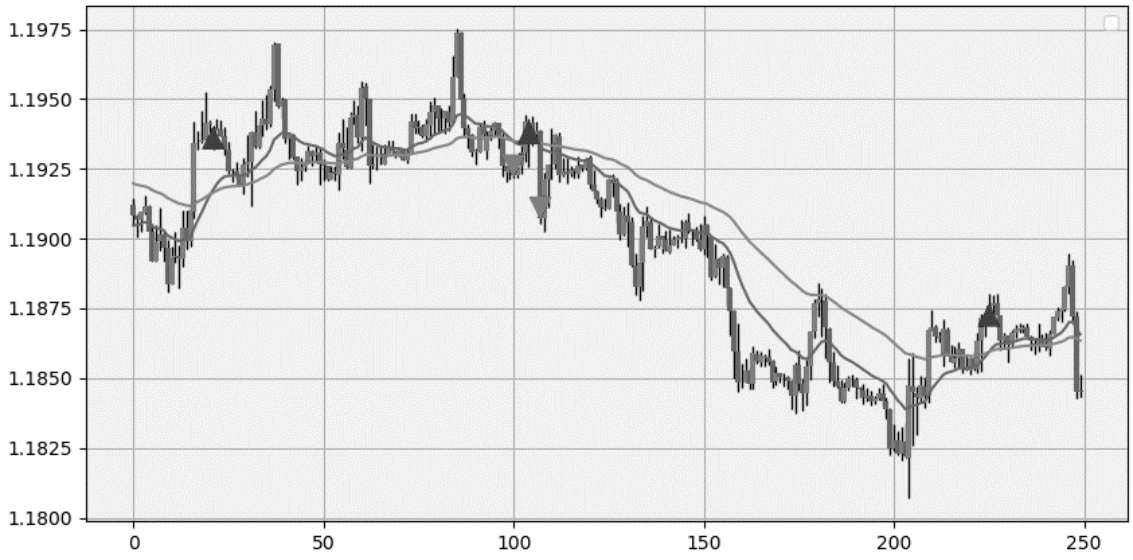
The below snippet shows how to calculate a 50-period exponential moving average and a 200-period exponential moving average on the OHLC array of historical data we have recently imported to the Python interpreter.

```
# Using the function
short_term_ma = 20
long_term_ma  = 60
my_data = ema(my_data, 2, short_term_ma, 3, 4)
my_data = ema(my_data, 2, long_term_ma,  3, 5)
```

The constant 2 in the exponential moving average's function refers to smoothing. By default, it is always equal to 2. However, it would be interesting to tweak it and see what kind of signals it gives when we put 3 or 4. I will leave this idea up to you to test it.

The full Python code to get signals from the moving average cross can be as follows:

```
def signal(Data, short_term_ema, long_term_ema, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, short_term_ema] > Data[i, long_term_ema] and \
            Data[i - 1, short_term_ema] < Data[i - 1, long_term_ema]:
            Data[i, buy] = 1
        elif Data[i, short_term_ema] < Data[i, long_term_ema] and \
            Data[i - 1, short_term_ema] > Data[i - 1, long_term_ema]:
            Data[i, sell] = -1
    return Data
```



The above is an example on the EURUSD with a cross between the 20-period exponential moving average and the 60-period exponential moving average.

The signals are quite clear. Whenever the short-term moving average is above the long-term moving average while the previous short-term moving is below the long-term moving average, then a bullish cross has occurred, and we have a buy signal. This is also called a Golden Cross. Similarly, whenever the short-term moving average is below the

long-term moving average while the previous short-term moving is above the long-term moving average, then a bearish cross has occurred, and we have a short sell signal. This is also called a Death Cross. Note that the words short-term and long-term are relative and not absolute, meaning a 200-period moving average can be short-term when compared to a 600-period moving average but long-term when compared to a 30-period moving average.

EXPONENTIAL TRIPLE MOVING AVERAGE CROSS

"Keep'em coming."

The main idea is that when three moving averages cross, a new trend may be emerging. When a short-term moving average crosses a medium-term and a long-term moving average, it can be a signal that the regime is switching. The conditions required for the triple moving average cross:

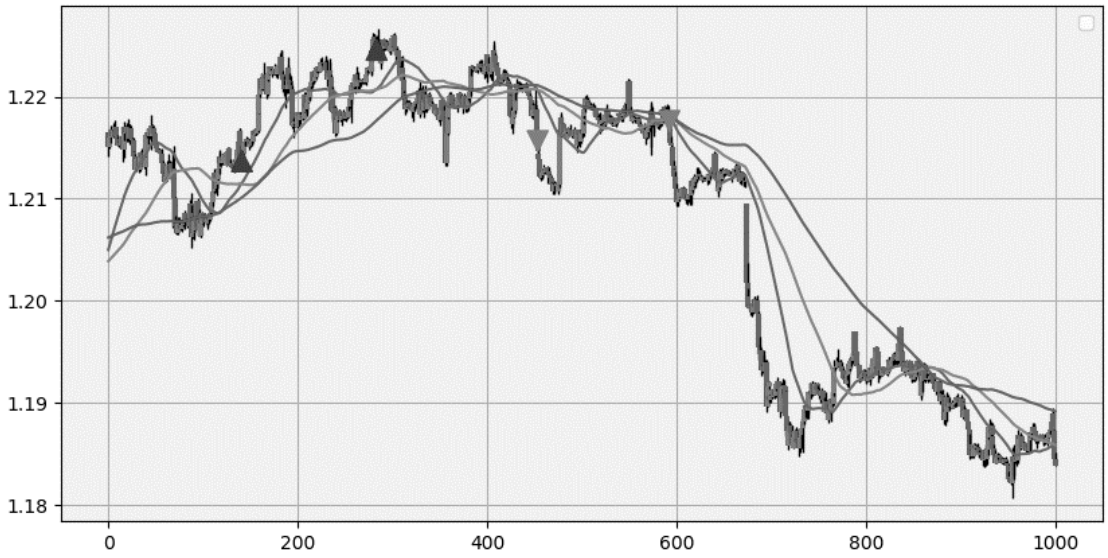
- For a bullish (Long) signal, the short-term moving average must surpass the medium-term moving average and the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be below the medium-term moving average.
- For a bearish (Short) signal, the short-term moving average must break the medium-term moving average and the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be above the medium moving average.

The below snippet shows how to calculate a 50-period simple moving average, a 100-period simple moving average, and a 200-period simple moving average on the OHLC array of historical data we have recently imported to the Python interpreter.

```
# Using the function
short_term_ma    = 50
medium_term_ma   = 100
long_term_ma     = 200
my_data = ema(my_data, 2, short_term_ma,   3, 4)
my_data = ema(my_data, 2, medium_term_ma,  3, 5)
my_data = ema(my_data, 2, long_term_ma,    3, 6)
```

The full Python code to get signals from the moving average cross can be as follows:

```
def signal(Data, short_term_ema, medium_term_ema, long_term_ema, buy, sell):
   Data = adder(Data, 10)
   for i in range(len(Data)):
      if Data[i, short_term_ema] > Data[i, medium_term_ema] and Data[i, short_term_ema] > Data[i, long_term_ema] and \
         Data[i - 1, short_term_ema] < Data[i - 1, medium_term_ema]:
         Data[i, buy] = 1
      elif Data[i, short_term_ema] < Data[i, medium_term_ema] and Data[i, short_term_ema] < Data[i, long_term_ema] and \
         Data[i - 1, short_term_ema] > Data[i - 1, medium_term_ema]:
         Data[i, sell] = -1
   return Data
```



Triple crosses are a bit more complex and are less frequent. However, they are often used by technical analysts as an evolved technique from the double cross so as to have an extra layer of confirmation.

SMOOTHED MOVING AVERAGE CROSS

"Smooth as silk, tough as nail."

The smoothed moving average has been presented by Welles Wilder, the creator of the famous RSI. It is a more stable version of the exponential moving average and is related to it with a simple transformation of the lookback period. Basically, to transform an exponential moving average to a smoothed moving average, we only have to change the lookback period following this formula:

```
smoothed = (exponential * 2) - 1 # From exponential to smoothed
```

The above code can be added before calling the exponential moving average function. It will automatically switch to the smoothed version. This is much simpler than presenting the intuition of the smoothed moving average as it is a bit complicated, which makes it a nice surprise that we have a basic transformation from an exponential moving average to a smoothed one.

The conditions required for the moving average cross:

- For a bullish (Long) signal, the market price must close above the current moving average while the previous market price closing below the previous moving average.
- For a bearish (Short) signal, the market price must close below the current moving average while the previous market price closing above the previous moving average.

The Python code to calculate a smoothed moving average and the signal function can be as follows. Note that it is important to have an OHLC data array and not a data frame. The code requires the definition of the exponential moving average first because the smoothed version only applies a transformation on the lookback period of the exponential version.

```
def ema(Data, alpha, lookback, what, where):
    alpha = alpha / (lookback + 1.0)
    beta  = 1 - alpha
    # First value is a simple SMA
    Data = ma(Data, lookback, what, where)
    # Calculating first EMA
    Data[lookback + 1, where] = (Data[lookback + 1, what] * alpha) + (Data[lookback,
where] * beta)
    # Calculating the rest of EMA
    for i in range(lookback + 2, len(Data)):
        try:
            Data[i, where] = (Data[i, what] * alpha) + (Data[i - 1, where] * beta)
        except IndexError:
            pass
    return Data
```

The transformation is applied before when choosing the lookback variable. The signal function to calculate the cross is as follows:

```
def signal(Data, close, sma_column, buy_col, sell_col):
    Data = adder(Data, 2)
    for i in range(len(Data)):
        # Scanning for Bullish signals
        if Data[i, close] > Data[i, sma_column] and Data[i - 1, close] < Data[i - 1, sma_column]:
            Data[i, buy_col] = 1
        # Scanning for Bearish signals
        elif Data[i, close] < Data[i, sma_column] and Data[i - 1, close] > Data[i - 1, sma_column]:
            Data[i, sell_col] = -1
    return Data
my_data = signal(my_data, 3, 4, 5, 6)
```

The function will output 1 in the column labelled as buy whenever the current market close is above the moving average and -1 in the column labelled as sell whenever the market close is below the moving average while imposing another important condition which requires the previous market close to be below the previous moving average, thus ensuring the cross.

The above chart uses the exponential moving average function but puts a lookback that is twice as needed (and then we subtract one from it). We can by now notice an issue with the cross strategy. Having seen until now three types of moving averages using the cross technique, the issue of whipsaws and constant re-integration arises. This means that when the market surpasses or breaks the moving average, it can do it successively many times before moving away from it, causing the algorithm to give out too many buy and sell signals around the same time period. We can remedy this by adding a backwards condition that states that the signal must be taken if the previous a specified number of periods do not have any signals. This ensures that when a signal is found, it should be the only one. However, it does not answer to the question of which is the correct signal? Also, who can say for certain that the market actually crossed the moving average? This is a huge pitfall in this strategy.

SMOOTHED MOVING AVERAGE DISTANCE

"With short lookback periods, there is no difference between it and the exponential distance strategy."

The conditions required for the moving average distance trade:

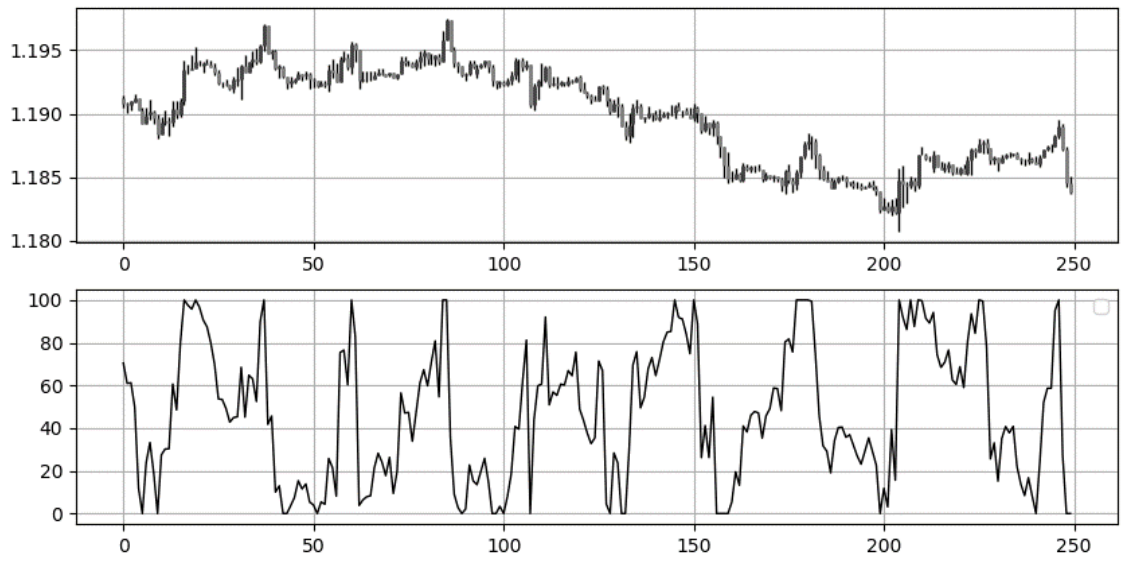
- For a bullish (Long) signal, the market price must close at the normalized lower barrier which represents the absolute negative distance.
- For a bearish (Short) signal, the market price must close at the normalized upper barrier which represents the absolute positive distance

As we have already seen how to calculate a smoothed moving average, we can now calculate the distance which is simply the difference between the market price and the current moving average. In two lines, we can get the desired results:

```
my_data = adder(my_data, 2)
my_data[:, 5] = my_data[:, 3] - my_data[:, 4]
```

The above code will create two new columns where the first one will be populated by the distance calculation discussed above and the second one will be the normalized value of the distance.

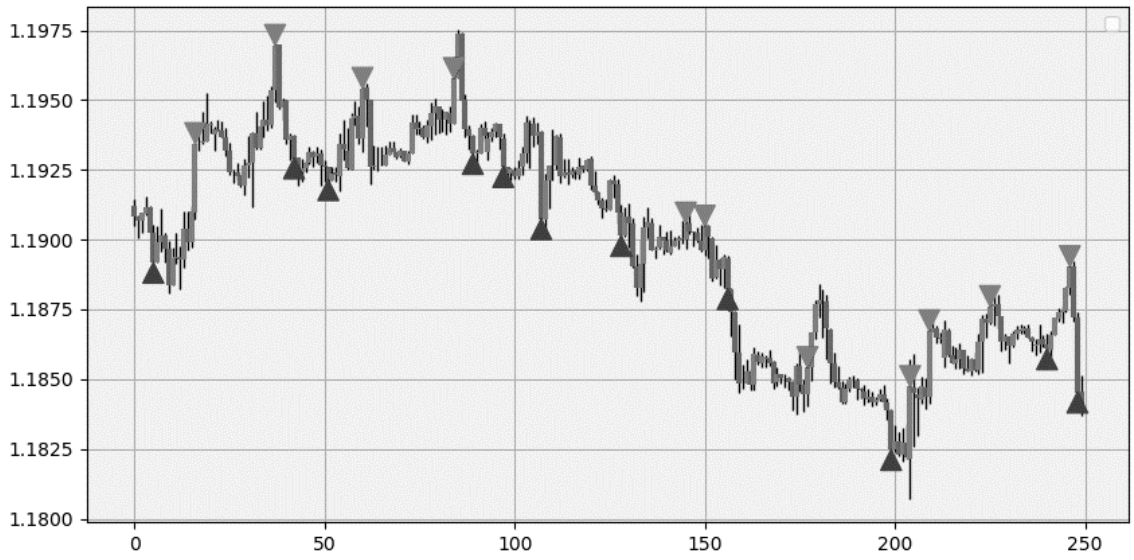
```
normalization_period = 20
my_data = stochastic(my_data, normalization_period, 5, 6, genre = 'Normalization')
```



The previous plot shows the hourly values of the EURUSD in the first panel with the 20-period normalized distance from its 20-period smoothed moving average (39-period exponential moving average). In layman's terms, whenever the reading in the second panel equals 100, it means that the current distance is the highest (on the positive side) since 20 periods ago. If it keeps increasing, then the reading will remain at 100. Similarly, whenever the reading in the second panel equals 0, it means that the current distance is the highest (on the negative side) since 20 periods ago. If it keeps increasing, then the reading will remain at 0. It becomes clear that now we should have the following trading conditions:

- For a bullish (Long) signal, the distance must be 0 in anticipation of mean-reversion.
- For a bearish (Short) signal, the distance must be 100 in anticipation of mean-reversion.

```
def signal(Data, normalized_column, buy, sell):
  Data = adder(Data, 10)
  for i in range(len(Data)):
    if Data[i, normalized_column] == 0 and Data[i - 1, buy] == 0:
      Data[i, buy] = 1
    elif Data[i, normalized_column] == 100 and Data[i - 1, sell] == 0:
      Data[i, sell] = -1
  return Data
```



The signals on the normalized distance between the market price and the 20-period smoothed moving average seem to have potential regardless of any performance metric. However, extensive optimization and back-testing is needed to enhance it. Recalling the previous discussion on why the distance strategy lies in the trend-following part and how we can use as a technique to follow the trend, we can apply it with a long-term moving average such as 200 and then only initiate signals from the distance algorithm if the market is above or below the long-term moving average. An example of the conditions can be the below:

- A bullish signal is triggered whenever the normalized 20-period distance of the 20-period moving average equals zero while the current price is above the 200-period moving average.
- A bearish signal is triggered whenever the normalized 20-period distance of the 20-period moving average equals 100 while the current price is below the 200-period moving average.

SMOOTHED DOUBLE MOVING AVERAGE CROSS

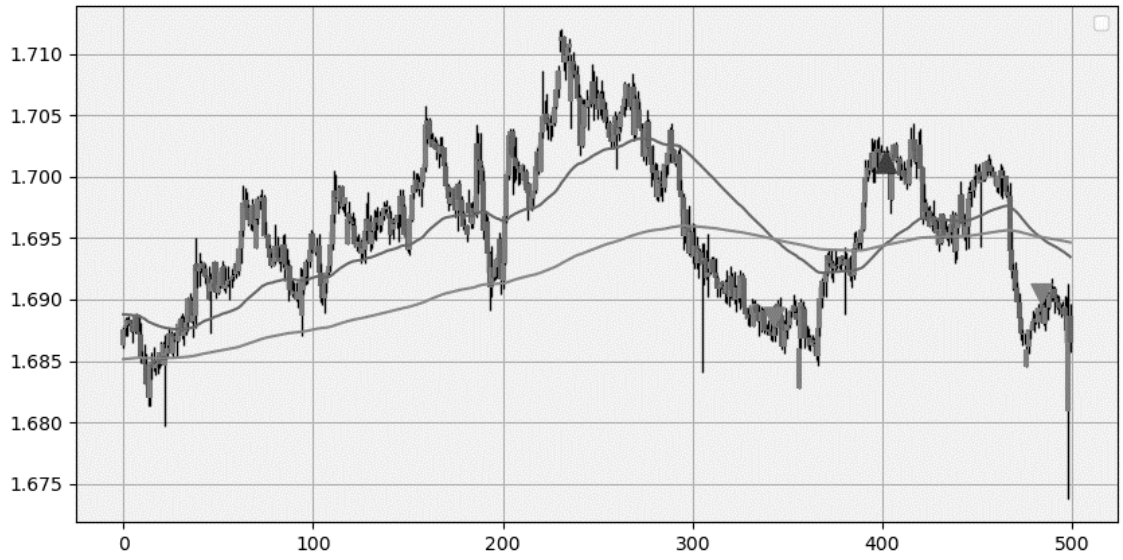
"Could it be the one?"

The main idea is that when two moving averages cross, a new trend may be emerging. When a short-term moving average crosses a long-term moving average, it can be a signal that the regime is switching. The conditions required for the moving average cross:

- For a bullish (Long) signal, the short-term moving average must surpass the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be below the long-term moving average.
- For a bearish (Short) signal, the short-term moving average must break the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be above the long-term moving average.

The below snippet shows how to calculate a 50-period smoothed moving average and a 200-period smoothed moving average on the OHLC array of historical data we have recently imported to the Python interpreter.

```
# Using the function
short_term_ma = 99 # To get a 50-period smoothed moving average
long_term_ma  = 399 # To get a 200-period smoothed moving average
my_data = ema(my_data, 2, short_term_ma, 3, 4)
my_data = ema(my_data, 2, long_term_ma, 3, 5)
```



The full Python code to get signals from the moving average cross can be as follows:

```
def signal(Data, short_term_ema, long_term_ema, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, short_term_ema] > Data[i, long_term_ema] and \
            Data[i - 1, short_term_ema] < Data[i - 1, long_term_ema]:
            Data[i, buy] = 1
        elif Data[i, short_term_ema] < Data[i, long_term_ema] and \
            Data[i - 1, short_term_ema] > Data[i - 1, long_term_ema]:
            Data[i, sell] = -1
    return Data
```

SMOOTHED TRIPLE MOVING AVERAGE CROSS

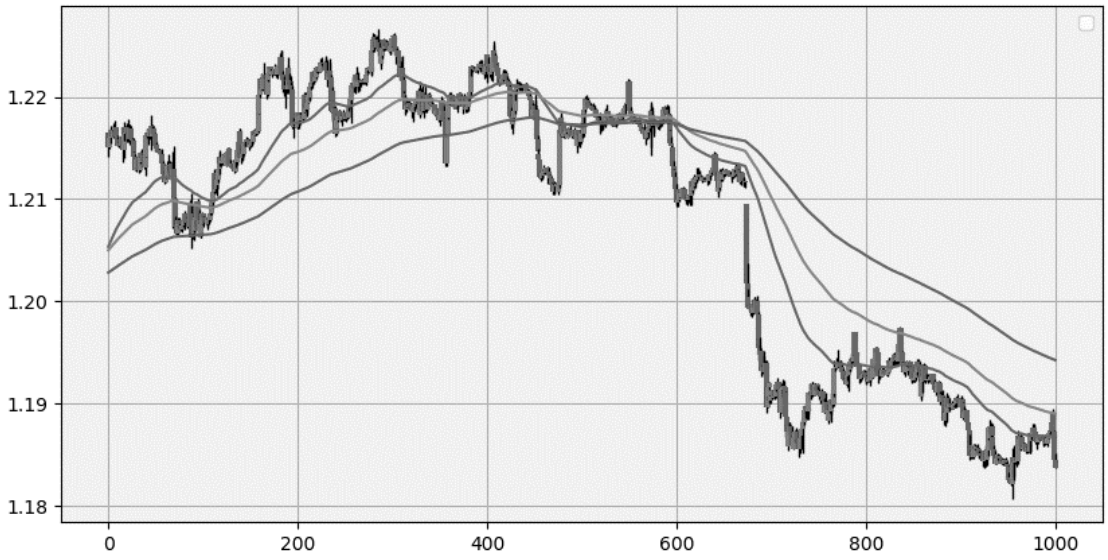
"Things are getting real."

The main idea is that when three moving averages cross, a new trend may be emerging. When a short-term moving average crosses a medium-term and a long-term moving average, it can be a signal that the regime is switching. The conditions required for the triple moving average cross:

- For a bullish (Long) signal, the short-term moving average must surpass the medium-term moving average and the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be below the medium-term moving average.
- For a bearish (Short) signal, the short-term moving average must break the medium-term moving average and the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be above the medium moving average.

The below snippet shows how to calculate a 50-period smoothed moving average, a 100-period smoothed moving average, and a 200-period smoothed moving average on the OHLC array of historical data we have recently imported to the Python interpreter.

```
# Using the function
ma_short = 99 # To get a 50-period smoothed moving average
ma_medium = 199 # To get a 100-period smoothed moving average
ma_long  = 399 # To get a 200-period smoothed moving average
my_data = ema(my_data, 2, ma_short, 3, 4)
my_data = ema(my_data, 2, ma_medium, 3, 5)
my_data = ema(my_data, 2, ma_long, 3, 6)
```



The full Python code to get signals from the moving average cross can be as follows:

```
def signal(Data, short_term_ma, medium_term_ma, long_term_ma, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, short_term_ma] > Data[i, medium_term_ma] and Data[i, short_term_ma] > Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] < Data[i - 1, medium_term_ma]:
            Data[i, buy] = 1
        elif Data[i, short_term_ma] < Data[i, medium_term_ma] and Data[i, short_term_ma] < Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] > Data[i - 1, medium_term_ma]:
            Data[i, sell] = -1
    return Data
```

ADAPTIVE MOVING AVERAGE CROSS

"Improvise, Adapt, Overcome. – Bear Grylls."

The adaptive moving average (or Kaufman's Adaptive Moving Average – KAMA) has been created to reduce the noise and whipsaw effects. It works the same as other moving averages do and follows the same intuition. The generation of false trading signals is one of the problems with moving averages and this is due to short-term sudden fluctuations that bias the calculation. The KAMA's primary objective is to reduce as much noise as possible.

The first concept we should measure is the Efficiency Ratio which is the absolute change of the current close relative to the change in the past 10 periods divided by a type of volatility calculated in a special manner. We can say that the Efficiency Ratio is the change divided by volatility.

$$Efficiency\ Ratio = \frac{|Close_n - Close_{n-10}|}{Sum\ of\ |Close_n - Close_{n-1}|}$$

Then we calculate a smoothing constant based on the following formula:

$$Smoothing\ Constant = (Efficiency\ Ratio\ x\ 0.6667)^2$$

Finally, to calculate the KAMA we use the below formula:

$$KAMA = KAMA_{n-1} + (Smoothing\ Constant\ .\ (Closing\ Price\ .\ KAMA_{n-1}))$$

The calculation may seem complicated, but it is easily automated, and you do not have to think about it much. Let us see how to code it in Python and then proceed by seeing some examples.

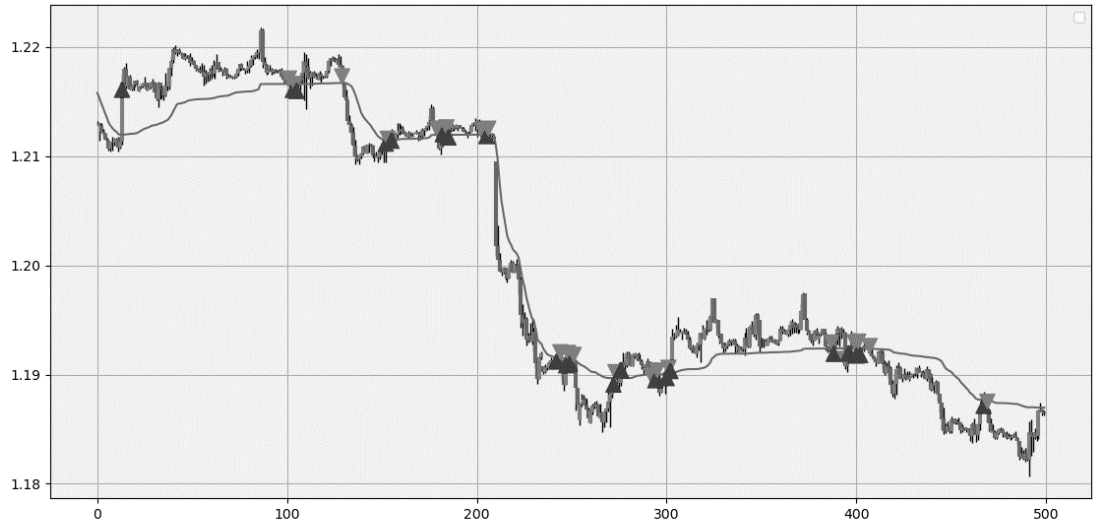
```
def kama(Data, what, where, change):
    # Change from previous period
    for i in range(len(Data)):
        Data[i, where] = abs(Data[i, what] - Data[i - 1, what])
    Data[0, where] = 0
    # Sum of changes
    for i in range(len(Data)):
        Data[i, where + 1] = (Data[i - change + 1:i + 1, where].sum())
    # Volatility
    for i in range(len(Data)):
        Data[i, where + 2] = abs(Data[i, 3] - Data[i - 10, 3])
    Data = Data[11:, ]
    # Efficiency Ratio
    Data[:, where + 3] = Data[:, where + 2] / Data[:, where + 1]
    for i in range(len(Data)):
        Data[i, where + 4] = np.square(Data[i, where + 3] * 0.6666666666666666667)
    for i in range(len(Data)):
        Data[i, where + 5] = Data[i - 1, where + 5] + (Data[i, where + 4] * (Data[i, 3] - Data[i - 1,
where + 5]))
        Data[11, where + 5] = 0
```

The next chart illustrates what the above function can give us. Notice the more stable tendency of the KAMA relative to other moving averages. It is an improvement with regards to whipsaws and false breaks.

The KAMA's main strength is that it is highly reactive to moves and is adapted to fast markets while the main weakness is detecting dynamic support and resistance levels.

The conditions required for the moving average cross:

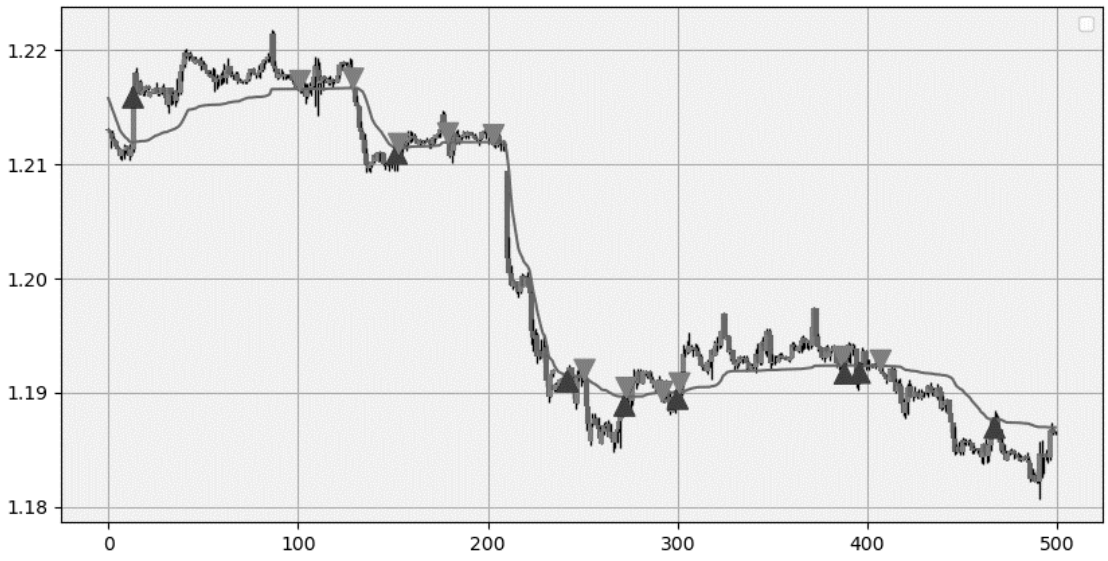
- For a bullish (Long) signal, the market price must close above the current moving average while the previous market price closing below the previous moving average.
- For a bearish (Short) signal, the market price must close below the current moving average while the previous market price closing above the previous moving average.




```
def signal(Data, close, ma_column, buy_col, sell_col):
    Data = adder(Data, 2)
    for i in range(len(Data)):
        # Scanning for Bullish signals
        if Data[i, close] > Data[i, ma_column] and Data[i - 1, close] < Data[i - 1, ma_column]:
            Data[i, buy_col] = 1
        # Scanning for Bearish signals
        elif Data[i, close] < Data[i, ma_column] and Data[i - 1, close] > Data[i - 1, ma_column]:
            Data[i, sell_col] = -1
    return Data
my_data = signal(my_data, 3, 4, 5, 6)
```

We can try to remedy the problem of successive trades in both direction by incorporating some conditions in the signal function.

```
def signal(Data, close, ma_col, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, close] > Data[i, ma_col] and Data[i - 1, close] < Data[i - 1, ma_col] and \
            Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0 and \
            Data[i - 5, buy] == 0 and Data[i - 6, buy] == 0 and Data[i - 7, buy] == 0 and \
            Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0 and \
            Data[i - 5, sell] == 0 and Data[i - 6, sell] == 0 and Data[i - 7, sell] == 0:
            Data[i, buy] = 1
        elif Data[i, close] < Data[i, ma_col] and Data[i - 1, close] > Data[i - 1, ma_col] and \
            Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0 and \
            Data[i - 5, buy] == 0 and Data[i - 6, buy] == 0 and Data[i - 7, buy] == 0 and \
            Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0 and \
            Data[i - 5, sell] == 0 and Data[i - 6, sell] == 0 and Data[i - 7, sell] == 0:
            Data[i, sell] = -1
    return Data
```



ADAPTIVE MOVING AVERAGE DISTANCE

"Everybody needs space and distance from time to time, even moving averages."

The conditions required for the moving average distance trade:

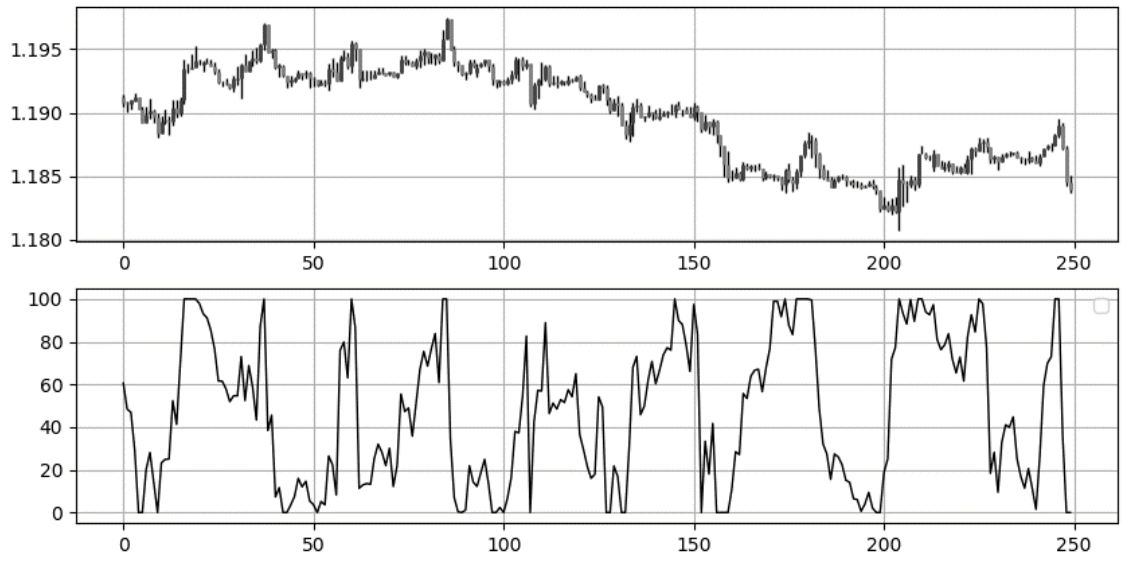
- For a bullish (Long) signal, the market price must close at the normalized lower barrier which represents the absolute negative distance.
- For a bearish (Short) signal, the market price must close at the normalized upper barrier which represents the absolute positive distance.

As we have already seen how to calculate a adaptive moving average, we can now calculate the distance which is simply the difference between the market price and the current moving average. In two lines, we can get the desired results:

```
my_data = adder(my_data, 2)
my_data[:, 5] = my_data[:, 3] - my_data[:, 4]
```

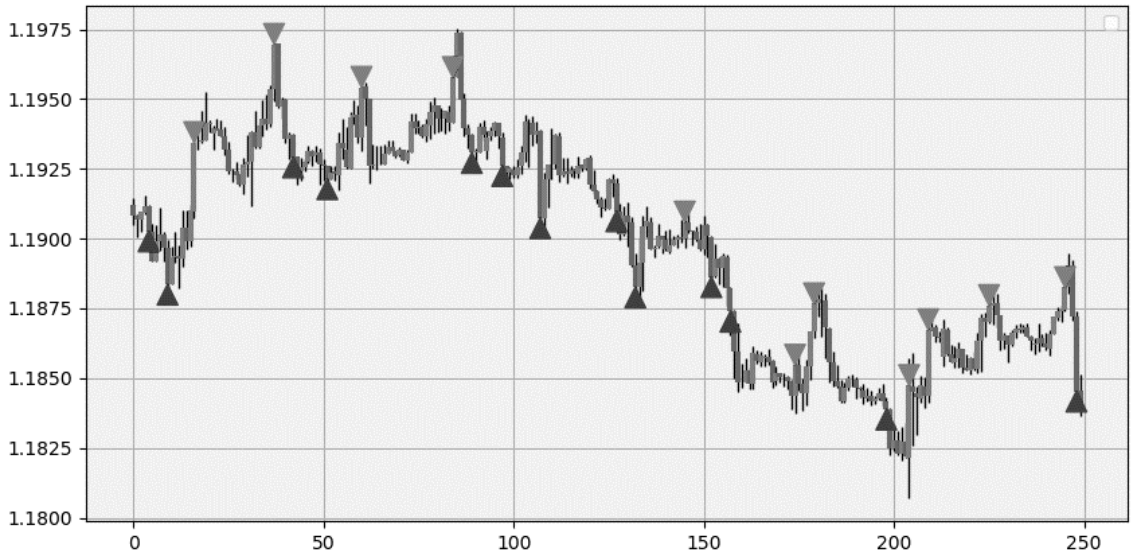
The above code will create two new columns where the first one will be populated by the distance calculation discussed above and the second one will be the normalized value of the distance. In the first part of the book, we have seen what normalization is.

```
normalization_period = 20
my_data = stochastic(my_data, normalization_period, 5, 6, genre = 'Normalization')
```



The above plot shows the hourly values of the EURUSD in the first panel with the 20-period normalized distance from its 20-period adaptive moving average. In layman's terms, whenever the reading in the second panel equals 100, it means that the current distance is the highest (on the positive side) since 20 periods ago. If it keeps increasing, then the reading will remain at 100. Similarly, whenever the reading in the second panel equals 0, it means that the current distance is the highest (on the negative side) since 20 periods ago. If it keeps increasing, then the reading will remain at 0. It becomes clear that now we should have the following trading conditions:

- For a bullish (Long) signal, the distance must be 0 in anticipation of mean-reversion.
- For a bearish (Short) signal, the distance must be 100 in anticipation of mean-reversion.



The signals generated by a 20-period normalized values of the 20-period adaptive moving average seem to be better than random. However, as stated, visual representation does not mean anything, and proper back-testing is needed.

ADAPTIVE DOUBLE MOVING AVERAGE CROSS

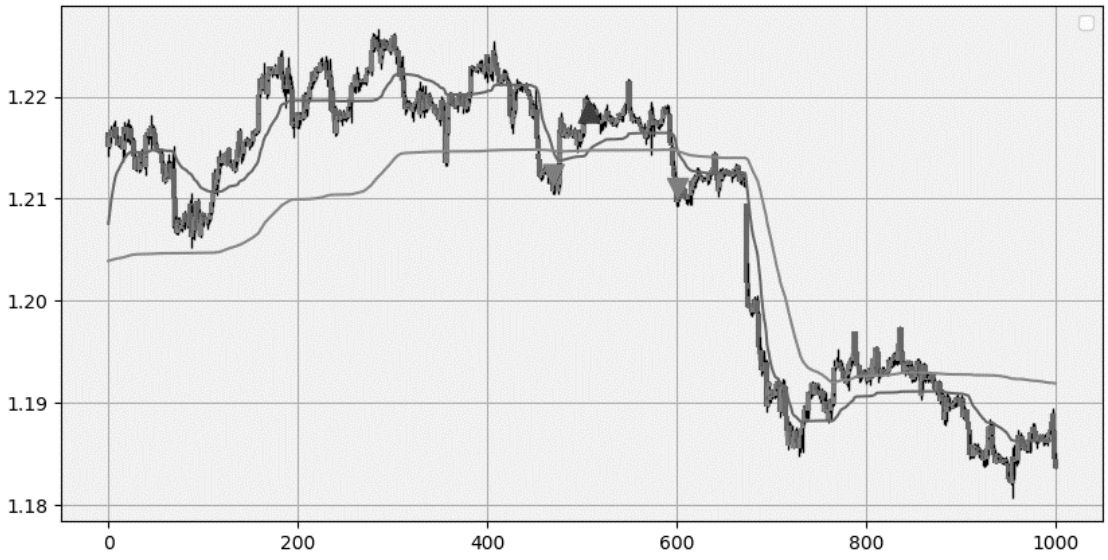
"To protect the world from devastation!"

The main idea is that when two moving averages cross, a new trend may be emerging. When a short-term moving average crosses a long-term moving average, it can be a signal that the regime is switching. The conditions required for the moving average cross:

- For a bullish (Long) signal, the short-term moving average must surpass the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be below the long-term moving average.
- For a bearish (Short) signal, the short-term moving average must break the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be above the long-term moving average.

The below snippet shows how to calculate a 50-period adaptive moving average and a 200-period adaptive moving average on the OHLC array of historical data we have recently imported to the Python interpreter.

```
# Using the function
short_term_ma = 50
long_term_ma  = 200
my_data = kama (my_data, 3, 4, short_term_ma)
my_data = kama (my_data, 3, 5, long_term_ma)
```



The same signal function is used as with other double moving average cross strategies. The adaptive version looks more stable, and its intention is to reduce a little the whipsaws which is why it is extremely important to do more research on this type of cross.

ADAPTIVE TRIPLE MOVING AVERAGE CROSS

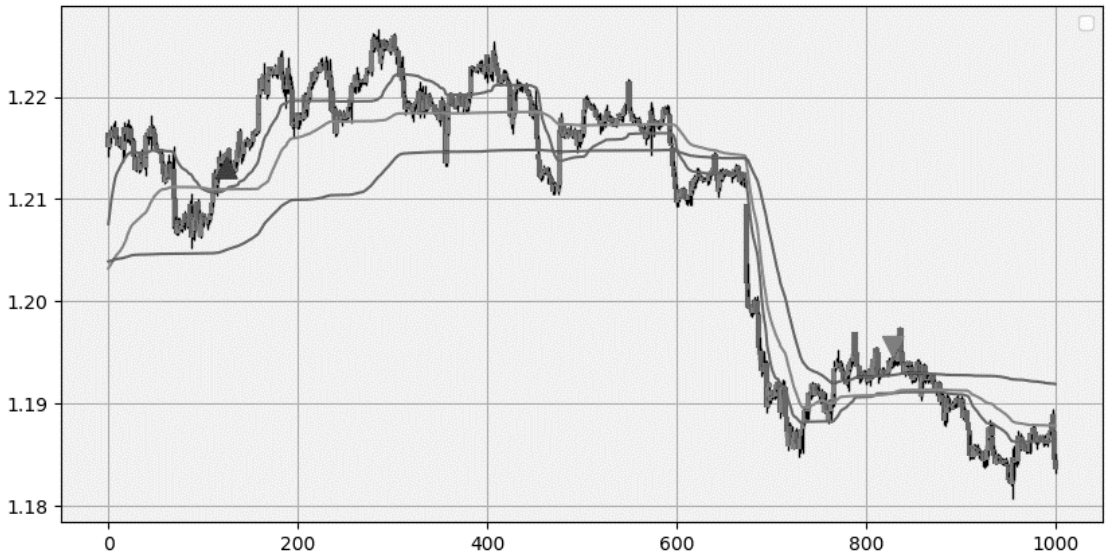
"To unite all peoples within our nation!"

The main idea is that when three moving averages cross, a new trend may be emerging. When a short-term moving average crosses a medium-term and a long-term moving average, it can be a signal that the regime is switching. The conditions required for the triple moving average cross:

- For a bullish (Long) signal, the short-term moving average must surpass the medium-term moving average and the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be below the medium-term moving average.
- For a bearish (Short) signal, the short-term moving average must break the medium-term moving average and the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be above the medium moving average.

The below snippet shows how to calculate a 50-period adaptive moving average, a 100-period adaptive moving average, and a 200-period adaptive moving average on the OHLC array of historical data we have recently imported to the Python interpreter.

```
# Using the function
ma_short   = 50
ma_medium   = 100
ma_long    = 200
my_data = kama(my_data, 3, 4, ma_short)
my_data = kama (my_data, 3, 5, ma_medium)
my_data = kama (my_data, 3, 6, ma_long)
```



The full Python code to get signals from the moving average cross can be as follows:

```
def signal(Data, short_term_ma, medium_term_ma, long_term_ma, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, short_term_ma] > Data[i, medium_term_ma] and Data[i, short_term_ma] > Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] < Data[i - 1, medium_term_ma]:
            Data[i, buy] = 1
        elif Data[i, short_term_ma] < Data[i, medium_term_ma] and Data[i, short_term_ma] < Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] > Data[i - 1, medium_term_ma]:
            Data[i, sell] = -1
    return Data
```

TRIANGULAR MOVING AVERAGE CROSS

"To denounce the evils of truth and love!"

What is called a Triangular Moving Average is simply the moving average of the moving average itself and this is to provide an extra layer of smoothing that can act as a dynamic support or resistance level. Surely it is not as reactive as the first moving average but in stable times, it can provide excellent reactionary levels.

The disadvantage of the Triangular Moving Average is that sometimes it tends to be far away from the market price when there is a strong trend. This situation makes it less useful to determine a close support or resistance level. The one seen in the next plot on the USDCAD with the first bearish trend. Notice how the simple 100-period moving average is close to the market price and can provide dynamic reactionary levels but the Triangular Moving Average is quite far.

To code the Triangular Moving Average, we simply write the ma function again but apply it on the moving average column.

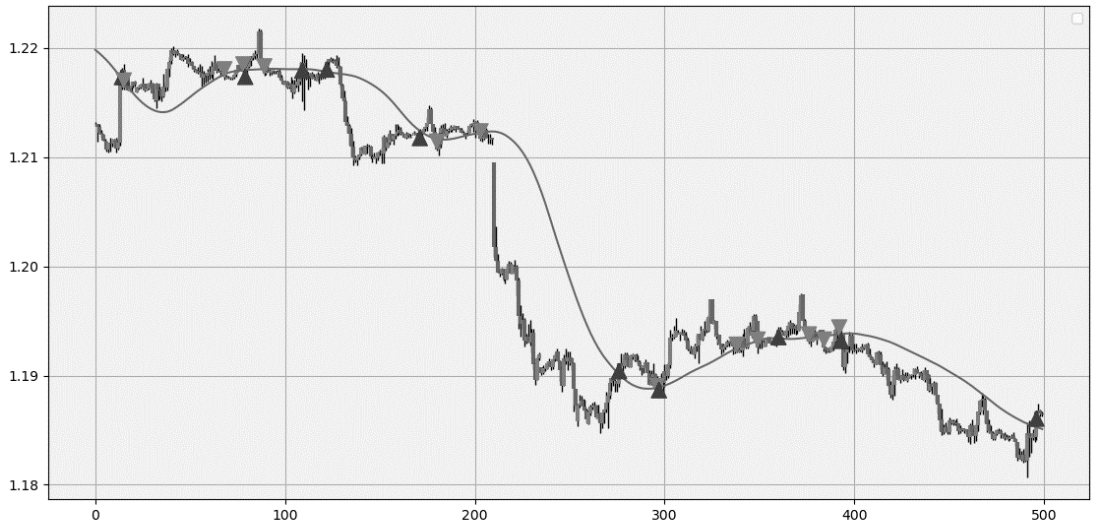
 # Consider my_data is the OHLC data array with two extra free columns

my_data = ma(my_data, lookback, closing_price, where_to_put_ma)

my_data = ma(my_data, lookback, where_to_put_ma, where_to_put_tma)

 # the variable where_to_put_tma refers to the index of the column where you want to put the Triangular Moving Average

Its main weakness is that it is very slow to react to the market's movements as it is an average of an average. However, it tends to act well as a dynamic support and resistance level. By using the condition fix from the last cross example on the adaptive moving average, we can get the following signal chart on the 30-period triangular moving average.



```
def signal(Data, close, ma_col, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, close] > Data[i, ma_col] and Data[i - 1, close] < Data[i - 1, ma_col] and \
            Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0 and \
            Data[i - 5, buy] == 0 and Data[i - 6, buy] == 0 and Data[i - 7, buy] == 0 and \
            Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0 and \
            Data[i - 5, sell] == 0 and Data[i - 6, sell] == 0 and Data[i - 7, sell] == 0:
            Data[i, buy] = 1
        elif Data[i, close] < Data[i, ma_col] and Data[i - 1, close] > Data[i - 1, ma_col] and \
            Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0 and \
            Data[i - 5, buy] == 0 and Data[i - 6, buy] == 0 and Data[i - 7, buy] == 0 and \
            Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0 and \
            Data[i - 5, sell] == 0 and Data[i - 6, sell] == 0 and Data[i - 7, sell] == 0:
            Data[i, sell] = -1
    return Data
```

TRIANGULAR MOVING AVERAGE DISTANCE

"Prepare for trouble!"

The conditions required for the moving average distance trade:

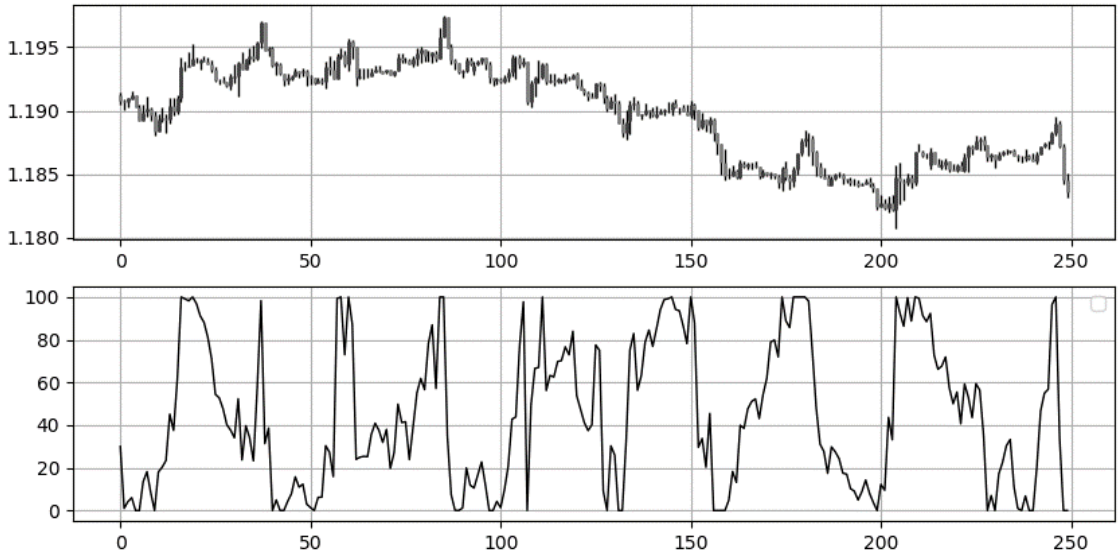
- For a bullish (Long) signal, the market price must close at the normalized lower barrier which represents the absolute negative distance.
- For a bearish (Short) signal, the market price must close at the normalized upper barrier which represents the absolute positive distance.

As we have already seen how to calculate a triangular moving average, we can now calculate the distance which is simply the difference between the market price and the current moving average. In two lines, we can get the desired results:

```
my_data = adder(my_data, 2)
my_data[:, 5] = my_data[:, 3] - my_data[:, 4]
```

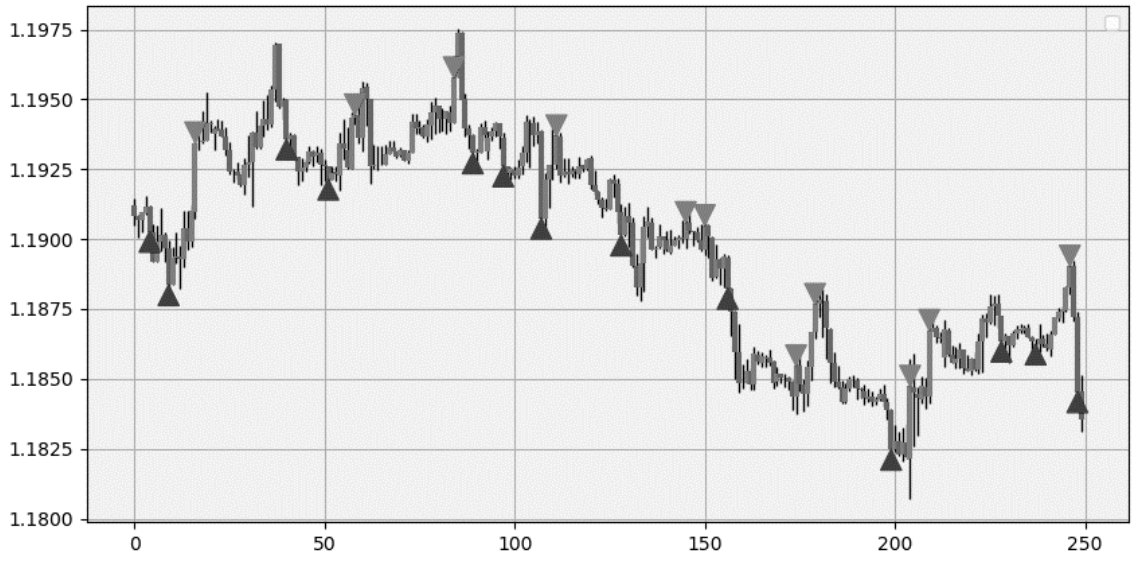
The above code will create two new columns where the first one will be populated by the distance calculation discussed above and the second one will be the normalized value of the distance. In the first part of the book, we have seen what normalization is.

```
normalization_period = 20
my_data = stochastic(my_data, normalization_period, 5, 6, genre = 'Normalization')
```



The above plot shows the hourly values of the EURUSD in the first panel with the 20-period normalized distance from its 20-period simple moving average. In layman's terms, whenever the reading in the second panel equals 100, it means that the current distance is the highest (on the positive side) since 20 periods ago. If it keeps increasing, then the reading will remain at 100. Similarly, whenever the reading in the second panel equals 0, it means that the current distance is the highest (on the negative side) since 20 periods ago. If it keeps increasing, then the reading will remain at 0. It becomes clear that now we should have the following trading conditions:

- For a bullish (Long) signal, the distance must be 0 in anticipation of mean-reversion.
- For a bearish (Short) signal, the distance must be 100 in anticipation of mean-reversion.



It looks like the triangular moving average also offers some stability like the adaptive version. Up until now, I think it is safe to see that two moving averages are worth pursuing:

- The adaptive moving average for its stability and account for volatility.
- The triangular moving average for its stability.

TRIANGULAR DOUBLE MOVING AVERAGE CROSS

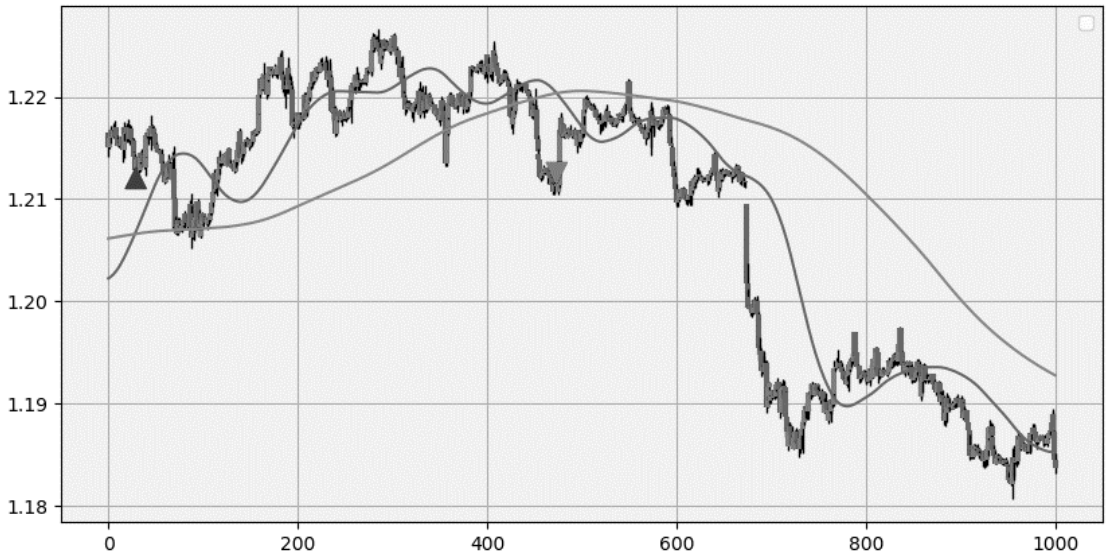
"Make it double!"

The main idea is that when two moving averages cross, a new trend may be emerging. When a short-term moving average crosses a long-term moving average, it can be a signal that the regime is switching. The conditions required for the moving average cross:

- For a bullish (Long) signal, the short-term moving average must surpass the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be below the long-term moving average.
- For a bearish (Short) signal, the short-term moving average must break the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be above the long-term moving average.

The below snippet shows how to calculate a 50-period simple triangular moving average and a 200-period simple triangular moving average on the OHLC array of historical data we have recently imported to the Python interpreter.

```
# Using the function
short_term_ma = 50
long_term_ma = 200
Data = ma(Data, short_term_ma, 3, 5)
Data = ma(Data, short_term_ma, 5, 6)
Data = deleter(Data, 5, 1)
Data = ma(Data, long_term_ma, 3, 6)
Data = ma(Data, long_term_ma, 6, 7)
Data = deleter(Data, 6, 1)
```



The full Python code to get signals from the moving average cross can be as follows:

```
def signal(Data, short_term_ma, long_term_ma, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, short_term_ma] > Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] < Data[i - 1, long_term_ma]:
            Data[i, buy] = 1
        elif Data[i, short_term_ma] < Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] > Data[i - 1, long_term_ma]:
            Data[i, sell] = -1
    return Data
```

TRIANGULAR TRIPLE MOVING AVERAGE CROSS

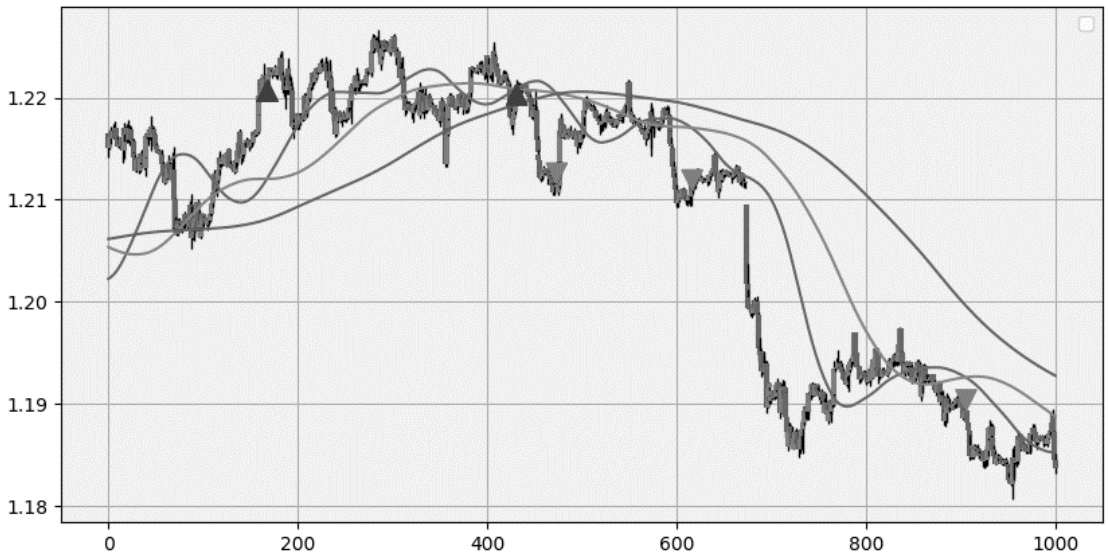
"To protect the world from devastation!"

The main idea is that when three moving averages cross, a new trend may be emerging. When a short-term moving average crosses a medium-term and a long-term moving average, it can be a signal that the regime is switching. The conditions required for the triple moving average cross:

- For a bullish (Long) signal, the short-term moving average must surpass the medium-term moving average and the long-term moving average.
- For a bearish (Short) signal, the short-term moving average must break the medium-term moving average and the long-term moving average.

The next snippet shows how to calculate a 50-period simple triangular moving average, a 100-period simple triangular moving average, and a 200-period simple triangular moving average on the OHLC array of historical data we have recently imported to the Python interpreter.

```
# Using the function
short_term_ma = 50
medium_term_ma = 100
long_term_ma = 200
Data = ma(Data, short_term_ma, 3, 5)
Data = ma(Data, short_term_ma, 5, 6)
Data = deleter(Data, 5, 1)
Data = ma(Data, medium_term_ma, 3, 6)
Data = ma(Data, medium_term_ma, 6, 7)
Data = deleter(Data, 6, 1)
Data = ma(Data, long_term_ma, 3, 7)
Data = ma(Data, long_term_ma, 7, 8)
Data = deleter(Data, 7, 1)
```



The full Python code to get signals from the moving average cross can be as follows:

```
def signal(Data, short_term_ma, medium_term_ma, long_term_ma, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, short_term_ma] > Data[i, medium_term_ma] and Data[i, short_term_ma] > Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] < Data[i - 1, medium_term_ma]:
            Data[i, buy] = 1
        elif Data[i, short_term_ma] < Data[i, medium_term_ma] and Data[i, short_term_ma] < Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] > Data[i - 1, medium_term_ma]:
            Data[i, sell] = -1
    return Data
```

WEIGHTED MOVING AVERAGE CROSS

"To unite all peoples within our nation!"

Also referred to as a linear-weighted moving average, the weighted moving average is a simple moving average that places more weight on recent data. The most recent observation has the biggest weight and each one prior to it has a progressively decreasing weight. Intuitively, it has less lag than the other moving averages, but it is also the least used, and hence, what it gains in lag reduction, it loses in popularity.

Mathematically speaking, it can be written down as:

$$\text{LWMA} = \frac{(P_n * W_1) + (P_{n-1} * W_2) + (P_{n-2} * W_3) \ldots}{\sum W}$$

Basically, if we have a dataset composed of two numbers [1, 2] and we want to calculate a linear weighted average, then we will do the following:

- (2 x 2) + (1 x 1) = 5
- 5 / 3 = 1.66

This assumes a time series with the number 2 as being the most recent observation.

```
def lwma(Data, lookback):
    weighted = []
    for i in range(len(Data)):
        try:
            total = np.arange(1, lookback + 1, 1)
            matrix = Data[i - lookback + 1: i + 1, 3:4]
            matrix = np.ndarray.flatten(matrix)
            matrix = total * matrix
            wma = (matrix.sum()) / (total.sum())
            weighted = np.append(weighted, wma)
        except ValueError:
            pass
    Data = Data[lookback - 1:, ]
    weighted = np.reshape(weighted, (-1, 1))
    Data = np.concatenate((Data, weighted), axis = 1)
    return Data
```

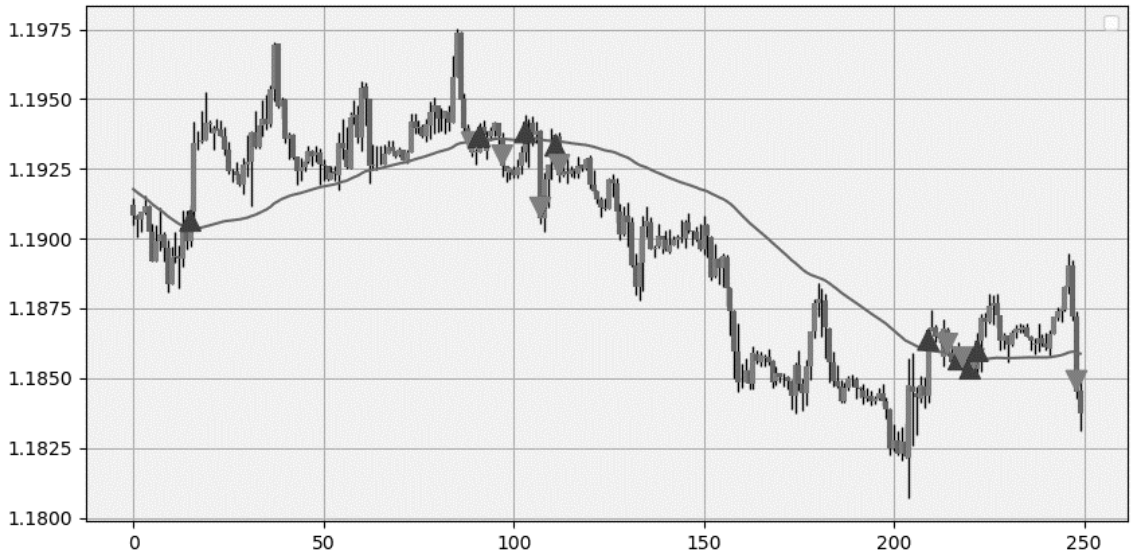
 # For this function to work, you need to have an OHLC array composed of the four usual columns, then you can use the below syntax to get a data array with the weighted moving average using the lookback you need

```
my_ohlc_data = lwma(my_ohlc_data, 20)
```

The linear-weighted moving average is quick to react and takes into account the latest values, but it is not very used in the field and not the best moving average to detect dynamic support and resistance levels. The conditions required for the moving average cross:

- For a bullish (Long) signal, the market price must close above the current moving average while the previous market price closing below the previous moving average.

- For a bearish (Short) signal, the market price must close below the current moving average while the previous market price closing above the previous moving average.



The above chart shows signals generated based on a 100-period weighted moving average using the same signal function as the original cross strategy. Try to code the whole strategy yourself using the concepts from the first part and then try to optimize it by looping through different lookback periods and tweaking the signal function.

WEIGHTED MOVING AVERAGE DISTANCE

"To denounce the evils of truth and love!"

The conditions required for the moving average distance trade:

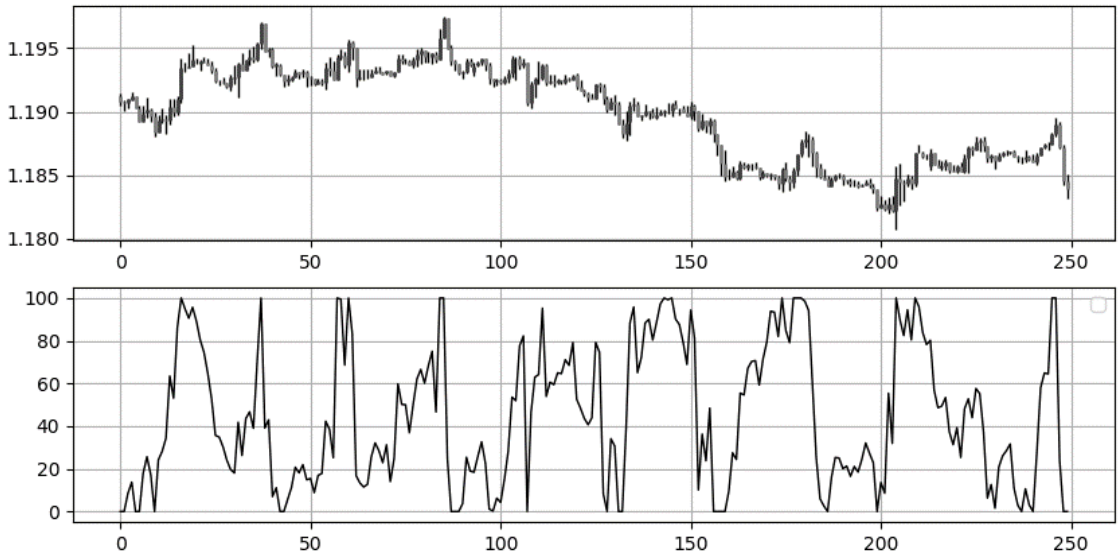
- For a bullish (Long) signal, the market price must close at the normalized lower barrier which represents the absolute negative distance.
- For a bearish (Short) signal, the market price must close at the normalized upper barrier which represents the absolute positive distance.

As we have already seen how to calculate a weighted moving average, we can now calculate the distance which is simply the difference between the market price and the current moving average. In two lines, we can get the desired results:

```
my_data = adder(my_data, 2)
my_data[:, 5] = my_data[:, 3] - my_data[:, 4]
```

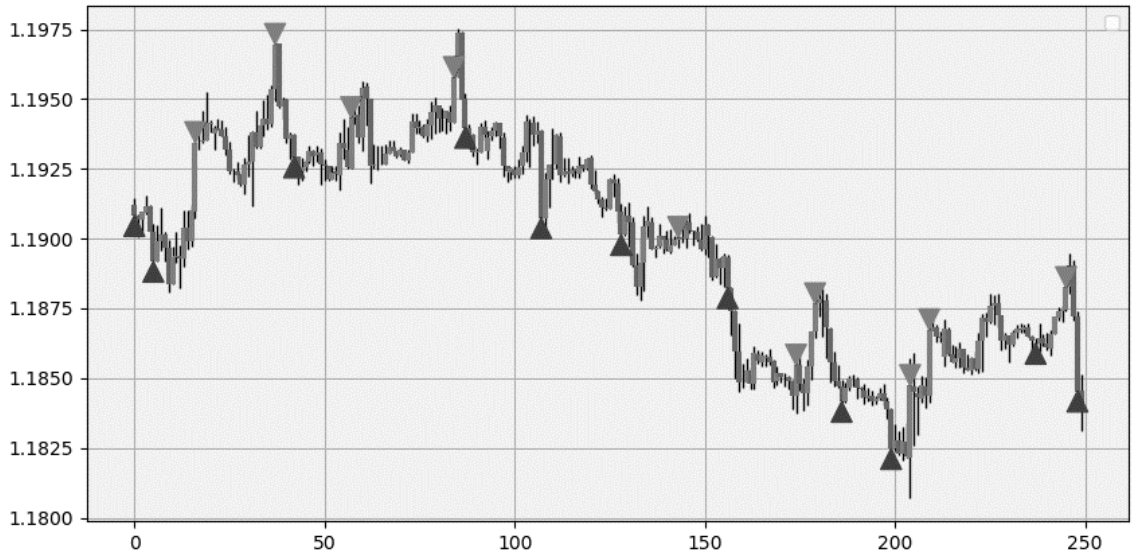
The above code will create two new columns where the first one will be populated by the distance calculation discussed above and the second one will be the normalized value of the distance. In the first part of the book, we have seen what normalization is.

```
normalization_period = 20
my_data = stochastic(my_data, normalization_period, 5, 6, genre = 'Normalization')
```



The above plot shows the hourly values of the EURUSD in the first panel with the 20-period normalized distance from its 20-period weighted moving average. It becomes clear that now we should have the following trading conditions:

- For a bullish (Long) signal, the distance must be 0 in anticipation of mean-reversion.
- For a bearish (Short) signal, the distance must be 100 in anticipation of mean-reversion.




```
def signal(Data, normalized_distance, buy, sell):
    Data = adder(Data, 2)
    for i in range(len(Data)):
        if Data[i, normalization_distance] == 0.0000 and Data[i - 1, buy] == 0 and \
            Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0:
            Data[i, buy] = 1
        elif Data[i, normalization_distance] == 100.0000 and Data[i - 1, sell] == 0 and \
            Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0:
            Data[i, sell] = -1
    return Data
my_data = signal(my_data, 6, 7, 8)
```

WEIGHTED DOUBLE MOVING AVERAGE CROSS

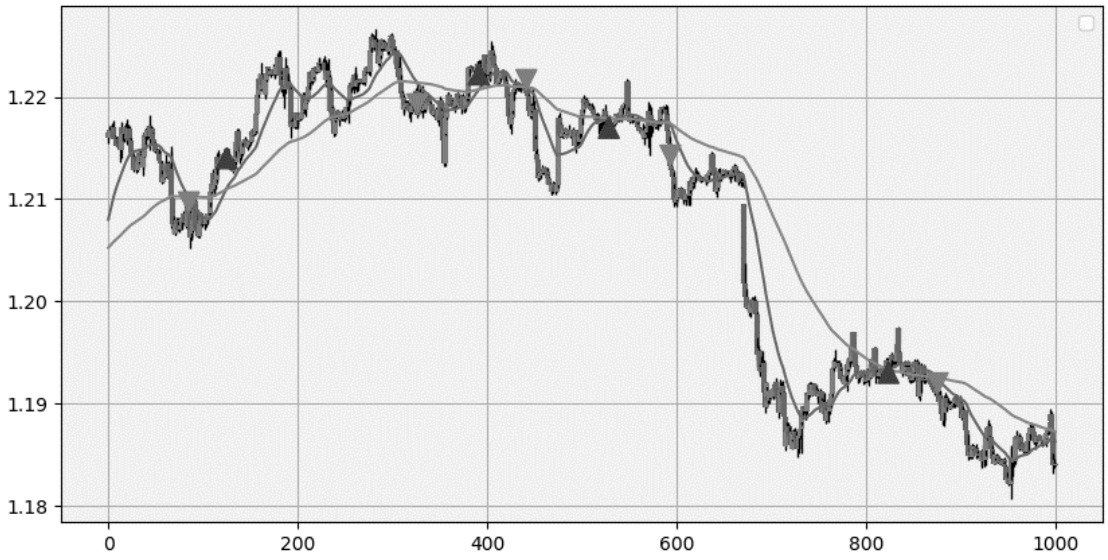
"To extend our reach to the stars above!"

The main idea is that when two moving averages cross, a new trend may be emerging. When a short-term moving average crosses a long-term moving average, it can be a signal that the regime is switching. The conditions required for the moving average cross:

- For a bullish (Long) signal, the short-term moving average must surpass the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be below the long-term moving average.
- For a bearish (Short) signal, the short-term moving average must break the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be above the long-term moving average.

The below snippet shows how to calculate a 50-period weighted moving average and a 200-period weighted moving average on the OHLC array of historical data we have recently imported to the Python interpreter.

```
# Using the function
short_term_ma = 50
long_term_ma = 200
my_data = lwma(my_data, short_term_ma, 3)
my_data = lwma(my_data, long_term_ma, 3)
```



The full Python code to get signals from the moving average cross can be as follows:

```
def signal(Data, short_term_ma, long_term_ma, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, short_term_ma] > Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] < Data[i - 1, long_term_ma]:
            Data[i, buy] = 1
        elif Data[i, short_term_ma] < Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] > Data[i - 1, long_term_ma]:
            Data[i, sell] = -1
    return Data
```

WEIGHTED TRIPLE MOVING AVERAGE CROSS

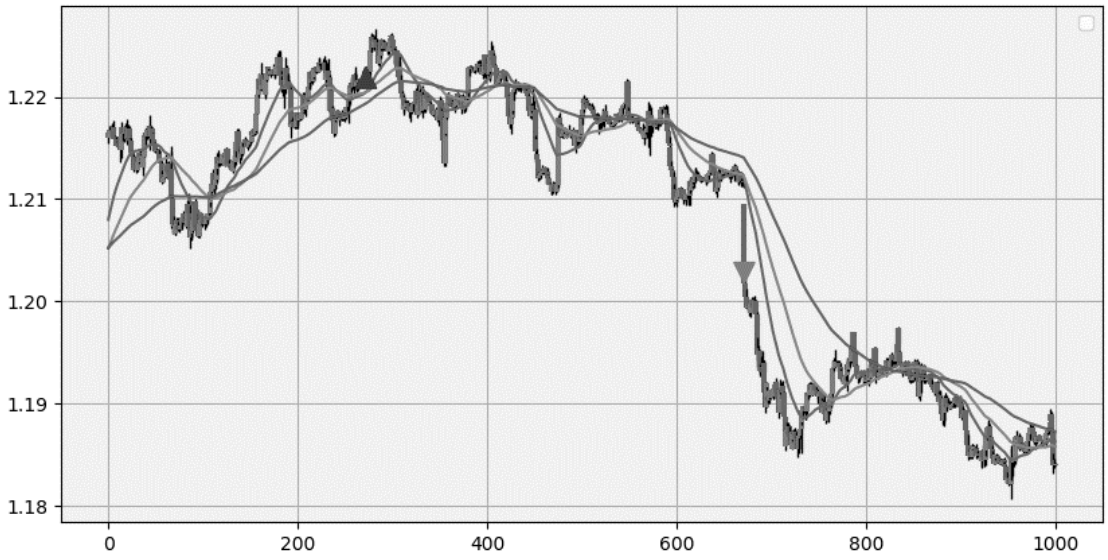
"Jessie! James!"

The main idea is that when three moving averages cross, a new trend may be emerging. When a short-term moving average crosses a medium-term and a long-term moving average, it can be a signal that the regime is switching. The conditions required for the triple moving average cross:

- For a bullish (Long) signal, the short-term moving average must surpass the medium-term moving average and the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be below the medium-term moving average.
- For a bearish (Short) signal, the short-term moving average must break the medium-term moving average and the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be above the medium moving average.

The below snippet shows how to calculate a 50-period weighted moving average, a 100-period weighted moving average, and a 200-period weighted moving average on the OHLC array of historical data we have recently imported to the Python interpreter.

```
# Using the function
short_term_ma = 50
medium_term_ma = 100
long_term_ma = 200
my_data = lwma(my_data, short_term_ma, 3)
my_data = lwma(my_data, medium_term_ma, 3)
my_data = lwma(my_data, long_term_ma, 3)
```



The full Python code to get signals from the moving average cross can be as follows:

```
def signal(Data, short_term_ma, medium_term_ma, long_term_ma, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, short_term_ma] > Data[i, medium_term_ma] and Data[i, short_term_ma] > Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] < Data[i - 1, medium_term_ma]:
            Data[i, buy] = 1
        elif Data[i, short_term_ma] < Data[i, medium_term_ma] and Data[i, short_term_ma] < Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] > Data[i - 1, medium_term_ma]:
            Data[i, sell] = -1
    return Data
```

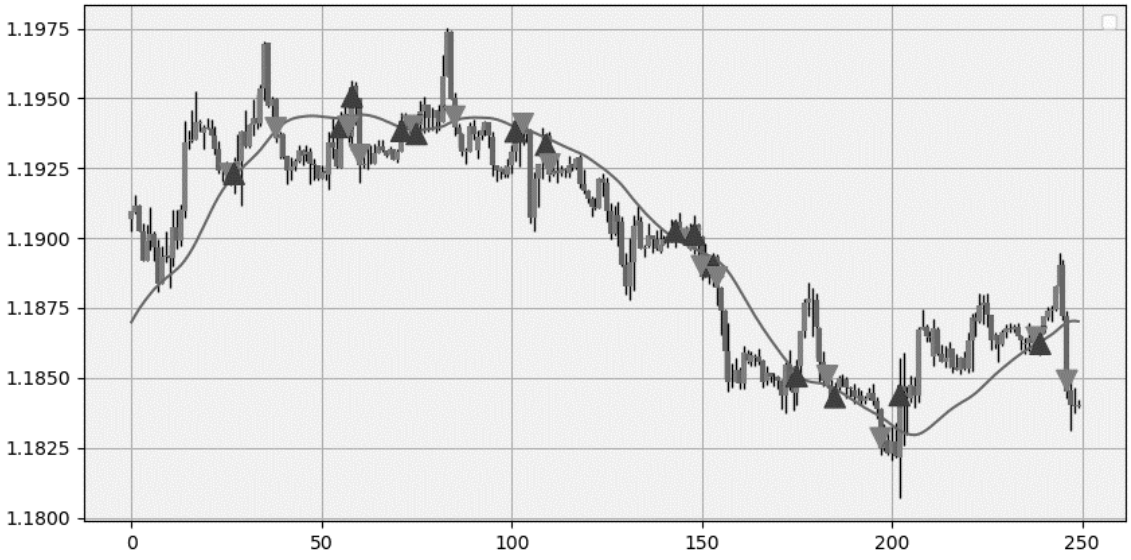
HULL MOVING AVERAGE CROSS

"Team Profit blasts off at the speed of light!"

The Hull moving average uses the weighted moving average as a building block, and it is calculated following the below steps:

- Choose a lookback period such as 20 or 100 and calculate the weighted moving average of the closing price.
- Divide the lookback period found in the first step and calculate the weighted moving average of the closing price using this new lookback period. If the number cannot by divided by two, then take the closest number before the comma (e.g. a lookback of 15 can be 7 or 8 as the second lookback).
- Multiply the second weighted moving average by two and subtract from it the first weighted moving average.
- As a final step, take the square root of the first lookback (e.g. if you have chosen a lookback of 100, then the third lookback period is 10) and calculate the weighted moving average on the latest result we have had in the third step. Be careful not to calculate it on the market price. Therefore, if we choose a lookback period of 100, we will calculate on 100 lookback period, then on 50, and finally, on 10 applied to the latest result.

```
def hull_moving_average(Data, what, lookback, where):
    Data = lwma(Data, lookback, what)
    second_lookback = round((lookback / 2), 1)
    second_lookback = int(second_lookback)
    Data = lwma(Data, second_lookback, what)
    Data = adder(Data, 1)
    Data[:, where + 2] = ((2 * Data[:, where + 1]) - Data[:, where])
    third_lookback = round(np.sqrt(lookback), 1)
    third_lookback = int(third_lookback)
    Data = lwma(Data, third_lookback, where + 2)
    return Data
```



Why do we say that the Hull moving average reduces lag? The answer comes mainly from the building blocks which are the weighted moving averages. They place more weights on more recent values. Furthermore, the lag is also reduced by offsetting one weighted moving average with the one that has half the lookback period. Finally, we take the square root of the lookback period and apply it on the moving average itself using the weighting method. This gives us a moving average close to the market price and serves as an early trend reversal indicator. However, is that a good thing for a cross

strategy? As can be seen from the above chart, there are a lot of false signals because the 100-period Hull moving average is passing through the price quite often. This can be fixed by tweaking the lookback period.

HULL MOVING AVERAGE DISTANCE

"Surrender now or prepare to lose!"

The conditions required for the moving average distance trade:

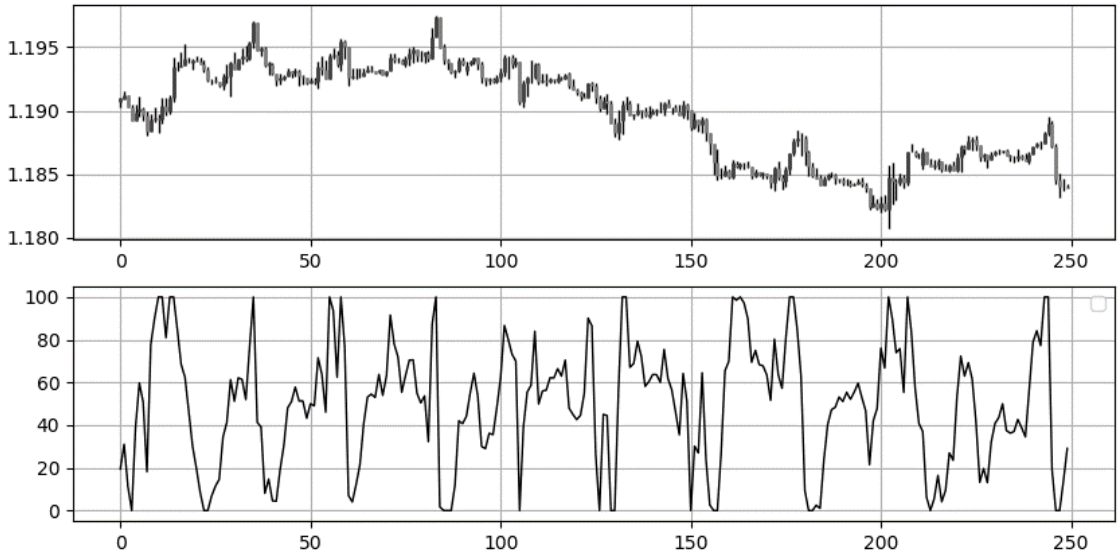
- For a bullish (Long) signal, the market price must close at the normalized lower barrier which represents the absolute negative distance from the current moving average.
- For a bearish (Short) signal, the market price must close at the normalized upper barrier which represents the absolute positive distance from the current moving average.

As we have already seen how to calculate a Hull moving average, we can now calculate the distance which is simply the difference between the market price and the current moving average. In two lines, we can get the desired results:

```
my_data = adder(my_data, 2)
my_data[:, 5] = my_data[:, 3] - my_data[:, 4]
```

The above code will create two new columns where the first one will be populated by the distance calculation discussed above and the second one will be the normalized value of the distance. In the first part of the book, we have seen what normalization is.

```
normalization_period = 20
my_data = stochastic(my_data, normalization_period, 5, 6, genre = 'Normalization')
```



The above plot shows the hourly values of the EURUSD in the first panel with the 20-period normalized distance from its 20-period Hull moving average. In layman's terms, whenever the reading in the second panel equals 100, it means that the current distance is the highest (on the positive side) since 20 periods ago. If it keeps increasing, then the reading will remain at 100. Similarly, whenever the reading in the second panel equals 0, it means that the current distance is the highest (on the negative side) since 20 periods ago. If it keeps increasing, then the reading will remain at 0. It becomes clear that now we should have the following trading conditions:

- For a bullish (Long) signal, the distance must be 0 in anticipation of mean-reversion.
- For a bearish (Short) signal, the distance must be 100 in anticipation of mean-reversion.


```
def signal(Data, normalized_distance, buy, sell):
    Data = adder(Data, 2)
    for i in range(len(Data)):
        if Data[i, normalization_distance] == 0.0000 and Data[i - 1, buy] == 0 and \
            Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0:
            Data[i, buy] = 1
        elif Data[i, normalization_distance] == 100.0000 and Data[i - 1, sell] == 0 and \
            Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0:
            Data[i, sell] = -1
    return Data
my_data = signal(my_data, 6, 7, 8)
```

HULL DOUBLE MOVING AVERAGE CROSS

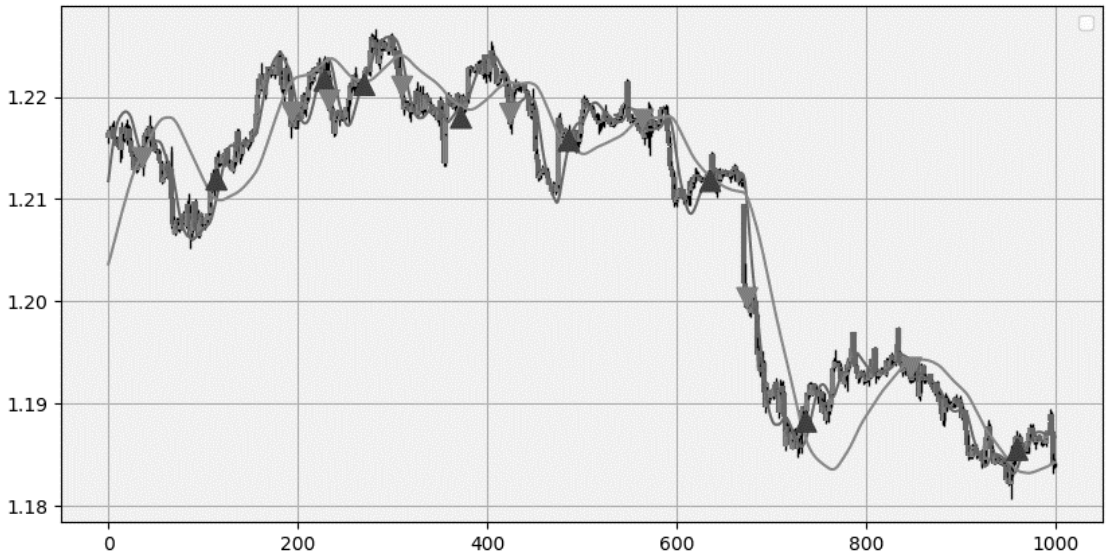
"Meowth! That's right!"

The main idea is that when two moving averages cross, a new trend may be emerging. When a short-term moving average crosses a long-term moving average, it can be a signal that the regime is switching. The conditions required for the moving average cross:

- For a bullish (Long) signal, the short-term moving average must surpass the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be below the long-term moving average.
- For a bearish (Short) signal, the short-term moving average must break the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be above the long-term moving average.

The below snippet shows how to calculate a 50-period Hull moving average and a 200-period Hull moving average on the OHLC array of historical data we have recently imported to the Python interpreter.

```
# Using the function
short_term_ma = 50
long_term_ma = 200
my_data = hull_moving_average (my_data, 3, short_term_ma, 4)
my_data = hull_moving_average (my_data, 3, long_term_ma, 5)
```



The full Python code to get signals from the moving average cross can be as follows:

```
def signal(Data, short_term_ma, long_term_ma, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, short_term_ma] > Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] < Data[i - 1, long_term_ma]:
            Data[i, buy] = 1
        elif Data[i, short_term_ma] < Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] > Data[i - 1, long_term_ma]:
            Data[i, sell] = -1
    return Data
```

HULL TRIPLE MOVING AVERAGE CROSS

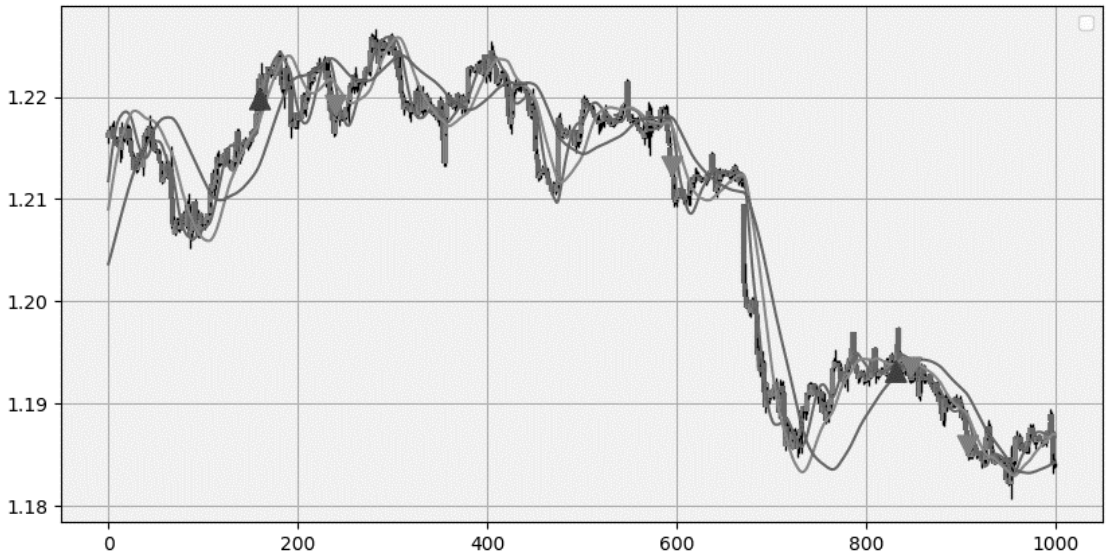
"The last triple thingy."

The main idea is that when three moving averages cross, a new trend may be emerging. When a short-term moving average crosses a medium-term and a long-term moving average, it can be a signal that the regime is switching. The conditions required for the triple moving average cross:

- For a bullish (Long) signal, the short-term moving average must surpass the medium-term moving average and the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be below the medium-term moving average.
- For a bearish (Short) signal, the short-term moving average must break the medium-term moving average and the long-term moving average. Algorithmically speaking, we need to impose a condition where the previous short-term moving average must be above the medium moving average.

The below snippet shows how to calculate a 50-period Hull moving average, a 100-period Hull moving average, and a 200-period Hull moving average on the OHLC array of historical data we have recently imported to the Python interpreter.

```
# Using the function
short_term_ma = 50
medium_term_ma = 100
long_term_ma = 200
my_data = hull_moving_average (my_data, 3, short_term_ma, 4)
my_data = hull_moving_average (my_data, 3, medium_term_ma, 5)
my_data = hull_moving_average (my_data, 3, long_term_ma, 6)
```



The full Python code to get signals from the moving average cross can be as follows:

```
def signal(Data, short_term_ma, medium_term_ma, long_term_ma, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, short_term_ma] > Data[i, medium_term_ma] and Data[i, short_term_ma] > Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] < Data[i - 1, medium_term_ma]:
            Data[i, buy] = 1
        elif Data[i, short_term_ma] < Data[i, medium_term_ma] and Data[i, short_term_ma] < Data[i, long_term_ma] and \
            Data[i - 1, short_term_ma] > Data[i - 1, medium_term_ma]:
            Data[i, sell] = -1
    return Data
```

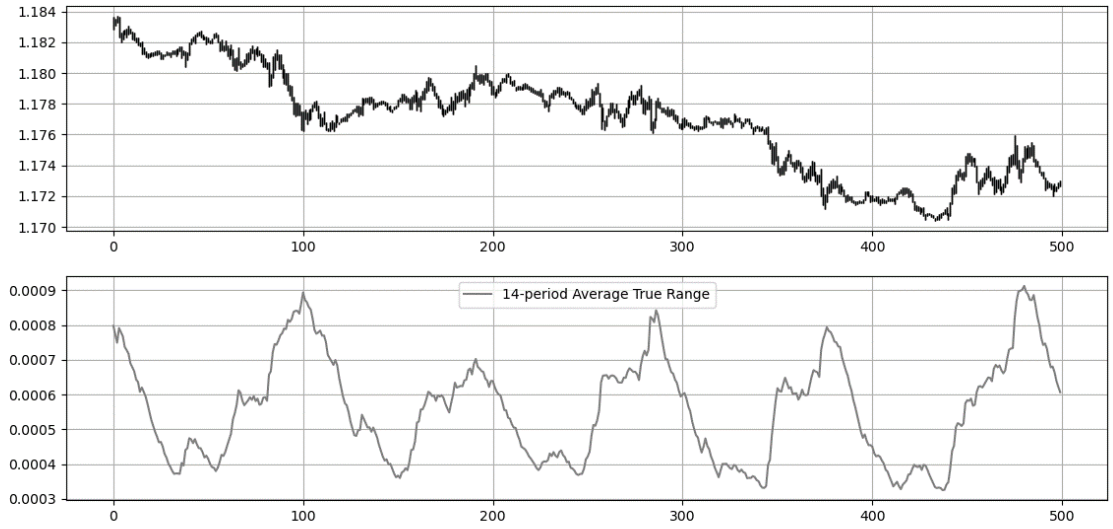
SUPERTREND CROSS

"To the moon!"

The first concept we should understand before creating the SuperTrend indicator is volatility. We sometimes measure volatility using the Average True Range. Although the ATR is considered a lagging indicator, it gives some insights as to where volatility is right now and where has it been last period. The ATR is described in detail in Appendix II at the end of the book. We will skim through it here since it is the main ingredient in the SuperTrend. But before that, we should understand how the True Range is calculated (the ATR is just the average of that calculation). The true range is simply the greatest of the three price differences:

- High — Low
- | High — Previous close |
- | Previous close — Low |

Once we have got the maximum out of the above three, we simply take an average of n periods of the true ranges to get the Average True Range. Generally, since in periods of panic and price depreciation we see volatility go up, the ATR will likely trend higher during these periods, similarly in times of steady uptrends or downtrends, the ATR will tend to go lower.



Now that we have understood what the ATR is and how to calculate it, we can proceed further with the SuperTrend indicator.

Now that we have our plan of attack laid out in front of us, we can proceed by understanding the SuperTrend and coding it.

- Calculate the ATR using the function provided above.
- Use the below formulas to calculate the SuperTrend.
- Plot the indicator, create the trading rules and analyze the results.

The indicator seeks to provide entry and exit levels for trend followers. You can think of it as a moving average or an MACD. Its uniqueness is its main advantage and although its calculation method is much more complicated than the other two indicators, it is intuitive in nature and not that hard to understand. Basically, we have two variables to choose from. The ATR lookback and the multiplier's value. The former is just the period used to calculate the ATR while the latter is generally an integer (usually 2 or 3).

$$Basic\ Upperband = \left(\frac{High + Low}{2}\right) + (multiplier\ .\ eATR)$$

$$Basic\ Lowerband = \left(\frac{High + Low}{2}\right) - (multiplier\ .\ eATR)$$

The first thing to do is to average the high and low of the price bar, then we will either add or subtract the average with the selected multiplier multiplied by the ATR as shown in the above formulas. This will give us two arrays, the basic upper band and the basic lower band which form the first building blocks in the SuperTrend indicator. The next step is to calculate the final upper band and the final lower band using the below formulas.

$$Final\ Upperband = If\ Current\ Basic\ Upperband\ < Previous\ Final\ Upperband\ OR$$
$$Previous\ Close > Previous\ Final\ Upperband\ Then$$
$$Current\ Basic\ Upperband\ Else\ Previous\ Final\ Upperband$$

$$Final\ Lowerband = If\ Current\ Basic\ Lowerband > Previous\ Final\ Lowerband\ OR$$
$$Previous\ Close < Previous\ Final\ Lowerband\ Then$$
$$Current\ Basic\ Lowerband\ Else\ Previous\ Final\ Lowerband$$

It may seem complicated but most of these conditions are repetitive and, in any case, I will provide the Python code so that you can play with the function and optimize it to your trading preferences. Finally, using the previous two calculations, we can find the SuperTrend.

$$SuperTrend = If\big((Previous\ SuperTrend = Previous\ Final\ Upperband) and\ (Current\ Close <= Current\ Final\ Upperband)\big) Then\ Current\ Final\ Upperband$$

$$If\big((Previous\ SuperTrend = Previous\ Final\ Upperband) and\ (Current\ Close > Current\ Final\ Upperband)\big)\ Then\ Current\ Final\ Lowerband$$

$$Else$$

$$If\big((Previous\ SuperTrend = Previous\ Final\ Lowerband) and\ (Current\ Close >= Current\ Final\ Lowerband)\big) Then\ Current\ Final\ Lowerband$$

$$Else$$

$$If\big((Previous\ SuperTrend = Previous\ Final\ Lowerband) and\ (Current\ Close < Current\ Final\ Upperband)\big) Then\ Current\ Final\ Upperband$$

```
def supertrend(Data, multiplier, atr_col, close, high, low, where):
    Data = adder(Data, 6)
    for i in range(len(Data)):
        # Average Price
        Data[i, where] = (Data[i, high] + Data[i, low]) / 2
        # Basic Upper Band
        Data[i, where + 1] = Data[i, where] + (multiplier * Data[i, atr_col])
        # Lower Upper Band
        Data[i, where + 2] = Data[i, where] - (multiplier * Data[i, atr_col])
    # Final Upper Band
    for i in range(len(Data)):
      if i == 0:
        Data[i, where + 3] = 0
      else:
        if (Data[i, where + 1] < Data[i - 1, where + 3]) or (Data[i - 1, close] > Data[i - 1,
where + 3]):
            Data[i, where + 3] = Data[i, where + 1]
        else:
            Data[i, where + 3] = Data[i - 1, where + 3]
    # Final Lower Band
    for i in range(len(Data)):
      if i == 0:
        Data[i, where + 4] = 0
      else:
        if (Data[i, where + 2] > Data[i - 1, where + 4]) or (Data[i - 1, close] < Data[i - 1,
where + 4]):
            Data[i, where + 4] = Data[i, where + 2]
        else:
            Data[i, where + 4] = Data[i - 1, where + 4]
```

```
    # SuperTrend
    for i in range(len(Data)):
        if i == 0:
            Data[i, where + 5] = 0
        elif (Data[i - 1, where + 5] == Data[i - 1, where + 3]) and (Data[i, close] <= Data[i,
where + 3]):
            Data[i, where + 5] = Data[i, where + 3]
        elif (Data[i - 1, where + 5] == Data[i - 1, where + 3]) and (Data[i, close] >  Data[i,
where + 3]):
            Data[i, where + 5] = Data[i, where + 4]
        elif (Data[i - 1, where + 5] == Data[i - 1, where + 4]) and (Data[i, close] >= Data[i,
where + 4]):
            Data[i, where + 5] = Data[i, where + 4]
        elif (Data[i - 1, where + 5] == Data[i - 1, where + 4]) and (Data[i, close] <  Data[i,
where + 4]):
            Data[i, where + 5] = Data[i, where + 3]
    Data = deleter(Data, where, 5)
    return Data
```

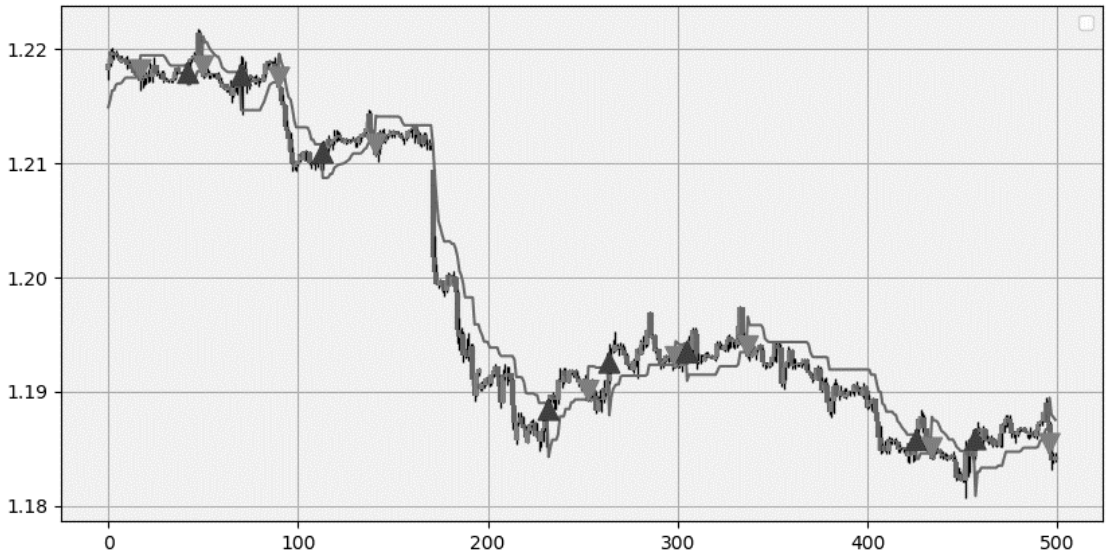
Applying the above function on an OHLC array will give us something like the next chart when we plot it. The way we should understand the indicator is that when it goes above the market price, we should be looking to short and when it goes below the market price, we should be looking to go long as we anticipate a bullish trend. Remember that the SuperTrend is a trend-following indicator. The aim here is to capture trends at the beginning and to close out when they are over.

The conditions required for the SuperTrend cross are as follows:

- Go long (Buy) whenever the market price turns above the SuperTrend Indicator.
- Go short (Sell) whenever the market price turns below the SuperTrend Indicator.

The below is the signal function that refers to the above conditions:

```
def super_trend_signals(Data, close, super_trend_column, buy, sell):
    for i in range(len(Data)):
        # Bullish SuperTrend cross
        if Data[i, close] > Data[i, super_trend_column] and Data[i - 1, close] < Data[i - 1,
super_trend_column]:
            Data[i, buy] = 1
        # Bearish SuperTrend cross
        if Data[i, close] < Data[i, super_trend_column] and Data[i - 1, close] > Data[i - 1,
super_trend_column]:
            Data[i, sell] = -1
    return Data
```

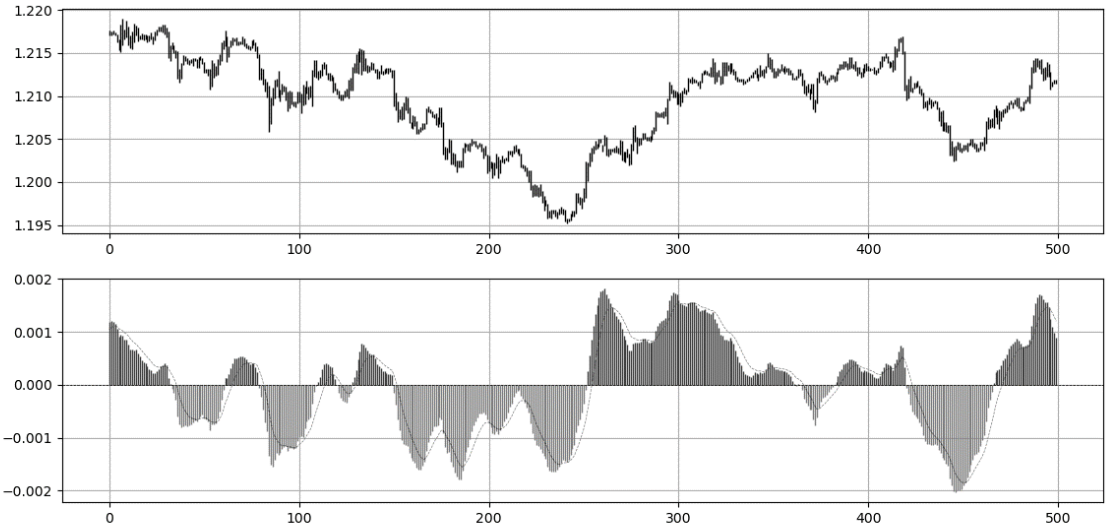


The SuperTrend indicator is a very powerful trend following technique which must be used right to harness its true power. Make sure you optimize the multiplier and play with the parameters of the Average True Range. Another thing to be aware of is that as any other trend following indicator out there, it must be used on trending markets and not when they are ranging.

MACD FLIP

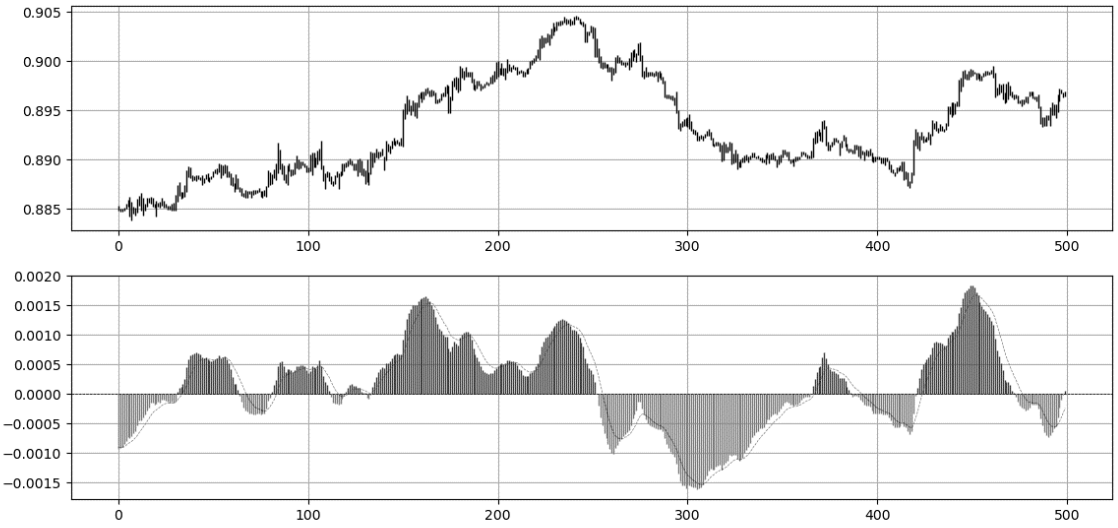
"It's pronounced Mak D."

The MACD is probably the second most known oscillator after the RSI. One that is heavily followed by traders. It stands for Moving Average Convergence Divergence, and it is used mainly for divergences and flips but also for crosses. Many people also consider it a trend-following indicator, but others use graphical analysis on it to find reversal points, making the MACD a versatile indicator.



How is the MACD calculated? It is the difference between the 26-period exponential moving Average applied to the closing price and the 12-period exponential moving average also applied to the closing price. The value found after taking the difference is called the MACD line. The 9-period exponential moving average of that calculation is called the MACD signal.

```
def macd(Data, what, long_ema, short_ema, signal_ema, where):
    Data = ema(Data, 2, long_ema,  what, where)
    Data = ema(Data, 2, short_ema, what, where + 1)
    Data[:, where + 2] = Data[:, where + 1] - Data[:, where]
    Data = jump(Data, long_ema)
    Data = ema(Data, 2, signal_ema, where + 2, where + 3)
    Data = deleter(Data, where, 2)
    Data = jump(Data, signal_ema)
    return Data
```

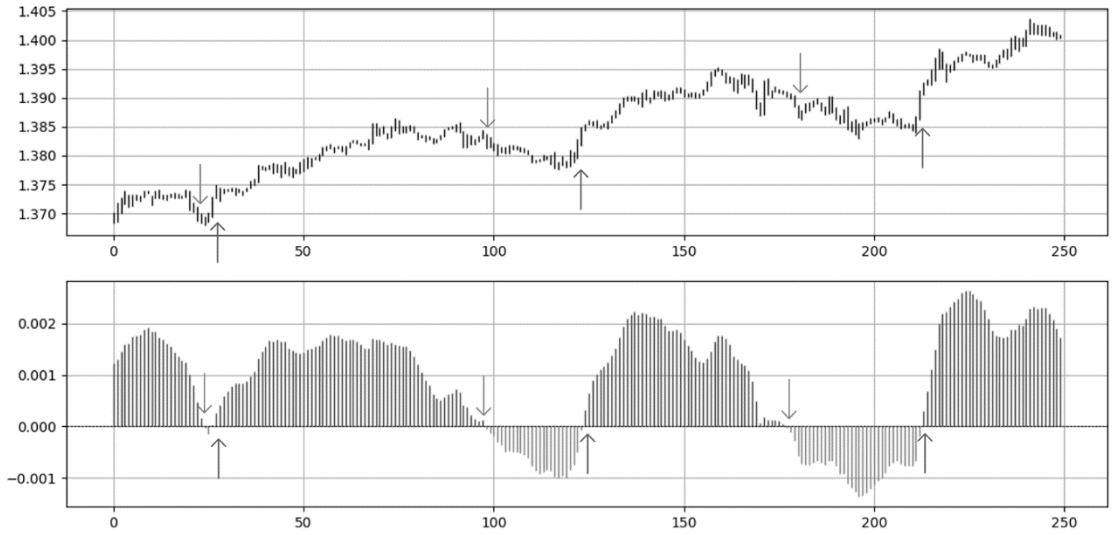


As a reminder, the MACD line is the difference between the two exponential moving averages which are plotted as histograms in green and red or for people reading this book, black and black. The MACD signal is simply the 9-period exponential moving average of the MACD line. It is the dashed line in the above plot. To plot the histograms, we can follow this function.

```
def indicator_plot_double_macd(Data, first, second, name = '', name_ind = '', window = 250):
    fig, ax = plt.subplots(2, figsize = (10, 5))
    Chosen = Data[-window:, ]
    for i in range(len(Chosen)):
        ax[0].vlines(x = i, ymin = Chosen[i, 2], ymax = Chosen[i, 1], color = 'black', linewidth = 1)
    ax[0].grid()
    for i in range(len(Chosen)):
        if Chosen[i, 5] > 0:
            ax[1].vlines(x = i, ymin = 0, ymax = Chosen[i, 5], color = 'green', linewidth = 1)
        if Chosen[i, 5] < 0:
            ax[1].vlines(x = i, ymin = Chosen[i, 5], ymax = 0, color = 'red', linewidth = 1)
        if Chosen[i, 5] == 0:
            ax[1].vlines(x = i, ymin = Chosen[i, 5], ymax = 0, color = 'black', linewidth = 1)
    ax[1].grid()
    ax[1].axhline(y = 0, color = 'black', linewidth = 0.5, linestyle = '--')
# Using the function
indicator_plot_double_macd(my_data, closing_price, macd_column_first, name = '', name_ind =
'MACD', window = 500)
plt.plot(my_data[-500:, macd_column_second], color = 'blue', linestyle = '--', linewidth = 0.5)
```

The conditions required for the MACD flip are:

- The current MACD line (not the signal line) is above the zero line with the previous MACD line below the zero line.
- The current MACD line (not the signal line) is below the zero line with the previous MACD line above the zero line.



The full Python code to get signals from the flip can be as follows:

```
def signal(Data, MACD, buy, sell):
    for i in range(len(Data)):
        if Data[i, MACD] > 0 and Data[i - 1, MACD] < 0:
            Data[i, buy] = 1
        if Data[i, MACD] < 0 and Data[i - 1, MACD] > 0:
            Data[i, sell] = -1
```

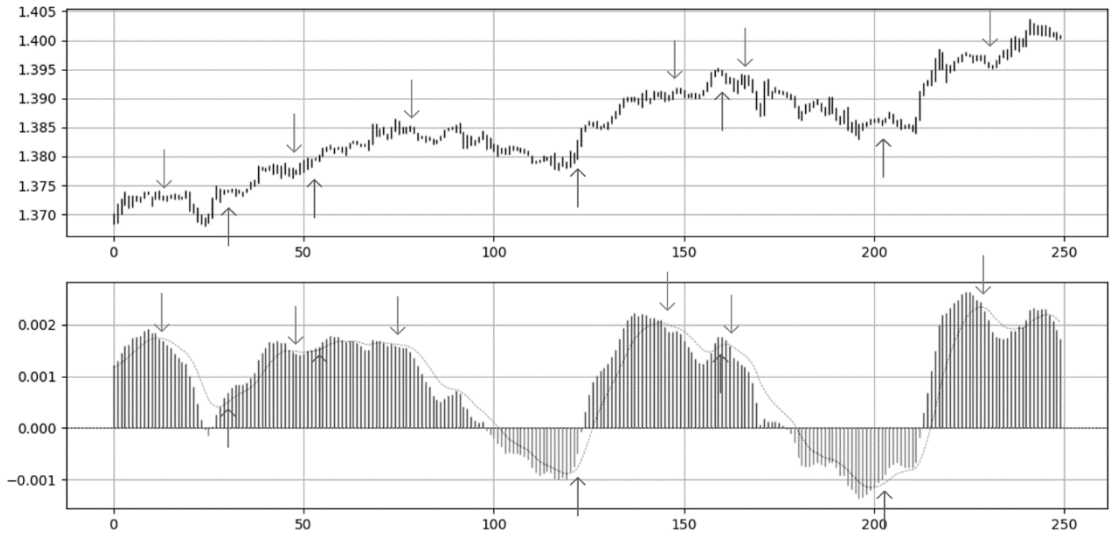
The MACD flip strategy is one of the most known trend confirmation strategies. Its simplicity makes it a favorite among retail traders. It does, however, have its limitations just as any other strategy out there. The most apparent one is the lag issue. As it is a cross between two moving averages, an embedded lag will always be there. We can try to fix this by trying to reduce the lag as much as possible. For example, we can use a Hull moving average in the calculation so that it reflects a more recent price action. Another solution is to lower the lookback periods and optimize them through back-testing.

MACD CROSS

"Seriously? Another cross?"

The main idea is that when the MACD signal line crosses over the MACD line, a trend confirmation signal is given. This is similar to the double cross strategies seen earlier in this part. The conditions required for the MACD cross are:

- Go long (Buy) whenever the MACD line (Histograms) surpasses the MACD signal (Dashed lines).
- Go short (Sell) whenever the MACD line (Histograms) breaks the MACD signal (Dashed lines).



The full Python code to get signals from the cross can be as follows:

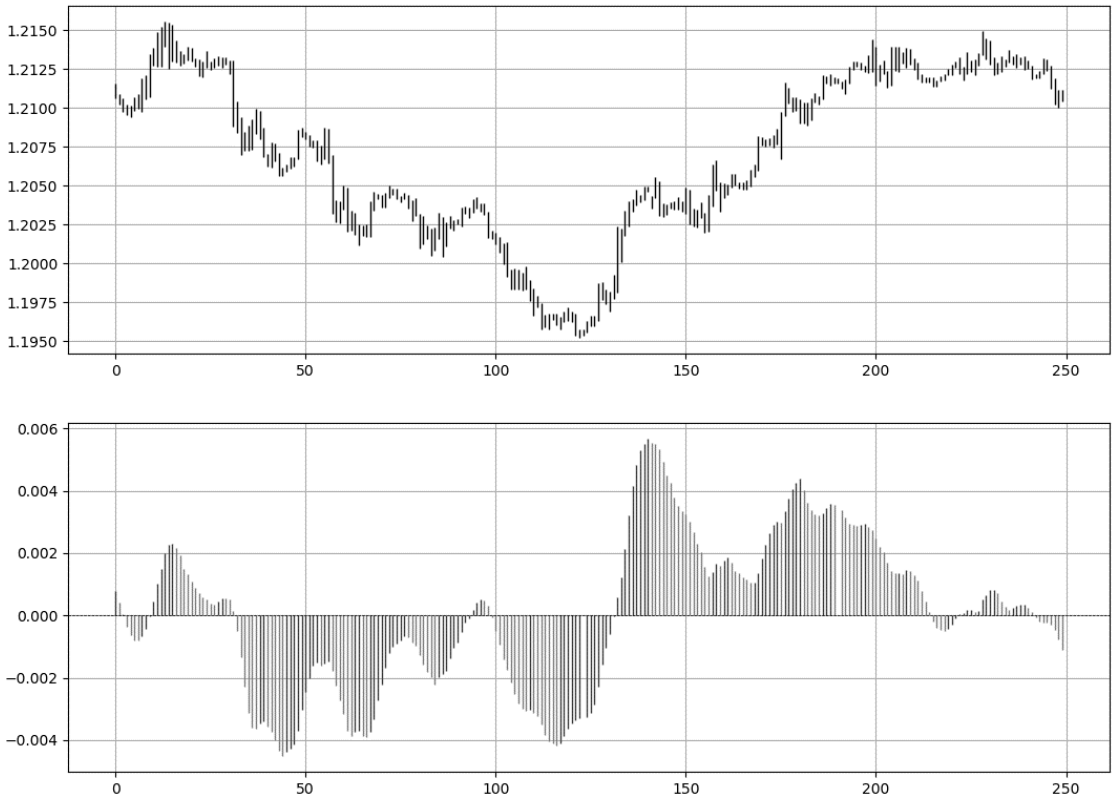
```
def signal(Data, macd_line, macd_signal, buy, sell):
    for i in range(len(Data)):
        if Data[i, macd_line] > Data[i, macd_signal] and Data[i - 1, macd_line] < Data[i - 1, macd_signal]:
            Data[i, buy] = 1
        if Data[i, macd_line] < Data[i, macd_signal] and Data[i - 1, macd_line] > Data[i - 1, macd_signal]:
            Data[i, sell] = -1
```

AWESOME OSCILLATOR FLIP

"The name is a bit misleading."

Created by Bill Williams, the Awesome Oscillator is a modified version of the MACD Oscillator. To construct the Awesome Oscillator, we can follow the below steps:

- Calculate the mid-point (current high minus the current low).
- Calculate a 5-period simple moving average on the mid-points.
- Calculate a 34-period simple moving average on the mid-points.
- Subtract the 5-period moving average from the 34-period moving average.



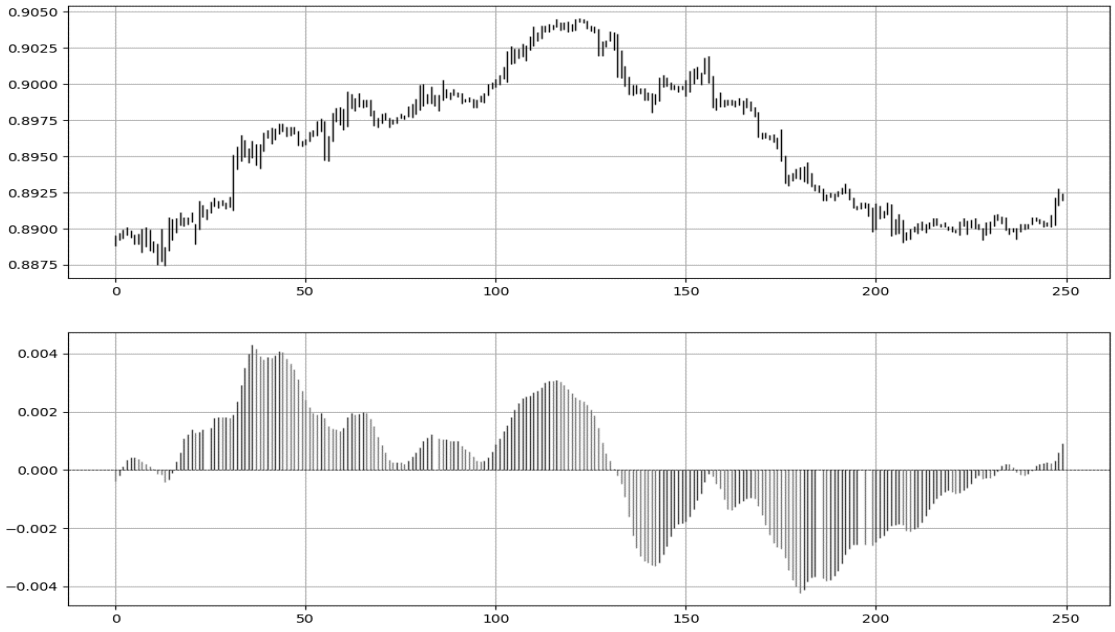
The above plot shows how the Awesome Oscillator looks. It has a color condition that states:

- Whenever the current value is greater than the previous value, the bar must have the color green.

- Whenever the current value is lower than the previous value, the bar must have the color red.

To code the Awesome Oscillator in python, we can use the below function on an OHLC array:

```
def awesome_oscillator(Data, high, low, long_ma, short_ma, where):
    # Mid-point Calculation
    Data[:, where] = (Data[:, high] + Data[:, low]) / 2
    # Calculating the short-term Simple Moving Average
    Data = ma(Data, short_ma, where, where + 1)
    # Calculating the long-term Simple Moving Average
    Data = ma(Data, long_ma, where, where + 2)
    # Calculating the Awesome Oscillator
    Data[:, where + 3] = Data[:, where + 1] - Data[:, where + 2]
    # Removing Excess Columns/Rows
    Data = jump(Data, long_ma)
    Data = deleter(Data, where, 3)
    return Data
```

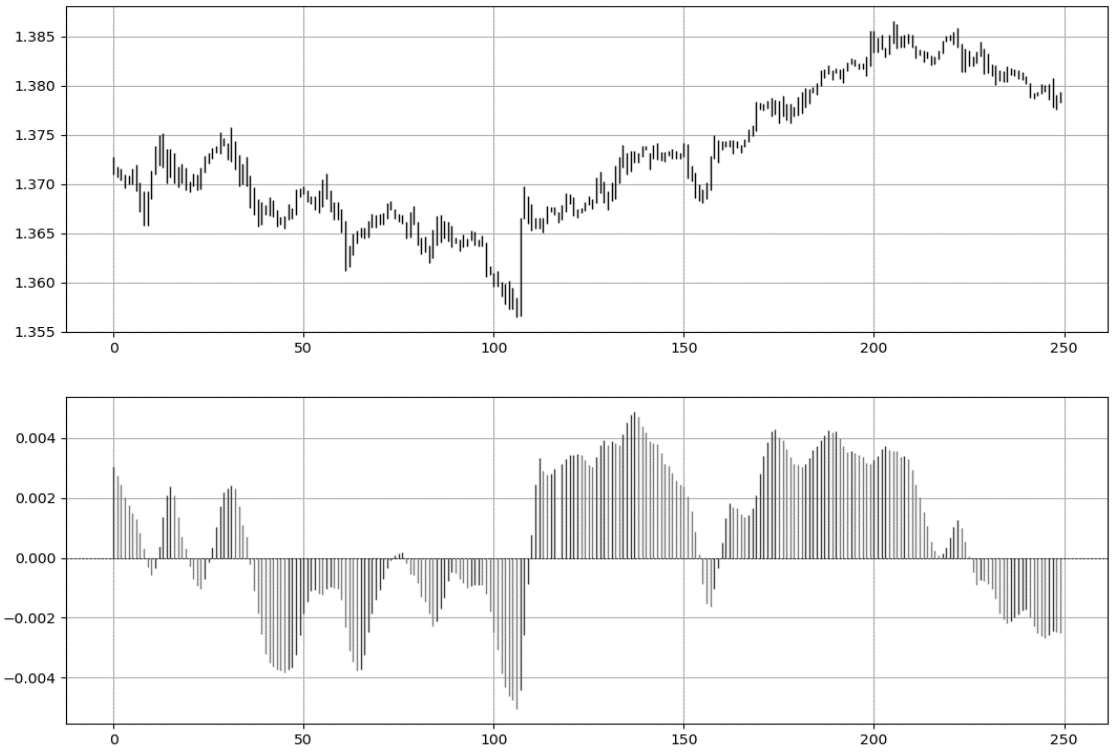


To generate the above plot with the colors, we can use the vertical lines trick:

```
def indicator_plot_double_awesome(Data, first, second, name = '', name_ind = '', window = 250):
fig, ax = plt.subplots(2, figsize = (10, 5))  Chosen = Data[-window:, ]
  for i in range(len(Chosen)):
   ax[0].vlines(x = i, ymin = Chosen[i, 2], ymax = Chosen[i, 1], color = 'black', linewidth = 1)
  ax[0].grid()
  for i in range(len(Chosen)):
   if Chosen[i, 5] > Chosen[i - 1, 5]:
    ax[1].vlines(x = i, ymin = 0, ymax = Chosen[i, 5], color = 'green', linewidth = 1)
   if Chosen[i, 5] < Chosen[i - 1, 5]:
    ax[1].vlines(x = i, ymin = Chosen[i, 5], ymax = 0, color = 'red', linewidth = 1)
  ax[1].grid()
  ax[1].axhline(y = 0, color = 'black', linewidth = 0.5)
```

 # Using the function to generate the plot

```
indicator_plot_double_awesome(my_data, 3, where_your_awesome_oscillator_lies, name = '',
name_ind = 'Awesome Oscillator', window = 250)
```



The full Python code to get signals from the flip can be as follows:

```
def signal(Data, awesome_column, buy, sell):
    for i in range(len(Data)):
        if Data[i, awesome_column] > 0 and Data[i - 1, awesome_column] < 0:
            Data[i, buy] = 1
        if Data[i, awesome_column] < 0 and Data[i - 1, awesome_column] > 0:
            Data[i, sell] = -1
```

HEIKIN-ASHI STRATEGY

"How is that spelled?"

The Heikin-Ashi (Also called Heiken-Ashi) candlesticks seek to clean out the picture and show a clearer trend by smoothing out the OHLC data. Here is how to calculate the Heikin-Ashi candlesticks:

$$Close\ Price = \frac{Open + High + Low + Close}{4}$$

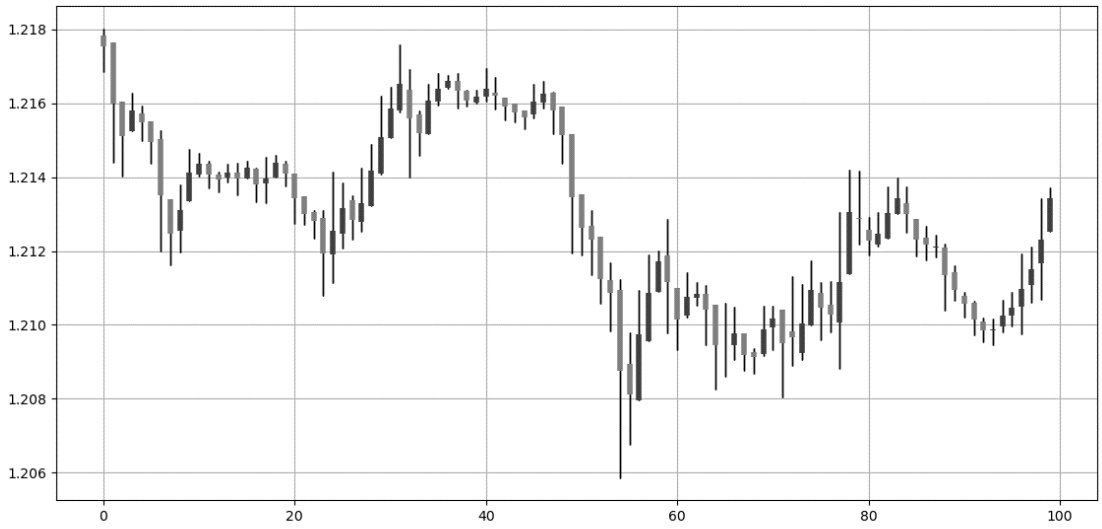
$$Open\ Price = \frac{Open\ of\ Previous\ Bar + Close\ of\ Previous\ Bar}{2}$$

And to calculate the high and low price, we take the maximum and minimum prices of the following:

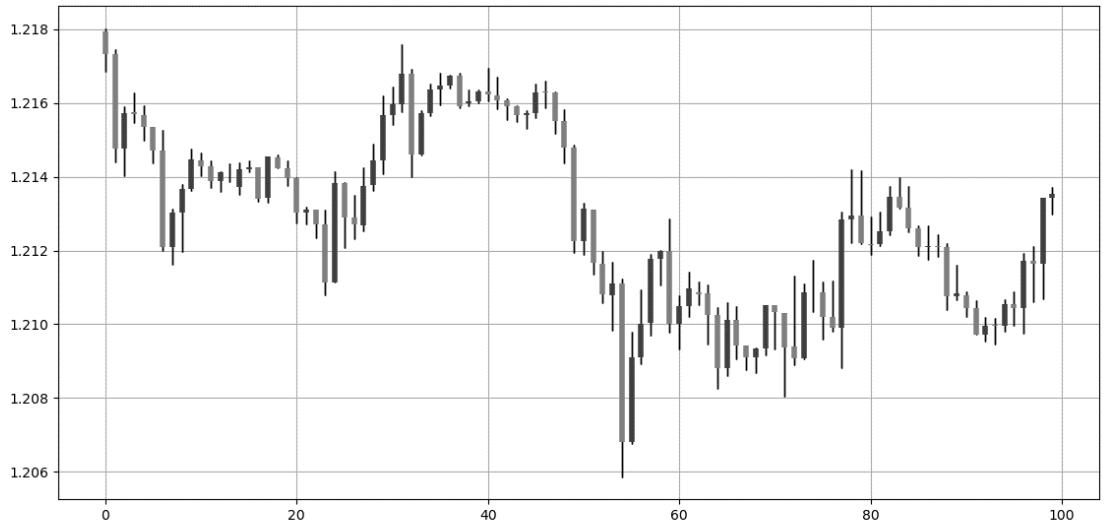
$$High\ Price = \max(Open, High, Close)$$

$$Low\ Price = \min(Open, High, Close)$$

The above formulas will smooth out the candles to give us a more defined and clearer trend.



The previous plot is a Heikin-Ashi chart on the EURUSD. Compared to the next regular candlestick chart, the Heikin-Ashi one does seem smoother. In Appendix I, we see how to create the candlestick chart.



To code a function in Python that adds 4 new columns containing OHLC Heikin-Ashi data, we can use the next code block:

```
def Heikin_ashi(Data, opening, high, low, close, where):
    # Heikin-Ashi Open
    try:
        for i in range(len(Data)):
            Data[i, where] = (Data[i - 1, opening] + Data[i - 1, close]) / 2
    except:
        pass
    # Heikin-Ashi Close
    for i in range(len(Data)):
        Data[i, where + 3] = (Data[i, opening] + Data[i, high] + Data[i, low] + Data[i, close]) / 4
```

```
    # Heikin-Ashi High
    for i in range(len(Data)):
        Data[i, where + 1] = max(Data[i, where], Data[i, where + 3], Data[i, high])
    # Heikin-Ashi Low
    for i in range(len(Data)):
        Data[i, where + 2] = min(Data[i, where], Data[i, where + 3], Data[i, low])
    return Data
# To be used on an OHLC array with a few columns to spare
my_ohlc_array = Heikin_ashi(my_ohlc_array, 0, 1, 2, 3, 4)
# The numbers signify in order: Open, High, Low, Close, then the column indexed at 4 is where the first new Heikin OHLC data will be populated (Meaning that the columns 4, 5, 6, and 7 will have a brand new OHLC data)
```

The basic strategy using the Heikin-Ashi chart is to simply buy and sell whenever the color of the candles changes. This should give us the following conditions:

- Go long (Buy) whenever the Heikin-Ashi candle changes from red to green. Hold the position until getting a contrarian signal. We have to be careful to initiate the position using the true market price and not the one displayed on the Heikin-Ashi chart.
- Go long (Buy) whenever the Heikin-Ashi candle changes from red to green. Hold the position until getting a contrarian signal. We have to be careful to initiate the position using the true market price and not the one displayed on the Heikin-Ashi chart.

We can code the signal function of the strategy as follows:

```
def signal(Data, Heikin_close, Heikin_open, buy, sell):
   for i in range(len(Data)):
      if Data[i, Heikin_close] > Data[i, Heikin_open] and Data[i - 1, Heikin_close] < Data[i - 1, Heikin_open]:
         Data[i, buy] = 1
      if Data[i, Heikin_close] < Data[i, Heikin_open] and Data[i - 1, Heikin_close] > Data[i - 1, Heikin_open]:
         Data[i, sell] = -1
```

 # The variable Heikin_close refers to the Heikin closing candle

 # The variable Heikin_open refers to the Heikin opening candle

PART 3

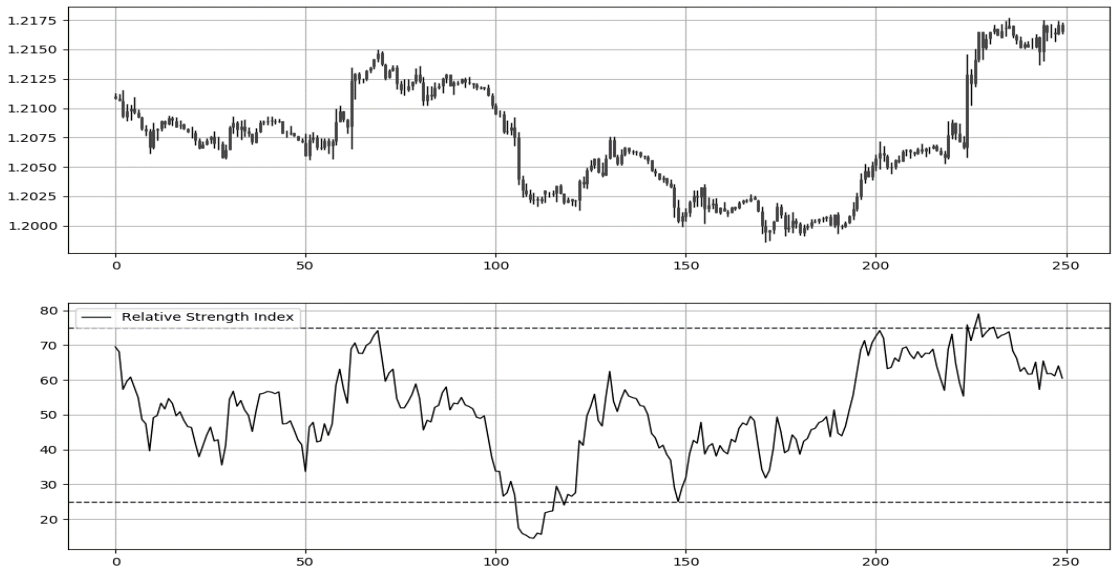
CONTRARIAN STRATEGIES

Market timing and mean-reversion are complex techniques that require more guts as you are going against the whole markets. The idea is that you have detected an anomaly that should make the market correct or even reverse. This can be done statistically, psychologically, or mathematically. As hard as contrarian strategies are, they remain prevalent in the arsenal of both retail and institutional traders. This part will deal with contrarian indicators and strategies and will try to present them in a structured manner. However, most complex contrarian strategies will be found in part 5. I recommend that you finish this part before moving on later to part 5 where more strategies and conditions are presented. One obvious weakness lies with contrarian strategies. They never work in trending markets and must be dynamically adjusted to account for volatility and market movements; therefore, it is never easy to maintain a contrarian strategy.

RELATIVE STRENGTH INDEX EXTREMES

"Enough of this strategy."

The RSI is without a doubt the most famous momentum indicator out there, and this is to be expected as it has many strengths especially in ranging markets. It is also bounded between 0 and 100 which makes it easier to interpret. Also, the fact that it is famous, contributes to its potential. This is because the more traders and portfolio managers look at the RSI, the more people will react based on its signals and this in turn can push market prices. Of course, we cannot prove this idea, but it is intuitive as one of the properties of Technical Analysis is that it is self-fulfilling. The RSI is calculated using a rather simple way. We first start by taking price differences of one period. This means that we have to subtract every closing price from the one before it. Then, we will calculate the smoothed average of the positive differences and divide it by the smoothed average of the negative differences. The last calculation gives us the RS which is then transformed into a measure between 0 and 100.



To calculate the RSI, we need an OHLC array (not a data frame). This means that we will be looking at an array of 4 columns. We have already discussed the smoothed moving average in the previous part, we will once again use it in the RSI's formula.

```
def rsi(Data, lookback, close, where, width = 1, genre = 'Smoothed'):
    # Adding a few columns
    Data = adder(Data, 5)
    # Calculating Differences
    for i in range(len(Data)):
        Data[i, where] = Data[i, close] - Data[i - width, close]
    # Calculating the Up and Down absolute values
    for i in range(len(Data)):
        if Data[i, where] > 0:
            Data[i, where + 1] = Data[i, where]
        elif Data[i, where] < 0:
            Data[i, where + 2] = abs(Data[i, where])
    # Calculating the Smoothed Moving Average on Up and Down absolute values
    if genre == 'Smoothed':
        lookback = (lookback * 2) - 1 # From exponential to smoothed
        Data = ema(Data, 2, lookback, where + 1, where + 3)
        Data = ema(Data, 2, lookback, where + 2, where + 4)
    if genre == 'Simple':
        Data = ma(Data, lookback, where + 1, where + 3)
        Data = ma(Data, lookback, where + 2, where + 4)
    # Calculating the Relative Strength
    Data[:, where + 5] = Data[:, where + 3] / Data[:, where + 4]
    # Calculate the Relative Strength Index
    Data[:, where + 6] = (100 - (100 / (1 + Data[:, where + 5])))
    # Cleaning
    Data = deleter(Data, where, 6)
    Data = jump(Data, lookback)
    return Data
```

The conditions for the strategy are as follows:

- A bullish signal is triggered whenever the indicator reaches or breaks the lower barrier (oversold zone).
- A bearish signal is triggered whenever the indicator reaches or surpasses the upper barrier (overbought zone).

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.

```
def signal(Data, indicator_column, buy, sell):
  Data = adder(Data, 2)
  for i in range(len(Data)):
    if Data[i, indicator_column] < lower_barrier and Data[i - 1, buy] == 0 and \
      Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0:
      Data[i, buy] = 1
    elif Data[i, indicator_column] > upper_barrier and Data[i - 1, sell] == 0 and \
      Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0:
      Data[i, sell] = -1
  return Data
```

Typically, we use either a 5-period, 14-period, or 21-period RSI and monitor overbought levels upwards of 70 and oversold levels downwards of 30.

```
lookback = 14
lower_barrier = 30
upper_barrier = 70
```

The next plot shows an example of a signal chart on the EURUSD using the conditions above.

Upward pointing arrows refer to bullish signals while downward pointing arrows refer to bearish signals. The RSI even though is a known technical indicator, strategies relying solely on it will likely underperform. Nevertheless, filters can help improve the strategy. One such filter can be the state of the market at the time of the signal. If we look at the chart above, we can see that bullish signals do not work in a bearish trend which is the prevalent one visually. Notice however, the quality of the bearish signals where the RSI seems to be capturing the tops quite well. Can a moving average filter be applied to remove the bad signals?

RELATIVE STRENGTH INDEX DIVERGENCE

"Detecting trend exhaustion is exhausting."

Divergences are a like the joker card that we like to see when searching for a good trade. We tend to see divergences a little before a strong correction or even a reversal. However, we can always optimize the way to detect a divergence so that we improve our chances of getting a profitable trade. Naturally, when prices are rising and making new tops while a price-based indicator is making lower tops, a momentum weakening is occurring and a possibility to change the bias from long to short can present itself. That is what we call a normal divergence. We know that:

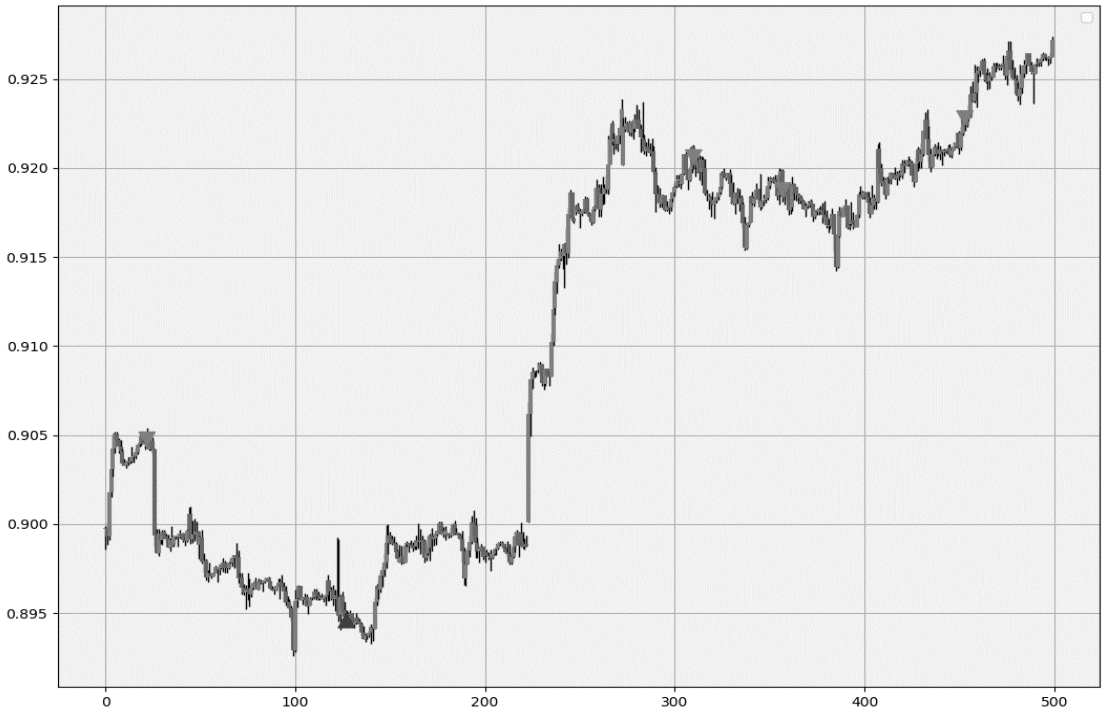
- When prices are making higher highs while the indicator is making lower highs, it is called a bearish divergence, and the market might stall.
- When prices are making lower lows while the indicator is making higher lows, it is called a bullish divergence, and the market might show some upside potential.

```
def divergence(Data, indicator, lower_barrier, upper_barrier, width, buy, sell):
    Data = adder(Data, 2)
    for i in range(len(Data)):
        try:
            if Data[i, indicator] < lower_barrier:
                for a in range(i + 1, i + width):
                    # First trough
                    if Data[a, indicator] > lower_barrier:
                        for r in range(a + 1, a + width):
                            if Data[r, indicator] < lower_barrier and \
                            Data[r, indicator] > Data[i, indicator] and Data[r, 3] < Data[i, 3]:
                                for s in range(r + 1, r + width):
                                    # Second trough
                                    if Data[s, indicator] > lower_barrier:
                                        Data[s, buy] = 1
                                        break
                                    else:
                                        break
                            else:
                                break
                        else:
                            break
                    else:
                        break
        except IndexError:
            pass
    for i in range(len(Data)):
        try:
            if Data[i, indicator] > upper_barrier:
                for a in range(i + 1, i + width):
```

```
            # First trough
            if Data[a, indicator] < upper_barrier:
              for r in range(a + 1, a + width):
                if Data[r, indicator] > upper_barrier and \
                Data[r, indicator] < Data[i, indicator] and Data[r, 3] > Data[i, 3]:
                  for s in range(r + 1, r + width):
                    # Second trough
                    if Data[s, indicator] < upper_barrier:
                      Data[s, sell] = -1
                      break
                    else:
                      break
                else:
                  break
              else:
                break
            else:
              break
      except IndexError:
        pass
    return Data
```

The intuition of the divergence can be captured in the previous function. However, it is just an example and other types of functions can also exist.

The previous plot shows an example of a signal chart on the EURUSD using the conditions of the divergence. The function can be tailored by tweaking the width variable which is simply the distance between the two tops or bottoms. In the above plot, only one signal has been detected, therefore, the parameters must be tweaked so that they capture more divergences like the above plot.



RELATIVE STRENGTH INDEX AVERAGE CROSS

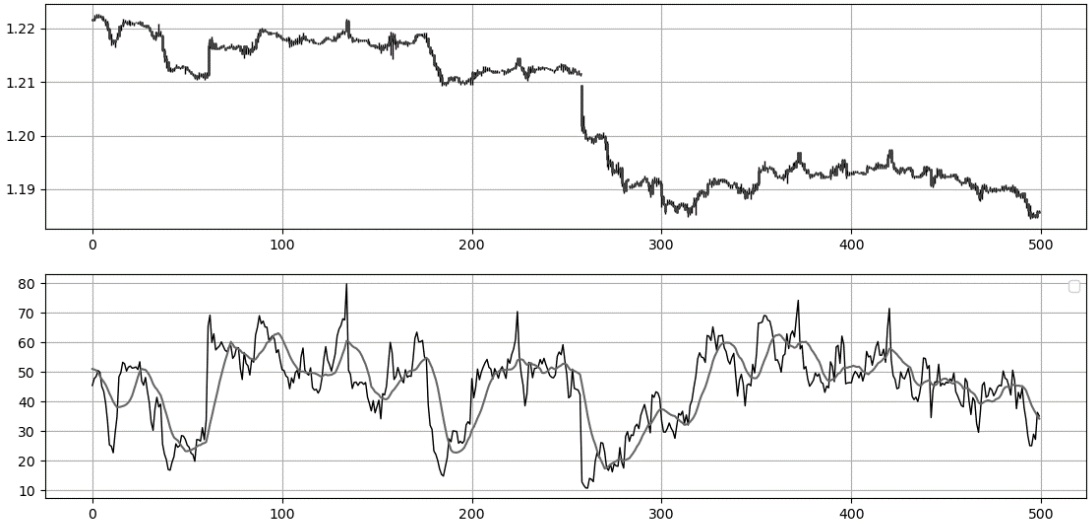
"We are told that this strategy works, but it does not."

Generally, technical indicators are accompanied by moving averages calculated on their values. This is particularly useful for unbounded indicators but also for bounded indicators such as the RSI. In this section, we will calculate a simple moving average on the values of the RSI and derive signals from their cross.

The conditions for the strategy are as follows:

- A bullish signal is triggered whenever the indicator surpasses its moving average.
- A bearish signal is triggered whenever the indicator breaks its moving average.

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.



The above plot shows the 14-period RSI with its 13-period simple moving average. The next plot shows the signals generated from the conditions above.


```
def signal(Data, rsi_col, ma_col, buy, sell):
    Data = adder(Data, 10)
    Data = rounding(Data, 5)
    for i in range(len(Data)):
        if Data[i, rsi_col] > Data[i, ma_col] and Data[i - 1, rsi_col] < Data[i - 1, ma_col]:
            Data[i, buy] = 1
        elif Data[i, rsi_col] < Data[i, ma_col] and Data[i - 1, rsi_col] > Data[i - 1, ma_col]:
            Data[i, sell] = -1
    return Data
```

The strategy is based on a famous premise in technical analysis that when an indicator crosses its moving average, a confirmation signal has occurred. This fails to account for the fact that we are applying a lagging indicator (the moving average) on another lagging indicator (the RSI). In real life, this strategy is unlikely to give a satisfactory result without huge optimization. Why is it presented then? The aim of the book is two folds:

- Inducing brainstorming and thinking outside the box when it comes to predictions.
- Demystifying and filtering out the bad strategies that are found everywhere and quoted as profitable strategies.

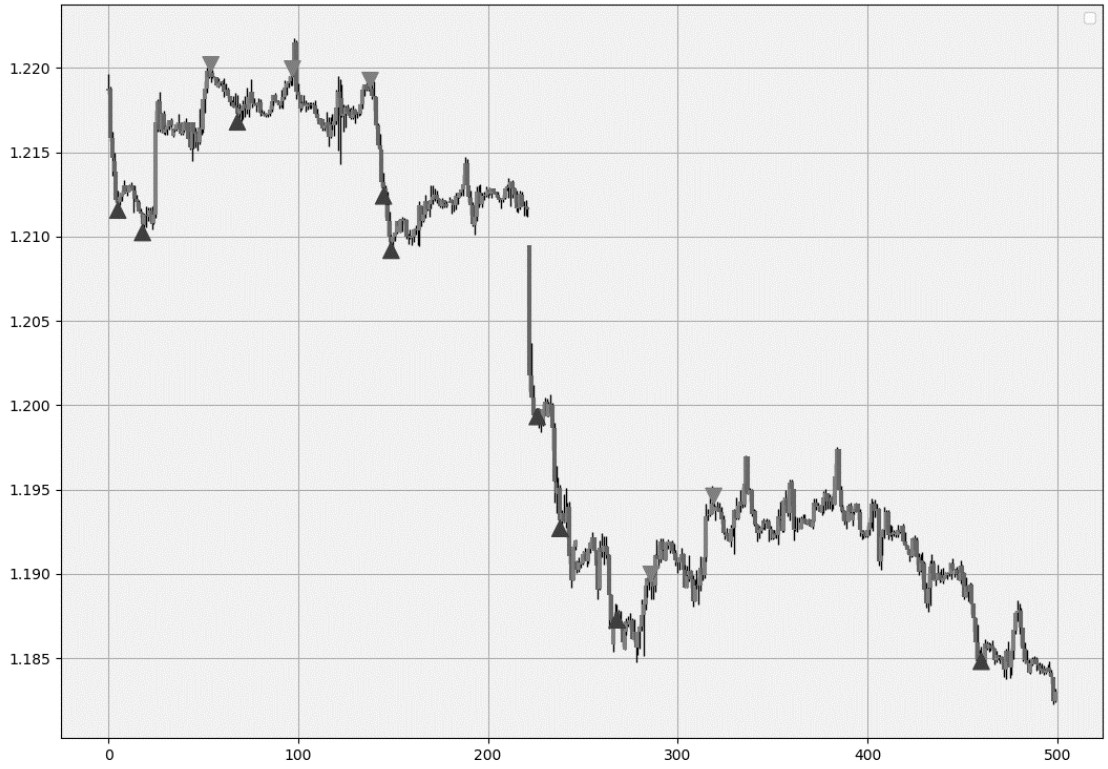
RELATIVE STRENGTH INDEX DURATION

"Time spent oversold or overbought is a valuable information."

The conditions for the strategy are as follows:

- A bullish signal is triggered whenever the indicator spends more time than expected in the oversold zone.
- A bearish signal is triggered whenever the indicator spends more time than expected in the overbought zone.

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.

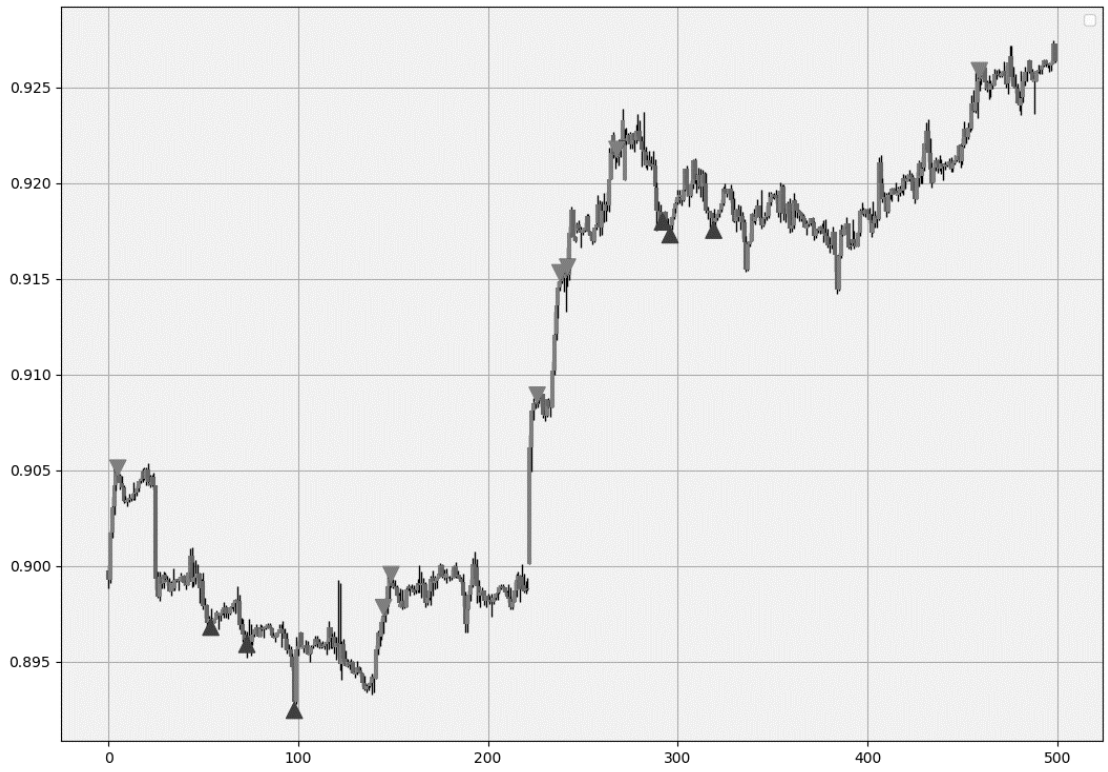


The above chart shows signals generated when the 5-period RSI spends 5 or more periods at the extremes. This is another way of saying that historically, the market spends 5 timesteps on the extremes before showing a reaction.

```
def extreme_duration(Data, indicator, upper_barrier, lower_barrier, where_upward_extreme,
where_downward_extreme, net_col):
  # Adding columns
  Data = adder(Data, 20)
  # Time Spent Overbought
  for i in range(len(Data)):
   if Data[i, indicator] > upper_barrier:
       Data[i, where_upward_extreme] = Data[i - 1, where_upward_extreme] + 1
   else:
       a = 0
       Data[i, where_upward_extreme] = a
  # Time Spent Oversold
  for i in range(len(Data)):
   if Data[i, indicator] < lower_barrier:
       Data[i, where_downward_extreme] = Data[i - 1, where_downward_extreme] + 1
   else:
       a = 0
       Data[i, where_downward_extreme] = a
  Data[:, net_col] = Data[:, where_upward_extreme] - Data[:, where_downward_extreme]
  Data = deleter(Data, 6, 2)
  return Data
```

The extreme duration function above calculates the time spent oversold or overbought on any type of indicator. Let us now use the Extreme Duration to generate contrarian signals. The next function is what defines a signal where we write the initial condition of being above or below a certain extreme for a chosen time period such as 5 periods, then it filters and eliminates duplicates. A duplicate example is when the RSI spends 8 time periods below the oversold level, giving us 4 signals at every time stamp.

```
def signal(Data, extreme, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, extreme] <= -5 and Data[i - 1, buy] == 0 and Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0:
            Data[i, buy] = 1
        elif Data[i, extreme] >= 5 and Data[i - 1, sell] == 0 and Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0:
            Data[i, sell] = -1
    return Data
```



The plot above shows the signals generated on the USDCHF hourly values. We can see that contrarian strategies are not fit for trending markets. This is because they are price-derived and do not really peek into the future. All they do is signal an anomaly in a neutral market.

STOCHASTIC OSCILLATOR EXTREMES

"The most basic contrarian trading strategy."

The Stochastic Oscillator seeks to find oversold and overbought zones by incorporating the highs and lows using the normalization formula as shown below:

$$\%K = \left(\frac{Close - Low}{High - Low}\right) x\ 100$$

An overbought level is an area where the market is perceived to be extremely bullish and is bound to consolidate. An oversold level is an area where market is perceived to be extremely bearish and is bound to bounce. Hence, the Stochastic Oscillator is a contrarian indicator that seeks to signal reactions of extreme movements. The conditions for the strategy are as follows:

- A bullish signal is triggered whenever the indicator reaches or breaks the lower barrier (oversold zone).
- A bearish signal is triggered whenever the indicator reaches or surpasses the upper barrier (overbought zone).

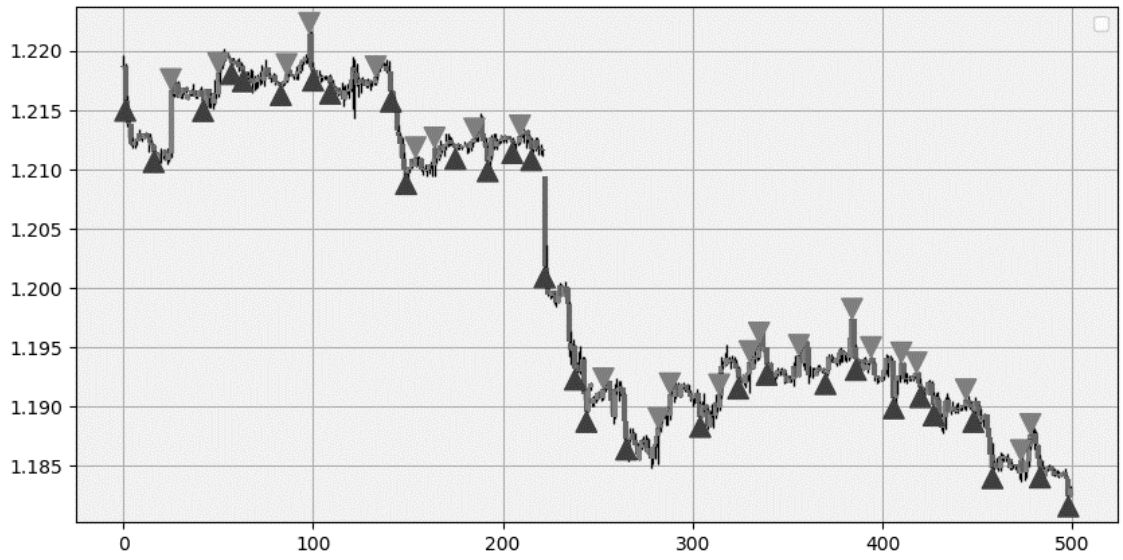
To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.

```
def signal(Data, indicator_column, buy, sell):
    Data = adder(Data, 2)
    for i in range(len(Data)):
        if Data[i, indicator_column] < lower_barrier and Data[i - 1, buy] == 0 and \
            Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0:
            Data[i, buy] = 1
        elif Data[i, indicator_column] > upper_barrier and Data[i - 1, sell] == 0 and \
            Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0:
            Data[i, sell] = -1
    return Data
```

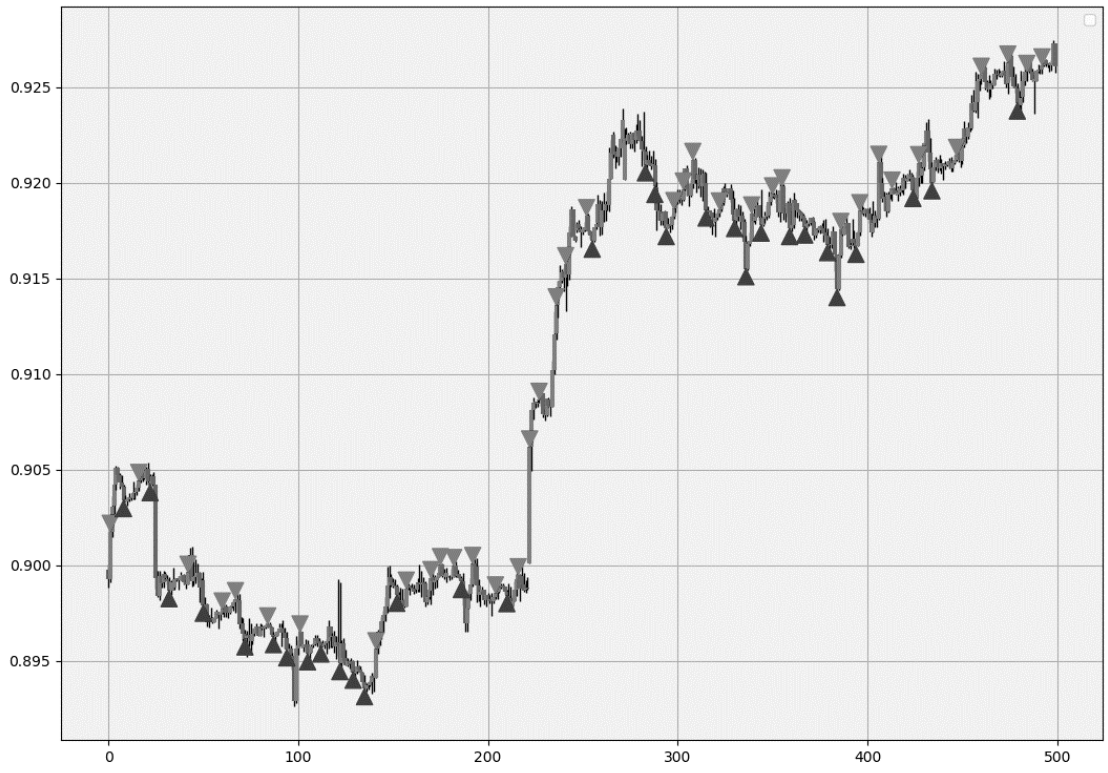
Typically, we use either a 5-period, 14-period, or 21-period Stochastic Oscillator and monitor overbought levels upwards of 80-90 and oversold levels downwards of 20-10.

```
lookback = 5
lower_barrier = 10
upper_barrier = 90
```

The next plot shows an example of a signal chart on the EURUSD using the conditions above.



Upward pointing arrows refer to bullish signals while downward pointing arrows refer to bearish signals. The Stochastic Oscillator even though is a known technical indicator, strategies relying solely on it will likely underperform. Nevertheless, filters can help improve the strategy. One such filter can be the state of the market at the time of the signal. If we look at the chart above, we can see that bullish signals do not work in a bearish trend which is the prevalent one visually. Notice however, the quality of the bearish signals where the Stochastic seems to be capturing the tops quite well.



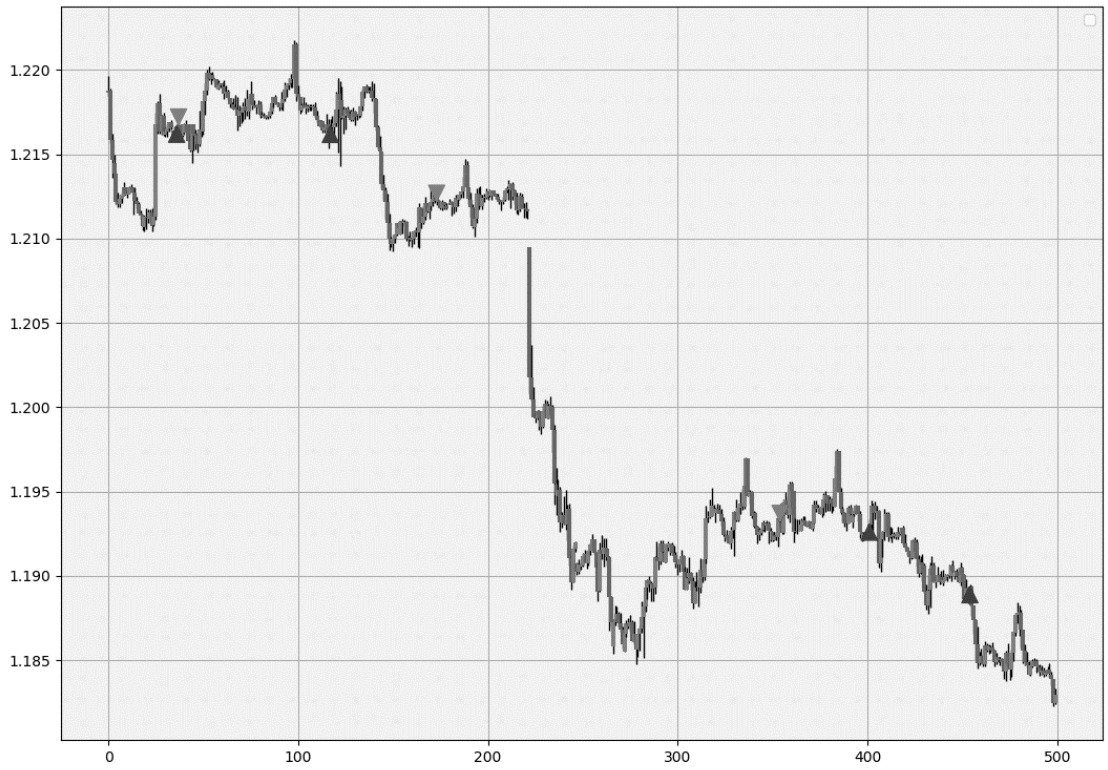
The above plot shows the signals on the USDCHF. In an uptrend, the contrarian bearish signals by the Stochastic Oscillator are not of high quality. This is why filters are important. Structured strategies will be seen in the last part of the book where we try to filter the signals to increase the probability of a winning trade,

STOCHASTIC OSCILLATOR DIVERGENCE

"A slightly worse version of the RSI's divergence strategy."

As previously seen, divergences are an integral technique in detecting trend exhaustions. Using the same function, we can apply it on the readings of the Stochastic Oscillator. This can be done using the following syntax.

```
width = 20
my_data = stochastic(my_data, lookback, 3, 5, genre = High-Low)
my_data = adder(my_data, 15)
my_data = divergence(my_data, stochastic_col, lower_barrier, upper_barrier, width, 7, 8)
```



The previous plot shows an example of a signal chart on the EURUSD using the conditions of the divergence. The function can be tailored by tweaking the width variable which is simply the distance between the two tops or bottoms.

The divergence technique works less well on the Stochastic Oscillator than on the RSI. The simplest reason is that the former is very volatile and can go from oversold to

overbought extremely quickly, making its divergences short-lived and unreliable. Some traders increase the lookback period on the Stochastic so that it becomes more stable, and its signals can start providing more intuitive sense. Generally, the preferred technique to use on the Stochastic Oscillator is the simple oversold/overbought method. However, the duration method may also provide some value as it is uncorrelated to the other techniques. This adds a diversification factor to the analysis. While we are at it, let us define and discuss the concept of correlation.

Correlation is the degree of linear relationship between two or more variables. It is bounded between -1 and 1 with one being a perfectly positive correlation, -1 being a perfectly negative correlation, and 0 as an indication of no linear relationship between the variables (they relatively go in random directions).

The measure is not perfect and can be biased by outliers and non-linear relationships, it does however provide quick glances to statistical properties. We can code the correlation function between two variables in Python using the below. Note that it must be an array and not a data frame:

```
def rolling_correlation(Data, first_data, second_data, lookback, where):
  for i in range(len(Data)):
    try:
      Data[i, where] = pearsonr(Data[i - lookback + 1:i + 1, first_data], Data[i - lookback + 1:i + 1,
second_data])[0]
    except ValueError:
      pass
  Data = jump(Data, lookback)
  return Data
```

We might need to import a library that has the function of the Pearson's correlation as used above.

```
from scipy.stats  import pearsonr
```

STOCHASTIC OSCILLATOR AVERAGE CROSS

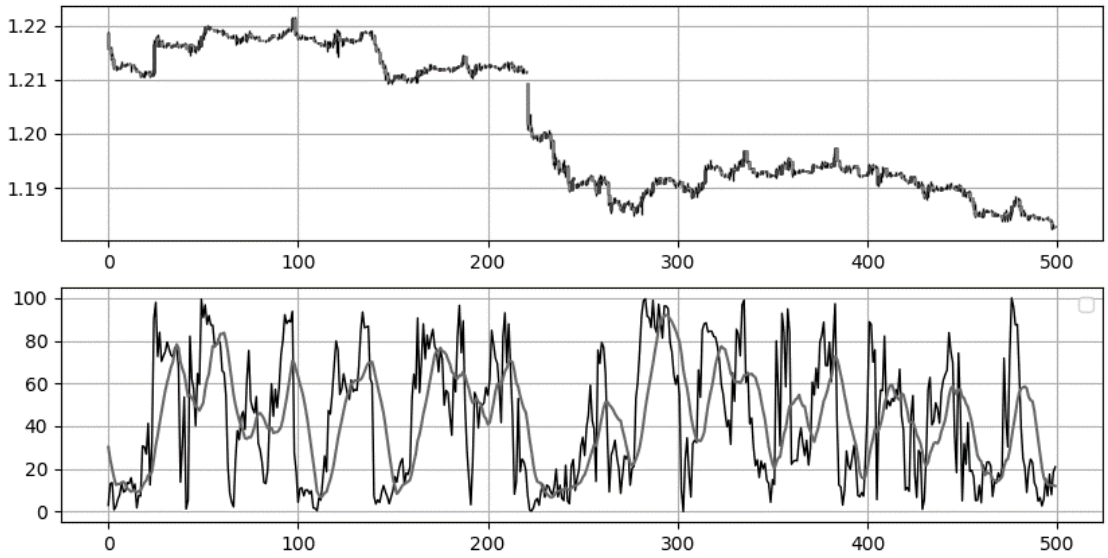
"Might work better in reverse."

Generally, technical indicators are accompanied by moving averages calculated on their values. This is particularly useful for unbounded indicators but also for bounded indicators such as the Stochastic Oscillator. In this section, we will calculate a simple moving average on the values of the Stochastic and derive signals from their cross.

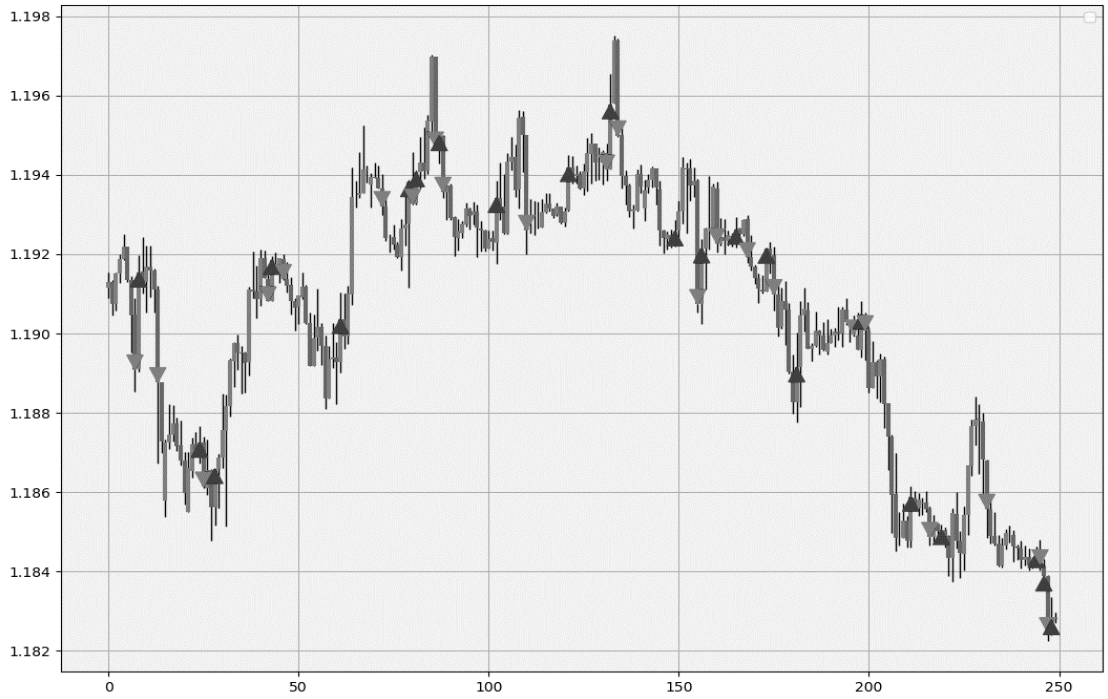
The conditions for the strategy are as follows:

- A bullish signal is triggered whenever the indicator surpasses its moving average.
- A bearish signal is triggered whenever the indicator breaks its moving average.

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.



The above plot shows the 14-period Stochastic with its 13-period simple moving average. The next plot shows the signals generated from the conditions above.




```
def signal(Data, stochastic_col, ma_col, buy, sell):
    Data = adder(Data, 10)
    Data = rounding(Data, 5)
    for i in range(len(Data)):
        if Data[i, stochastic_col] > Data[i, ma_col] and Data[i - 1, stochastic_col] < Data[i - 1, ma_col]:
            Data[i, buy] = 1
        elif Data[i, stochastic_col] < Data[i, ma_col] and Data[i - 1, stochastic_col] > Data[i - 1, ma_col]:
            Data[i, sell] = -1
    return Data
```

Make sure to use the concepts learnt in Part 1 to back-test the strategies presented in this book. If you have any trouble with copying the code, you can refer to the GitHub link [7]where you should find what you need. Note that it will be continuously updated therefore, even new strategies will be published there. If you feel that you have a strategy worth back-testing, do not hesitate to contact me.

[7] https://github.com/sofienkaabar/the-book-of-more-back-tests

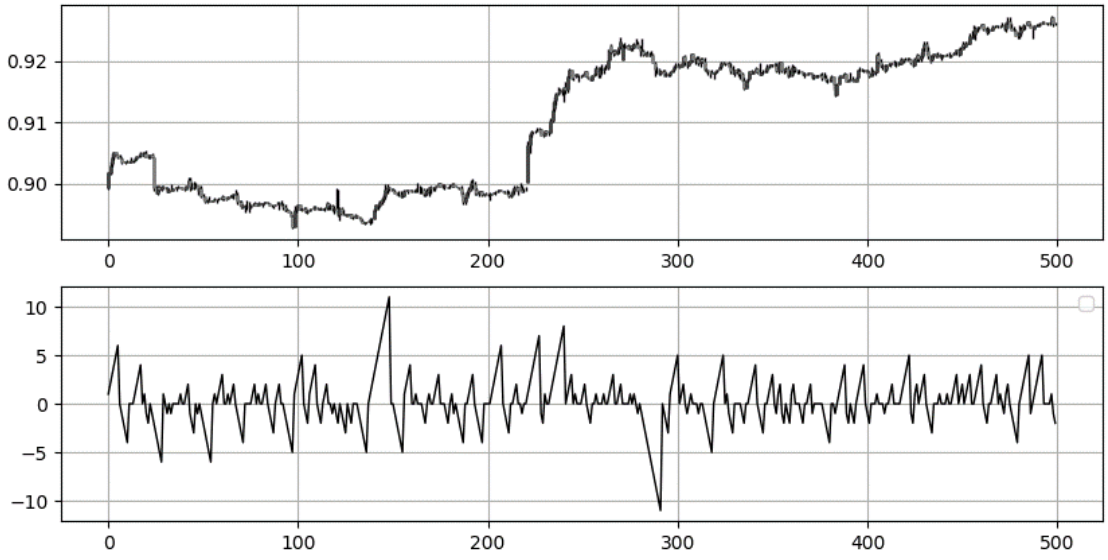
STOCHASTIC OSCILLATOR DURATION

"Again, time spend oversold or overbought is important."

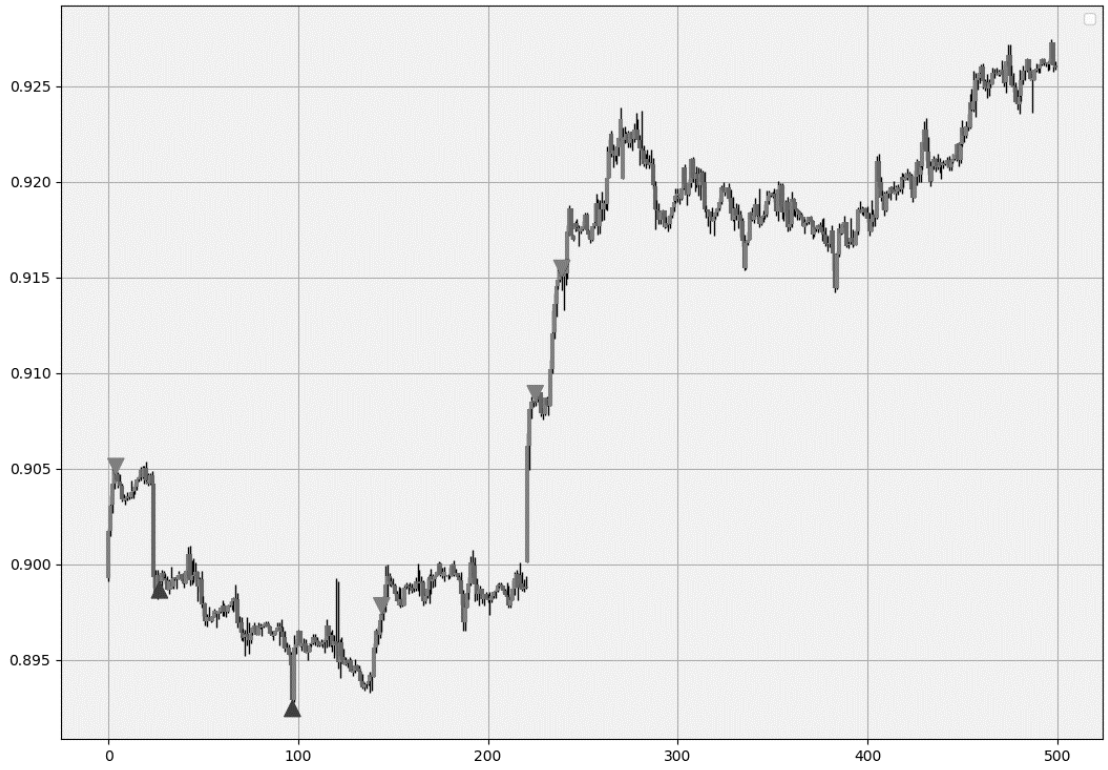
The conditions for the strategy are as follows:

- A bullish signal is triggered whenever the indicator spends more time than expected in the oversold zone.
- A bearish signal is triggered whenever the indicator spends more time than expected in the overbought zone.

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.



The above chart shows the hourly USDCHF values with the extreme technique seen previously, applied on the 5-period Stochastic Oscillator. The next chart shows signals generated when the 5-period Stochastic Oscillator spends 5 or more periods at the extremes. This is another way of saying that historically, the market spends 5 timesteps on the extremes before showing a reaction. Let us choose the 70/30 barriers to keep consistency.



The next function is what defines a signal where we write the initial condition of being above or below a certain extreme for a chosen time period such as 5 periods, then it filters and eliminates duplicates. A duplicate example is when the Stochastic Oscillator spends 7 time periods below the oversold level, giving us 3 signals at every time stamp.

```
def signal(Data, extreme, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, extreme] <= -5 and Data[i - 1, buy] == 0 and Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0:
            Data[i, buy] = 1
        elif Data[i, extreme] >= 5 and Data[i - 1, sell] == 0 and Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0:
            Data[i, sell] = -1
    return Data
```

REAL RANGE EXTREMES

"Can such a simple formula work?"

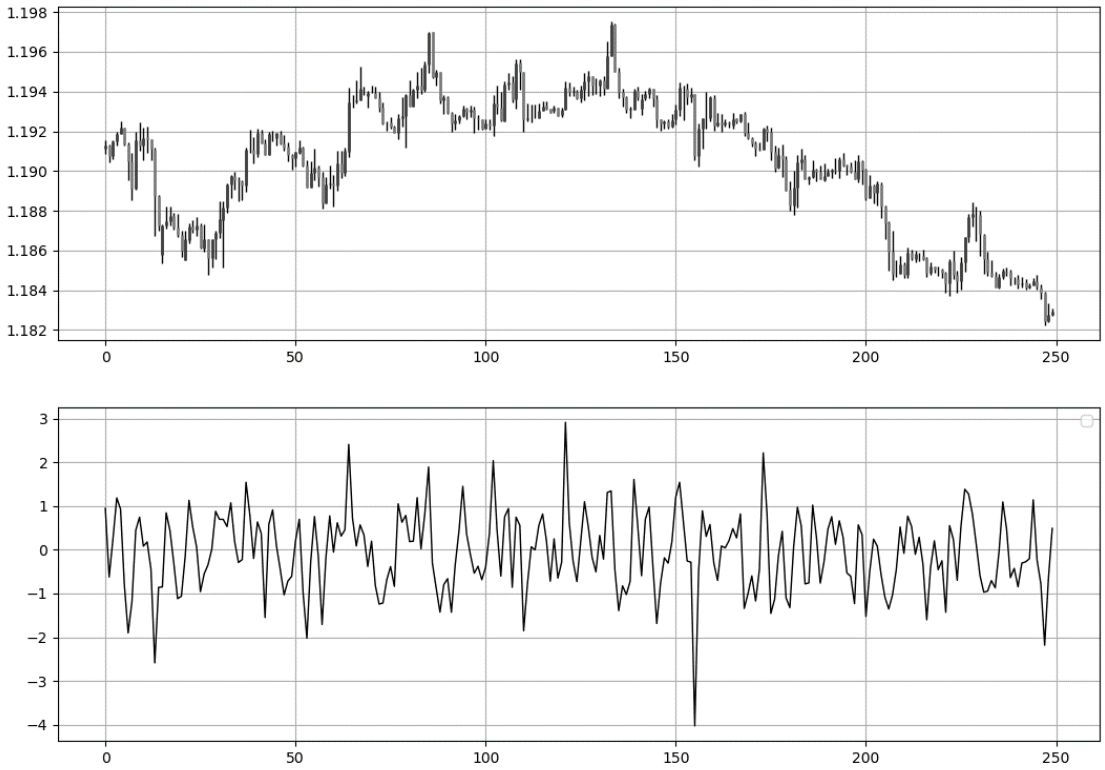
This formula can also be found when calculating the Relative Vigor Index and thus I take no credit in it. The Real Range indicator is calculated following this formula:

$$RRI = \frac{Close - Open}{High - Low}$$

The function in Python is very simple. We have to use it on an OHLC (Open, High, Low, Close) array, then the function will select a column using the where variable and then apply the formula. Remember that the fourth column Data[i, 3] refers to the closing price, the first column Data[i, 0] refers to the opening price, the second column Data[i, 1] refers to the high, and the third column Data[i, 2] refers to the low. This is because Python's indexing starts at zero.

```
def rri(Data, lookback, where):
    # Adding a column
    Data = adder(Data, 1)
    for i in range(len(Data)):
        Data[i, where] = (Data[i, 3] - Data[i - lookback, 0]) / (Data[i - lookback, 1] - Data[i - lookback, 2])
        if Data[i - lookback, 1] == Data[i - lookback, 2]:
            Data[i, where] = 0
    return Data
```

The above function creates a column populated by the values of the Real Range. When charted, it should look noisy like the next graph.



The RRI is unbounded and therefore, any extremes strategy is highly subjective. We will see below barriers of 2.00 and -2.00.

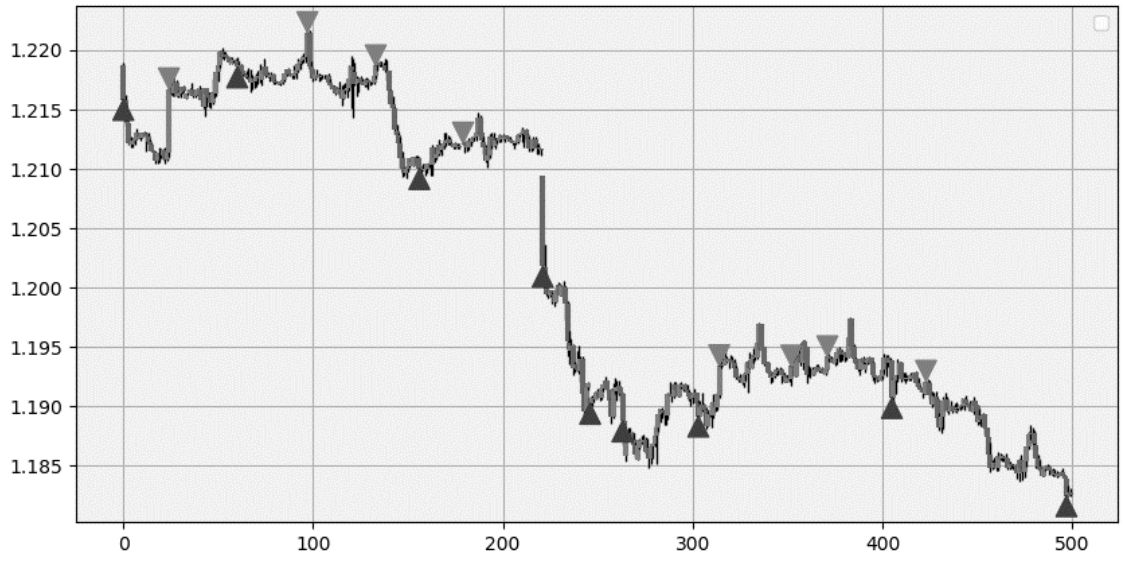
An overbought level is an area where the market is perceived to be extremely bullish and is bound to consolidate. An oversold level is an area where market is perceived to be extremely bearish and is bound to bounce. The conditions for the strategy are as follows:

- A bullish signal is triggered whenever the indicator reaches or breaks the lower barrier (oversold zone). In our case, -2.00.
- A bearish signal is triggered whenever the indicator reaches or surpasses the upper barrier (overbought zone), in our case, 2.00.

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.

```
def signal(Data, indicator_column, buy, sell):
    Data = adder(Data, 2)
    for i in range(len(Data)):
        if Data[i, indicator_column] < lower_barrier and Data[i - 1, buy] == 0 and \
            Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0:
            Data[i, buy] = 1
        elif Data[i, indicator_column] > upper_barrier and Data[i - 1, sell] == 0 and \
            Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0:
            Data[i, sell] = -1
    return Data
```

The next plot shows an example of a signal chart on the EURUSD using the conditions above.




```
upper_barrier    = 2
lower_barrier    = -2
```

COUNTDOWN INDICATOR EXTREMES

"It's the Final Countdown! – Europe."

The Countdown Indicator is a contrarian indicator that I have developed for mostly discretionary trading. The Countdown Indicator have been created in order to include the totality of the OHLC data inside a framework that calculates when we are around an extreme and when we are bound to reverse. I have made some optimizations and finally arrived at the following ways to construct the indicator. All we need, is OHLC data. The Countdown Indicator should be used in conjunction with other technical indicators to get a better confirmation. The below are the steps used to create it with the function in Python later given so that it can be copied and pasted.

Calculate the Upside Pressure. The calculation is simply a summation of two conditions:

- If the current close is greater than the opening price, then we have 1 point in the Upside Pressure calculation. Next, if the current high is greater than the high from the last period, we have another point. At its maximum, the Upside Pressure can only have 2 points while at its minimum, it can have zero points. If the current close is greater than the open while the current high is lower than the previous high, the Upside Pressure should have 1 point.

$$Upside\ Pressure = \sum \begin{cases} 1\ if\ Close > Open \\ 1\ if\ High > High_{i-1} \end{cases}$$

$$Upside\ Pressure = \{0, 2\}$$

Calculate the Downside Pressure. The calculation is simply a summation of two conditions:

- If the current close is lower than the opening price, then we have 1 point in the Downside Pressure calculation. Next, if the current low is lower than the low from the last period, we have another point. At its maximum, the Downside Pressure can only have 2 points while at its minimum, it can have zero points. If

the current close is lower than the open while the current low is higher than the previous low, the Downside Pressure should have 1 point.

$$Downside\ Pressure = \sum \begin{cases} 1\ if\ Close < Open \\ 1\ if\ Low_{i-1} > Low \end{cases}$$

$$Downside\ Pressure = \{0, 2\}$$

- Calculate the 8-period Cumulative Upside and Downside pressures. This is a simple summation of the last 8 pressures including the current one:

$$Cumulative\ Upside\ Pressure = \sum_{i=1}^{n} Upside\ Pressure_i$$

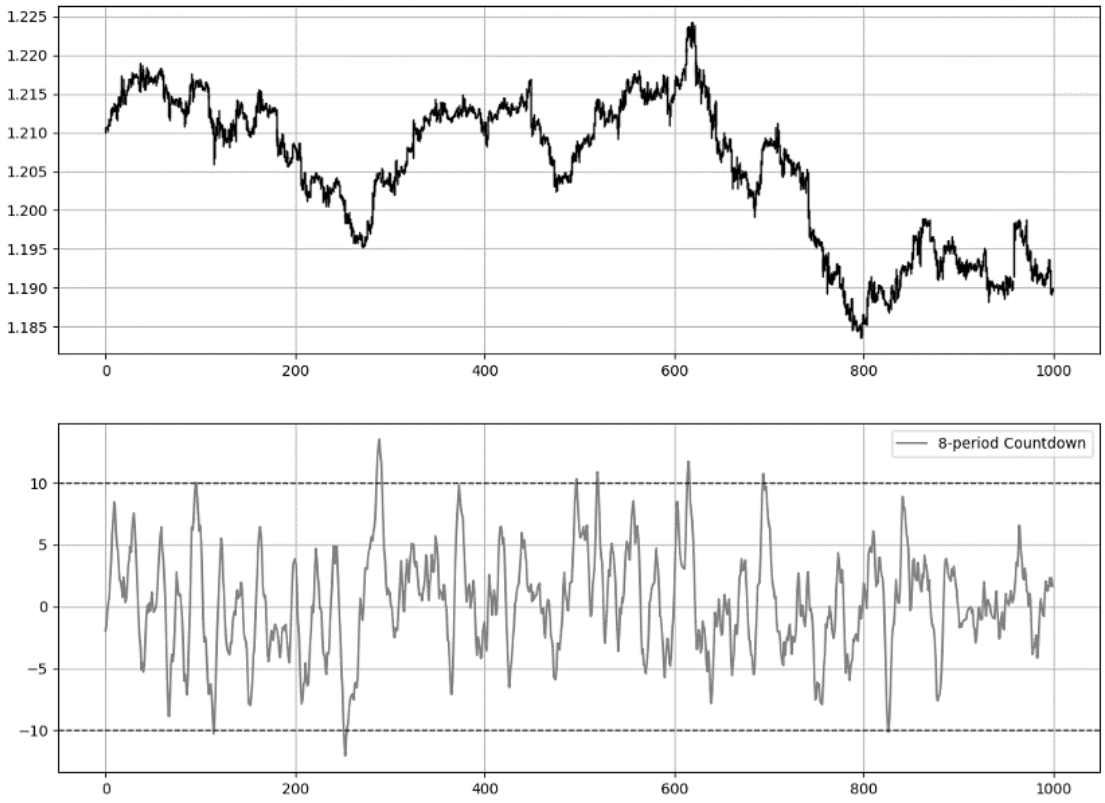
$$Cumulative\ Downside\ Pressure = \sum_{i=1}^{n} Downside\ Pressure_i$$

- Next, the Countdown Indicator (Raw form) is simply the difference between the Cumulative Upside Pressure and the Cumulative Downside Pressure which then we can simply calculate a 3-period Exponential Moving Average on:

$$Countdown\ Indicator\ Raw = CUP - CDP$$

$$Countdown\ Indicator = Exponential\ Moving\ Average_3(CIR)$$

An example of the Countdown Indicator is shown in the next chart versus the EURUSD hourly values.



To code the indicator, we need to have an OHLC array.

```
def countdown_indicator(Data, lookback, ma_lookback, opening, high, low, close,
where):
  # Adding columns
  Data = adder(Data, 20)
  # Calculating Upside Pressure
  for i in range(len(Data)):
    if Data[i, close] > Data[i, opening]:
      Data[i, where] = 1if Data[i, high] > Data[i - 1, high]:
      Data[i, where + 1] = 1
  Data[:, where + 2] = Data[:, where] + Data[:, where + 1]
  Data = deleter(Data, where, 2)
```

```
    # Calculating Downside Pressure
      for i in range(len(Data)):
       if Data[i, close] < Data[i, opening]:
         Data[i, where + 1] = 1if Data[i, low] < Data[i - 1, low]:
            Data[i, where + 2] = 1
      Data[:, where + 3] = Data[:, where + 1] + Data[:, where + 2]
      Data = deleter(Data, where + 1, 2)
      # Calculate Cumulative Upside Pressure
      for i in range(len(Data)):
       Data[i, where + 2] = Data[i - lookback + 1:i + 1, where].sum()
      # Calculate Cumulative Downside Pressure
      for i in range(len(Data)):
       Data[i, where + 3] = Data[i - lookback + 1:i + 1, where + 1].sum()
      # Calculate the Countdown Indicator
      Data[:, where + 4] = Data[:, where + 2] - Data[:, where + 3]
      Data = ema(Data, 2, ma_lookback, where + 4, where + 5)
      Data = deleter(Data, where, 5)
      Data = jump(Data, lookback)
      return Data
    # Using the Function
    my_data = countdown_indicator(my_data, lookback, ma_lookback, 0, 1, 2, 3, 4)
```

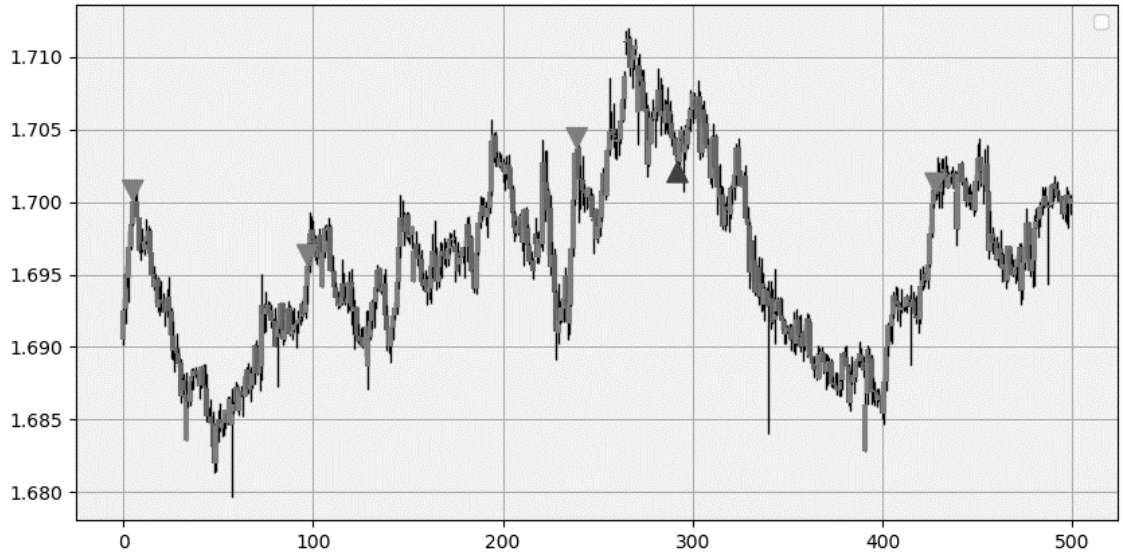
An overbought level is an area where the market is perceived to be extremely bullish and is bound to consolidate. An oversold level is an area where market is perceived to be extremely bearish and is bound to bounce. The conditions for the strategy are as follows:

- Go long (Buy) whenever the Countdown Indicator reaches -10.00 with the four previous readings above -10.00.
- Go short (Sell) whenever the Countdown Indicator reaches 10.00 with the four previous readings below 10.00.

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.

```
def signal(Data, indicator_column, buy, sell):
    Data = adder(Data, 2)
    for i in range(len(Data)):
        if Data[i, indicator_column] < lower_barrier and Data[i - 1, buy] == 0 and \
            Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0:
            Data[i, buy] = 1
        elif Data[i, indicator_column] > upper_barrier and Data[i - 1, sell] == 0 and \
            Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0:
            Data[i, sell] = -1
    return Data
```

These signal charts show the last hourly signals of the Countdown Indicator. I recommend using it in discretionary trading using 8 periods as a lookback and a 3-period exponential moving average. Of course, it is possible to transform it into a systematic trading strategy, but the lookback periods will change.



COUNTDOWN INDICATOR AVERAGE CROSS

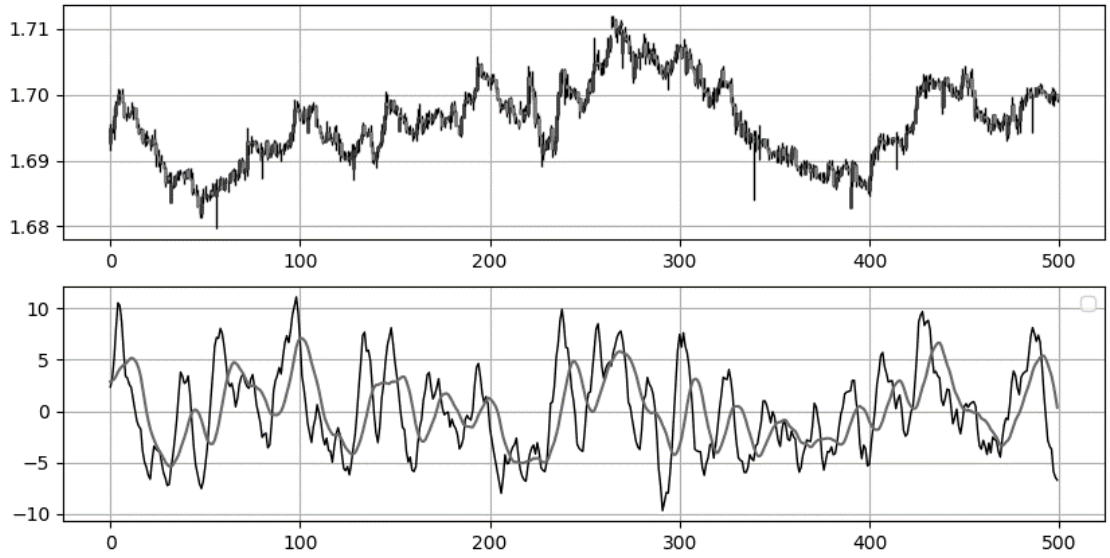
"Nope."

Generally, technical indicators are accompanied by moving averages calculated on their values. This is particularly useful for unbounded indicators such as the Countdown Indicator. In this section, we will calculate a simple moving average on the values of the Countdown Indicator and derive signals from their cross.

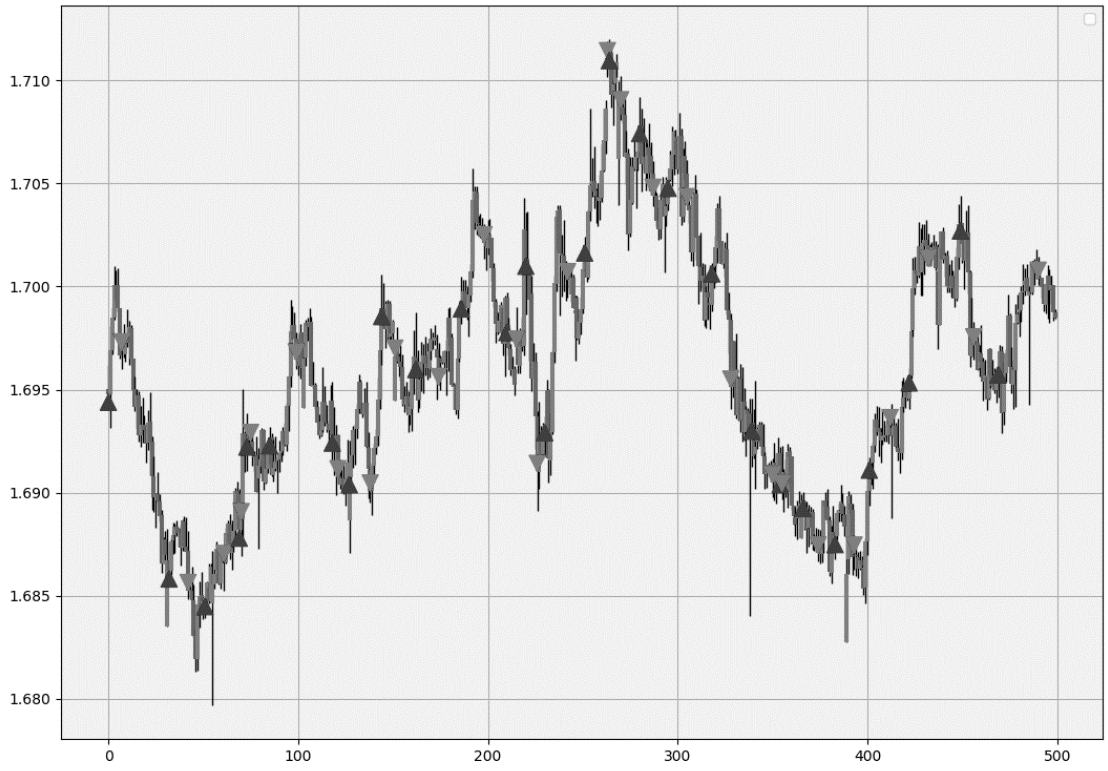
The conditions for the strategy are as follows:

- A bullish signal is triggered whenever the indicator surpasses its moving average.
- A bearish signal is triggered whenever the indicator breaks its moving average.

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.



The above plot shows the 8-period Countdown with its 13-period simple moving average. The next plot shows the signals generated from the conditions above.




```
def signal(Data, countdown_col, ma_col, buy, sell):
    Data = adder(Data, 10)
    Data = rounding(Data, 5)
    for i in range(len(Data)):
        if Data[i, countdown_col] > Data[i, ma_col] and Data[i - 1, countdown_col] < Data[i - 1, ma_col]:
            Data[i, buy] = 1
        elif Data[i, countdown_col] < Data[i, ma_col] and Data[i - 1, countdown_col] > Data[i - 1, ma_col]:
            Data[i, sell] = -1
    return Data
```

The strategy has frequent signals, but the question remains whether if they are profitable or not. Generally, as mentioned previously, moving average crosses are interesting but require more in-depth analysis so that we try to remove the lagging effect as much as possible.

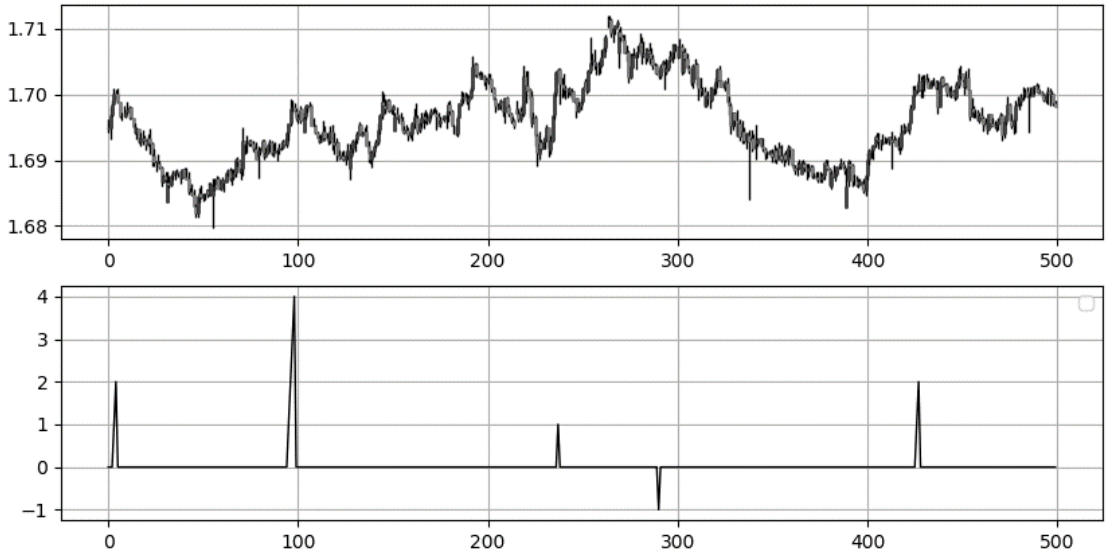
COUNTDOWN INDICATOR DURATION

"Probably works better in theory."

The conditions for the strategy are as follows:

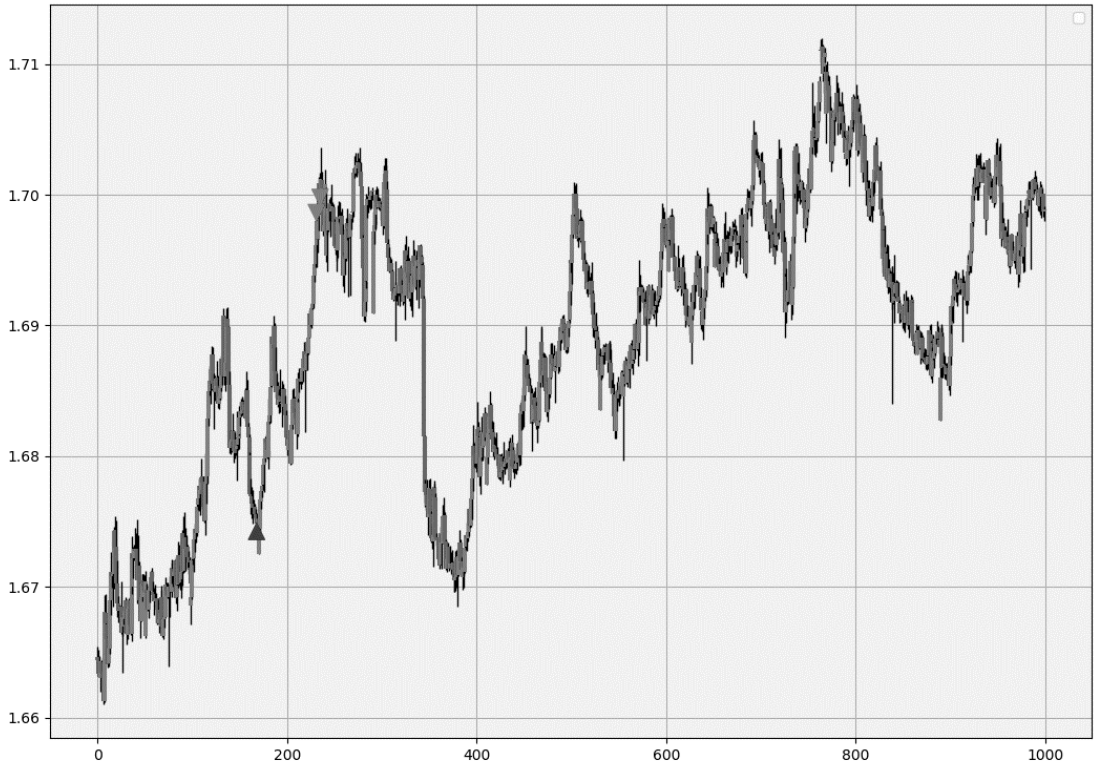
- A bullish signal is triggered whenever the indicator spends more time than expected in the oversold zone.
- A bearish signal is triggered whenever the indicator spends more time than expected in the overbought zone.

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.



The above chart shows the hourly EURNZD values with the extreme technique seen previously, applied on the 8-period Countdown Indicator. The next chart shows signals generated when the indicator spends 5 or more periods at the extremes. This is another way of saying that historically, the market spends 5 timesteps on the extremes before showing a reaction. Let us choose the same barriers a -10/10 to keep consistency.

We can see that the Countdown Indicator tends to not spend much time at the extremes and reverts back quickly to normality. This quality can be exploited by finding a historical average length of extreme time that is more accurate than the other indicators due to less outliers. The chart below shows the signals generated based on a 8-period Countdown Indicator where the surpass or break of -10/10 for over 5 time periods triggers a trade. We notice only 3 signals (one buy signal at the bottom and two sell signals at the top).



The next function is what defines a signal where we write the initial condition of being above or below a certain extreme for a chosen time period such as 5 periods, then it filters and eliminates duplicates.

```
def signal(Data, extreme, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, extreme] <= -5 and Data[i - 1, buy] == 0 and Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0:
            Data[i, buy] = 1
        elif Data[i, extreme] >= 5 and Data[i - 1, sell] == 0 and Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0:
            Data[i, sell] = -1
    return Data
```

DEMARKER INDICATOR EXTREMES

"Another masterpiece from the great Demark."

Another well-known indicator worth-studying is the Demarker coming from the famous Tom Demark. It is a contrarian indicator that resembles the Relative Strength Index. Created by Tom Demark, the Demarker oscillator belongs to the family of contrarian indicators like the RSI and the Stochastic. It is created with the sake of determining demand to find oversold and overbought conditions.

It resembles the Relative Strength Index in that it is bounded between 0 and 100 (The RSI is bounded between 0 and 100) as well as having oversold/overbought values generally set to 0.30 and 0.70 (The RSI's commonly agreed on levels are 30 and 70).

To create the Demarker Indicator, we can follow these steps which later are succeeded by the full Python code for the indicator's function.

- Select a lookback period on which the calculations will be based on. For example, if you select 14, then the calculations will be done on the last 14 observations counting the current one. Let us start with the standard 14 periods.
- Calculate the two main variables in the indicator which are referred to as the DeMAX and the DeMIN. Their formulas are as follows:

$$DeMAX = \begin{cases} High - Previous\ High & if > 0 \\ 0\ if\ High - Previous < 0 \end{cases}$$

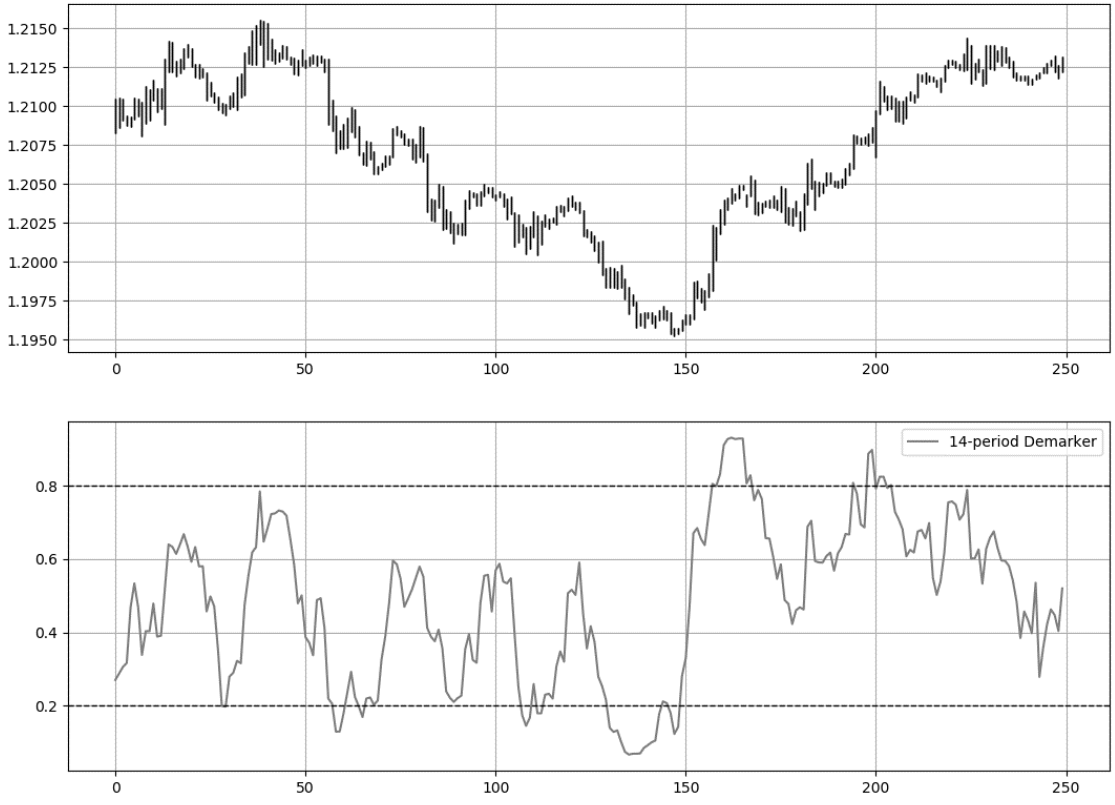
$$DeMIN = \begin{cases} Previous\ Low - Low & if > 0 \\ 0\ if\ Previous\ Low - Low < 0 \end{cases}$$

The above formula states that the DeMAX value equals the current high minus the previous high if it is greater than zero. Otherwise, it equals zero. Similarly, the DeMIN value equals the previous low minus the current low if it is greater than zero.

- Finally, to calculate the DeMarker Indicator, we simply divide the 14-period moving average of the DeMAX by the summation of the DeMAX and DeMIN average values. This is illustrated by the formula below:

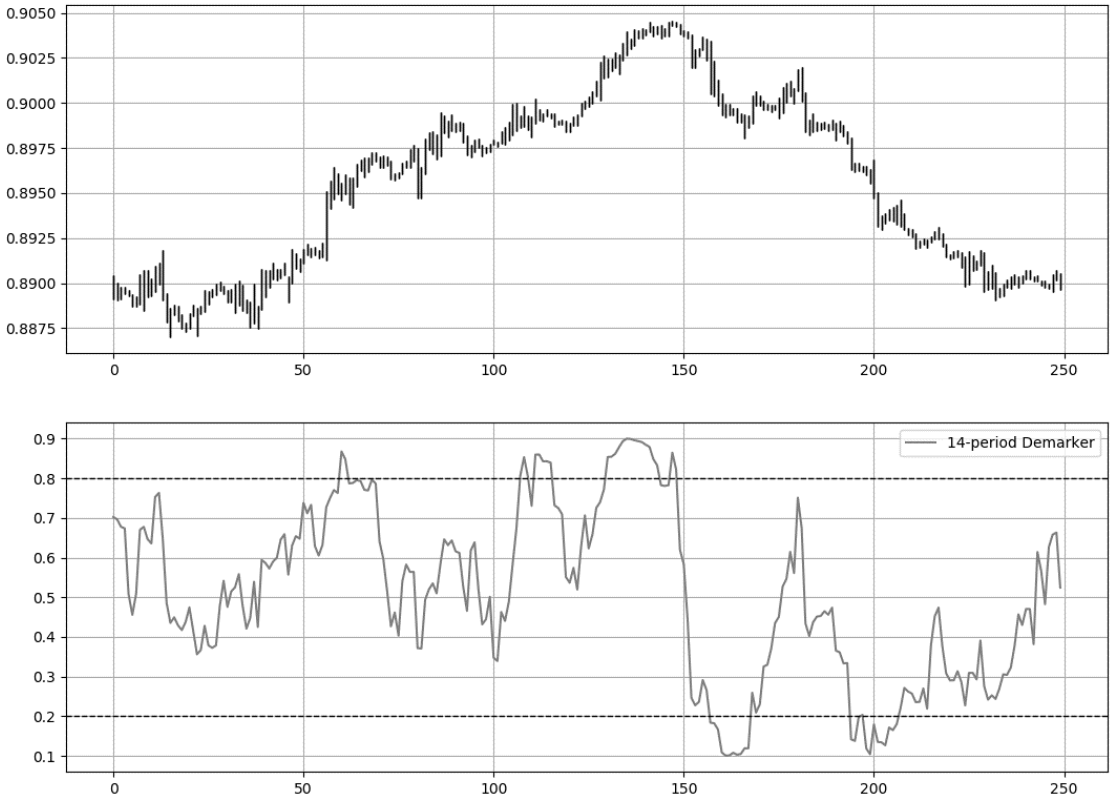
$$DeMarker = \frac{SMA_n(DeMAX)}{SMA_n(DeMAX) + SMA_n(DeMIN)}$$

This calculation should give us a plot that resembles the below when applied on the EURUSD hourly data.



Now, we can finally and easily code this in Python this way:

```
def demarker(Data, lookback, high, low, where):
    # Calculating DeMAX
    for i in range(len(Data)):
        if Data[i, high] > Data[i - 1, high]:
            Data[i, where] = Data[i, high] - Data[i - 1, high]
        else:
            Data[i, where] = 0
    # Calculating the Moving Average on DeMAX
    Data = ma(Data, lookback, where, where + 1)
    # Calculating DeMIN
    for i in range(len(Data)):
        if Data[i - 1, low] > Data[i, low]:
            Data[i, where + 2] = Data[i - 1, low] - Data[i, low]
        else:
            Data[i, where + 2] = 0
    # Calculating the Moving Average on DeMIN
    Data = ma(Data, lookback, where + 2, where + 3)
    # Calculating DeMarker
    for i in range(len(Data)):
        Data[i, where + 4] = Data[i, where + 1] / (Data[i, where + 1] + Data[i, where + 3])
    # Removing Excess Columns
    Data = deleter(Data, where, 4)
    return Data
```

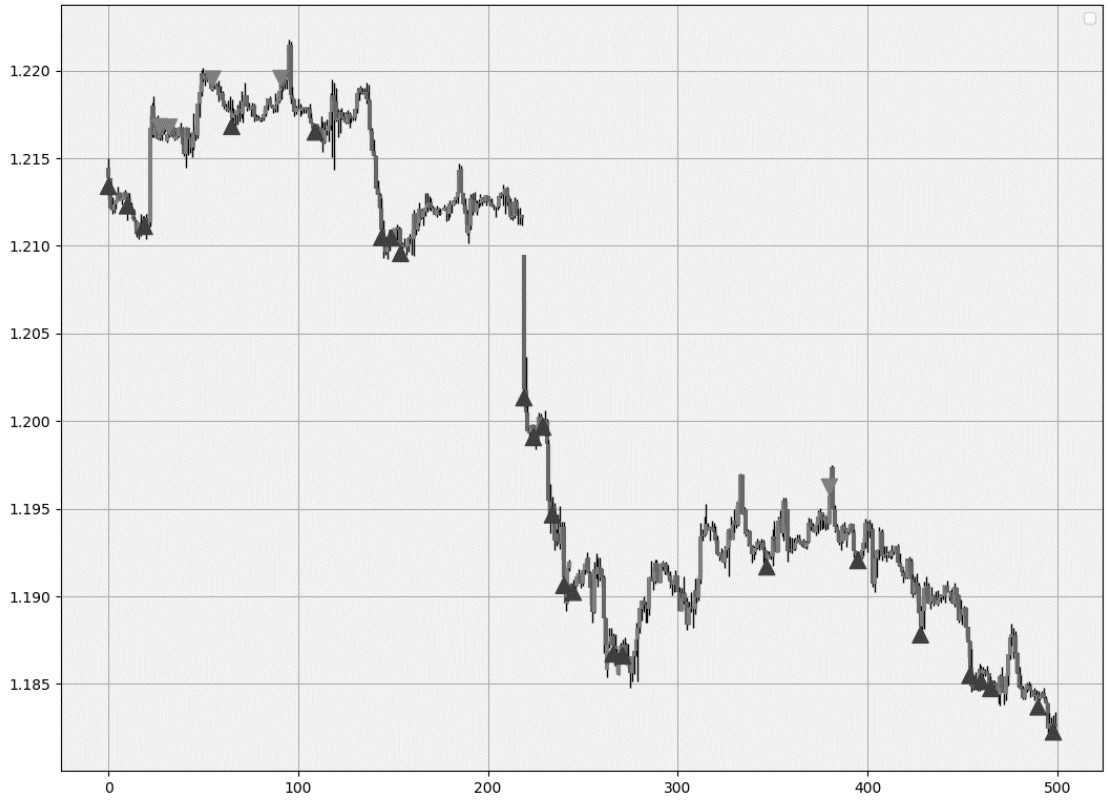


An overbought level is an area where the market is perceived to be extremely bullish and is bound to consolidate. An oversold level is an area where market is perceived to be extremely bearish and is bound to bounce. The conditions for the strategy are as follows:

- Go long (Buy) whenever the Demarker reaches 0.30 with the four previous readings above -0.30.
- Go short (Sell) whenever the Demarker reaches 0.70 with the four previous readings below 0.70.

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.

```
def signal(Data, indicator_column, buy, sell):
    Data = adder(Data, 2)
    for i in range(len(Data)):
        if Data[i, indicator_column] < lower_barrier and Data[i - 1, buy] == 0 and \
            Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0:
            Data[i, buy] = 1
        elif Data[i, indicator_column] > upper_barrier and Data[i - 1, sell] == 0 and \
            Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0:
            Data[i, sell] = -1
    return Data
```



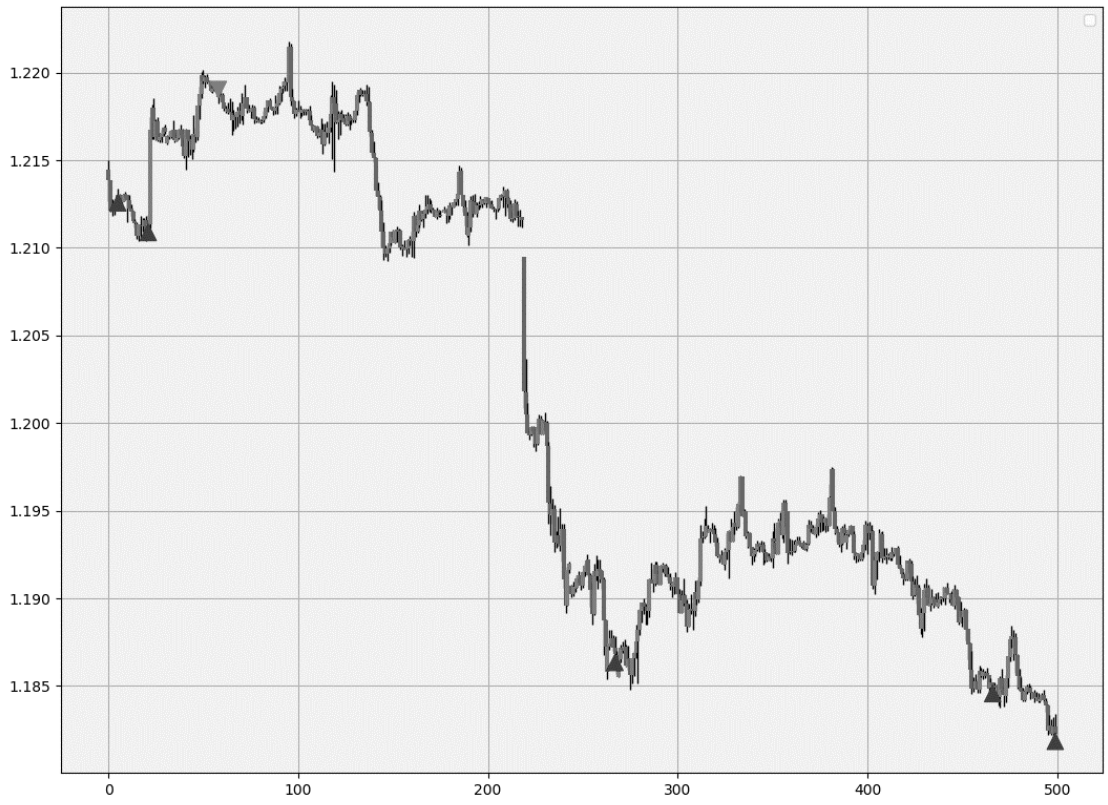
The above chart shows signals generated by the 14-period Demarker following the 0.20/0.80 rule. The values are of the EURUSD pair.

DEMARKER INDICATOR DIVERGENCE

"Trying out the divergence technique on the Demarker."

As previously seen, divergences are an integral technique in detecting trend exhaustions. Using the same function, we can apply it on the readings of the Demarker. This can be done using the following syntax.

```
width = 20
my_data = demarker(my_data, lookback, 1, 2, 4)
my_data = adder(my_data, 15)
my_data = divergence(my_data, demarker_col, lower_barrier, upper_barrier, width, 7, 8)
```



The previous plot shows an example of a signal chart on the EURUSD using the conditions of the divergence. The function can be tailored by tweaking the width variable which is simply the distance between the two tops or bottoms.

The divergence method works better on the Demarker that it does on the Stochastic Oscillator, however, more research must be made to determine if it delivers on average better signals than the RSI or not. In anyway, one could look at the divergence signals from both perspectives. Having the same divergence signal on both indicators is a conviction enhancer.

DEMARKER INDICATOR AVERAGE CROSS

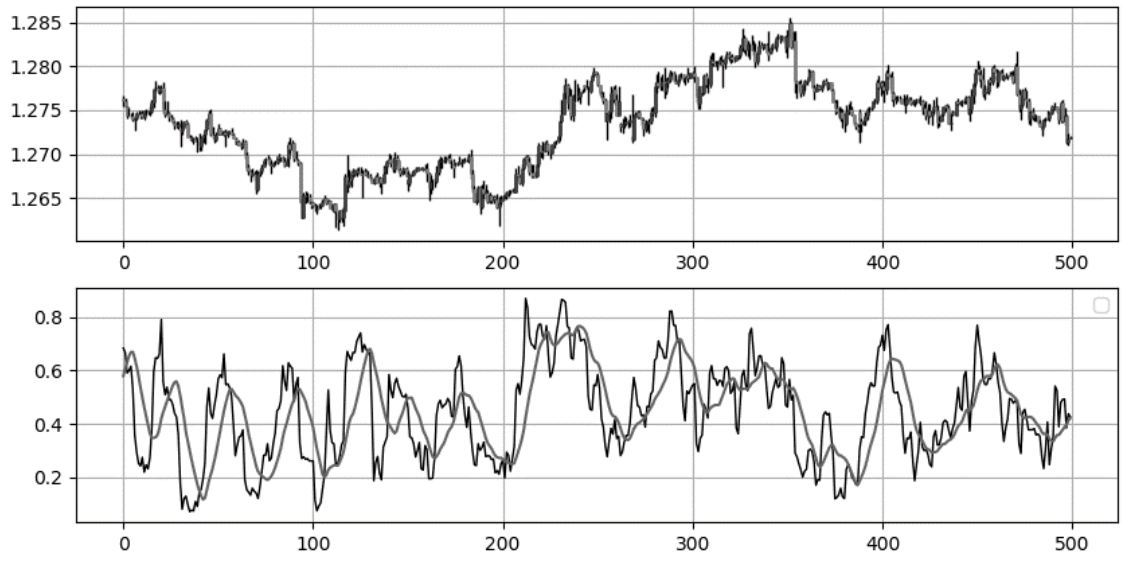
"Trying out the cross strategy on the Demarker."

Generally, technical indicators are accompanied by moving averages calculated on their values. In this section, we will calculate a simple moving average on the values of the Demarker and derive signals from their cross.

The conditions for the strategy are as follows:

- A bullish signal is triggered whenever the indicator surpasses its moving average.
- A bearish signal is triggered whenever the indicator breaks its moving average.

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.



The above plot shows the 14-period Demarker with its 13-period simple moving average. The next plot shows the signals generated from the conditions above.


```
def signal(Data, countdown_col, ma_col, buy, sell):
    Data = adder(Data, 10)
    Data = rounding(Data, 5)
    for i in range(len(Data)):
        if Data[i, countdown_col] > Data[i, ma_col] and Data[i - 1, countdown_col] < Data[i - 1, ma_col]:
            Data[i, buy] = 1
        elif Data[i, countdown_col] < Data[i, ma_col] and Data[i - 1, countdown_col] > Data[i - 1, ma_col]:
            Data[i, sell] = -1
    return Data
```

The strategy has frequent signals, but the question remains whether if they are profitable or not. Generally, as mentioned previously, moving average crosses are interesting but require more in-depth analysis so that we try to remove the lagging effect as much as possible.

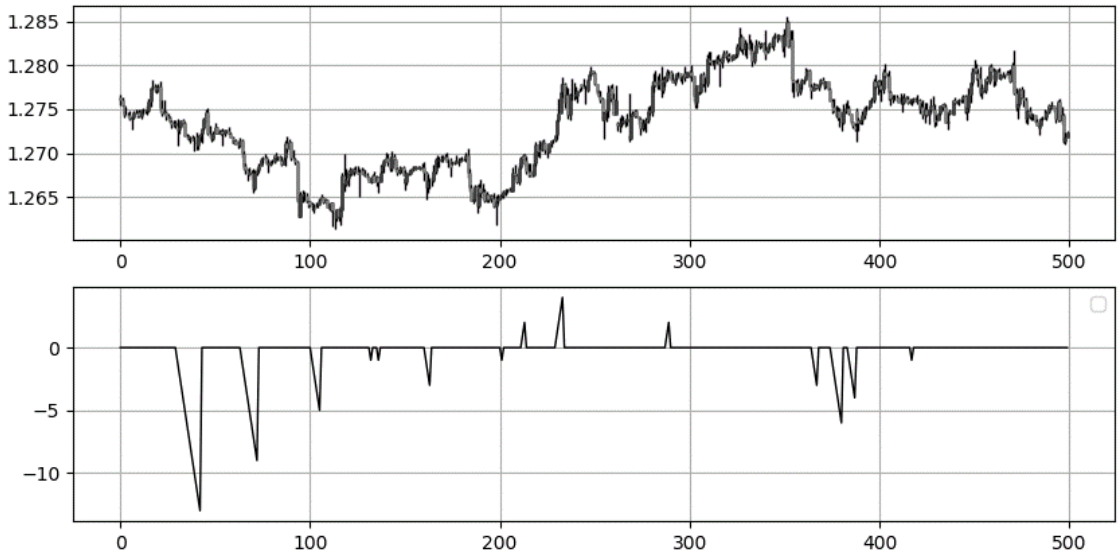
DEMARKER INDICATOR DURATION

"Will this be a good strategy?"

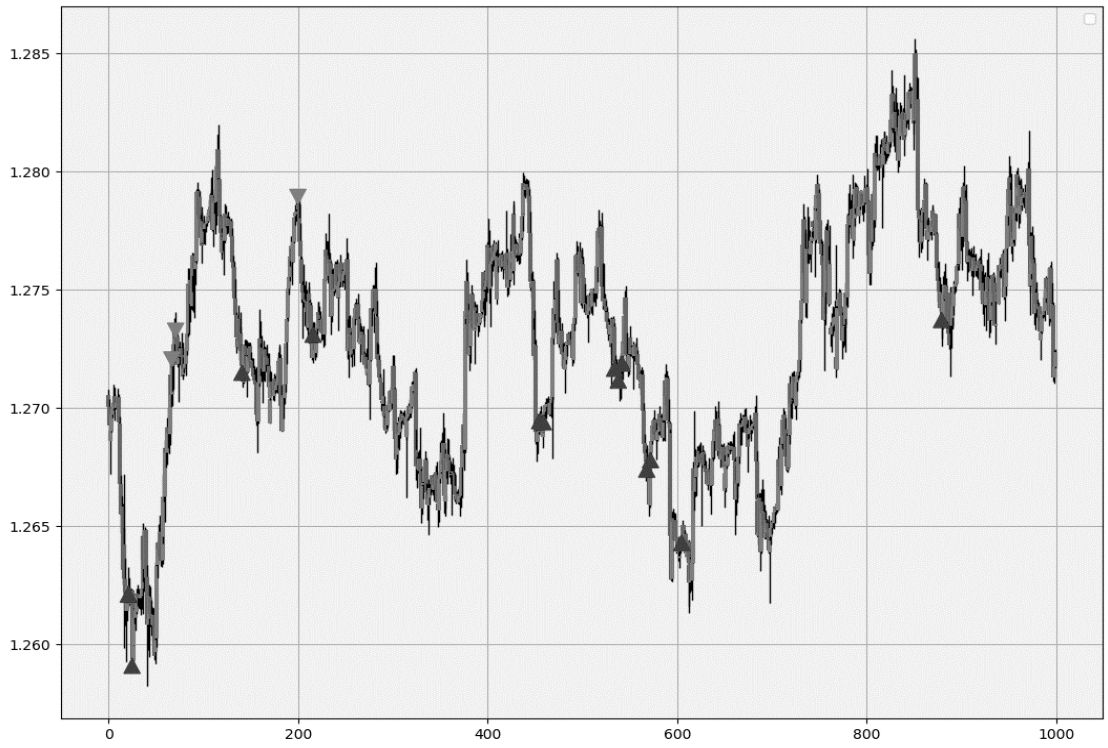
The conditions for the strategy are as follows:

- A bullish signal is triggered whenever the indicator spends more time than expected in the oversold zone.
- A bearish signal is triggered whenever the indicator spends more time than expected in the overbought zone.

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.



The signal chart on the strategy with the same conditions as its predecessors can be found in the next plot.



The next function is what defines a signal where we write the initial condition of being above or below a certain extreme for a chosen time period such as 5 periods, then it filters and eliminates duplicates. A duplicate example is when the Stochastic Oscillator spends 7 time periods below the oversold level, giving us 3 signals at every time stamp.

```
def signal(Data, extreme, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, extreme] <= -5 and Data[i - 1, buy] == 0 and Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0:
            Data[i, buy] = 1
        elif Data[i, extreme] >= 5 and Data[i - 1, sell] == 0 and Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0:
            Data[i, sell] = -1
    return Data
```

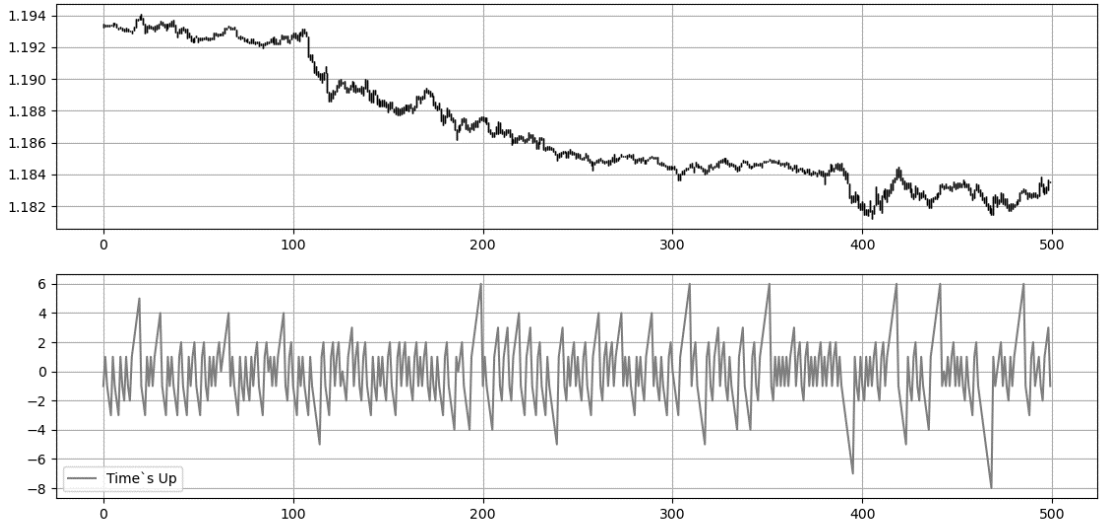
TIME'S UP INDICATOR EXTREMES

"An interesting concept and indicator."

What if we think of an extremely simple concept? Counting the number of positive and negative days and deriving trading signals from them is something we can try to demystify and see whether it is profitable or not. The main idea of the Time's Up Indicator is to count the consecutive up and down days and see whether a pattern is formed on specific barriers. For example, does having five consecutive up days consistently provide a reaction to the downside?

As mentioned in the introduction, the Time's Up Indicator will be a counter of the consecutive positive and negative days. For example, if we have three days where the closing price is greater than the opening price, then the Time's Up Indicator will show a reading of 3. The idea is to search for reversal patterns and back-test them. Later, we can simply try to optimize the barriers by looping around them. We can sum up the steps required to calculate this indicator:

- Calculate the price difference between the current market price and the one preceding it. This gives us the change in price.
- Create a column where we input the following condition: if the change is positive, then the current value in the column equals the previous one plus one.
- Create a column where we input the following condition: if the change is negative, then the current value in the column equals the previous one plus one. We can multiply the result by -1 to obtain negative values.
- Sum the values from the second column and the third column to arrive at the Time's Up Indicator.



Of course, to easily calculate the indicator, we need an OHLC array. The width seen in the function later is there to change the lookback period. For example, if it were set to 1, the calculation would be for the current closing price and the one preceding it, while if it were set to 4, it would be the one 4 periods ago.

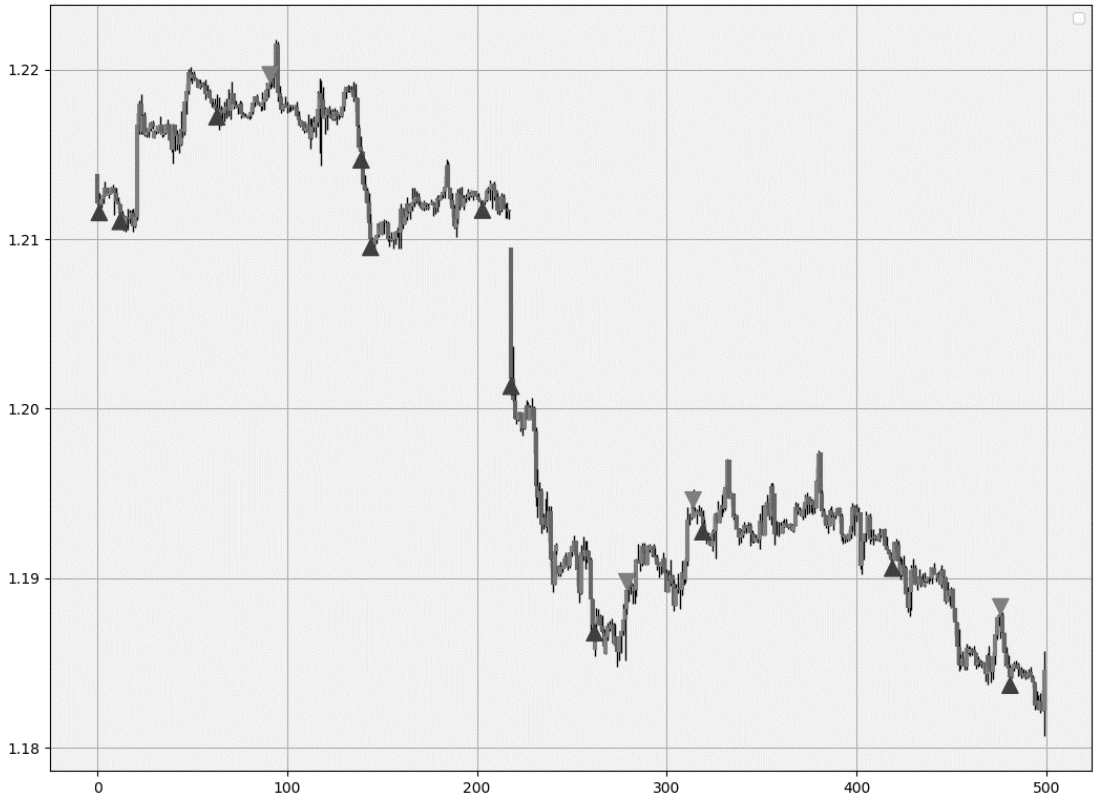
```
def signal(Data, indicator_column, buy, sell):
  Data = adder(Data, 2)
  for i in range(len(Data)):
    if Data[i, indicator_column] < lower_barrier and Data[i - 1, buy] == 0 and \
      Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0:
      Data[i, buy] = 1
    elif Data[i, indicator_column] > upper_barrier and Data[i - 1, sell] == 0 and \
      Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0:
      Data[i, sell] = -1
  return Data
```

```
def time_up(Data, width, what, where):
    # Adding the required columns
    Data = adder(Data, 4)
    # Calculating the difference in prices
    for i in range(len(Data)):
        Data[i, where] = Data[i, what] - Data[i - width, what]
    # Upward Timing
    for i in range(len(Data)):
        Data[0, where + 1] = 1
        if Data[i, where] > 0:
            Data[i, where + 1] = Data[i - width, where + 1] + 1
        else:
            Data[i, where + 1] = 0
    # Downward Timing
    for i in range(len(Data)):
        Data[0, where + 2] = 1
        if Data[i, where] < 0:
            Data[i, where + 2] = Data[i - width, where + 2] + 1
        else:
            Data[i, where + 2] = 0
    # Changing signs
    Data[:, where + 2] = -1 * Data[:, where + 2]
    # Time's Up Indicator
    Data[:, where + 3] = Data[:, where + 1] + Data[:, where + 2]
    # Cleaning rows/columns
    Data = deleter(Data, where, 3)
    return Data
```

The conditions for the strategy are as follows:

- A bullish signal is triggered whenever the indicator reaches or breaks the lower barrier (oversold zone).
- A bearish signal is triggered whenever the indicator reaches or surpasses the upper barrier (overbought zone).

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.



The Time's Up indicator is very intuitive, simple, and tweakable. It is also a valuable indicator to look at in tandem with other techniques.

THE DISPARITY INDEX EXTREMES

"A simple indicator based on moving averages."

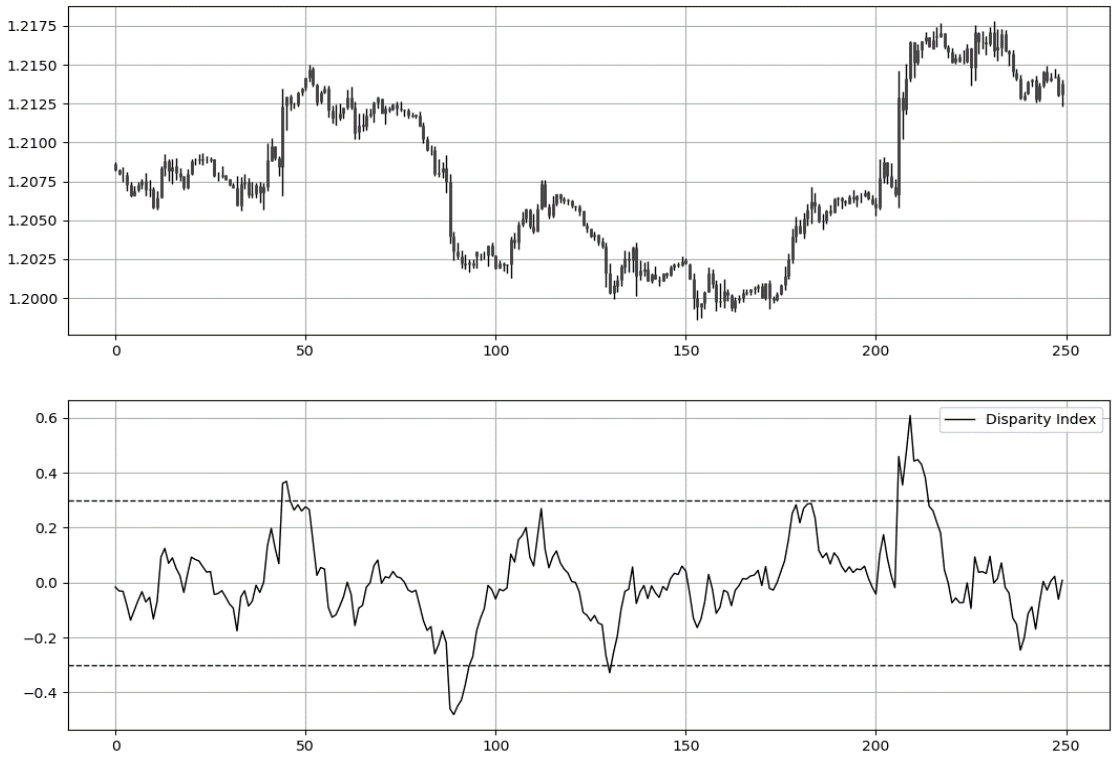
Distance analysis is sometimes very important to determine whether the market is about to reverse or correct. Statistically, we expect to remain in normality more often than extremes, therefore, when we do reach the extremes, we expect some form of reversion to the mean unless Euphoria is in hyperdrive, and participants are excessively buying or selling following big news.

Created by Steve Nison, the Disparity Index is a very simple calculation that measures the distance between the price and its moving average using the following formula:

$$Disparity\ Index = \left(\frac{Current\ Closing\ Price}{Current\ Moving\ Average\ value} - 1\right)\ x\ 100$$

In Python, we can code the Disparity Index following this syntax:

```
def disparity_index(Data, lookback, close, where):
    # Adding a column
    Data = adder(Data, 2)
    # Calculating the moving average on closing prices
    Data = ma(Data, lookback, close, where)
    # Calculating the Disparity Index
    for i in range(len(Data)):
     Data[i, where + 1] = ((Data[i, close] / Data[i, where]) - 1) * 100
    # Cleaning
    Data = deleter(Data, where, 1)
    return Data
```



The indicator is extremely simple to understand and code. It is always worth back-testing such indicators and seeing if they can be optimized.

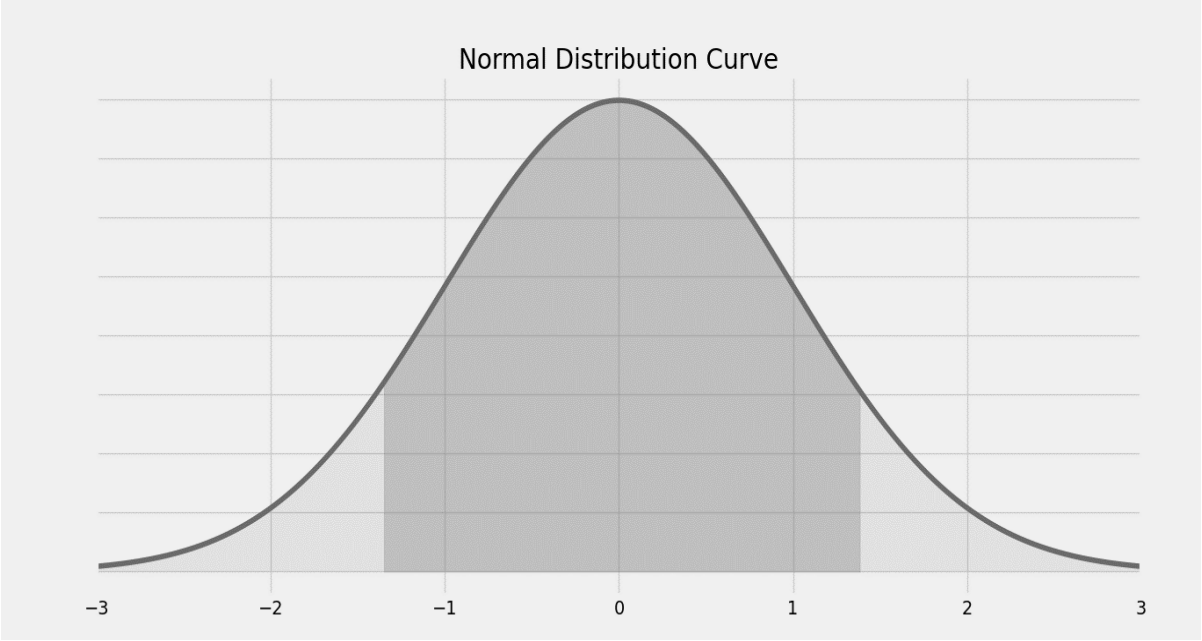
```
def signal(Data, indicator_column, buy, sell):
  Data = adder(Data, 2)
  for i in range(len(Data)):
    if Data[i, indicator_column] < lower_barrier and Data[i - 1, buy] == 0 and \
      Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0:
      Data[i, buy] = 1
    elif Data[i, indicator_column] > upper_barrier and Data[i - 1, sell] == 0 and \
      Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0:
      Data[i, sell] = -1
  return Data
```

The indicator resembles the distance strategy we have seen in Part 1, but it is not normalized. However, back-testing results showed that the distance strategy performs better than the Disparity Index which is why I recommend sticking to the strategy from the first part.

THE FISHER TRANSFORM EXTREMES

"I don't know why it works, but it works."

One of the pillars of descriptive statistics is the normal distribution curve. It describes how random variables are distributed and centered around a central value. It often resembles a bell. Some data in the world are said to be normally distributed. This means that their distribution is symmetrical with 50% of the data lying to the left of the mean and 50% of the data lying to the right of the mean. Its mean, median, and mode are also equal as seen in the below curve.



The above curve shows the number of values within a number of standard deviations. For example, the area shaded in red represents around 1.33x of standard deviations away from the mean of zero. We know that if data is normally distributed then:

- About 68% of the data falls within 1 standard deviation of the mean.
- About 95% of the data falls within 2 standard deviations of the mean.
- About 99% of the data falls within 3 standard deviations of the mean.

Presumably, this can be used to approximate the way to use financial returns data, but studies show that financial data is not normally distributed. For the moment, we can

assume it is so that we can use such indicators. The flaw of the method does not hinder much its usefulness. Let us now see how to create the Modified Fisher Transformation which is basically like the original one but with some minor changes to enhance the results and make them easier to obtain.

Created by John F. Ehlers, the indicator seeks to transform the price into a normal Gaussian (normal) distribution. This is very helpful in detecting reversals. The steps to create the Modified Fisher Transformation are somewhat similar to the original Fisher Transformation. Here is how we do it:

Select a lookback period and calculate a normalized version of the OHLC data using the original Stochastic formula as seen in the formula below:

$$x_{normalized} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Trap the values from the first step between -1 and +1 using the following normalization formula:

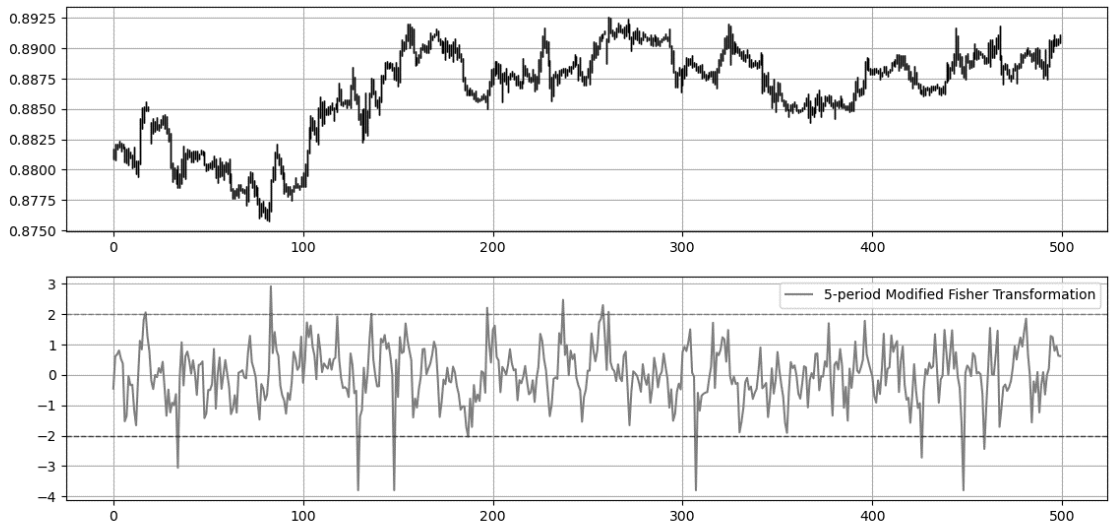
$$x_{new} = 2 * x_{normalized} - 1$$

Create a condition that eliminates the -1.00's and +1.00's and transforms them into -0.999's and +0.999's so that we do not get infinite values. This also serves to make the indicator bounded between two levels we will see later.

Apply the following formula to the results from the last step:

$$Fisher\ Transformation = \frac{1}{2} * \ln\left(\frac{1+X}{1-X}\right)$$

Now that we have the Modified Fisher Transformation Indicator, we can proceed by coding it in Python before starting the back-tests. The below plot shows a 5-period Modified Fisher Transformation with subjective boundaries at -2.00 and +2.00.



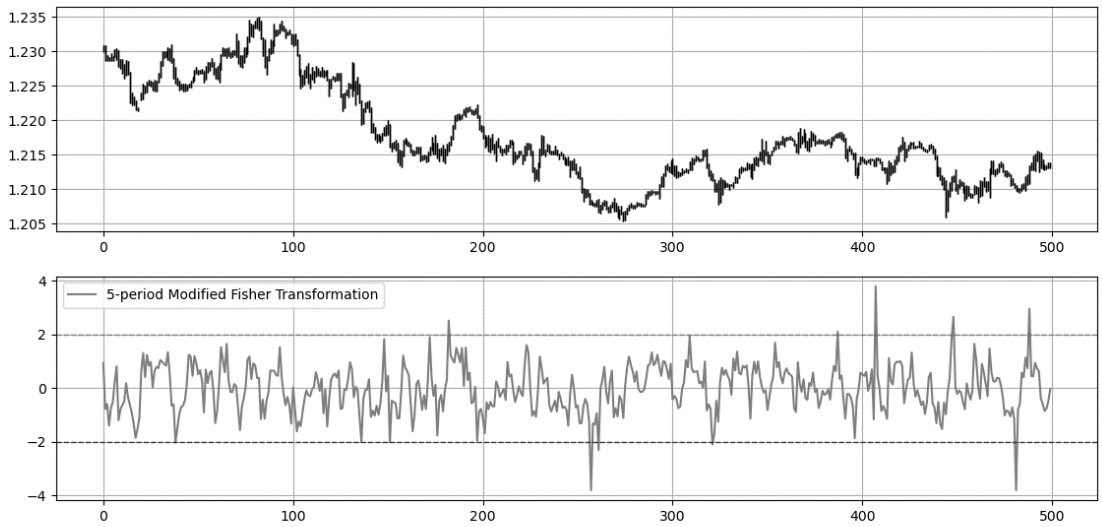
Naturally with the correction I have added, the maximum values will be 3.80 and the minimum values will be -3.80. Therefore, we can say that the indicator is now bounded even though the original version was not shown to be bounded.

```
def signal(Data, indicator_column, buy, sell):
  Data = adder(Data, 2)
  for i in range(len(Data)):
    if Data[i, indicator_column] < lower_barrier and Data[i - 1, buy] == 0 and \
      Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0:
      Data[i, buy] = 1
    elif Data[i, indicator_column] > upper_barrier and Data[i - 1, sell] == 0 and \
      Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0:
      Data[i, sell] = -1
  return Data
```

```
def modified_fisher_transform(Data, lookback, what, where):
  Data = stochastic(Data, lookback, what, where)
  Data[:, where] = Data[:, where] / 100
  Data[:, where] = (2 * Data[:, where]) - 1
  for i in range(len(Data)):
    if Data[i, where] == 1:
      Data[i, where] = 0.999
    if Data[i, where] == -1:
      Data[i, where] = -0.999
  for i in range(len(Data)):
   Data[i, where + 1] = 0.5 * (np.log((1 + Data[i, where]) / (1 - Data[i, where])))
  return Data
```

 # Using the Transformation on an OHLC array with a few extra columns

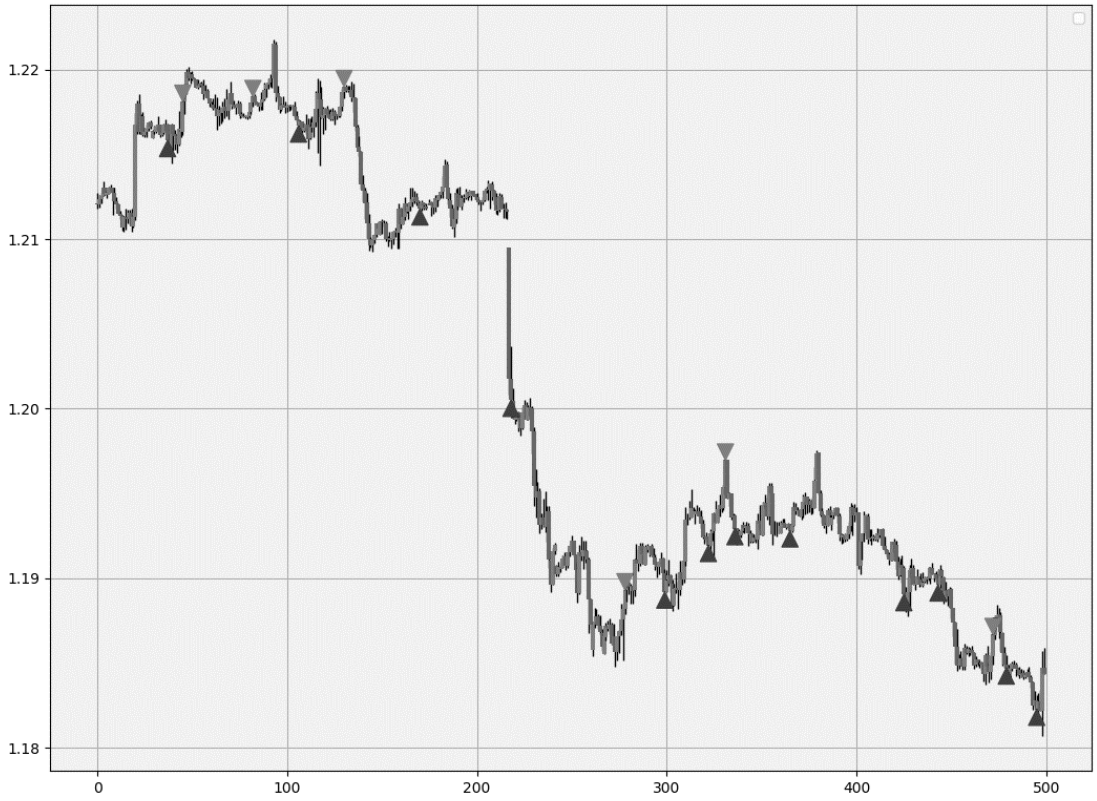
```
my_ohlc_data = modified_fisher_transform(my_ohlc_data, 5, 3, 4)
```



An overbought level is an area where the market is perceived to be extremely bullish and is bound to consolidate. An oversold level is an area where market is perceived to be extremely bearish and is bound to bounce. The conditions for the strategy are as follows:

- Go long (Buy) whenever the indicator reaches -2 with the four previous readings above -2.
- Go short (Sell) whenever the indicator reaches 2 with the four previous readings below 2.

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.



The Fisher Transform is an amazing indicator created by one of the leading researchers in the world of finance. Personally, I use this indicator constantly. Even though it is correlated to other technical indicators such as the RSI, it still manages to be unique and deliver quality signals.

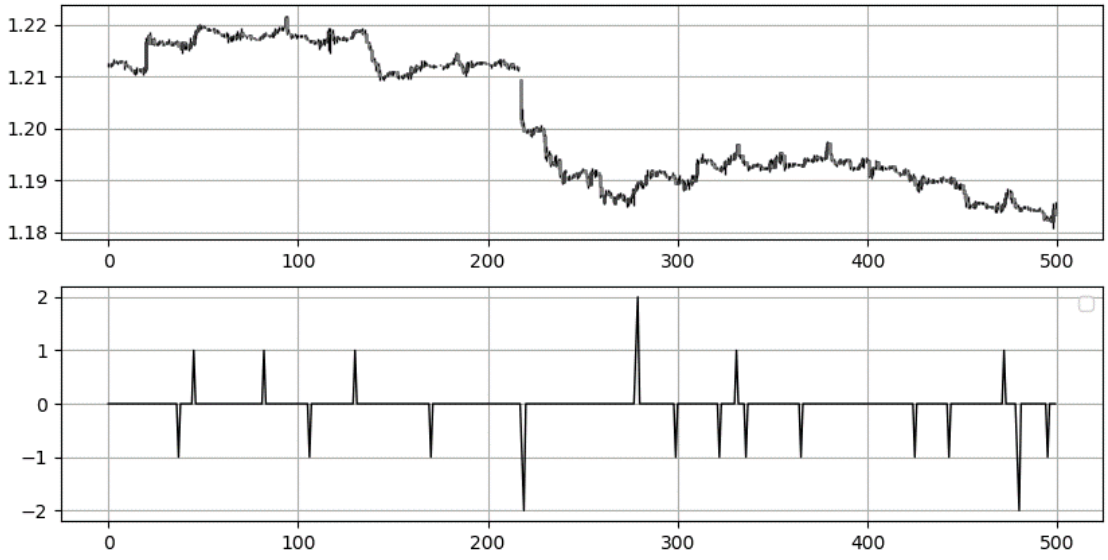
THE FISHER TRANSFORM DURATION

"Very complex math."

The conditions for the strategy are as follows:

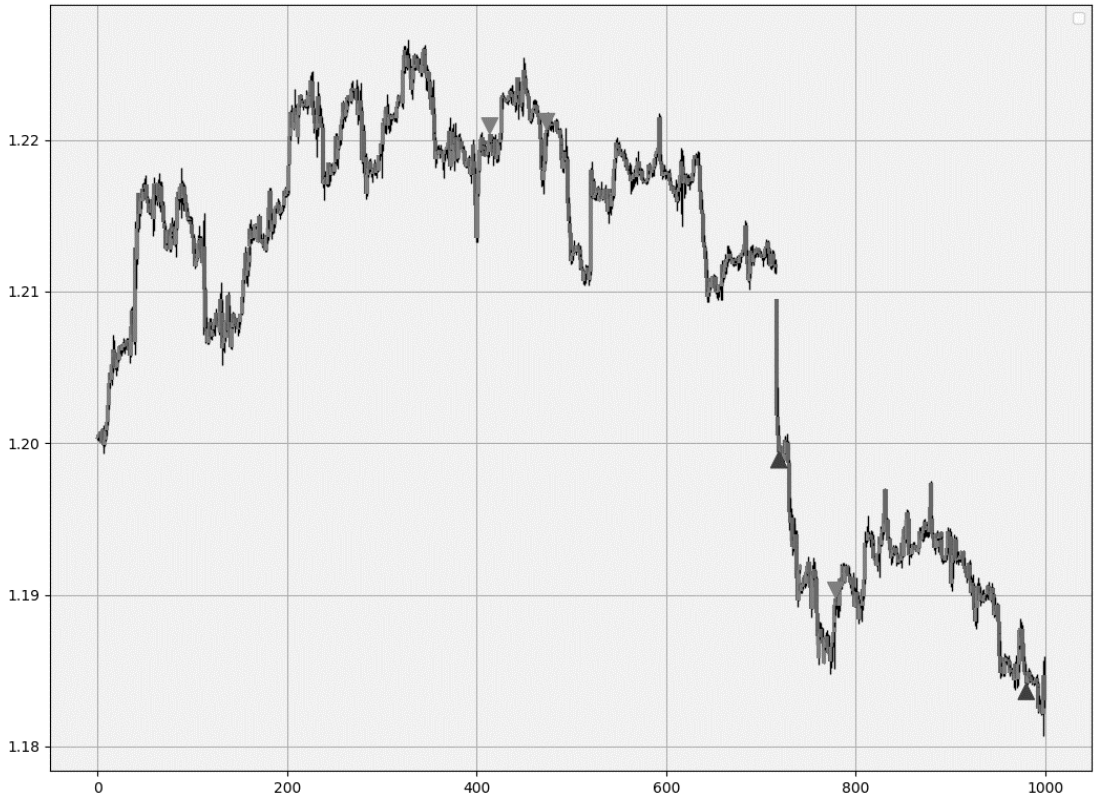
- A bullish signal is triggered whenever the indicator spends more time than expected in the oversold zone.
- A bearish signal is triggered whenever the indicator spends more time than expected in the overbought zone.

To avoid duplicate and successive trades, we can create a condition where the algorithm only acts on the signal if the previous time periods do not have any signals of the same type. For example, if we get a signal while we already had a signal two periods ago, the signal is not accounted for.



The above chart shows the hourly EURUSD values with the extreme technique seen previously, applied on the 5-period Modified Fisher Transform. The next chart shows signals generated when the indicator spends 5 or more periods at the extremes. This is another way of saying that historically, the market spends 5 timesteps on the extremes before showing a reaction. Let us choose the same barriers a -2/2 to keep consistency.

We can see that the indicator tends to not spend much time at the extremes and reverts quickly to normality. This quality can be exploited by finding a historical average length of extreme time that is more accurate than the other indicators due to less outliers. The chart below shows the signals generated based on a 5-period Fisher indicator where the surpass or break of -2/2 for over 5 time periods triggers a trade.



The next function is what defines a signal where we write the initial condition of being above or below a certain extreme for a chosen time period such as 5 periods, then it filters and eliminates duplicates.

```
def signal(Data, extreme, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, extreme] <= -5 and Data[i - 1, buy] == 0 and Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0:
            Data[i, buy] = 1
        elif Data[i, extreme] >= 5 and Data[i - 1, sell] == 0 and Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0:
            Data[i, sell] = -1
    return Data
```

THE TSABM INDICATOR STRATEGY

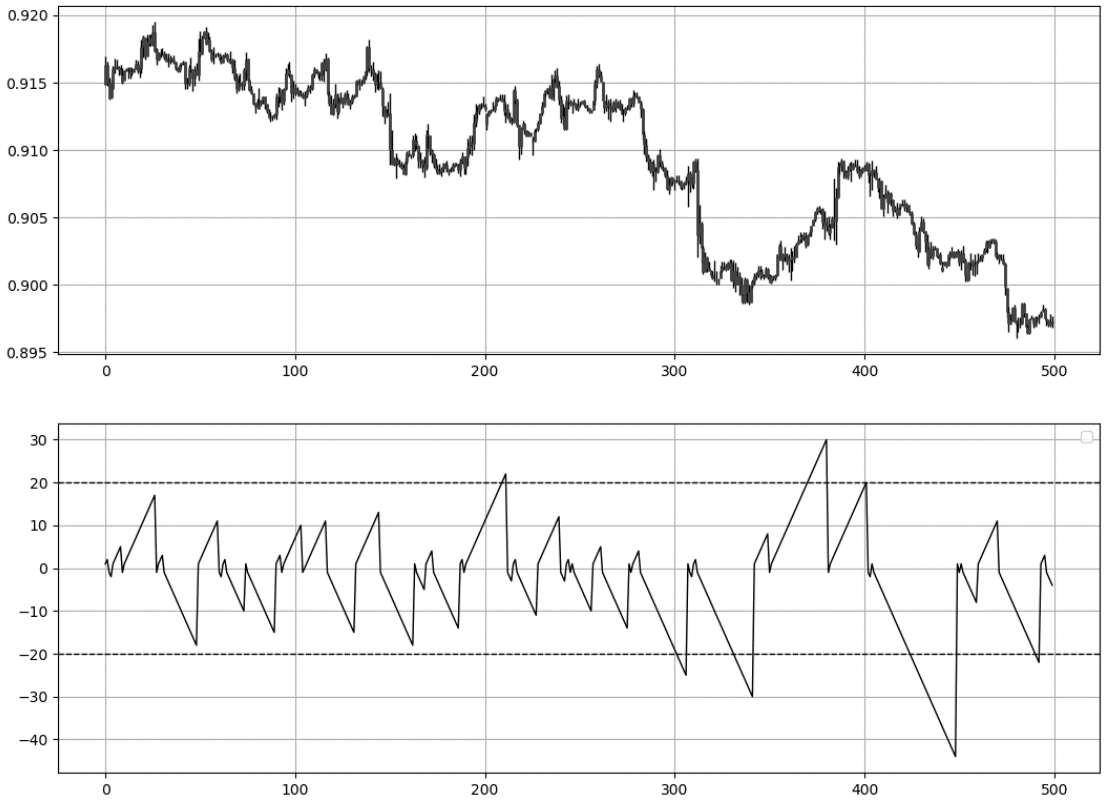
"Time is important."

Statistical properties of time series may from time to time show a characteristic known as mean-reversion. This characteristic is defined as variables going towards the direction of their mean in case, they diverge from it for a certain period of time. While we can measure the divergence or statistical deviation from the mean using volatility bands, we do not really have a way to measure the time spent away from this mean. A simple way to remedy this is to address the notion of the time spent below or above the mean. The aim here is to get a grasp on the mechanism of time with regards to the moving average and see whether it can signal trend exhaustion or not. There is a popular technique in trading called Divergence. This is where the market price continues in one direction, but the price-derived technical indicator goes into another direction. We are often used to seeing this on the Relative Strength Index as a signal of trend exhaustion and that a stabilization may occur soon. It is not a reversal signal per-se but simply an indication that the strength of the current trend is deteriorating. Hence, intuitively, the market losing momentum and may be switching from one regime to another. We will try to measure the exact same thing with the Time Spent Above/Below Mean (TSABM) calculation. The aim is to say that we are seeing a weakening on the current trend and thus may be used in tandem with a divergence signal. As it can be understood from the previous statement, it is a contrarian signal and even though it measures time, it is not a timing pattern.

To measure the TSABM, we should follow the below steps:

- Calculate a 20-period moving average on the market price. By default, I have found that this period works better than other periods, but it is encouraged to test out other periods.
- If the current market price is above its current 20-period moving average, then the counter increases by 1 each time. The starting period is 1.
- If the current market price is below its current 20-period moving average, then the counter decreases by -1 each time. The starting period is -1.

Therefore, whenever the market price switches side with respect to its moving average, the TSABM resets and follows the new condition.

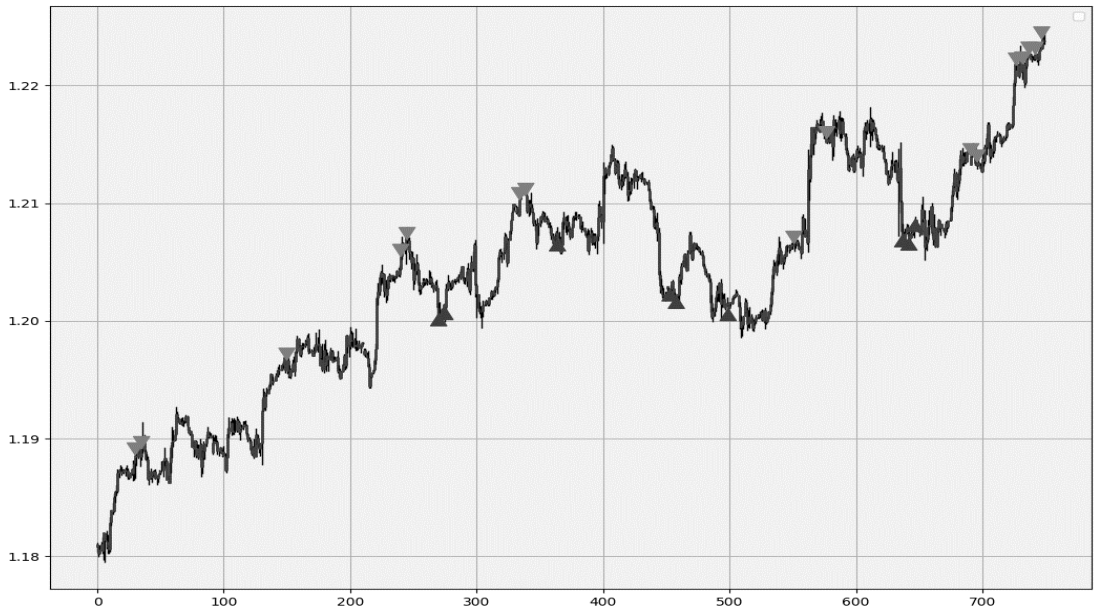


The interesting thing is that the barriers that signal trend exhaustion are equal to the lookback period. This means that with a 20-period TSABM, the barrier that signals a bullish trend exhaustion lies at 20 (bearish signal) and the barrier that signals a bearish trend exhaustion lies at -20 (bullish signal).

```
def time_spent_above_below_mean(Data, lookback, close, where):
    # Adding the required columns
    Data = adder(Data, 4)
    # Calculating the moving average
    Data = ma(Data, lookback, close, where)
```

```
# Time Spent Above the Mean
  for i in range(len(Data)):
    Data[0, where + 1] = 1
    if Data[i, close] > Data[i, where]:
      Data[i, where + 1] = Data[i - 1, where + 1] + 1
    else:
      Data[i, where + 1] = 0
  # Time Spent Below the Mean
  for i in range(len(Data)):
    Data[0, where + 2] = -1
    if Data[i, close] < Data[i, where]:
      Data[i, where + 2] = Data[i - 1, where + 2] - 1
    else:
      Data[i, where + 2] = 0
  # Time Spent Above/Below Mean
  Data[:, where + 3] = Data[:, where + 1] + Data[:, where + 2]
  # Cleaning
  Data = deleter(Data, where, 3)
  return Data
```

With the barriers made at 20 and -20, we can take a look at the next charts. Note that the TSABM is not a trading system. Just like the Divergence method, it simply indicates some extremes in the current trend and a possible consolidation.



The above chart on the EURUSD hourly values shows exhaustion signals where the red arrows indicate a weakening in the bullish move and the green arrows indicate a weakening in the bearish move. The disadvantage of the TSABM is that it does not give any indication on when the expected move will happen. It also does not provide a stop or profit potential, and this is to be expected as it is not a trading indicator but simple a whistleblower on the prevailing trend. I recommend giving it minimal weight in the trading framework.

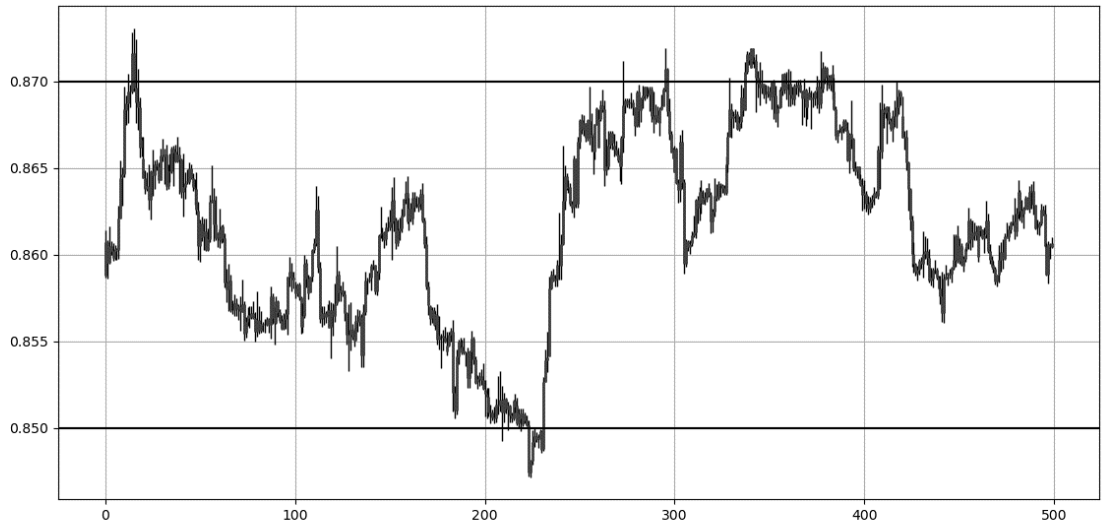
```
def signal(Data, indicator_column, buy, sell):
  Data = adder(Data, 2)
  for i in range(len(Data)):
    if Data[i, indicator_column] < lower_barrier and Data[i - 1, buy] == 0 and \
      Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0:
      Data[i, buy] = 1
    elif Data[i, indicator_column] > upper_barrier and Data[i - 1, sell] == 0 and \
      Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0:
      Data[i, sell] = -1
  return Data
```

PSYCHOLOGICAL LEVELS STRATEGY

"Round numbers serve as checkpoints and positioning."

Psychological levels form an important part in any analysis. The reason for this is because mentally, they are given more attention than other levels. For instance, which price would you retain more in your mind if you come across it? 1.1500 on the EURUSD or 1.3279 on the GBPUSD?

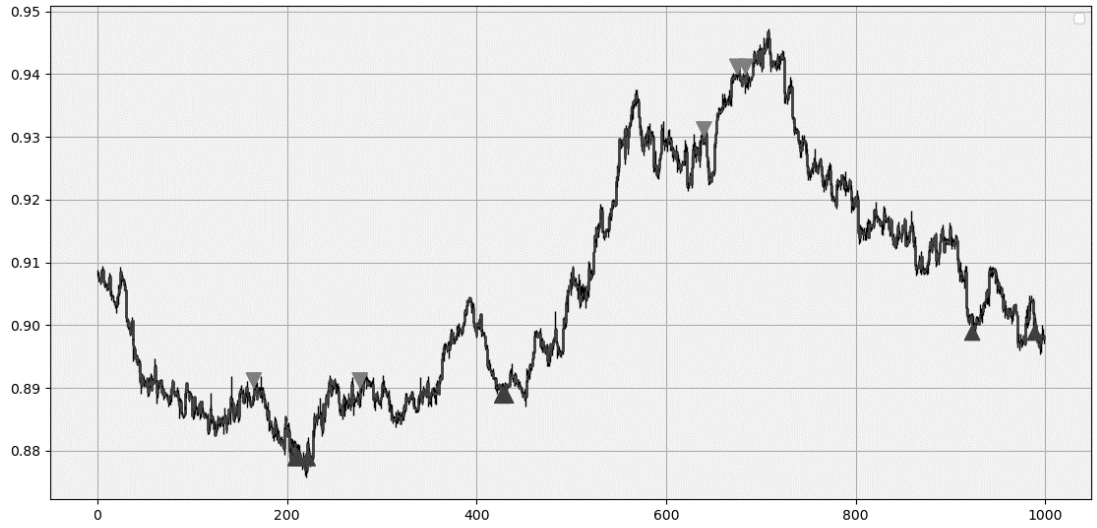
Clearly, round numbers are the first part of a psychological price. The other part is simply significance. For example, the par-value or par-level such as 1.000 on the USDCHF or 100.00 on the USDJPY is considered a par-level. The basic idea is that around these levels, the market participants may choose to place their orders and stops, hence, a form of reaction may happen.



Our aim is to develop an algorithm that trades whenever the market reaches a psychological level. This is done using a very simple loop function in Python discussed next.

```
def psychological_levels_scanner(Data, trend, signal, buy, sell):
    # Adding buy and sell columns
    Data = adder(Data, 15)
    # Rounding for ease of use
    Data = rounding(Data, 4)
    # Scanning for Psychological Levels
    for i in range(len(Data)):
        if  Data[i, 3] == 0.6000 or Data[i, 3] == 0.6100 or Data[i, 3] == 0.6200 or Data[i, 3] == 0.6300
or Data[i, 3] == 0.6400 or Data[i, 3] == 0.6500 or Data[i, 3] == 0.6600 or Data[i, 3] == 0.6700 or
Data[i, 3] == 0.6800 or Data[i, 3] == 0.6900 or Data[i, 3] == 0.7000 or Data[i, 3] == 0.7100 or
Data[i, 3] == 0.7200 or Data[i, 3] == 0.7300 or Data[i, 3] == 0.7400 or Data[i, 3] == 0.7500 or
Data[i, 3] == 0.7600 or Data[i, 3] == 0.7700 or Data[i, 3] == 0.7800 or Data[i, 3] == 0.7900 or
Data[i, 3] == 0.8000 or Data[i, 3] == 0.8100 or Data[i, 3] == 0.8200 or Data[i, 3] == 0.8300 or
Data[i, 3] == 0.8400 or Data[i, 3] == 0.8500 or Data[i, 3] == 0.8600 or Data[i, 3] == 0.8700 or
Data[i, 3] == 0.8800 or Data[i, 3] == 0.8900 or Data[i, 3] == 0.9000 or Data[i, 3] == 0.9100 or
Data[i, 3] == 0.9200 or Data[i, 3] == 0.9300 or Data[i, 3] == 0.9400 or Data[i, 3] == 0.9500 or
Data[i, 3] == 0.9600 or Data[i, 3] == 0.9700 or Data[i, 3] == 0.9800 or Data[i, 3] == 0.9900 or
Data[i, 3] == 1.0000 or Data[i, 3] == 1.0100 or Data[i, 3] == 1.0200 or Data[i, 3] == 1.0300 or
Data[i, 3] == 1.0400 or Data[i, 3] == 1.0500 or Data[i, 3] == 1.0600 or Data[i, 3] == 1.0700 or
Data[i, 3] == 1.0800 or Data[i, 3] == 1.0900 or Data[i, 3] == 1.1000 or Data[i, 3] == 1.1100 or
Data[i, 3] == 1.1200 or Data[i, 3] == 1.1300 or Data[i, 3] == 1.1400 or Data[i, 3] == 1.1500 or
Data[i, 3] == 1.1600 or Data[i, 3] == 1.1700 or Data[i, 3] == 1.1800 or Data[i, 3] == 1.1900 or
Data[i, 3] == 1.2000 or Data[i, 3] == 1.2100 or Data[i, 3] == 1.2300 or Data[i, 3] == 1.2400 or
Data[i, 3] == 1.2500 or Data[i, 3] == 1.2600 or Data[i, 3] == 1.2700 or Data[i, 3] == 1.2800 or
Data[i, 3] == 1.2900 or Data[i, 3] == 1.3000 or Data[i, 3] == 1.3100 or Data[i, 3] == 1.3200 or
Data[i, 3] == 1.3300 or Data[i, 3] == 1.3400 or Data[i, 3] == 1.3500 or Data[i, 3] == 1.3600 or
Data[i, 3] == 1.3700 or Data[i, 3] == 1.3800 or Data[i, 3] == 1.3900 or Data[i, 3] == 1.4000 or
Data[i, 3] == 1.4100 or Data[i, 3] == 1.4200 or Data[i, 3] == 1.4300 or Data[i, 3] == 1.4400 or
Data[i, 3] == 1.4500 or Data[i, 3] == 1.4600 or Data[i, 3] == 1.4700 or Data[i, 3] == 1.4800 or
Data[i, 3] == 1.4900 or Data[i, 3] == 1.5000 or Data[i, 3] == 1.5100 or Data[i, 3] == 1.5200 or
Data[i, 3] == 1.5300 or Data[i, 3] == 1.5400 or Data[i, 3] == 1.5500 or Data[i, 3] == 1.5600 or
Data[i, 3] == 1.5700 or Data[i, 3] == 1.5800 or Data[i, 3] == 1.5900 or Data[i, 3] == 1.6000 or
Data[i, 3] == 1.6100 or Data[i, 3] == 1.6200 or Data[i, 3] == 1.6300 or Data[i, 3] == 1.6400 or
Data[i, 3] == 1.6500 or Data[i, 3] == 1.6600 or Data[i, 3] == 1.6700 or Data[i, 3] == 1.6800 or
Data[i, 3] == 1.6900 or Data[i, 3] == 1.7000 or Data[i, 3] == 1.7100 or Data[i, 3] == 1.7200 or
Data[i, 3] == 1.7300 or Data[i, 3] == 1.7400 or Data[i, 3] == 1.7500 or Data[i, 3] == 1.7600 or
Data[i, 3] == 1.7700 or Data[i, 3] == 1.7800 or Data[i, 3] == 1.7900 or Data[i, 3] == 1.8000:
            Data[i, signal] = 1
return Data
```

Obviously, the above code can use some better way to define a psychological level such as modulo technique, division by 5, or a search for values where the last two digits is double zero. However, the idea is to show that even with simplistic syntax, we can arrive at what we are trying to do.



The above plot also shows the signal chart on the USDCHF H3 values. We can notice that the signals tend to be good on average due to them occurring around key reversal points. This is of course not sustainable nor is it profitable on its own but value-wise, it is definitely an addition to the framework.

```
# Signal
for i in range(len(Data)):
    if Data[i, 5] == 1 and Data[i, 3] < Data[i - trend, 3]:
        Data[i, buy] = 1
    elif Data[i, 5] == 1 and Data[i, 3] > Data[i - trend, 3]:
        Data[i, sell] = -1
# Column 5 refers to where we have applied the previous function
# Column 3 is the closing price
# The trend variable is how long should we look into the past to understand the trend
```

BOLLINGER BANDS EXTREMES

"The legendary bands."

One of the pillars of descriptive statistics or any basic analysis method is the concept of averages. Averages give us a glance of the next expected value given a historical trend. They can also be a representative number of a larger dataset that helps us understand the data quickly. Another pillar is the concept of volatility. Volatility is the average deviation of the values from their mean. Let us create the simple hypothetical table with different random variables.

Data
5
10
15
5
10

Let us suppose that the above is a timely ordered time series. If we had to naively guess the next value that comes after 10, then one of the best guesses can be the average the data. Therefore, if we sum up the values {5, 10, 15, 5, 10} and divide them by their quantity (i.e. 5), we get 9. This is the average of the above dataset. In python, we can generate the list or array and then calculate the average easily:

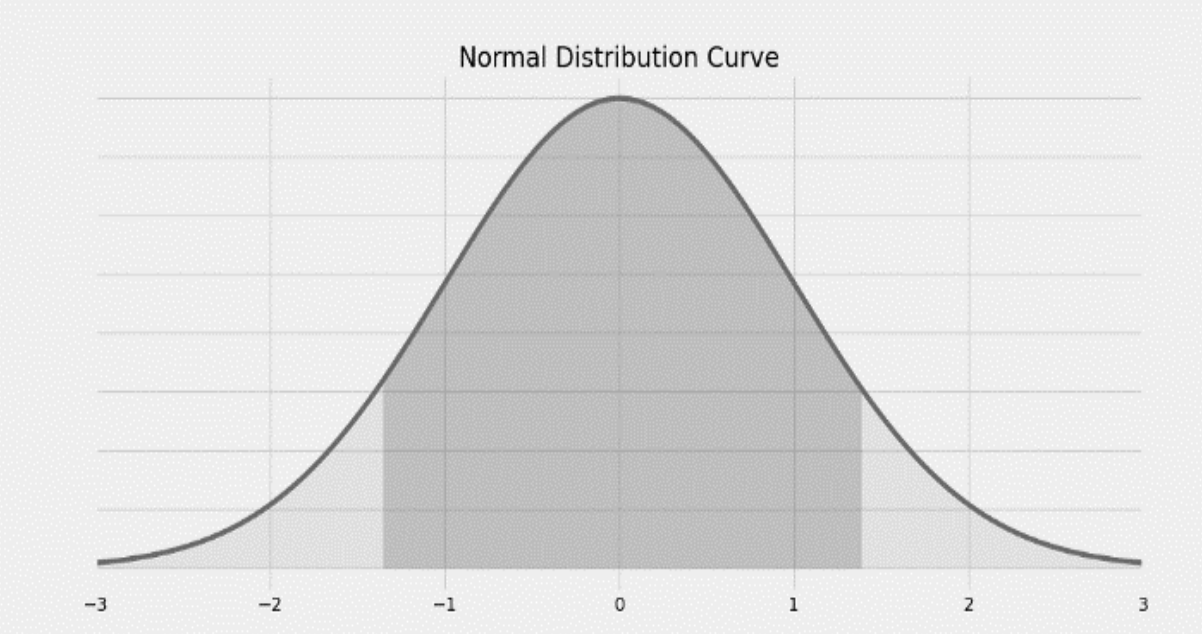
```
# Importing the required library
import numpy as np
# Creating the array
array = [5, 10, 15, 5, 10]
array = np.array(array)
# Calculating the mean
array.mean()
```

Now that we have calculated the mean value, we can see that no value in the dataset really equals 9. How do we know that the dataset is generally not close to the dataset? This is measured by what we call the Standard Deviation (Volatility).

The Standard Deviation simply measures the average distance away from the mean by looping through the individual values and comparing their distance to the mean.

```
# Calculating the mean
array.std()
```

The above code snippet calculates the Standard Deviation of the dataset which is around 3.74. This means that on average, the individual values are 3.74 units away from 9 (the mean). Now, let us move on to the normal distribution shown in the below curve.



The above curve shows the number of values within a number of standard deviations. For example, the area shaded in red represents around 1.33x of standard deviations away from the mean of zero. We know that if data is normally distributed then:

- About 68% of the data falls within 1 standard deviation of the mean.
- About 95% of the data falls within 2 standard deviations of the mean.
- About 99% of the data falls within 3 standard deviations of the mean.

Presumably, this can be used to approximate the way to use financial returns data, but studies show that financial data is not normally distributed but at the moment we can assume it is so that we can use such indicators. The weakness of the method does not hinder much its usefulness. Now, with the information below, we are ready to start creating the Bollinger Bands indicator:

- Financial time series data can have a moving average that calculates a rolling mean window. For example, a 20-period moving average calculates each time a 20-period mean that refreshes each time a new bar is formed.
- On this rolling mean window, we can calculate the Standard Deviation of the same lookback period on the moving average.

What are Bollinger Bands? When prices move, we can calculate a moving average (mean) around them so that we better understand their position regarding their mean. By doing this, we can also calculate where do they stand statistically.

Some say that the concept of volatility is the most important one in the financial markets industry. Trading the volatility bands is using some statistical properties to aid you in the decision-making process, hence, you know you are in good hands.

The idea of the Bollinger Bands is to form two barriers calculated from a constant multiplied by the rolling Standard Deviation. They are in essence barriers that give out a probability that the market price should be contained within them. The lower Bollinger Band can be considered as a dynamic support while the upper Bollinger Band can be considered as a dynamic resistance. Hence, the Bollinger bands are simple a combination of a moving average that follows prices and a moving standard deviation(s) band that moves alongside the price and the moving average.

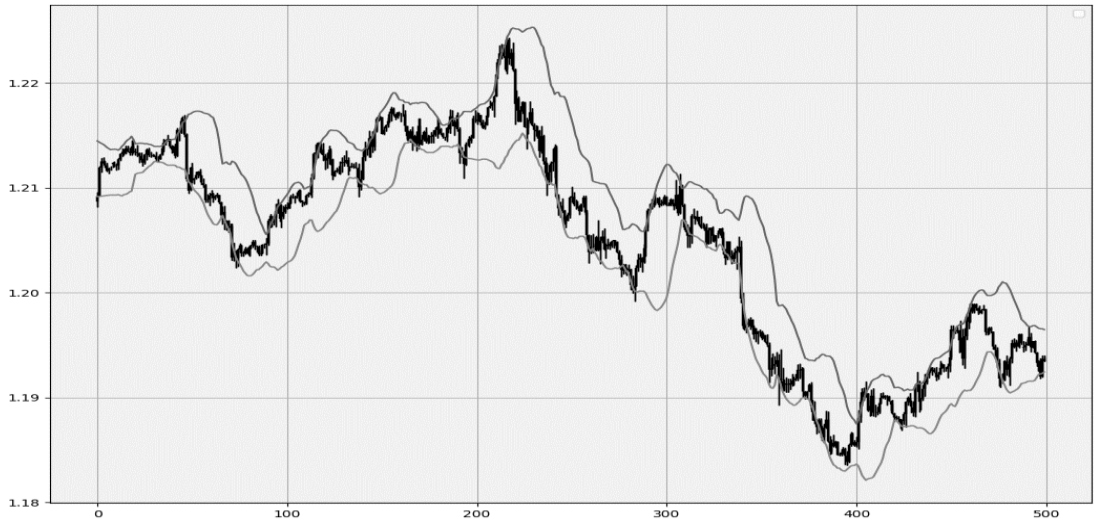
To calculate the two Bands, we use the following relatively simple formulas:

$$Upper\ Band = Moving\ Average + Constant\ \sqrt{\frac{\sum_{i=1}^{n}(y_i - Moving\ Average)^2}{n}}$$

$$Lower\ Band = Moving\ Average - Constant\ \sqrt{\frac{\sum_{i=1}^{n}(y_i - Moving\ Average)^2}{n}}$$

With the constant being the number of standard deviations that we choose to envelop prices with. By default, the indicator calculates a 20-period simple moving average and two standard deviations away from the price, then plots them together to get a better understanding of any statistical extremes. This means that on any time, we can calculate the mean and standard deviations of the last 20 observations we have and then multiply the standard deviation by the constant. Finally, we can add and subtract it from the mean to find the upper and lower band.

Clearly, the below chart seems easy to understand. Every time the price reaches one of the bands, a contrarian position is most suited, and this is evidenced by the reactions we tend to see when prices hit these extremes. So, whenever the EURUSD reaches the upper band, we can say that statistically, it should consolidate and when it reaches the lower band, we can say that statistically, it should bounce.



To create the Bollinger Bands in Python, we need to define the moving average function, the standard deviation function, and then the Bollinger Bands function which will use the former two functions. Consider an array containing OHLC data. We should define the following primal functions first and then we can code the Bollinger function:

```
# Adding a few columns
Data = adder(Data, 20)
def ma(Data, lookback, what, where):
    for i in range(len(Data)):
     try:
       Data[i, where] = (Data[i - lookback + 1:i + 1, what].mean())
         except IndexError:
             pass
    return Data
def volatility(Data, lookback, what, where):
    for i in range(len(Data)):
     try:
       Data[i, where] = (Data[i - lookback + 1:i + 1, what].std())
       except IndexError:
          pass
    return Data
```

```
def BollingerBands(Data, boll_lookback, standard_distance, what, where):
    # Calculating mean
    ma(Data, boll_lookback, what, where)
    # Calculating volatility
    volatility(Data, boll_lookback, what, where + 1)
    Data[:, where + 2] = Data[:, where] + (standard_distance *  Data[:, where + 1])
    Data[:, where + 3] = Data[:, where] - (standard_distance * Data[:, where + 1])
    return Data
# Using the function to calculate a 20-period Bollinger Band with 2 Standard Deviations
Data = BollingerBands(Data, 20, 2, 3, 4)
```

The rules for the Bollinger Strategy are simple. They revolve around selling the strength around the upper Band and buying the dips around the lower band:

- Go long (Buy) whenever the market price closes at or below the lower Bollinger Band. Hold this position until getting a new signal or getting stopped out by the risk management system.
- Go short (Sell) whenever the market price closes at or above the upper Bollinger Band. Hold this position until getting a new signal or getting stopped out by the risk management system.

```
def signal(Data, close, upper_boll, lower_boll, buy, sell):
    for i in range(len(Data)):
        if Data[i, close] < Data[i, lower_boll] and Data[i - 1, close] > Data[i - 1, lower_boll]:
            Data[i, buy] = 1
        if  Data[i, close] > Data[i, upper_boll] and Data[i - 1, close] < Data[i - 1, upper_boll]:
            Data[i, sell] = -1
```

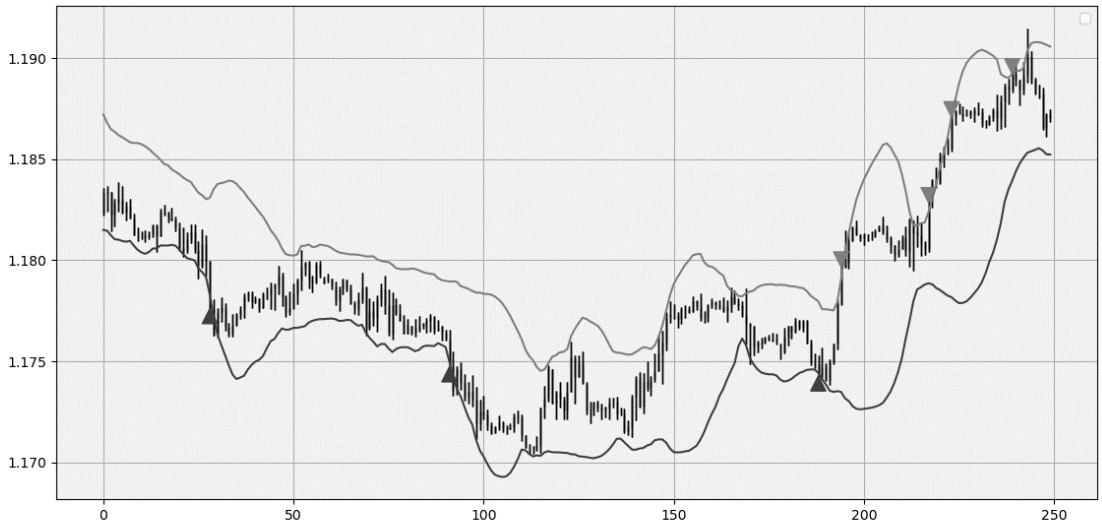
AUGMENTED BOLLINGER BANDS EXTREMES

"A modified version of the original Bollinger Bands."

The idea of an Augmented Bands came to me when I was trying to get the statistical information for as up to date as possible. It is not a big change from the original Bollinger Bands, but it uses the exponential moving average of the highs and lows then it applies the standard deviation bands using the highs and lows, respectively. Let us discuss it in a step-by-step manner:

- First, we calculate the 20-period exponential moving average of the highs. Then, we do the same for the lows.
- To find the upper band (Resistance), we will calculate the standard deviation of the exponential moving average that is applied to the highs and add a constant. In our case, it will be the number 2 which is also the original constant in the original Bollinger Bands.
- To find the lower band (Support), we will calculate the standard deviation of the exponential moving average that is applied to the lows and add the same constant.
- Finally, as an optional step, the two exponential moving averages can be averaged to form the middle line of the Volatility Bands. The middle line is usually the 20-period simple moving average that we find in the original Bollinger Bands indicator.

```
def augmented_BollingerBands(Data, boll_lookback, standard_distance, what, high, low, where):
    # Calculating means
    ema(Data, 2, boll_lookback, high, where)
    ema(Data, 2, boll_lookback, low, where + 1)
    volatility(Data, boll_lookback, high, where + 2)
    volatility(Data, boll_lookback, low, where + 3)
    Data[:, where + 4] = Data[:, where] + (standard_distance *  Data[:, where + 2])
    Data[:, where + 5] = Data[:, where + 1] - (standard_distance * Data[:, where + 3])
    return Data
```



PART 4

PATTERN RECOGNITION STRATEGIES

Pattern recognition is the search and identification of recurring patterns with approximately similar outcomes. This means that when we manage to find a pattern, we have an expected outcome that we want to see and act on through our trading. For example, a head and shoulders pattern is a classic technical configuration that signals an imminent trend reversal. The literature differs on the predictive ability of this famous pattern. In this part, we will present many price patterns that can be found in the literature or in other places. The goal is to present as many as possible so that you can find the one you are most comfortable with after having done the proper back-tests. I do not recommend you just blindly believe that after a Doji pattern, the market will reverse as the back-testing result will prove otherwise. This is primordial in understanding what to use and what not to use. Always be critical of what you are taught, otherwise, you will not be able to innovate or enhance your methods. The part is divided into candlestick patterns, Tom Demark's patterns, and other personal timing patterns I have found to be interesting within a trading framework.

INTRODUCTION TO CANDLESTICKS PATTERNS

Candlestick charts are among the most famous ways to analyze the time series visually. They contain more information than a simple line chart and have more visual interpretability than bar charts. Many libraries in Python offer charting functions but being someone who suffers from malfunctioning import of libraries and functions alongside their fogginess, I have created my own simple function that charts candlesticks manually with no exogenous help needed. The details on how to chart your own candlesticks in Python are found in Appendix I.

OHLC data is an abbreviation for Open, High, Low, and Close price. They are the four main ingredients for a timestamp. It is always better to have these four values together so that our analysis reflects more the reality. Here is a table that summarizes the OHLC data of hypothetical security:

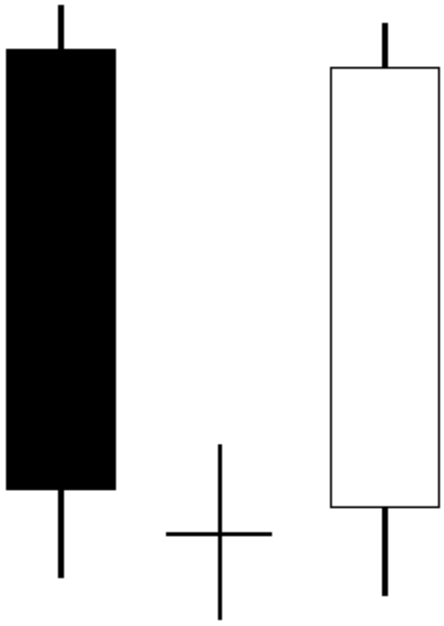
Time Stamp	Open	High	Low	Close
01/01/2021	1.4294	1.4297	1.4272	1.4285
02/01/2021	1.4285	1.4299	1.4282	1.4290
03/01/2021	1.4289	1.4308	1.4287	1.4300
04/01/2021	1.4300	1.4309	1.4280	1.4295
05/01/2021	1.4295	1.4307	1.4269	1.4303
06/01/2021	1.4303	1.4338	1.4300	1.4337
07/01/2021	1.4337	1.4395	1.4336	1.4395

THE DOJI PATTERN

"A Neutral pattern that signals indecision."

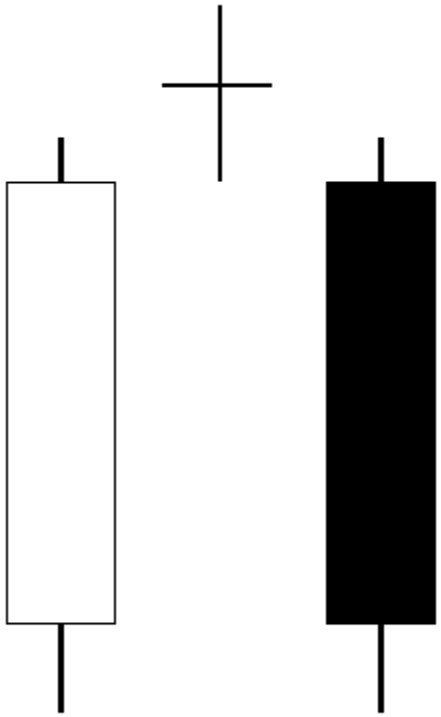
Candlestick patterns deserve to be studied thoroughly and even though a strategy relying solely on them will be unstable and unprofitable, they can be a valuable addition into a full trading system that uses other techniques. The Doji pattern is known as the neutral or indecisive pattern. This is because it occurs when the opening price is the same as the closing price, meaning that the market has not reached a consensus yet on the direction.

The Bullish Doji pattern is composed of a candle that has its closing price equal to its opening price. It usually occurs after downward trending price action and is considered a bullish reversal or a correction pattern.



The Bullish Doji pattern is based on the psychology that the balance of power has been equalized after trending in one direction. Theoretically, we are supposed to buy at the close of the second and last candle to validate the pattern. For a bullish Doji pattern to be valid, we need a bullish candle after it.

The Bearish Doji pattern is composed of a candle that has its closing price equal to its opening price.



The Bearish Doji pattern is based on the psychology that the balance of power has been equalized after trending in one direction. Theoretically, we are supposed to sell at the close of the second and last candle to validate the pattern.

- To find a potential short-term bottom and a long opportunity:
- The current opening price must be equal to the current closing price.

The current price action is greater than the last 2 closing prices. This should confirm a bullish previous trend.

To find a potential short-term top and a short-sell opportunity:

- The current opening price must be equal to the current closing price.
- The current price action is lower than the last 2 closing prices. This should confirm a bearish previous trend.

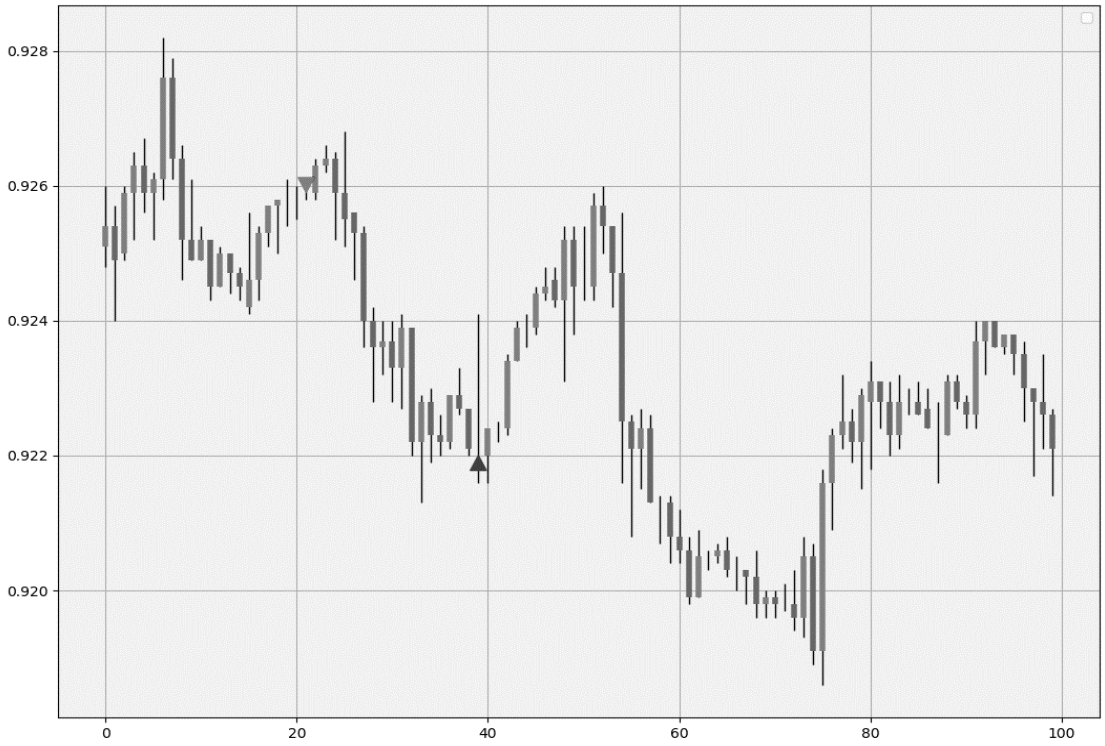
One thing before we start writing the scanning (signal) function is that FX pairs are now quoted with 5 decimals. Doji patterns will be almost impossible to occur if we use 5 decimals, therefore, we need to round the time series to 4 decimals.

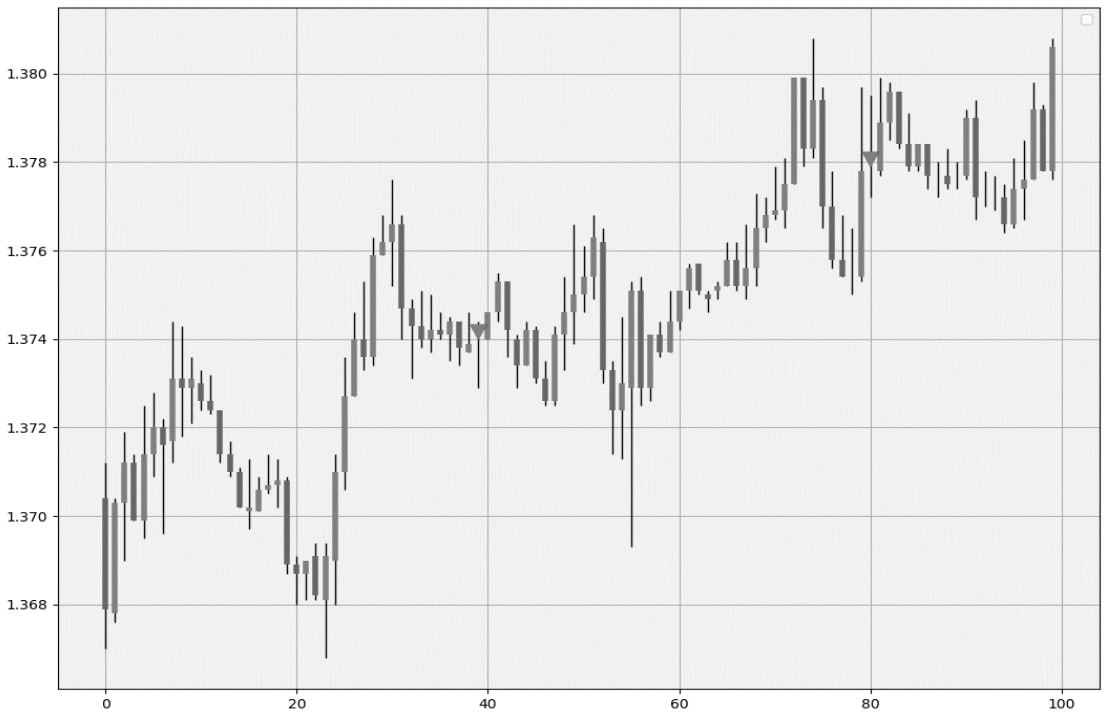
```
# Code to add before the signal function
def rounding(Data, how_far):
  Data = Data.round(decimals = how_far)
  return Data
my_data = rounding(my_data, 4)
```

The below function takes an OHLC data array with multiple empty columns to spare and populates columns 6 (Buy) and 7 (Sell) with the conditions that we discussed earlier.

```
def signal(Data):
    for i in range(len(Data)):
    # Bullish Doji
    if Data[i, 3] == Data[i, 0] and Data[i, 3] < Data[i - 1, 3]    and Data[i, 3] < Data[i - 2, 3]:
        Data[i, 6] = 1
    # Bearish Doji
    if Data[i, 3] == Data[i, 0] and Data[i, 3] > Data[i - 1, 3] and Data[i, 3] > Data[i - 2, 3]:
        Data[i, 7] = -1
```

We want to input 1's in the column we call "buy" and -1 in the column we call "sell". This later allows you to create a function that calculates the profit and loss by looping around these two columns and taking differences in the market price to find the profit and loss of a close-to-close strategy. Then you can use a risk management function that uses stops and profit orders. The next charts show an example of the signals generated using the Doji function above. The downward pointing grey arrows refer to short sell signals while the slightly darker upward pointing arrows refer to bullish signals. Remember, the code can be found in the GitHub link presented earlier in the book.

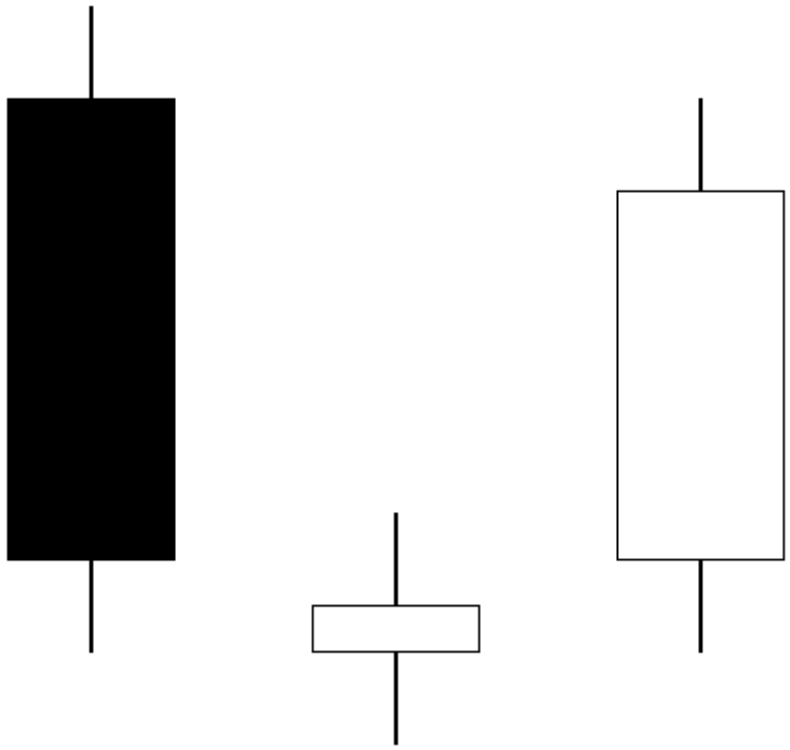

0.928
0.926
0.924
0.922
0.920
0
20
40
60
80
100


1.380
1.378
1.376
1.374
1.372
1.370
1.368
0
20
40
60
80
100

THE MORNING STAR & EVENING STAR PATTERNS

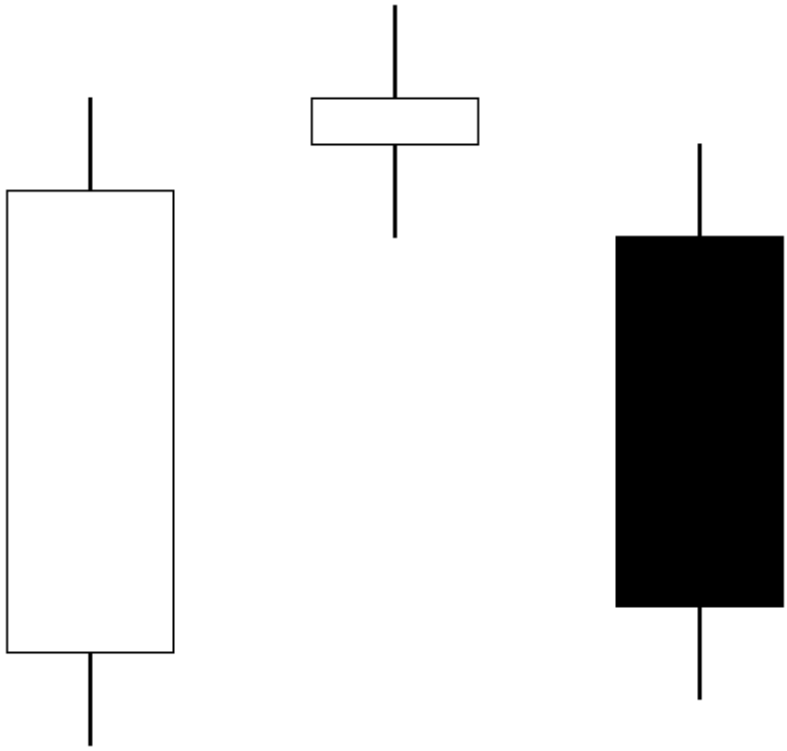
"An intuitive reversal structure."

The Morning Star pattern is composed of three candles where the first is a relatively big bearish candle, the second is a small body candle, and the third is a relatively big bullish candle.



The Morning Star pattern is a reversal pattern based on the psychology that the sentiment has drastically shifted from bearish to bullish. Theoretically, we are supposed to buy at the close of the third candle.

The Evening Star pattern is composed of three candles where the first is a relatively big bullish candle, the second is a small body candle, and the third is a relatively big bearish candle.



The Evening Star pattern is a reversal pattern based on the psychology that the sentiment has drastically shifted from bullish to bearish. Theoretically, we are supposed to sell at the close of the third candle.

To find a potential short-term bottom and a long opportunity:

- The first candle must be bearish with a big body.
- The second candle must be a candle with a small body. Preferably, a Doji candle.
- The third candle must be bullish with a big body.

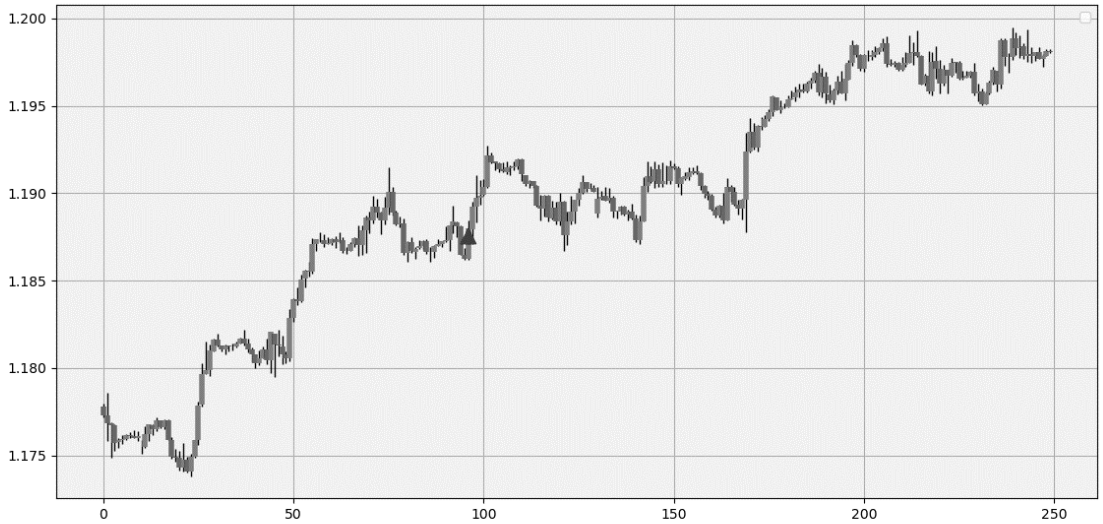
To find a potential short-term top and a short-sell opportunity:

- The first candle must be bullish with a big body.
- The second candle must be a candle with a small body. Preferably, a Doji candle.
- The third candle must be bearish with a big body.

Ideally, there should be a gap between the small bodied-candle and the other candles.

```
# Defining the minimum width of the first and third candles

side_body = 0.0010

# Defining the maximum width of the second candle

middle_body = 0.0003

# Signal function

def signal(Data):

   for i in range(len(Data)):

      # Morning Star

      if Data[i - 2, 3] < Data[i - 2, 0] and (Data[i - 2, 0] - Data[i - 2, 3]) > side_body and
(abs(Data[i - 1, 3] - Data[i - 1, 0])) <= middle_body and Data[i, 3] > Data[i, 0] and (Data[i, 3]
- Data[i, 0]) > side_body:

            Data[i, 6] = 1

      # Evening Star

      if Data[i - 2, 3] > Data[i - 2, 0] and (Data[i - 2, 3] - Data[i - 2, 0]) > side_body and
(abs(Data[i - 1, 3] - Data[i - 1, 0])) <= middle_body and Data[i, 3] < Data[i, 0] and (Data[i, 3]
- Data[i, 0]) > side_body:

            Data[i, 7] = -1
```

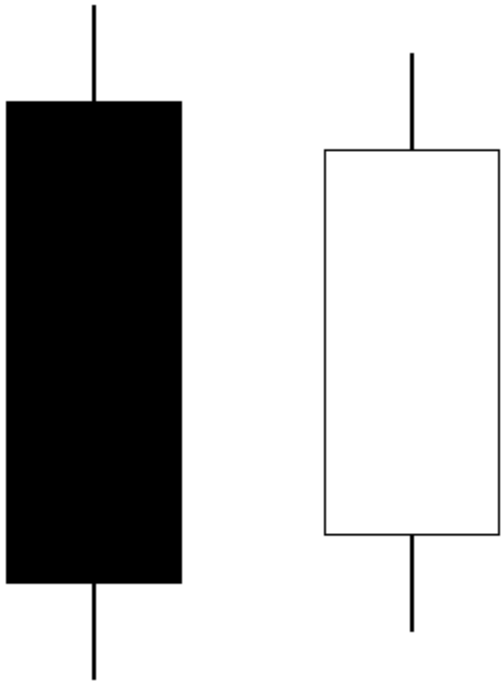
In the example above, we have chosen a minimum with of 10 pips for the big body candles which can be adjusted for time frame or the trader's preferences. Also, a maximal length of the middle candle can be chosen. In the function above, an arbitrary 3 pips have been chosen.



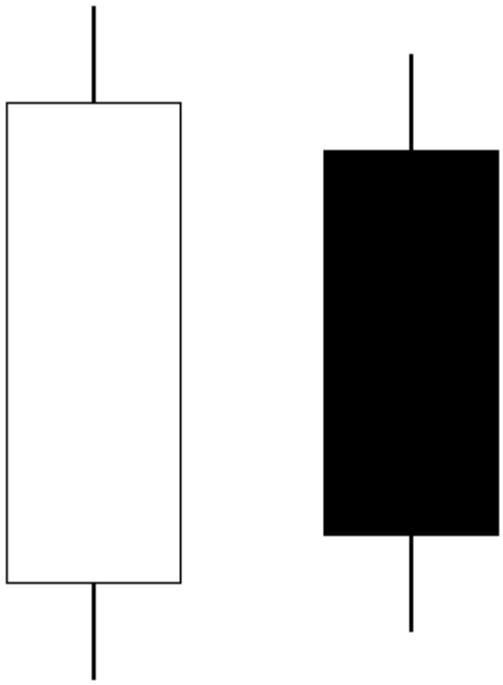
THE HARAMI PATTERN

"The pattern means pregnant in Japanese."

The Bullish Harami pattern is composed of two candles with the first (Bearish) fully englobing the second. Meaning that the high of the first candle is greater than the high of the second candle and the low of the first candle is also lower than the low of the second candle. Similarly, the opening price of the first candle is greater than the closing price of the second candle and the closing price of the first candle is lower than the opening price of the second.



The Bullish Harami is an upside continuation or reversal pattern depending on the prevalent trend. The word Harami means pregnant in Japanese which is what the pattern looks like. The Bearish Harami pattern is composed of two candles with the first (Bullish) fully englobing the second. Meaning that the high of the first candle is greater than the high of the second candle and the low of the first candle is also lower than the low of the second candle. Similarly, the closing price of the first candle is greater than the opening price of the second candle and the opening price of the first candle is lower than the closing price of the second candle.



The Bearish Harami is a downside continuation or reversal pattern depending on the prevalent trend. Our aim is to create an algorithm that detects this pattern and places theoretical buy and sell orders so that we back-test the strategy and optimize it if possible. But first, we need to code the intuition of the patterns. Let us review what we will need for the bullish Harami:

- The first high must be higher than the second high.
- The first opening price must be higher than the second closing price.
- The first closing price must be lower than the second opening price.
- The first low must be lower than the second low.

Similarly, for the bearish Harami, we need the following conditions:

- The first high must be higher than the second high.
- The first closing price must be higher than the second opening price.
- The first opening price must be lower than the second closing price.
- The first low must be lower than the second low.


```
def signal(Data):
   for i in range(len(Data)):
      # Bullish Harami
      if Data[i, 1] < Data[i - 1, 1] and Data[i, 2] > Data[i - 1, 2] and Data[i, 3] < Data[i - 1, 0] and
Data[i, 0] > Data[i - 1, 3] and Data[i, 3] > Data[i, 0] and Data[i - 1, 3] < Data[i - 1, 0]:
            Data[i, 6] = 1
      # Bearish Harami
      if Data[i, 1] < Data[i - 1, 1] and Data[i, 2] > Data[i - 1, 2] and Data[i, 3] > Data[i - 1, 0] and
Data[i, 0] < Data[i - 1, 3] and Data[i, 3] < Data[i, 0] and Data[i - 1, 3] > Data[i - 1, 0]:
            Data[i, 7] = -1
```

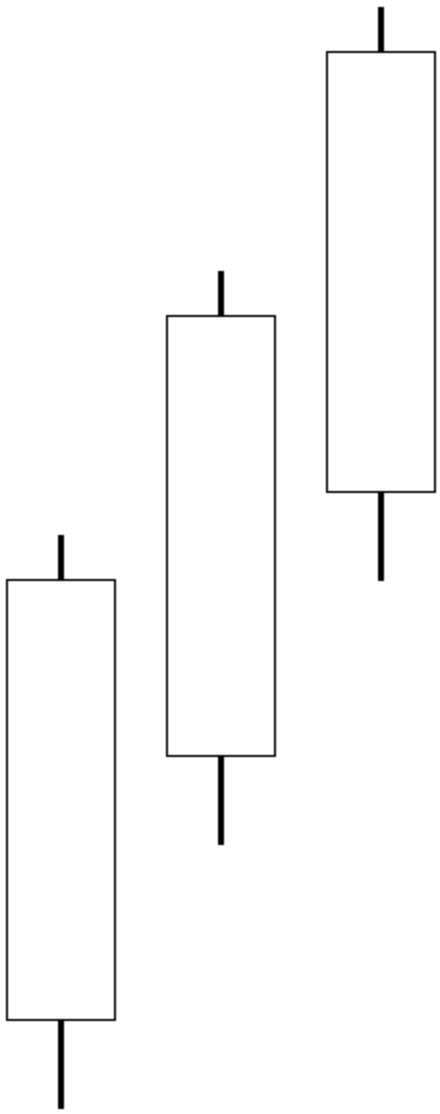
The above function takes an OHLC data array with multiple empty columns to spare and populates columns 6 (Buy) and 7 (Sell) with the conditions that we discussed earlier.

We want to input 1's in the column we call "buy" and -1 in the column we call "sell". This later allows you to create a function that calculates the profit and loss by looping around these two columns and taking differences in the market price to find the profit and loss of a close-to-close strategy. Then you can use a risk management function that uses stops and profit orders.

THE THREE WHITE SOLDIERS & BLACK CROWS PATTERNS

"A candle-shaped Momentum indicator."

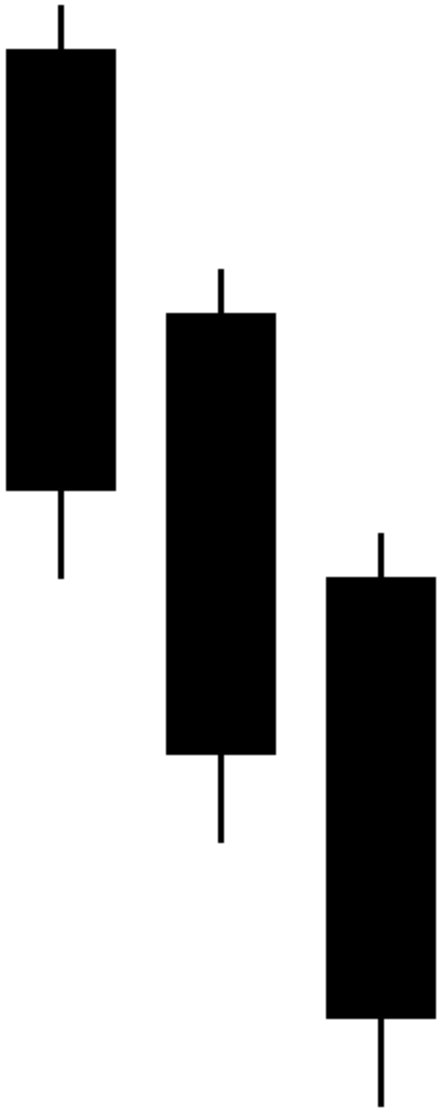
The three white soldiers pattern is a pattern composed of three consecutive bullish candles that typically have their openings within the prior session's body (and not tails) but this is optional and will be left out as in the currencies market, gaps a less common than in other markets.



The three white soldiers pattern is a bullish continuation based on the psychology that the momentum is healthy and strong which signifies the buyers are still in control and

that the market is likely to continue higher. Theoretically, we are supposed to buy at the close of the third and last candle to validate the pattern.

The three black crows pattern is a pattern composed of three consecutive bearish candles that typically have their openings within the prior session's body (and not tails) but this is optional and will be left out as in the currencies market, gaps a less common than in other markets.



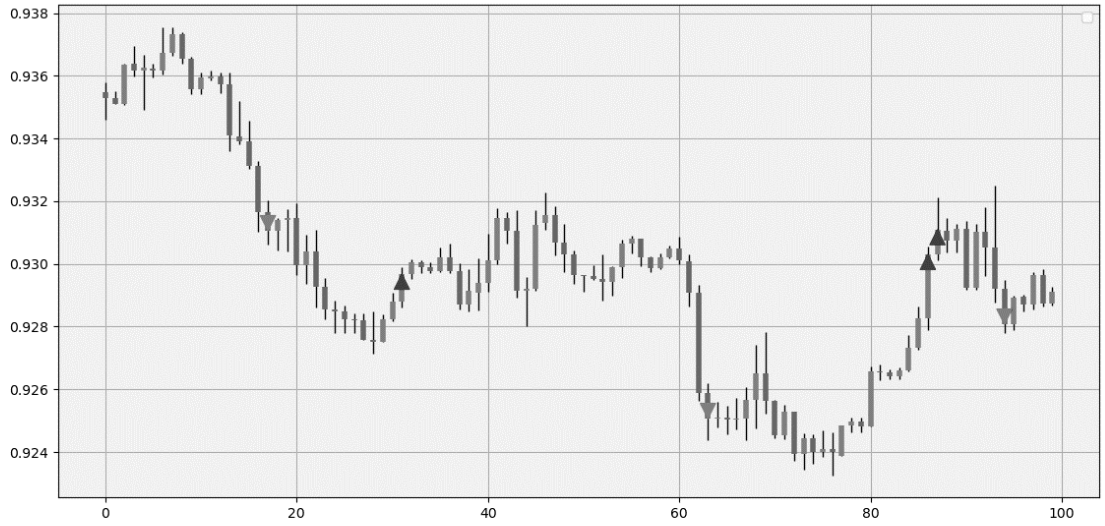
The three black crows pattern is a bearish continuation based on the psychology that the momentum is healthy and strong which signifies the sellers are still in control and

that the market is likely to continue lower. Theoretically, we are supposed to sell short at the close of the third and last candle to validate the pattern. Let us review what we will need for the bullish pattern:

- The three candles must be green (Bullish). Therefore, for each candle, we have to check whether the closing price is greater than the opening price.
- Each candle must close higher than the previous candle. Therefore, for each candle, we must check whether the closing price is greater than the previous closing price.
- Finally, the body of the candle must be big enough to be qualified. Sometimes, we can have a bullish candle, but it can be too small to be valid in the pattern. This is done by creating a variable called body where it is the subtraction of the closing price from the opening price. The bigger the body, the less common are the signals but the more the pattern resembles a true pattern. We will choose a body of 5 pips on hourly data.

Similarly, for the bearish three black crows, we need the following conditions:

- The three candles must be red (Bearish). Therefore, for each candle, we have to check whether the closing price is lower than the opening price.
- Each candle must close lower than the previous candle. Therefore, for each candle, we must check whether the closing price is lower than the previous closing price.
- Finally, the body of the candle must be big enough to be qualified. Sometimes, we can have a bearish candle, but it can be too small to be valid in the pattern. This is done by creating a variable called body where it is the subtraction of the closing price from the opening price. The bigger the body, the less common are the signals but the more the pattern resembles a true pattern. We will choose a body of 5 pips on hourly data.



 # Defining the minimum width of the candle

body = 0.0005

def signal(Data, body):

for i in range(len(Data)):

 # Three White Soldiers

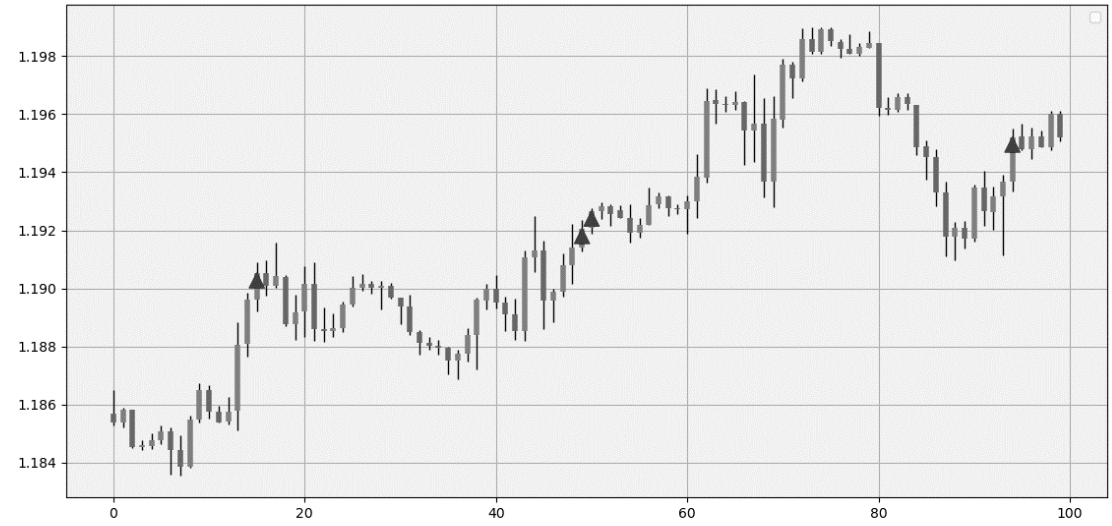
if Data[i, 3] > Data[i, 0] and (Data[i, 3] - Data[i, 0]) >= body and Data[i, 3] > Data[i - 1, 3] and Data[i - 1, 3] > Data[i - 1, 0] and (Data[i - 1, 3] - Data[i - 1, 0]) >= body and Data[i - 1, 3] > Data[i - 2, 3] and Data[i - 2, 3] > Data[i - 2, 0] and (Data[i - 2, 3] - Data[i - 2, 0]) >= body and Data[i - 2, 3] > Data[i - 3, 3]:

Data[i, 6] = 1

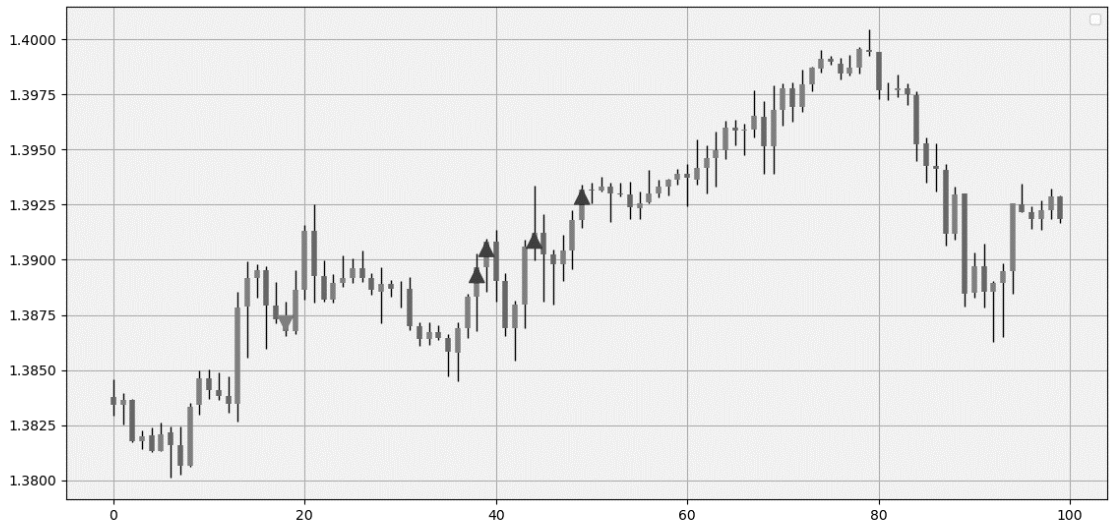
 # Three Black Crows

if Data[i, 3] < Data[i, 0] and (Data[i, 0] - Data[i, 3]) >= body and Data[i, 3] < Data[i - 1, 3] and Data[i - 1, 3] < Data[i - 1, 0] and (Data[i - 1, 0] - Data[i - 1, 3]) >= body and Data[i - 1, 3] < Data[i - 2, 3] and Data[i - 2, 3] < Data[i - 2, 0] and (Data[i - 2, 0] - Data[i - 2, 3]) >= body and Data[i - 2, 3] < Data[i - 3, 3]:

Data[i, 7] = -1



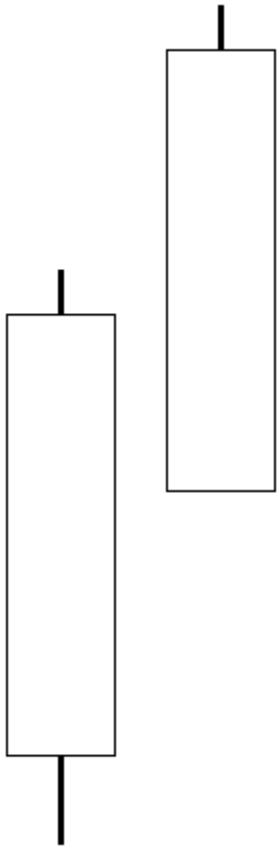
The previous function takes an OHLC data array with multiple empty columns to spare and populates columns 6 (Buy) and 7 (Sell) with the conditions that we discussed earlier.



THE BOTTLE PATTERN

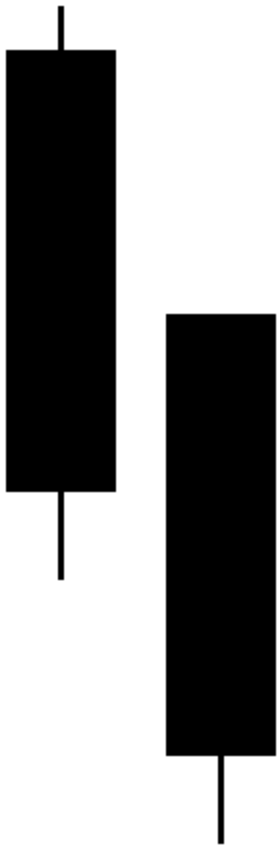
"Just another pattern worth-testing."

The Bullish Bottle pattern is composed of a candle with no wick on the low side but with a wick on the high side. The first candle's color can be bullish or bearish.



The Bullish Bottle is an upside continuation pattern. It can be used to confirm upside continuations. The psychology of the pattern is that the market has failed to make a new low after its opening, suggesting strong bullish pressure.

The Bearish Bottle pattern is composed of a candle with no wick on the high side but with a wick on the low side. The first candle's color can be bullish or bearish.



The Bearish Bottle is a downside continuation pattern. It can be used to confirm downside continuations. The psychology of the pattern is that the market has failed to make a new high after its opening, suggesting strong bearish pressure.

What is the difference between an Inverted Hammer and the Bottle pattern? Similarly, what is the difference between a Hanging Man pattern and the Inverted Bottle pattern?

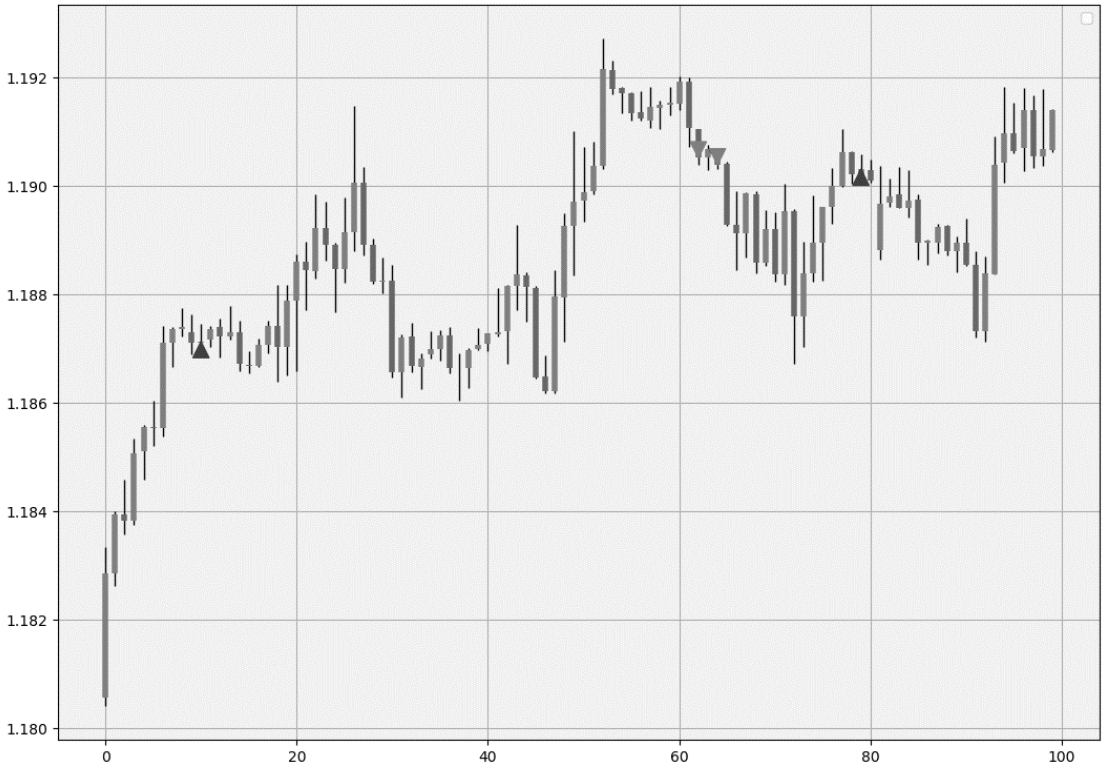
The Bottle is a continuation pattern where we should expect further strength after the Bottle and further weakness after the Inverted Bottle. This can seem confusing but as we have however, the aim is to create a more objective Technical Analysis backed by more intuition, research, and results. Therefore, I believe that by using a form of risk management and a proper back-test we can challenge some classical patterns and back-test our own intuitions and see if they work or not.

This is in no way a fool proof technique, but simply an attempt to do-it-yourself that can either yield a good or bad strategy. Therefore, I highly encourage you to do the same and if you find other result.

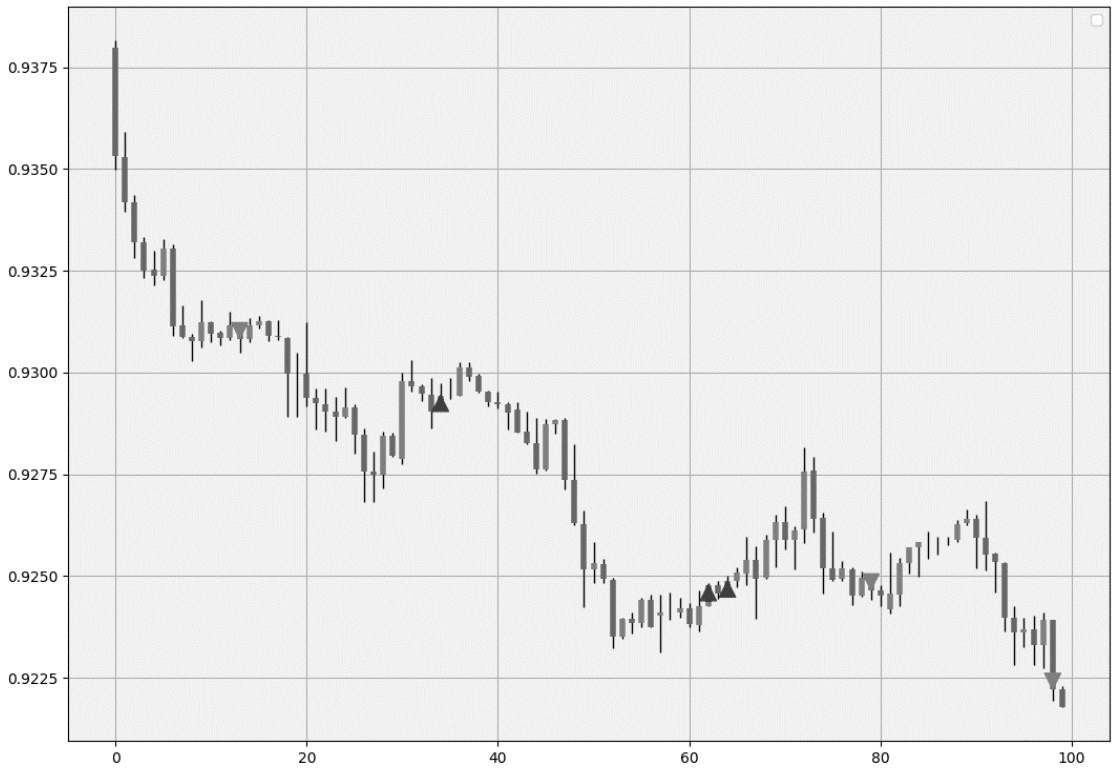
- Let us review what we will need for the bullish Bottle pattern:
- The closing price must be higher than the opening price.

The opening price must equal the low price. Similarly, for the bearish Bottle pattern (Inverted Bottle), we need the following conditions:

- The closing price must be lower than the opening price.
- The opening price must equal the high price.



```
def signal(Data):
    for i in range(len(Data)):
        # Bullish Bottle
        if Data[i, 3] > Data[i, 0] and Data[i, 0] == Data[i, 2]:
            Data[i, 6] = 1
        # Bearish Bottle
        if Data[i, 3] < Data[i, 0] and Data[i, 0] == Data[i, 1]:
            Data[i, 7] = -1
```



THE MARUBOZU PATTERN

"No wicks, just a straightforward candlestick."

The Bullish Marubozu pattern is composed of a candle without wicks. Meaning that its close is greater than its opening price while the high equals the close and the low equals the opening price.

The Bullish Marubozu is a neutral or trend-continuation pattern based on the psychology that the current trend is strong enough to continue. The market has failed to make a lower low than the opening price and it closed at its highest point as a result of significant one-sided strength.

The Bearish Marubozu pattern is composed of a candle without wicks. Meaning that its close is lower than its opening price while the high equals the opening price and the low equals the close.

The Bearish Marubozu is a neutral or trend-continuation pattern based on the psychology that the current trend is strong enough to continue. The market has failed to make a higher high than the opening price and it closed at its lowest point as a result

of significant one-sided strength. Let us review what we will need for the bullish Marubozu:

- The close must be greater than the opening price.
- The high must equal the closing price.
- The low must equal the opening price.

Similarly, for the bearish Marubozu, we need the following conditions:

- The close must be lower than the opening price.
- The high must equal the closing price.
- The low must equal the opening price.

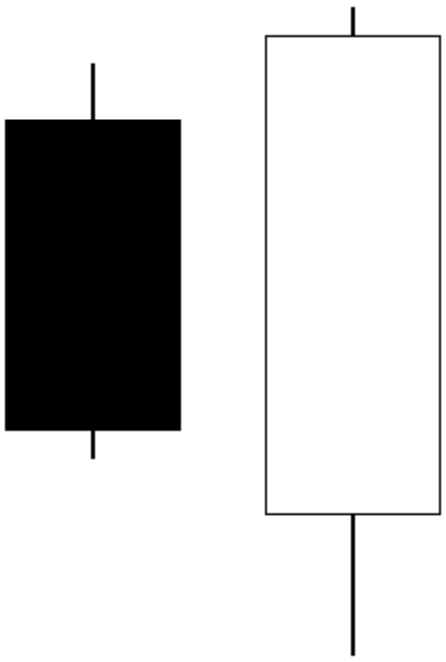

```
def signal(Data, body):
   for i in range(len(Data)):
      # Bullish Marubozu
      if Data[i, 3] > Data[i, 0] and Data[i, 1] == Data[i, 3] and Data[i, 2] == Data[i, 0]:
            Data[i, 6] = 1
      # Bearish Marubozu
      if Data[i, 3] < Data[i, 0] and Data[i, 1] == Data[i, 0] and Data[i, 2] == Data[i, 3]:
            Data[i, 7] = -1
```


1.1910
1.1909
1.1908
1.1907
1.1906
1.1905
1.1904
1.1903
1.1902
0
20
40
60
80
100

THE ENGULFING PATTERN

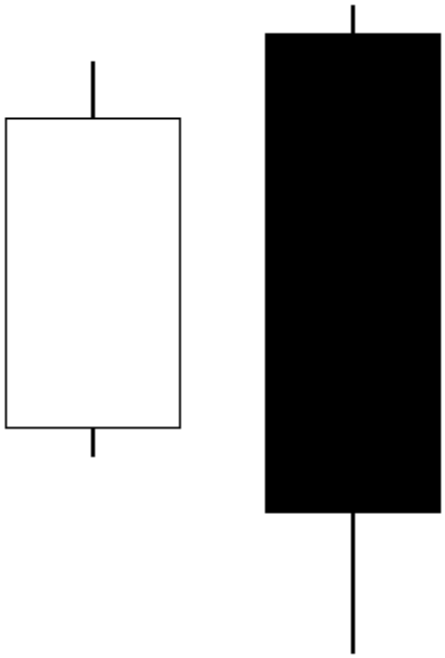
"When the big boys take over."

The Bullish Engulfing pattern is composed of two candles where the first is a bearish relatively small candle and the second is a bullish candle that fully englobes the previous one. Meaning that it has a higher high and close as well as a lower opening and low price.



The Bullish Engulfing is a reversal pattern based on the psychology that the sentiment has drastically shifted from bearish to bullish. Theoretically, we are supposed to buy at the close of the second and last candle to validate the pattern.

The Bearish Engulfing pattern is composed of two candles where the first is a bullish relatively small candle and the second is a bearish candle that fully englobes the previous one. Meaning that it has a higher high and open as well as a lower close and low price.



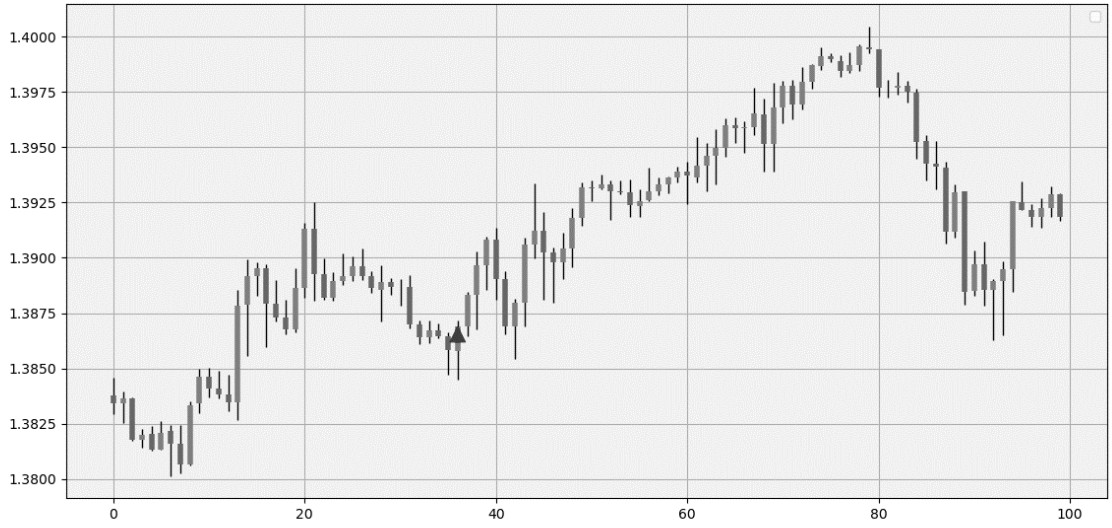
The Bearish Engulfing is a reversal pattern based on the psychology that the sentiment has drastically shifted from bullish to bearish. Theoretically, we are supposed to sell at the close of the second and last candle to validate the pattern.

Let us review what we will need for the bullish Engulfing:

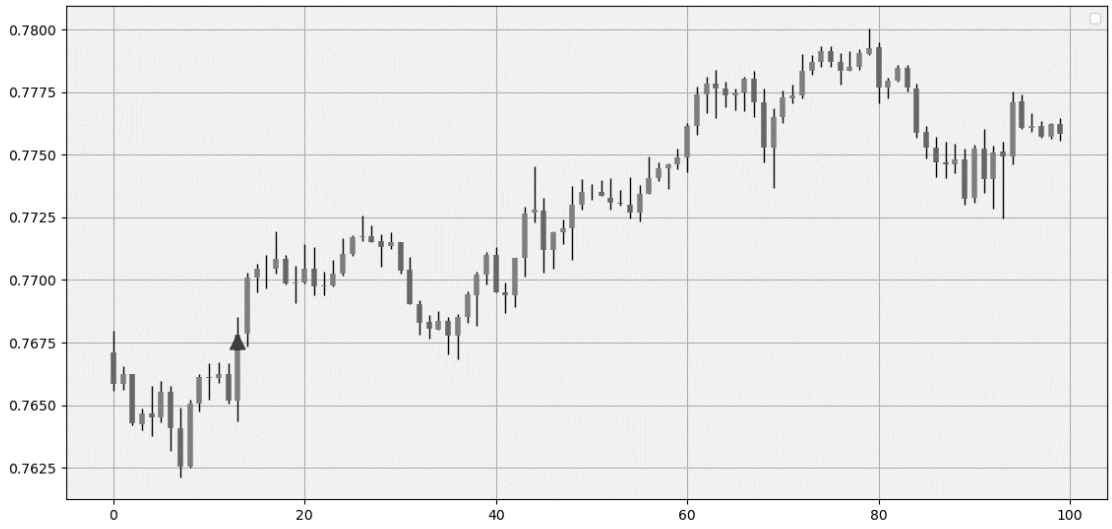
- The first candle must be bearish, and the second candle must be bullish.
- The high of the second candle must be higher than the high of the first candle and the low of the second candle must be lower than the low of the first candle.
- The close of the second candle must be higher than the opening of the first candle and the opening of the second candle must be lower than the close of the first candle.
- Finally, the body of the candle must be big enough to be qualified. Sometimes, we can have a bullish candle, but it can be too small to be valid in the pattern. This is done by creating a variable called body where it is the subtraction of the closing price from the opening price. The bigger the body, the less common are the signals but the more the pattern resembles a true pattern. We will choose a body of 5 pips on hourly data.

Similarly, for the bearish Engulfing, we need the following conditions:

- The first candle must be bullish, and the second candle must be bearish.
- The high of the second candle must be higher than the high of the first candle and the low of the second candle must be lower than the low of the first candle.
- The close of the second candle must be lower than the opening of the first candle and the opening of the second candle must be higher than the close of the first candle.
- Finally, the body of the candle must be big enough to be qualified. Sometimes, we can have a bearish candle, but it can be too small to be valid in the pattern. This is done by creating a variable called body where it is the subtraction of the closing price from the opening price. The bigger the body, the less common are the signals but the more the pattern resembles a true pattern. We will choose a body of 5 pips on hourly data.

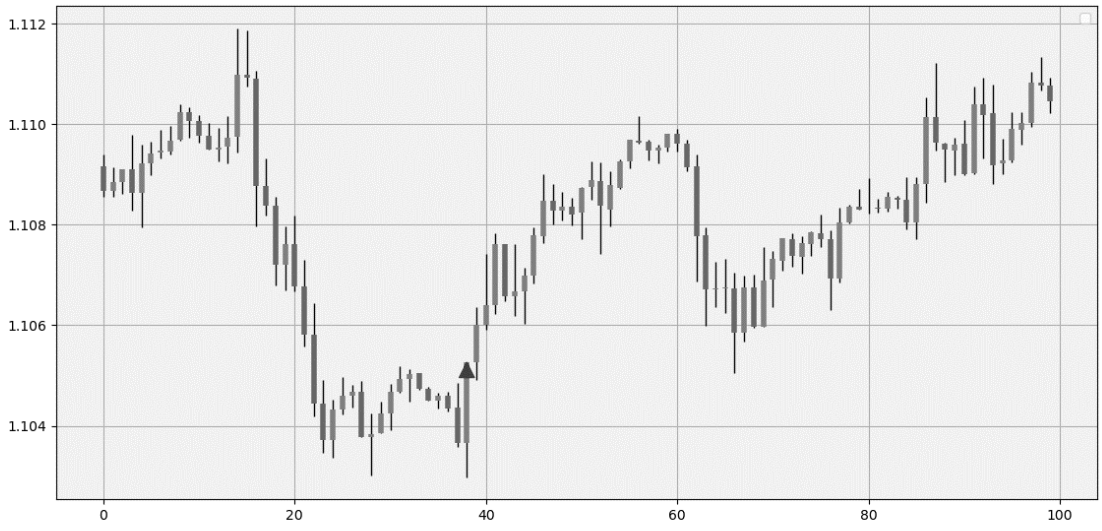


```
# Defining the minimum width of the candle

body = 0.0005

def signal(Data, body):

   for i in range(len(Data)):

      # Bullish Engulfing

      if Data[i, 3] > Data[i, 0] and Data[i - 1, 3] < Data[i - 1, 0] and (Data[i, 3] - Data[i, 0]) >= body
and \

         Data[i, 1] > Data[i - 1, 1] and Data[i, 2] < Data[i - 1, 2] and Data[i, 3] > Data[i - 1, 0] and \

            Data[i, 0] < Data[i - 1, 3]:

             Data[i, 6] = 1

      # Bearish Engulfing

      if Data[i, 3] < Data[i, 0] and Data[i - 1, 3] > Data[i - 1, 0] and (Data[i, 3] - Data[i, 0]) >= body
and \

         Data[i, 1] < Data[i - 1, 1] and Data[i, 2] > Data[i - 1, 2] and Data[i, 3] < Data[i - 1, 0] and \

            Data[i, 0] > Data[i - 1, 3] :

            Data[i, 7] = -1
```



The above function takes an OHLC data array with multiple empty columns to spare and populates columns 6 (Buy) and 7 (Sell) with the conditions that we discussed earlier.

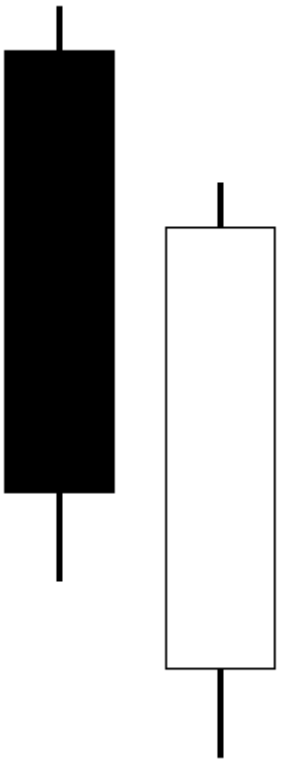
We want to input 1's in the column we call "buy" and -1 in the column we call "sell". This later allows you to create a function that calculates the profit and loss by looping around these two columns and taking differences in the market price to find the profit and loss of a close-to-close strategy. Then you can use a risk management function that uses stops and profit orders.



THE PIERCING & DARK CLOUD PATTERNS

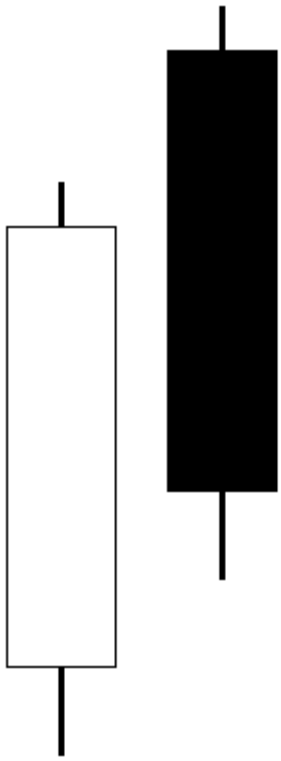
"A pattern that may suggest a shift in sentiment."

The Bullish Piercing pattern is composed of two candles with the second candle opening below the first candle's close but closing around its body, giving the image of piercing it. This pattern has a gap embedded into it and it is the opening price of the second candle relative to the closing price of the first candle.



The Bullish Piercing is an upside reversal pattern after a downside move. It can be used to confirm reactions or corrections in a bearish trend.

The Bearish Piercing pattern is composed of two candles with the second candle closing below the first candle's close but opening above its closing price, giving the image of piercing it. This pattern is also referred to as a Dark Cloud.

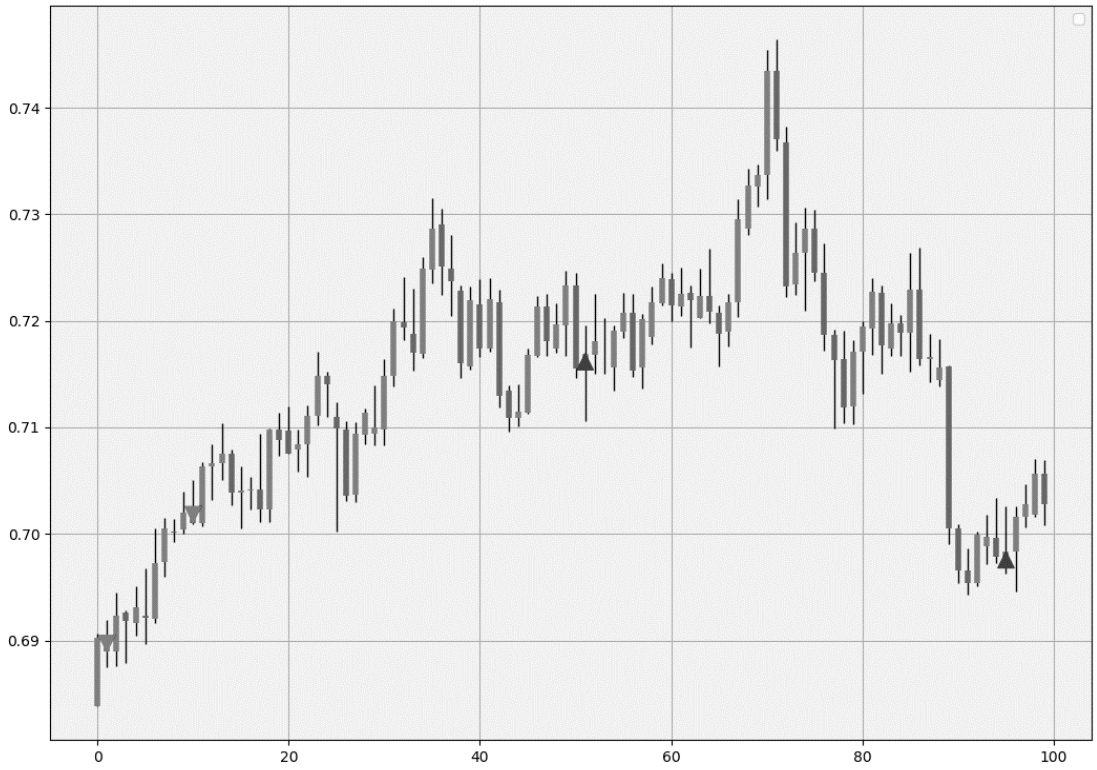


The Bearish Piercing is a downside reversal pattern after an upside move. It can be used to confirm reactions or corrections in a bullish trend. Our aim is to create an algorithm that detects this pattern and places theoretical buy and sell orders so that we back-test the strategy and optimize it if possible. But first, we need to code the intuition of the patterns. Let us review what we will need for the bullish Piercing pattern:

- The first closing price must be higher than the second opening price.
- The first opening price must be higher than the second closing price.
- The first closing price must be lower than the second closing price.
- The first high must be higher than the second high.
- The first low must be higher than the second low.
- The first candle must be bearish, and the second candle must be bullish.

Similarly, for the bearish piercing pattern (Dark Cloud), we need the following conditions:

- The first closing price must be lower than the second opening price.
- The first opening price must be lower than the second closing price.
- The first closing price must be higher than the second closing price.
- The first high must be lower than the second high.
- The first low must be lower than the second low.
- The first candle must be bullish, and the second candle must be bearish.



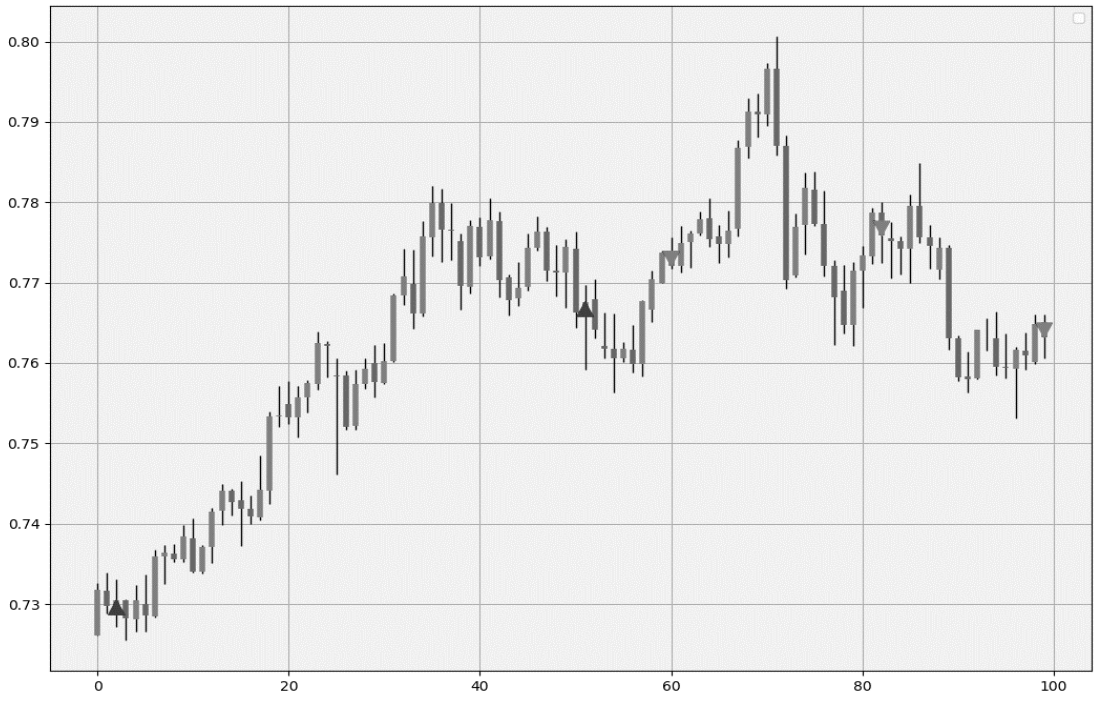
The body variable is the candle's width. Sometimes, the algorithm can confuse a Doji and a piercing, and the variable aims to fix that problem. Note: Candlestick patterns that have gaps embedded in them are more common in either Daily horizons or very short-term horizons (as opposed to hourly and 3-hour charts).

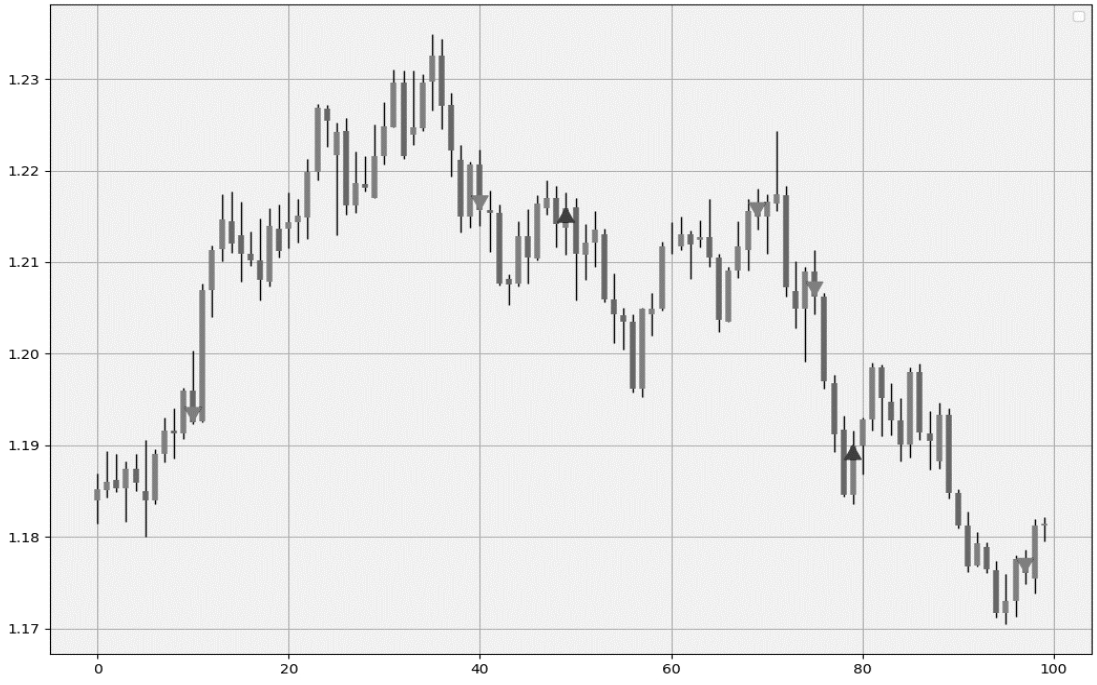
```
body = 0.0005
def signal(Data):
  for i in range(len(Data)):
    # Bullish Piercing
    if Data[i - 1, 3] > Data[i, 0] and \
      Data[i - 1, 0] > Data[i, 3] and \
      Data[i - 1, 3] < Data[i, 3] and \
      Data[i - 1, 1] > Data[i, 1] and \
      Data[i - 1, 2] > Data[i, 2] and \
      Data[i - 1, 3] < Data[i - 1, 0] and \
      Data[i, 3] > Data[i, 0] and \
      Data[i, 3] - Data[i, 0] > body:
        Data[i, 6] = 1
    # Bearish Piercing
    if Data[i - 1, 3] < Data[i, 0] and \
      Data[i - 1, 0] < Data[i, 3] and \
      Data[i - 1, 3] > Data[i, 3] and \
      Data[i - 1, 1] < Data[i, 1] and \
      Data[i - 1, 2] < Data[i, 2] and \
      Data[i - 1, 3] > Data[i - 1, 0] and \
      Data[i, 3] < Data[i, 0] and \
      Data[i, 0] - Data[i, 3] > body:
        Data[i, 7] = -1
```

The above function takes an OHLC data array with multiple empty columns to spare and populates columns 6 (Buy) and 7 (Sell) with the conditions that we discussed earlier.

We want to input 1's in the column we call "buy" and -1 in the column we call "sell". This later allows you to create a function that calculates the profit and loss by looping around these two columns and taking differences in the market price to find the profit

and loss of a close-to-close strategy. Then you can use a risk management function that uses stops and profit orders.

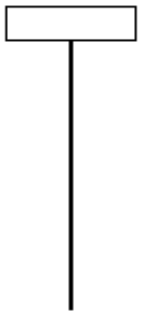




THE HAMMER & SHOOTING STAR PATTERNS

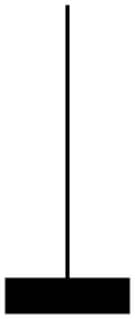
"It looks like a hammer."

The Hammer pattern is a bullish candlestick configuration that resembles the letter T. The basic intuition is that the bearish pressure has made a new low but at the end could not keep it as the market has closed closer to the high of the day.



The specificity for the Hammer pattern is a long low wick and a high that is equal to either the open or the high. Hence, in this study, the Hammer can be a bullish or a bearish candle but the forecast of the price action after it is bullish.

The specificity for the Hammer pattern is a long low wick and a high that is equal to either the open or the high. Hence, in this study, the Hammer can be a bullish or a bearish candle but the forecast of the price action after it is bullish.



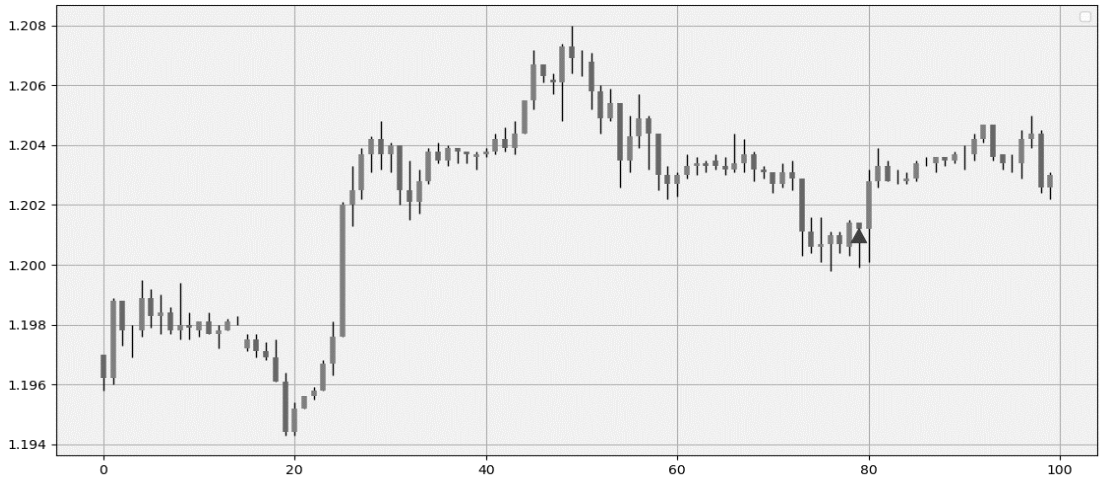
The specificity of the Shooting Star pattern is a long high wick and a low that is equal to either the open or the high. Hence, in this study, the Shooting Star can be a bullish or a bearish candle but the forecast of the price action after it is bearish.

Let us review what we will need for the Hammer:

- The low must be at least twice the length of the body.
- The high must be equal to the closing price or the opening price.

Similarly, for the Shooting Star, we need the following conditions:

- The high must be at least twice the length of the body.
- The low must be equal to the closing price or the opening price.

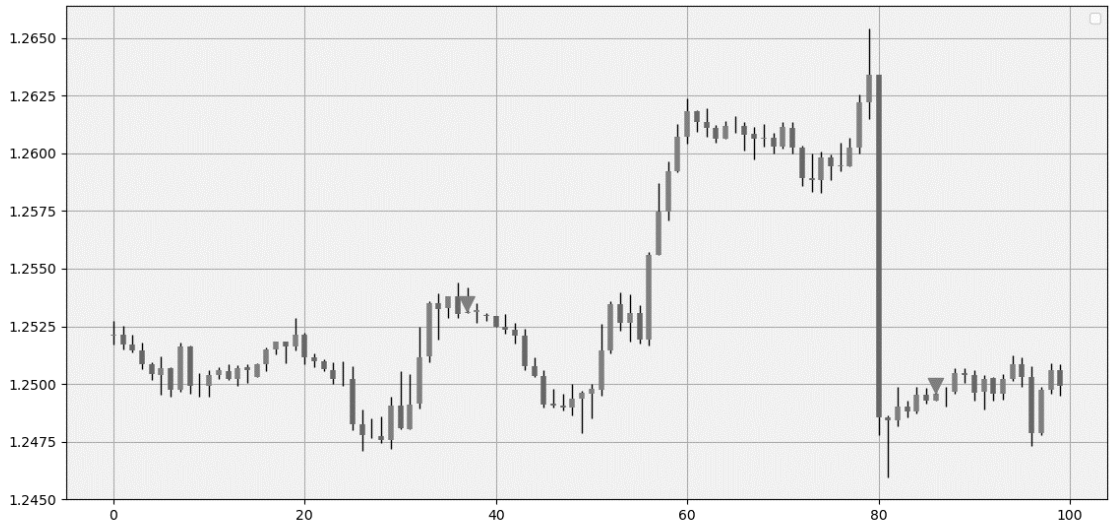



```
# Defining the minimum width of the candle
body = 0.0004
wick = body * 2
def signal(Data):
  for i in range(len(Data)):
    # Hammer
    if abs(Data[i, 3] - Data[i, 0]) < body and abs(Data[i, 2] - Data[i, 0]) > wick and Data[i, 1] == Data[i, 0]:
        Data[i, 6] = 1
    # Hammer
    if abs(Data[i, 3] - Data[i, 0]) < body and abs(Data[i, 2] - Data[i, 0]) > wick and Data[i, 1] == Data[i, 3]:
        Data[i, 6] = 1
```

```
    # Shooting Star
    if abs(Data[i, 3] - Data[i, 0]) < body and abs(Data[i, 1] - Data[i, 0]) > wick and Data[i, 2] ==
Data[i, 0]:
        Data[i, 7] = -1
    # Shooting Star
    if abs(Data[i, 3] - Data[i, 0]) < body and abs(Data[i, 1] - Data[i, 0]) > wick and Data[i, 2] ==
Data[i, 3]:
        Data[i, 7] = -1
```

The above function takes an OHLC data array with multiple empty columns to spare and populates columns 6 (Buy) and 7 (Sell) with the conditions that we discussed earlier.

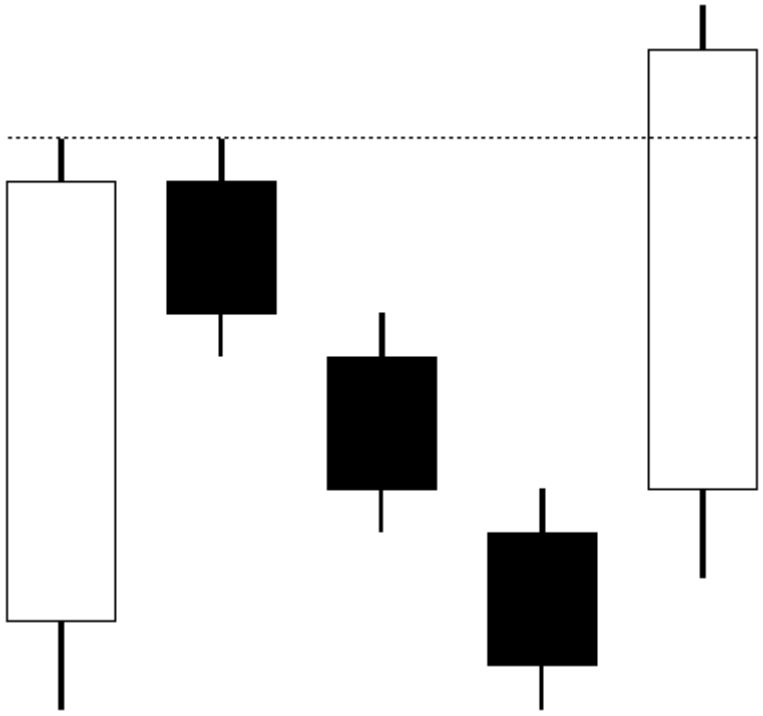
We want to input 1's in the column we call "buy" and -1 in the column we call "sell". This later allows you to create a function that calculates the profit and loss by looping around these two columns and taking differences in the market price to find the profit and loss of a close-to-close strategy. Then you can use a risk management function that uses stops and profit orders.



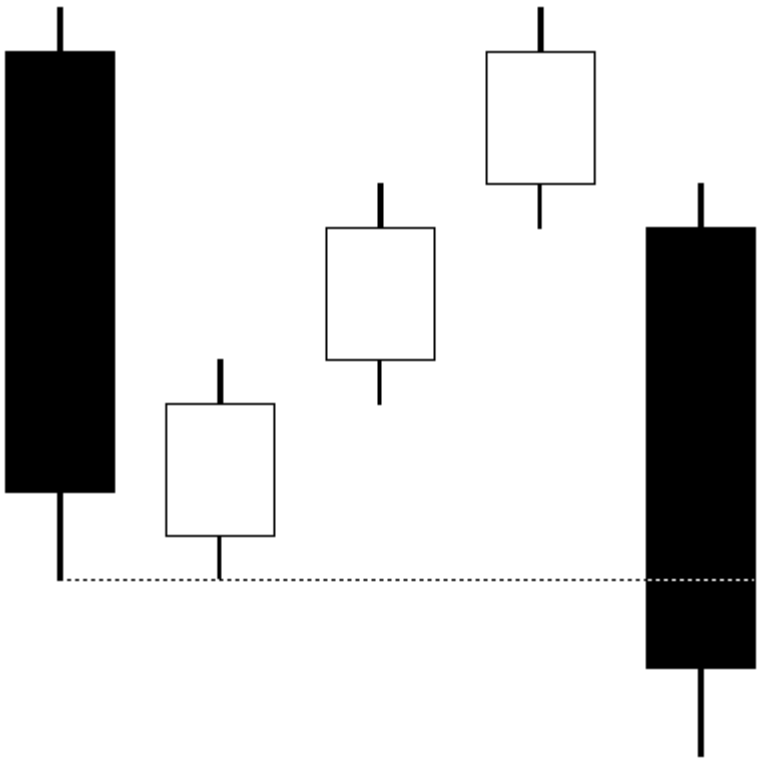
THE THREE METHODS PATTERN

"The pause within the trend."

The Rising Three Methods pattern is a complex pattern mainly composed of five candles where the first one is bullish generally within an uptrend, followed by three corrective bearish candles that can be considered as a profit-taking time period. The last step involves a big bullish candle that surpasses the high of the previous step. Psychologically speaking, after the bulls took some profits, they have managed to push through the highs they have made and seem to be in a healthy phase to continue higher.



Theoretically, we are supposed to buy at the close of the last candle to validate the pattern. The Falling Three Methods pattern is a complex pattern mainly composed of five candles where the first one is bearish generally within a downtrend, followed by three corrective bullish candles that can be considered as a profit-taking time period. The last step involves a big bearish candle that breaks the low of the previous step. Psychologically speaking, after the bears took some profits, they have managed to push through the lows they have made and seem to be in a healthy phase to continue lower.

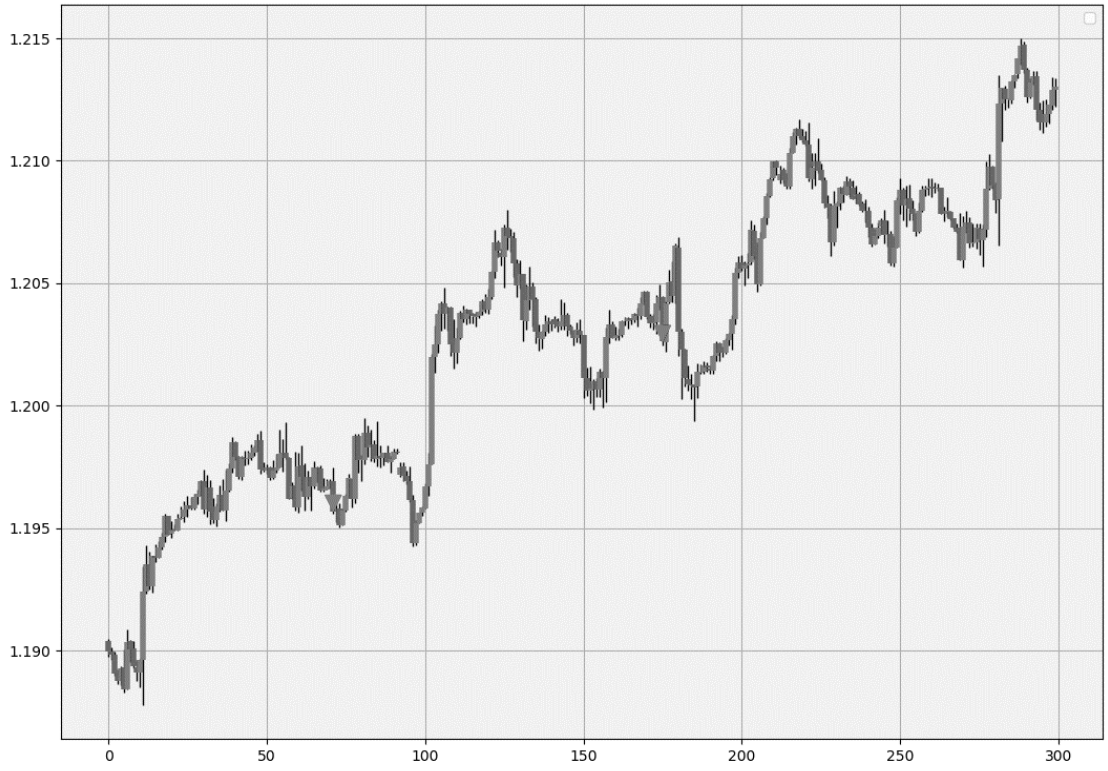


Theoretically, we are supposed to go short at the close of the last candle to validate the pattern. Let us review what we will need for the Rising Three Methods:

- The first candle must be bullish.
- The next three candles must be bearish and each lower than the previous one.
- The last candle must be a big bullish candle that surpasses the high of the corrective phase.

Similarly, for the Falling Three Methods, we need the following conditions:

- The first candle must be bearish.
- The next three candles must be bullish and each higher than the previous one.
- The last candle must be a big bearish candle that breaks the low of the corrective phase.



 # Defining the minimum width of the candle

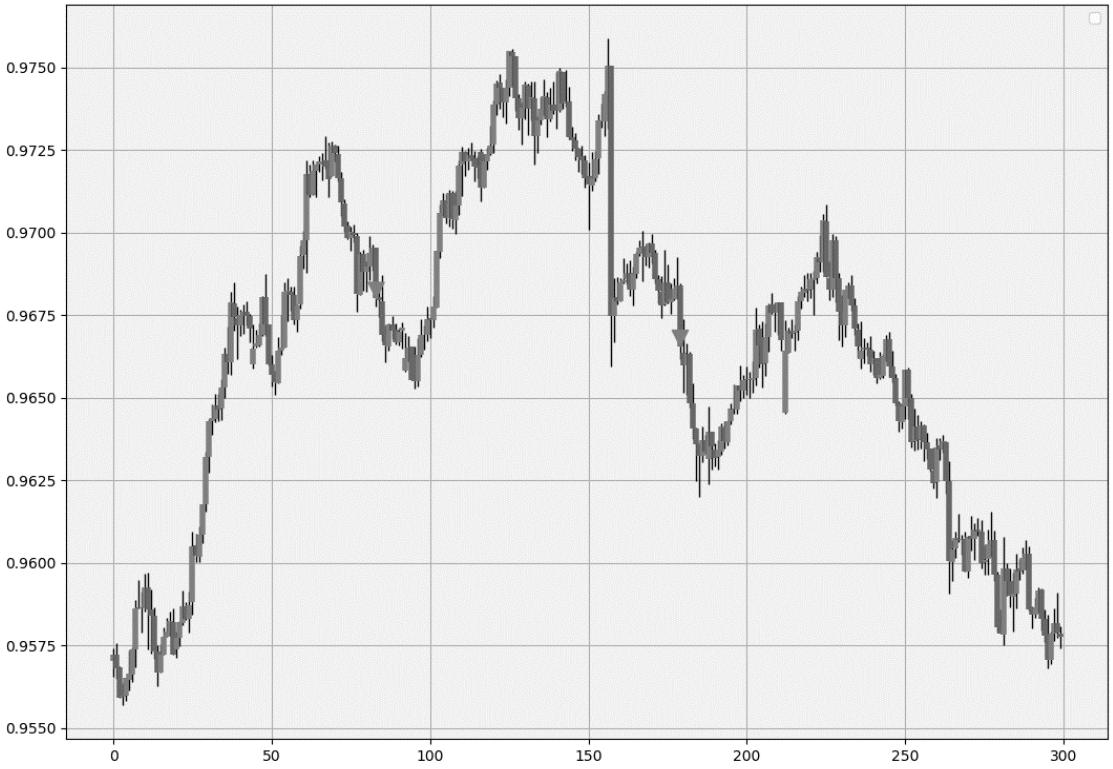
body = 0.0005

 # Defining the window of trend

window = 5

The above are variables that can be adjusted. The body refers to the length of the candle while the window refers to the lookback period.

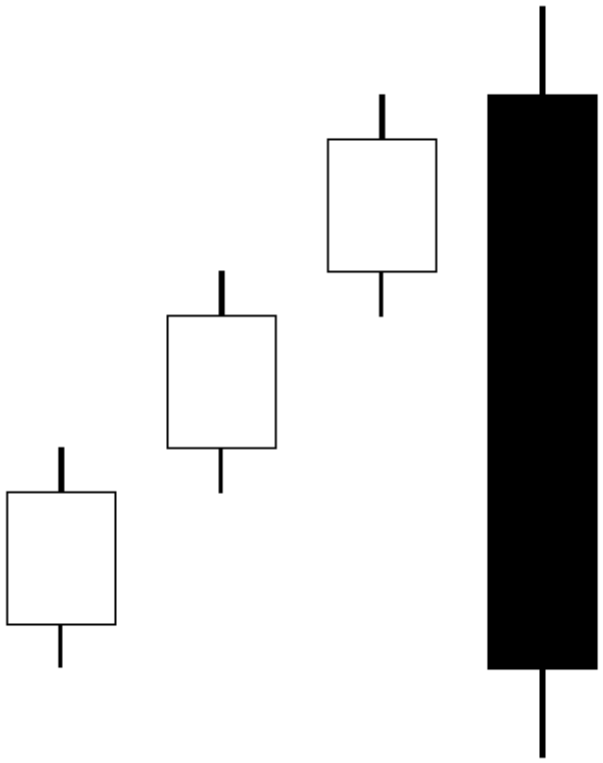
```
def signal(Data):
    for i in range(len(Data)):
        # Falling Three Methods
        if Data[i, 3] < Data[i, 0] and Data[i, 3] < Data[i - 3, 2] and (Data[i, 0] - Data[i, 3]) > body and \
            Data[i - 1, 3] > Data[i - 1, 0] and Data[i - 1, 3] > Data[i - 2, 3] and \
            Data[i - 2, 3] > Data[i - 2, 0] and Data[i - 2, 3] > Data[i - 3, 3] and \
            Data[i - 1, 3] > Data[i - 3, 0] and Data[i - 3, 3] > Data[i - 4, 3] and \
            Data[i - 3, 3] < Data[i - window, 3]:
                Data[i, 7] = -1
        # Rising Three Methods
        if Data[i, 3] > Data[i, 0] and Data[i, 3] > Data[i - 3, 2] and (Data[i, 0] - Data[i, 3]) > body and \
            Data[i - 1, 3] < Data[i - 1, 0] and Data[i - 1, 3] < Data[i - 2, 3] and \
            Data[i - 2, 3] < Data[i - 2, 0] and Data[i - 2, 3] < Data[i - 3, 3] and \
            Data[i - 1, 3] < Data[i - 3, 0] and Data[i - 3, 3] < Data[i - 4, 3] and \
            Data[i - 3, 3] > Data[i - window, 3]:
                Data[i, 6] = 1
```



THE THREE LINE STRIKE PATTERN

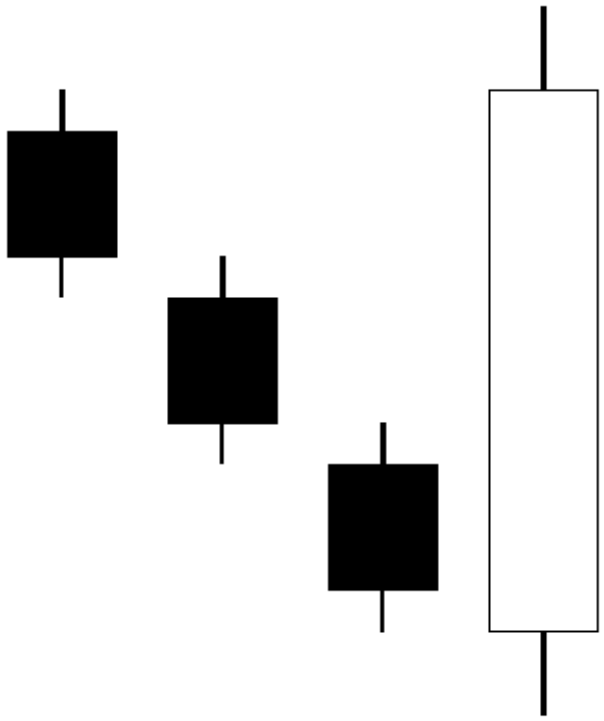
"That pattern that resembles the previous one."

The Bullish Three Line Strike pattern is composed of four candles where the first three are rising and the last one is a big bearish candle that englobes the previous three.



The Bullish Three Line Strike is a continuation pattern based on the psychology that the market has taken profit and is now around good buying levels to continue the trend upwards.

The Bearish Three Line Strike pattern is composed of four candles where the first three are going down and the last one is a big bullish candle that englobes the previous three.

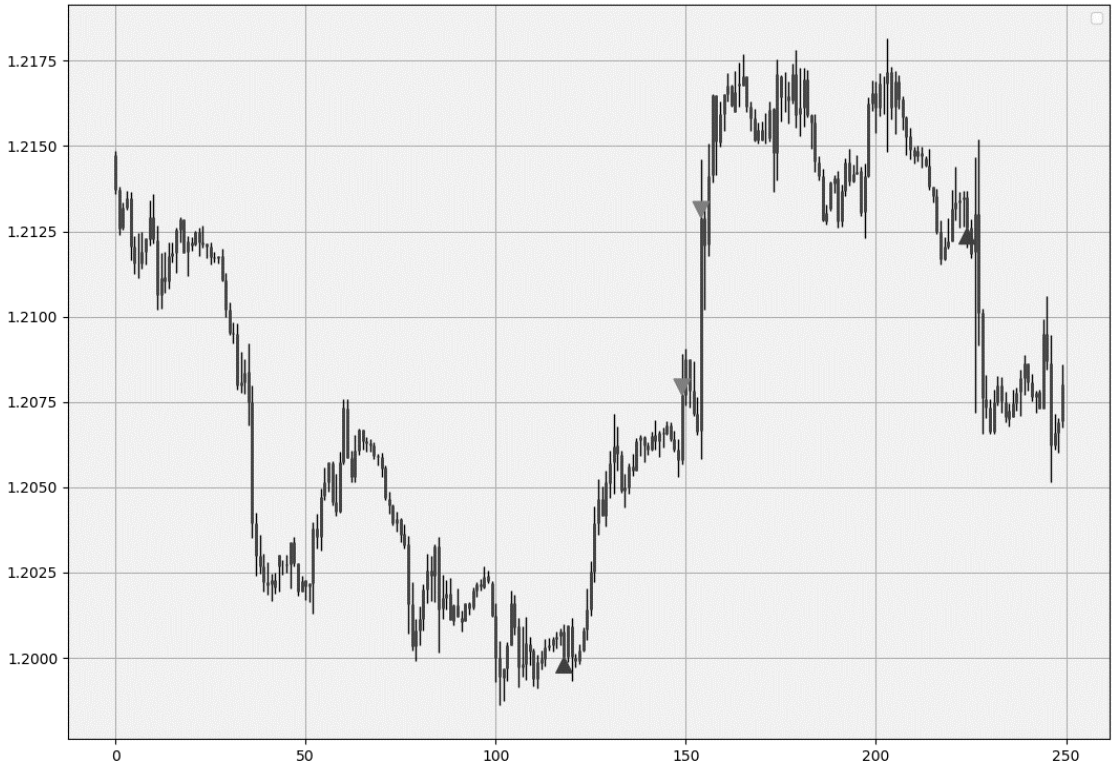


The Bearish Three Line Strike is a continuation pattern based on the psychology that the market has taken profit and is now around good selling levels to continue the trend downwards in case of short-selling pressure. Let us review what we will need for the bullish Three Line Strike:

- The first candle must be bullish.
- The second candle must be bullish and greater than the previous candle.
- The third candle must be bullish and greater than the previous candle.
- The fourth candle must be bearish, and its close must be lower than the close of the first candle.

Similarly, for the bearish Three Line Strike, we need the following conditions:

- The first candle must be bearish.
- The second candle must be bearish and less than the previous candle.
- The third candle must be bearish and less than the previous candle.
- The fourth candle must be bullish, and its close must be greater than the high of the first candle.




```
def signal(Data):
    Data = adder(Data, 10)
    Data = rounding(Data, 5)
    for i in range(len(Data)):
        if Data[i - 3, 3] > Data[i - 3, 0] and Data[i - 2, 3] > Data[i - 3, 3]  and Data[i - 2, 3] > Data[i - 3, 3] and Data[i - 1, 3] > Data[i - 2, 3] and Data[i, 3] < Data[i - 3, 2] and Data[i, 1] > Data[i - 1, 3]:
            Data[i, 6] = 1
        if Data[i - 3, 3] < Data[i - 3, 0] and Data[i - 2, 3] < Data[i - 3, 3]  and Data[i - 2, 3] < Data[i - 3, 3] and Data[i - 1, 3] < Data[i - 2, 3] and Data[i, 3] > Data[i - 3, 1] and Data[i, 2] < Data[i - 1, 3]:
            Data[i, 7] = -1
    return Data
```

TD WALDO #2 PATTERN

"Where's Waldo?"

Tom Demark, the renowned famous technical analyst has created many indicators and patterns such as the Demarker discussed earlier in the book. Among his many discoveries, he came up with 7 different price patterns that he would dub as "Waldo patterns".

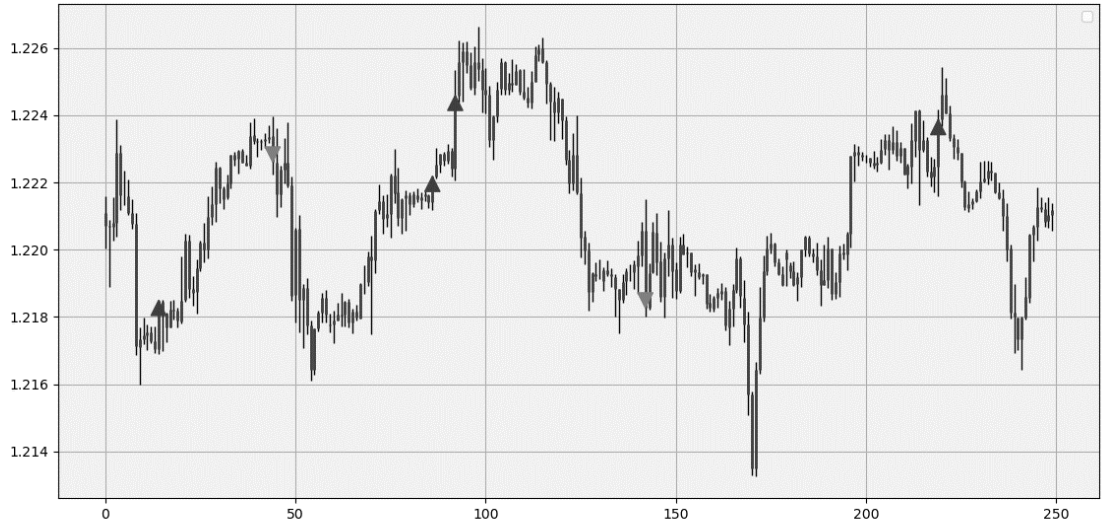
Inspired by the cartoon Where's Waldo? Tom has developed these patterns to find short-term tops and bottoms. Any trader or even a receiver of such information would be excited knowing about this, but with hindsight and confirmation bias, the person seeing this, will test the patterns on the chart and focus on the ones that work while disregarding the ones that do not. The only way to remedy this problem is by developing a systematic trading strategy based on these patterns, and that is precisely what we are going to do.

The TD Waldo patterns are a set of simple rules that give out trading signals. The below are the details for the Waldo #2 pattern. However, for simplification issues, the conditions have been relaxed by yours truly as some conditions seem to be the result of overfitting considering the results and the specificities.

To find a potential short-term bottom and a long opportunity:

- A new low is recorded but the close of the bar is higher than the four previous closes.
- To find a potential short-term top and a short-sell opportunity:
- A new high is recorded but the close of the bar is lower than the four previous closes.

We can code the signal function that gives us the necessary trading triggers as follows:



Assuming we have an OHLC array, we can define and use the scanner below that will look for the pattern and input 1 in case of a bullish Waldo #2 or input -1 in case of a bearish Waldo #2.

```
def td_waldo_2(Data, high, low, close, buy, sell):
    # Adding a few columns
    Data = adder(Data, 10)
    for i in range(len(Data)):
        # Short-term Bottom
        if Data[i, 2] < Data[i - 1, 2] and Data[i, 2] < Data[i - 2, 2] and Data[i, 2] < Data[i - 3, 2] and Data[i, 2] < Data[i - 4, 2] and Data[i, 3] > Data[i - 1, 3] and Data[i, 3] > Data[i - 2, 3] and Data[i, 3] > Data[i - 3, 3] and Data[i, 3] > Data[i - 4, 3]:
            Data[i, buy] = 1
        # Short-term Top
        if Data[i, 1] > Data[i - 1, 1] and Data[i, 1] > Data[i - 2, 1] and Data[i, 1] > Data[i - 3, 1] and Data[i, 1] > Data[i - 4, 1] and Data[i, 3] < Data[i - 1, 3] and Data[i, 3] < Data[i - 2, 3] and Data[i, 3] < Data[i - 3, 3] and Data[i, 3] < Data[i - 4, 3]:
            Data[i, sell] = -1
    return Data
```

TD WALDO #5 PATTERN

"I don't think we'll find him."

Continuing in the spirit of the Waldo patterns, the below are the requirements for a TD Waldo #3 pattern.

To find a potential short-term bottom and a long opportunity:

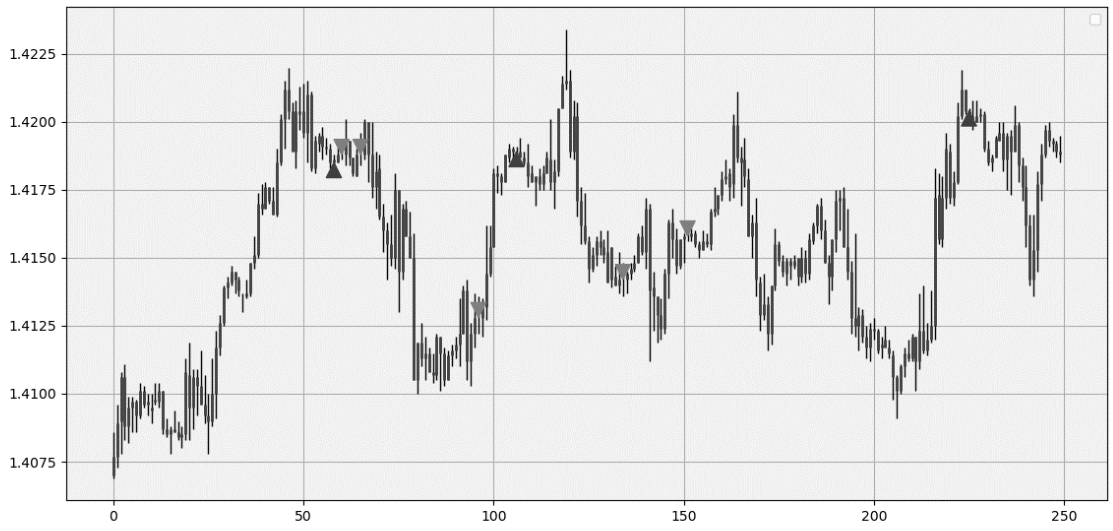
- The close of the current bar should be equal to the close of the previous bar.
- The close of the previous bar should be lower than the close of the bar before that.

To find a potential short-term top and a short-sell opportunity:

- The close of the current bar should be equal to the close of the previous bar.
- The close of the previous bar should be lower than the close of the bar before that.

Assuming we have an OHLC array, we can define and use the scanner that will look for the pattern and input 1 in case of a bullish Waldo #5 or input -1 in case of a bearish Waldo #5.

```
def td_waldo_5(Data, high, low, close, buy, sell):
    # Adding a few columns
    Data = adder(Data, 10)
    Data = rounding(Data, 4)
    for i in range(len(Data)):
        # Short-term Bottom
        if Data[i, 3] == Data[i - 1, 3] and Data[i - 1, 3] < Data[i - 2, 3]:
            Data[i, buy] = 1
        # Short-term Top
        if Data[i, 3] == Data[i - 1, 3] and Data[i - 1, 3] > Data[i - 2, 3]:
            Data[i, sell] = -1
    return Data
```



TD WALDO #6 PATTERN

"This is already too much."

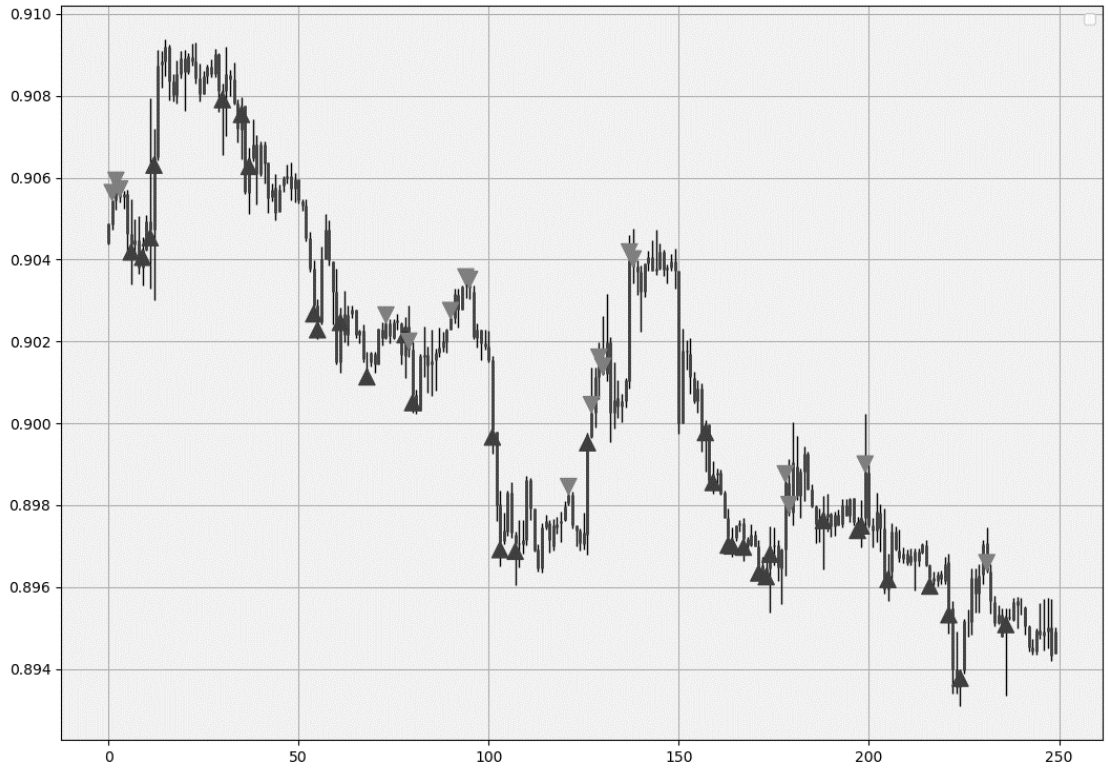
Continuing in the spirit of the Waldo patterns, the below are the requirements for a TD Waldo #3 pattern.

To find a potential short-term bottom and a long opportunity:

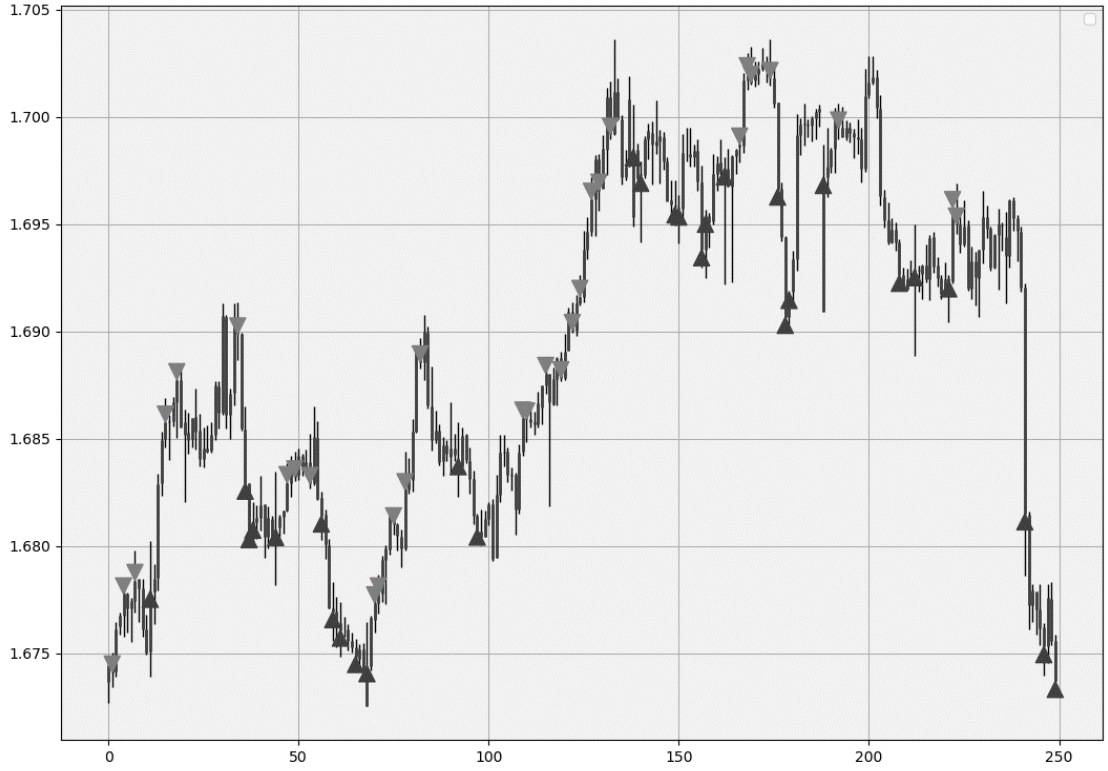
- The low of the current bar is lower than all eight prior lows.
- The difference between the current close and its low is greater than the difference between the previous close and its low.

To find a potential short-term top and a short-sell opportunity:

- The high of the current bar is higher than all eight prior highs.
- The difference between the current close and its high is greater than the difference between the previous close and its high.



```
def td_waldo_6(Data, high, low, close, buy, sell):
    # Adding a few columns
    Data = adder(Data, 10)
    for i in range(len(Data)):
        # Short-term Bottom
        if Data[i, 2] < Data[i - 1, 2] and \
            Data[i, 2] < Data[i - 2, 2] and \
            Data[i, 2] < Data[i - 3, 2] and \
            Data[i, 2] < Data[i - 4, 2] and \
            Data[i, 2] < Data[i - 5, 2] and \
            Data[i, 2] < Data[i - 6, 2] and \
            Data[i, 2] < Data[i - 7, 2] and \
            Data[i, 2] < Data[i - 8, 2] and \
            abs(Data[i, 3] - Data[i, 2]) - abs(Data[i - 1, 3] - Data[i - 1, 2]) > 0:
                Data[i, buy] = 1
        # Short-term Top
        if Data[i, 1] > Data[i - 1, 1] and \
            Data[i, 1] > Data[i - 2, 1] and \
            Data[i, 1] > Data[i - 3, 1] and \
            Data[i, 1] > Data[i - 4, 1] and \
            Data[i, 1] > Data[i - 5, 1] and \
            Data[i, 1] > Data[i - 6, 1] and \
            Data[i, 1] > Data[i - 7, 1] and \
            Data[i, 1] > Data[i - 8, 1] and \
            abs(Data[i, 3] - Data[i, 1]) - abs(Data[i - 1, 3] - Data[i - 1, 1]) > 0:
                Data[i, sell] = -1
        return Data
```


1.705
1.700
1.695
1.690
1.685
1.680
1.675
0
50
100
150
200
250

TD WALDO #8 PATTERN

"The last attempt."

Continuing in the spirit of the Waldo patterns, the below are the requirements for a TD Waldo #3 pattern.

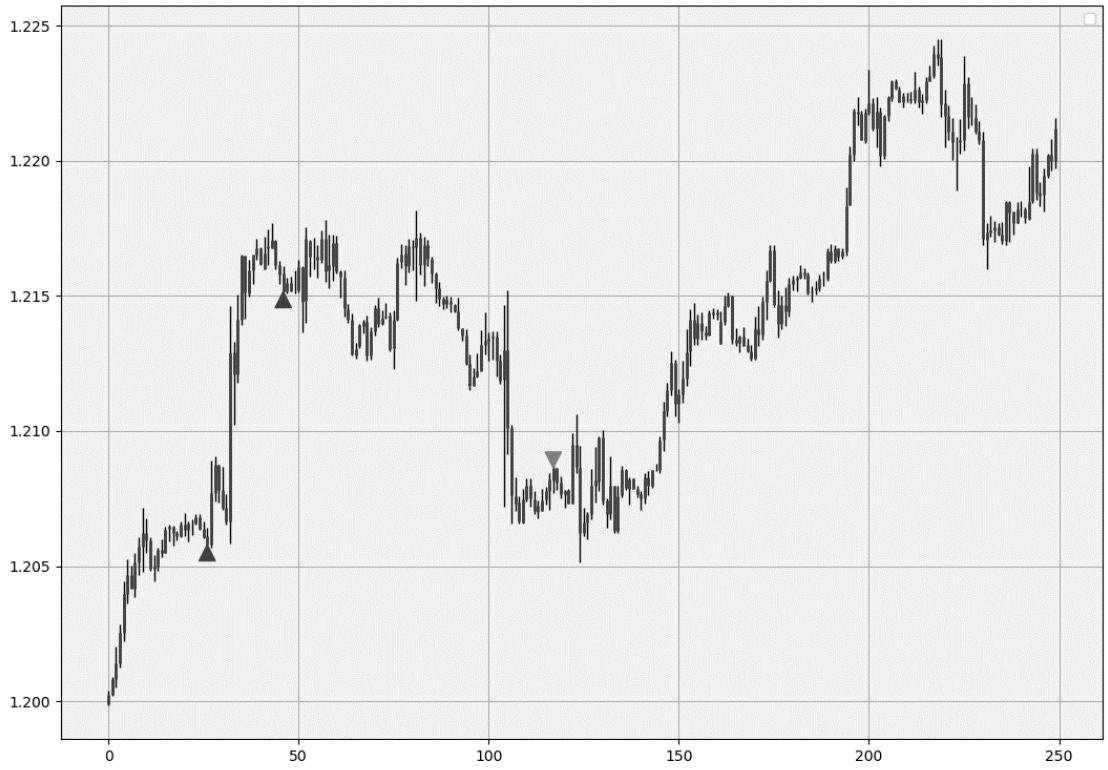
To find a potential short-term bottom and a long opportunity:

- The close of the current bar must be higher than the close of 12 bars earlier.
- The close of the current bar must be lower than all 7 prior lows.

To find a potential short-term top and a short-sell opportunity:

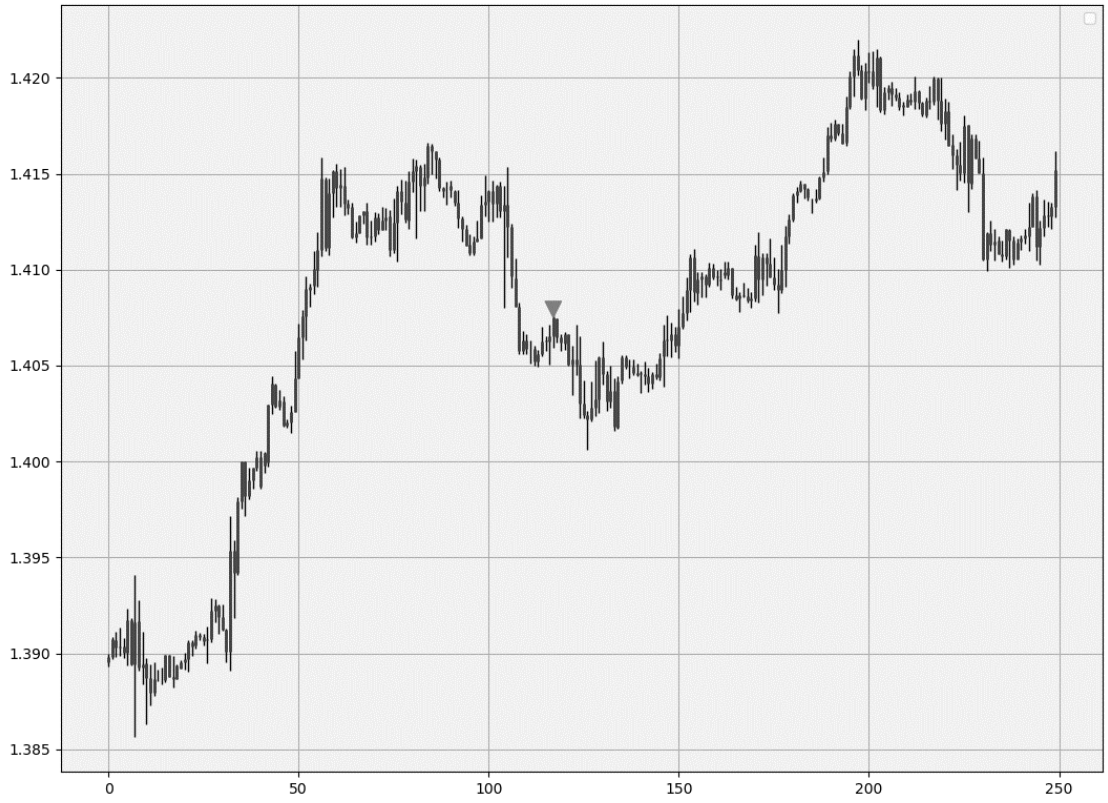
- The close of the current bar must be lower than the close of 12 bars earlier.
- The close of the current bar must be higher than all 7 prior highs.

We can code the signal function that gives us the necessary trading triggers as follows:



Assuming we have an OHLC array, we can define and use the scanner below that will look for the pattern and input 1 in case of a bullish Waldo #8 or input -1 in case of a bearish Waldo #8.

```
def td_waldo_8(Data, high, low, close, buy, sell):
    # Adding a few columns
    Data = adder(Data, 3)
    for i in range(len(Data)):
        # Short-term Bottom
        if Data[i, 3] < Data[i - 1, 2] and \
            Data[i, 3] < Data[i - 2, 2] and \
            Data[i, 3] < Data[i - 3, 2] and \
            Data[i, 3] < Data[i - 4, 2] and \
            Data[i, 3] < Data[i - 5, 2] and \
            Data[i, 3] < Data[i - 6, 2] and \
            Data[i, 3] < Data[i - 7, 2] and \
            Data[i, 3] > Data[i - 12, 3]:
                Data[i, buy] = 1
        # Short-term Top
        if Data[i, 3] > Data[i - 1, 1] and \
            Data[i, 3] > Data[i - 2, 1] and \
            Data[i, 3] > Data[i - 3, 1] and \
            Data[i, 3] > Data[i - 4, 1] and \
            Data[i, 3] > Data[i - 5, 1] and \
            Data[i, 3] > Data[i - 6, 1] and \
            Data[i, 3] > Data[i - 7, 1] and \
            Data[i, 3] < Data[i - 12, 3]:
                Data[i, sell] = -1
    return Data
```


1.420
1.415
1.410
1.405
1.400
1.395
1.390
1.385
0
50
100
150
200
250

TD DIFFERENTIAL PATTERN

"A pattern based on pressure."

The TD Differential group has been created (or found?) in order to find short-term reversals or continuations. They are supposed to help confirm our biases by giving us an extra conviction factor. For example, let us say that you expect a rise on the USDCAD pair over the next few weeks. You have your justifications for the trade, and you find some patterns on the higher time frame that seem to confirm what you are thinking. This will definitely make you more comfortable taking the trade.

Before we start presenting the patterns individually, we need to understand the concept of buying and selling pressure from the perception of the Differentials group. To calculate the Buying Pressure, we use the below formulas:

$$True\ Low = Min(Current\ Low,\ Previous\ Low)$$

$$Buying\ Pressure = Current\ Close - True\ Low$$

To calculate the Selling Pressure, we use the below formulas:

$$True\ High = Max(Current\ High,\ Previous\ High)$$

$$Selling\ Pressure = Current\ Close - True\ High$$

The TD Differential pattern seeks to find short-term trend reversals; therefore, it can be seen as a predictor of small corrections and consolidations.

To find a potential short-term bottom and a long opportunity:

- Two closes each less than the prior.
- Current buying pressure > previous buying pressure.
- Current selling pressure < previous selling pressure.

To find a potential short-term top and a short-sell opportunity:

- Two closes each greater than the prior.
- Current buying pressure < previous buying pressure.
- Current selling pressure > previous selling pressure.

The full Python code will be presented along the other two Differential patterns united in one function seen after the discussion.

TD REVERSE-DIFFERENTIAL PATTERN

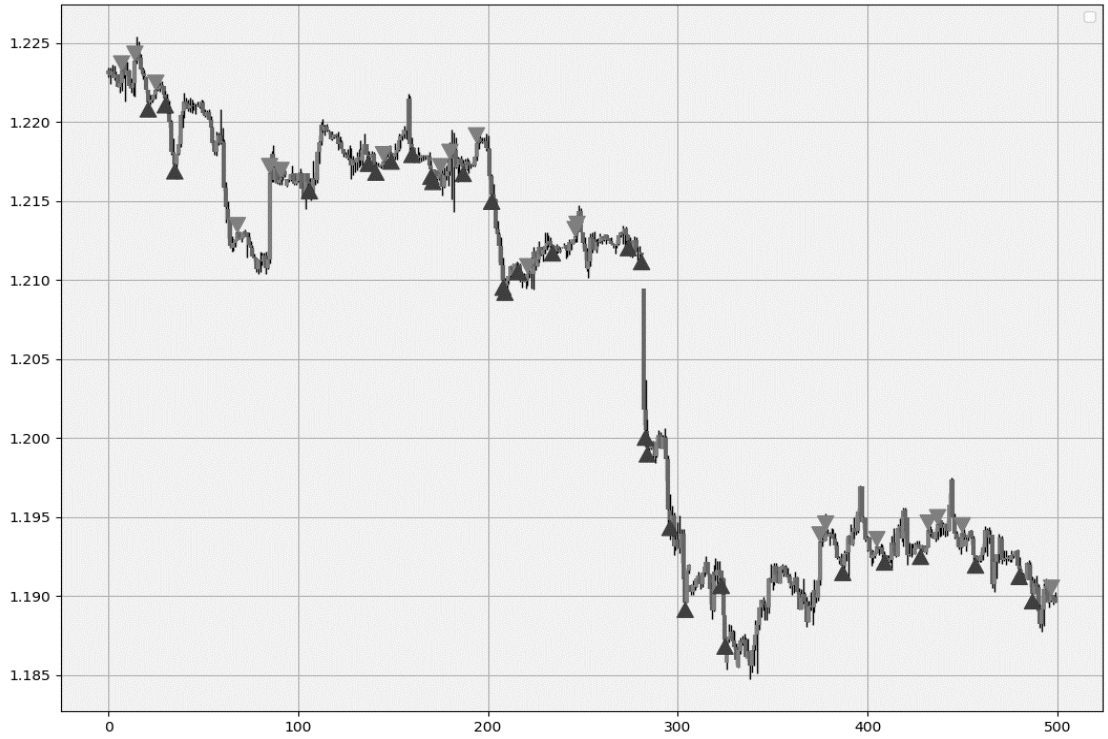
"Another pattern based on pressure."

This pattern seeks to find short-term trend continuations; therefore, it can be seen as a predictor of when the trend is strong enough to continue. It is useful because as we know it, the trend is our friend, and by adding another friend to the group, we may have more chance to make a profitable strategy. Let us check the conditions and how to code it: To find a potential short-term bottom and a long opportunity:

- Two closes each less than the prior.
- Current buying pressure < previous buying pressure.
- Current selling pressure > previous selling pressure.

To find a potential short-term top and a short-sell opportunity:

- Two closes each greater than the prior.
- Current buying pressure > previous buying pressure.
- Current selling pressure < previous selling pressure.



TD ANTI-DIFFERENTIAL PATTERN

"I don't get the name either."

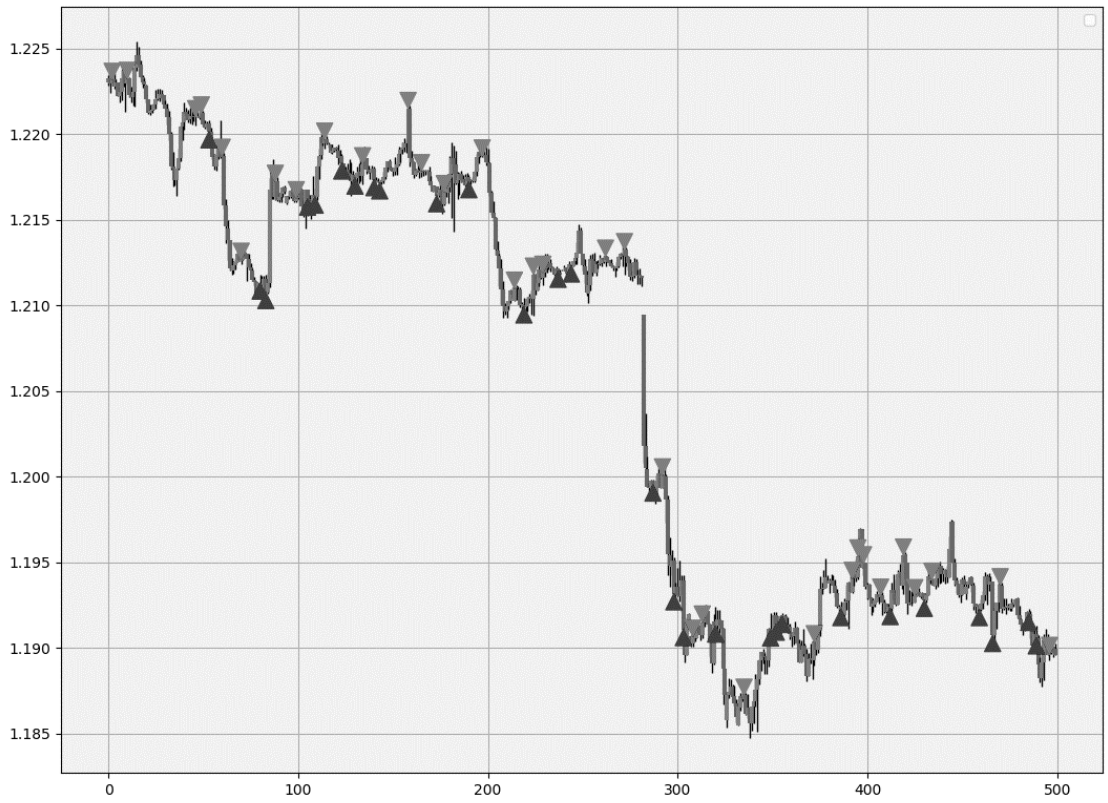
This pattern also seeks to find short-term trend reversals; therefore, it can be seen as a predictor of small corrections and consolidations. It is similar to the TD Differential pattern.

To find a potential short-term bottom and a long opportunity:

- Two closes each one is lower than the previous one then,
- A higher close than the previous close then,
- A lower close relative to the close of the previous bar.

To find a potential short-term top and a short-sell opportunity:

- Two closes each one is higher than the previous one then,
- A lower close than the previous close then,
- A close higher relative to the close of the previous bar.



```
def differentials(Data, what, true_low, true_high, buy, sell, differential = 1):
    if differential == 1:
        for i in range(len(Data)):
            # True low
            Data[i, true_low] = min(Data[i, 2], Data[i - 1, what])
            Data[i, true_low] = Data[i, what] - Data[i, true_low]
            # True high
            Data[i, true_high] = max(Data[i, 1], Data[i - 1, what])
            Data[i, true_high] = Data[i, what] - Data[i, true_high]
            # TD Differential
            if Data[i, what] < Data[i - 1, what] and Data[i - 1, what] < Data[i - 2, what] and Data[i, true_low] > Data[i - 1, true_low] and Data[i, true_high] < Data[i - 1, true_high]:
                Data[i, buy] = 1
            if Data[i, what] > Data[i - 1, what] and Data[i - 1, what] > Data[i - 2, what] and Data[i, true_low] < Data[i - 1, true_low] and Data[i, true_high] > Data[i - 1, true_high]:
                Data[i, sell] = -1
    if differential == 2:
        for i in range(len(Data)):
            # True low
            Data[i, true_low] = min(Data[i, 2], Data[i - 1, what])
            Data[i, true_low] = Data[i, what] - Data[i, true_low]
            # True high
            Data[i, true_high] = max(Data[i, 1], Data[i - 1, what])
            Data[i, true_high] = Data[i, what] - Data[i, true_high]
            # TD Reverse Differential
            if Data[i, what] < Data[i - 1, what] and Data[i - 1, what] < Data[i - 2, what] and Data[i, true_low] < Data[i - 1, true_low] and Data[i, true_high] > Data[i - 1, true_high]:
                Data[i, buy] = 1
            if Data[i, what] > Data[i - 1, what] and Data[i - 1, what] > Data[i - 2, what] and Data[i, true_low] > Data[i - 1, true_low] and Data[i, true_high] < Data[i - 1, true_high]:
                Data[i, sell] = -1
```

```
    if differential == 3: # TD Anti-Differential
        for i in range(len(Data)):
            if Data[i, what] < Data[i - 1, what] and Data[i - 1, what] > Data[i - 2, what] and Data[i - 2, what] < Data[i - 3, what] and Data[i - 3, what] < Data[i - 4, what]:
                Data[i, buy] = 1
            if Data[i, what] > Data[i - 1, what] and Data[i - 1, what] < Data[i - 2, what] and Data[i - 2, what] > Data[i - 3, what] and Data[i - 3, what] > Data[i - 4, what]:
                Data[i, sell] = -1
    Data = deleter(Data, 5, 1)
    return Data
```

The code above lets us choose between one of the three patterns and inputs the signals accordingly. The coding is therefore 1 for the TD Differential, 2 for the TD Reverse-Differential, and 3 for the TD Anti-Differential. The function is very straightforward, and the variables are easy to understand.

TD CAMOUFLAGE PATTERN

"One of the timing patterns created by Tom Demark."

This pattern seeks to find short-term trend reversals. For a bullish opportunity:

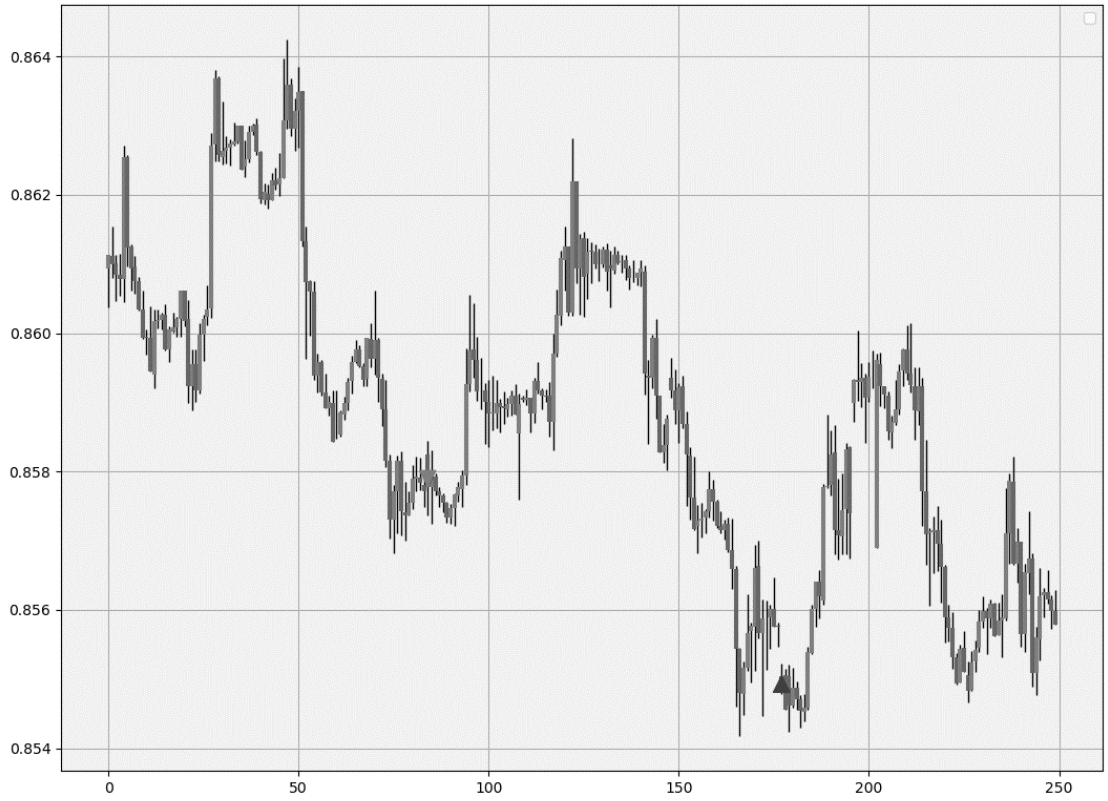
- Two closes each lower than the previous one then, a higher close than the previous close.
- A lower close relative to the close of the previous bar.

To find a potential short-term top and a short-sell opportunity:

- Two closes each higher than the previous one then, a lower close than the previous close.
- A close higher relative to the close of the previous bar.

```
def td_camouflage(Data):
    Data = adder(Data, 20)
    # True Low Calculation
    for i in range(len(Data)):
        Data[i, 5] = min(Data[i, 2], Data[i - 2, 2])
    # True High Calculation
    for i in range(len(Data)):
        Data[i, 6] = max(Data[i, 1], Data[i - 2, 1])
    # Bullish signal
    for i in range(len(Data)):
        if Data[i, 3] < Data[i - 1, 3] and Data[i, 3] > Data[i, 0] and Data[i, 2] < Data[i - 2, 5]:
            Data[i, 7] = 1
    # Bearish signal
    for i in range(len(Data)):
        if Data[i, 3] > Data[i - 1, 3] and Data[i, 3] < Data[i, 0] and Data[i, 1] > Data[i - 2, 6]:
            Data[i, 8] = -1
        return Data
```

The pattern is somewhat rare and unlikely to provide huge value to trading due to its simplicity and little conditions. One way to enhance it is to try and tweak its parameters and periods. Optimization can sometimes be in the form of looping around different periods and modifiable conditions so that one combination works well on average across similar assets. The plot above shows the signals generated from the pattern. We can notice the rarity of the pattern and the less than impressive timing skill.



The market's structure is too complicated to be captured by simplistic patterns. Tom Demark, a great technical analyst has discovered many patterns and created a huge number of indicators, all have proven efficacy and added value is featured a lot of times in this book and the credit must be given to him for the Demarker, the TD Waldo patterns, and all the TD indicators found elsewhere.

TD CLOP PATTERN

"I had to google the meaning of Clop. I still don't know that it means."

This pattern also seeks to find short-term trend reversals; therefore, it can be seen as a predictor of small corrections and consolidations.

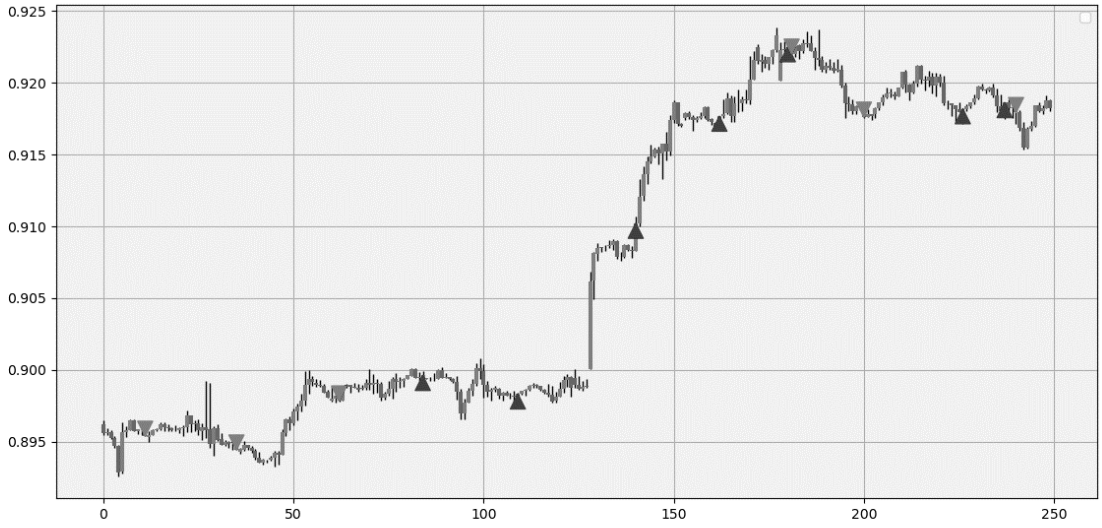
To find a potential short-term bottom and a long opportunity:

- The open of the current price bar must be below the close and open of the previous price bar.
- The market must close above both the open and close of the previous bar.

To find a potential short-term top and a short-sell opportunity:

- The open of the current price bar must be above the close and open of the previous price bar.
- The market must close below both the open and close of the previous bar.

```
def td_clop(Data):
  # Adding columns
  Data = adder(Data, 20)
  # Bullish signal
  for i in range(len(Data)):
    if Data[i, 0] < Data[i - 1, 3] and Data[i, 0] < Data[i - 1, 0] and Data[i, 3] > Data[i - 1, 0] and Data[i, 3] > Data[i - 1, 3]:
      Data[i, 6] = 1
  # Bearish signal
  for i in range(len(Data)):
    if Data[i, 0] > Data[i - 1, 3] and Data[i, 0] > Data[i - 1, 0] and Data[i, 3] < Data[i - 1, 0] and Data[i, 3] < Data[i - 1, 3]:
      Data[i, 7] = -1
  return Data
```


0.925
0.920
0.915
0.910
0.905
0.900
0.895
0
50
100
150
200
250

TD CLOPWIN PATTERN

"Don't even try to google the meaning."

This pattern also seeks to find short-term trend reversals; therefore, it can be seen as a predictor of small corrections and consolidations.

To find a potential short-term bottom and a long opportunity:

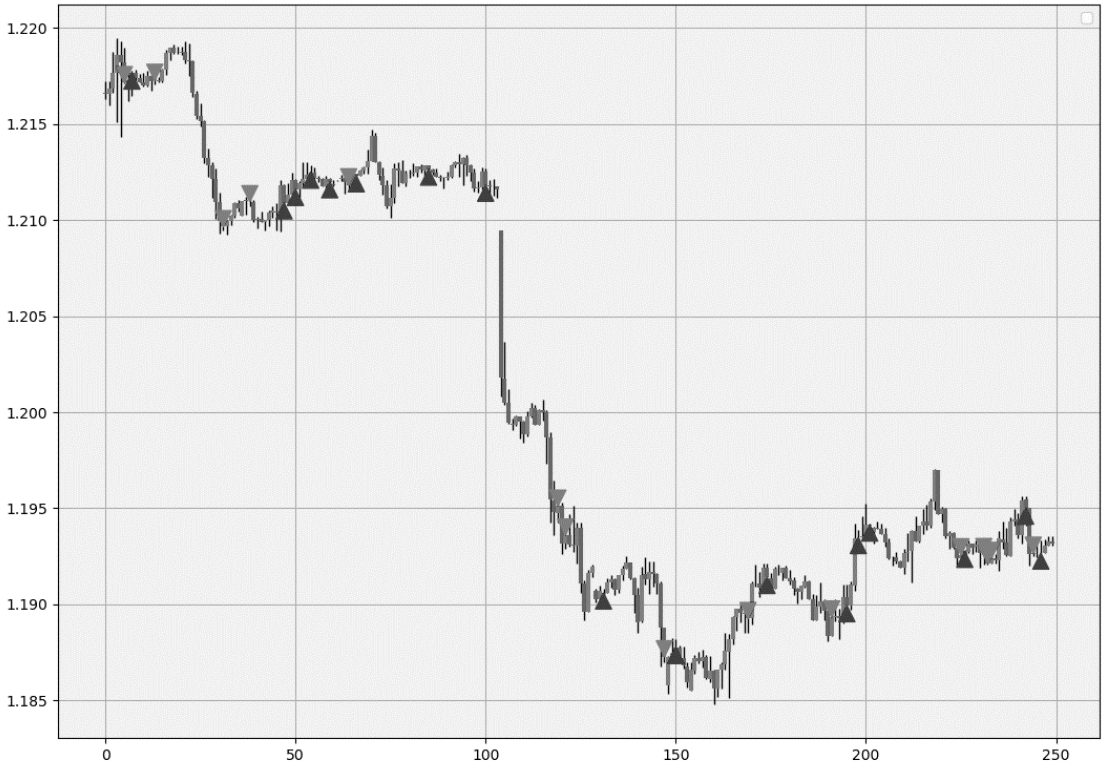
- The open and close of the current price bar must be contained within the open and close range of the previous price bar.
- The close of the current price bar must be above the close of the prior price bar.

To find a potential short-term top and a short-sell opportunity:

- The open and close of the current price bar must be contained within the open and close range of the previous price bar.
- The close of the current price bar must be below the close of the prior price bar.

```
def td_clopwin(Data):
    # Adding columns
    Data = adder(Data, 20)
    # Bullish signal
    for i in range(len(Data)):
        if Data[i, 1] < Data[i - 1, 1] and Data[i, 2] > Data[i - 1, 2] and Data[i, 3] > Data[i - 2, 3]:
            Data[i, 6] = 1
    # Bearish signal
    for i in range(len(Data)):
        if Data[i, 1] < Data[i - 1, 1] and Data[i, 2] > Data[i - 1, 2] and Data[i, 3] < Data[i - 2, 3]:
            Data[i, 7] = -1
    return Data
```

The above code calculates the pattern and populates columns 6 and 7 with buy and sell signals. We can visualize the signals given in the next graph.


1.220
1.215
1.210
1.205
1.200
1.195
1.190
1.185
0
50
100
150
200
250

TD TRAP PATTERN

"Will this be a good pattern?"

This pattern also seeks to find short-term trend reversals; therefore, it can be seen as a predictor of small corrections and consolidations.

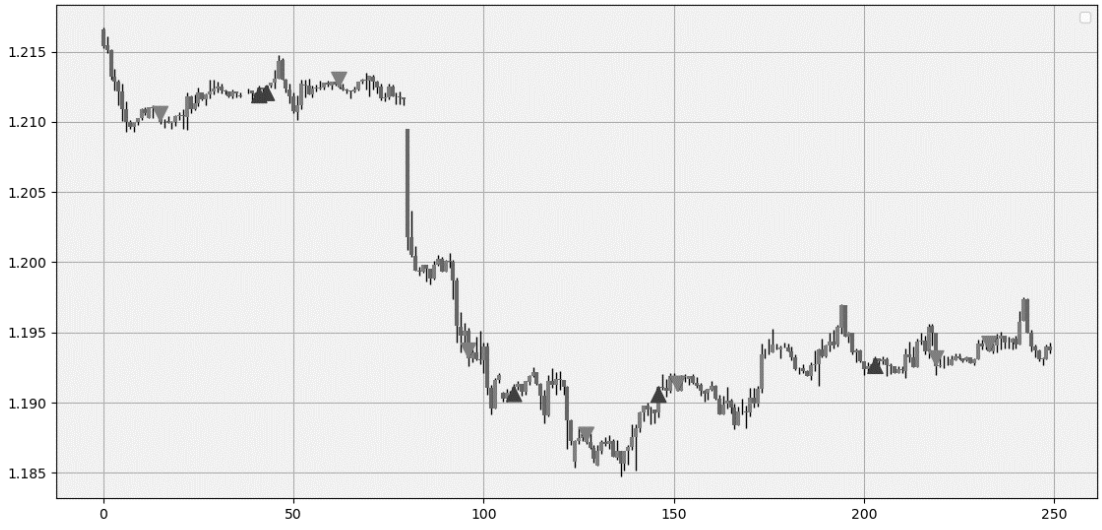
To find a potential short-term bottom and a long opportunity:

- Must be within the range of the previous price bar.
- Must then break above the high of that range.

To find a potential short-term top and a short-sell opportunity:

- Must be within the range of the previous price bar.
- Must then break below the low of that range.

```
def td_trap(Data):
    # Adding columns
    Data = adder(Data, 20)
    # Bullish signal
    for i in range(len(Data)):
        if Data[i, 1] < Data[i - 1, 1] and Data[i, 2] > Data[i - 1, 2] and Data[i, 3] > Data[i - 1, 1]:
            Data[i, 6] = 1
    # Bearish signal
    for i in range(len(Data)):
        if Data[i, 1] < Data[i - 1, 1] and Data[i, 2] > Data[i - 1, 2] and Data[i, 3] < Data[i - 1, 2]:
            Data[i, 7] = -1
    return Data
```


1.215
1.210
1.205
1.200
1.195
1.190
1.185
0
50
100
150
200
250

TD OPEN PATTERN

"The final Tom Demark pattern we will discuss in this book."

This pattern also seeks to find short-term trend reversals; therefore, it can be seen as a predictor of small corrections and consolidations.

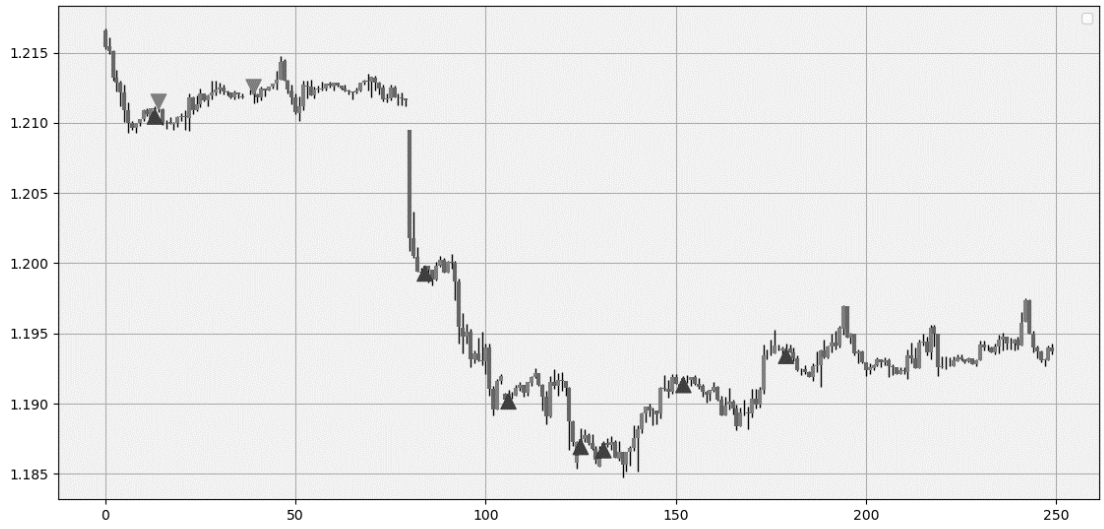
To find a potential short-term bottom and a long opportunity:

- The current price bar's open must be less than the low of the prior price bar.
- It must then trade above that low.

To find a potential short-term top and a short-sell opportunity:

- The current price bar's open must be greater than the high of the prior price bar.
- It must then trade below that high.

```
def td_open(Data):
    # Adding columns
    Data = adder(Data, 20)
    # Bullish signal
    for i in range(len(Data)):
        if Data[i - 1, 0] < Data[i - 2, 2] and Data[i, 3] > Data[i - 1, 2]:
            Data[i, 6] = 1
    # Bearish signal
    for i in range(len(Data)):
        if Data[i - 1, 0] > Data[i - 2, 1] and Data[i, 3] < Data[i - 1, 1]:
            Data[i, 7] = -1
    return Data
```


1.215
1.210
1.205
1.200
1.195
1.190
1.185
0
50
100
150
200
250

FIBONACCI TIMING PATTERN

"My first attempt to present a discretionary timing pattern to the world."

The pattern combines time, price, and the Fibonacci sequence in order to show whether they provide reversal points or not. Here is the basic intuition:

- For a bullish Fibonacci Timing Pattern, we need 8 closes where each close is lower than the close 5 periods ago, lower than the close 3 periods ago, and lower than the close 1 period ago. Upon the completion of this pattern, we will have a bullish signal. Any interruption in the sequence will invalidate the pattern.
- For a bearish Fibonacci Timing Pattern, we need 8 closes where each close is higher than the close 5 periods ago, higher than the close 3 periods ago, and higher than the close 1 period ago. Upon the completion of this pattern, we will have a bearish signal. Any interruption in the sequence will invalidate the pattern.

As seen in previous sections, the Fibonacci sequence is a very interesting mathematical series which can be found in many areas in life. From nature to astronomy and trading, the sequence can help us detect market reactions whether through retracements or as inputs in indicators such as patterns and moving averages. When I have discovered this pattern, I immediately thought about its correlation with other indicators and how it can add value into the trading framework. When we use two indicators to make a decision, we should try to make them as uncorrelated as possible otherwise, they would give the same signals. Giving the same signals is not very useful as it is as if someone gives his opinion twice. The value added is null. However, the pattern, when added to other mathematical indicators such as the Relative Strength Index, the signals do seem uncorrelated, therefore, when a planetary alignment occurs, the conviction should be stronger.

```
def fibonacci_timing_pattern(Data, count, step, step_two, step_three, close, buy, sell):
    # Bullish Fibonacci Timing Pattern
    counter = -1
for i in range(len(Data)):
        if Data[i, close] < Data[i - step, close] and [i, close] < Data[i - step_two, close] and \
          Data[i, close] < Data[i - step_three, close]:
           Data[i, buy] = counter
           counter += -1
           if counter == -count - 1:
              counter = 0
           else:
              continue
        elif Data[i, close] >= Data[i - step, close]:
           counter = -1
           Data[i, buy] = 0
    # Bearish Fibonacci Timing Pattern
    counter = 1
    for i in range(len(Data)):
        if Data[i, close] > Data[i - step, close] and Data[i, close] > Data[i - step_two, close] and \
          Data[i, close] > Data[i - step_three, close]:
           Data[i, sell] = counter
           counter += 1
           if counter == count + 1:
              counter = 0
           else:
              continue
        elif Data[i, close] <= Data[i - step, close]:
           counter = 1
           Data[i, sell] = 0
    return Data
```

The regular Fibonacci Timing Pattern has the following parameters:

 # Using the function

count = 8

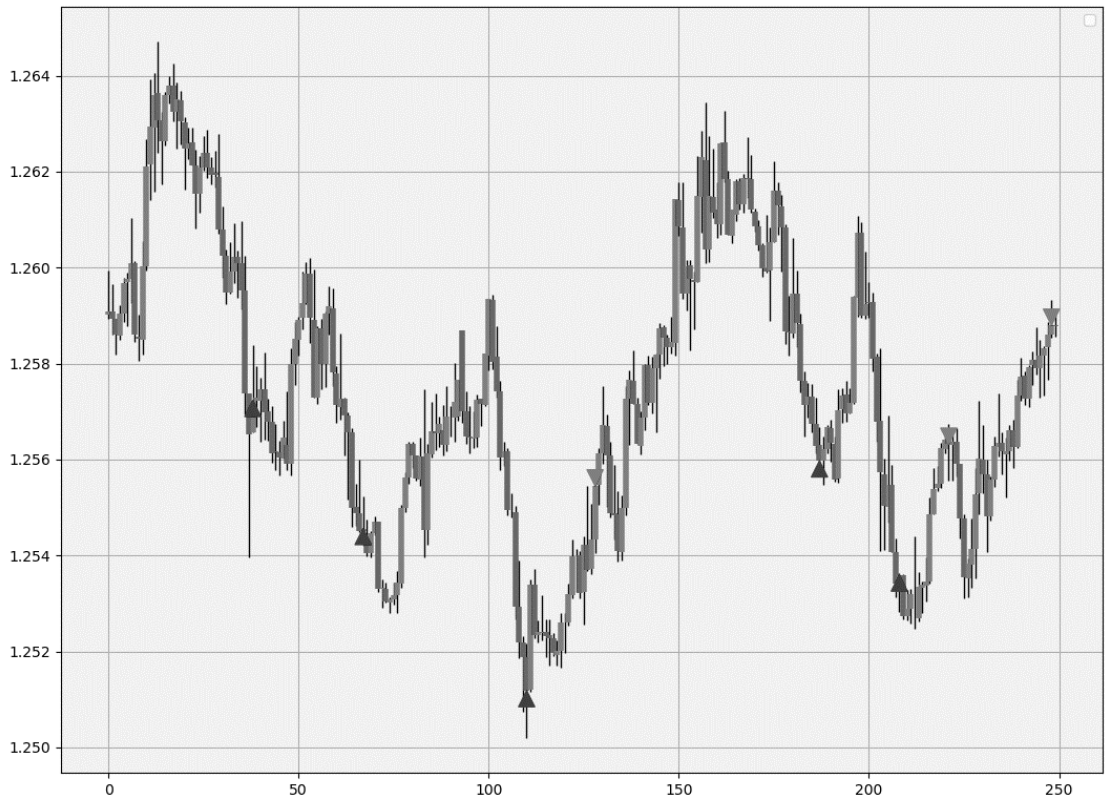
step = 5

step_two = 3

step_three = 2

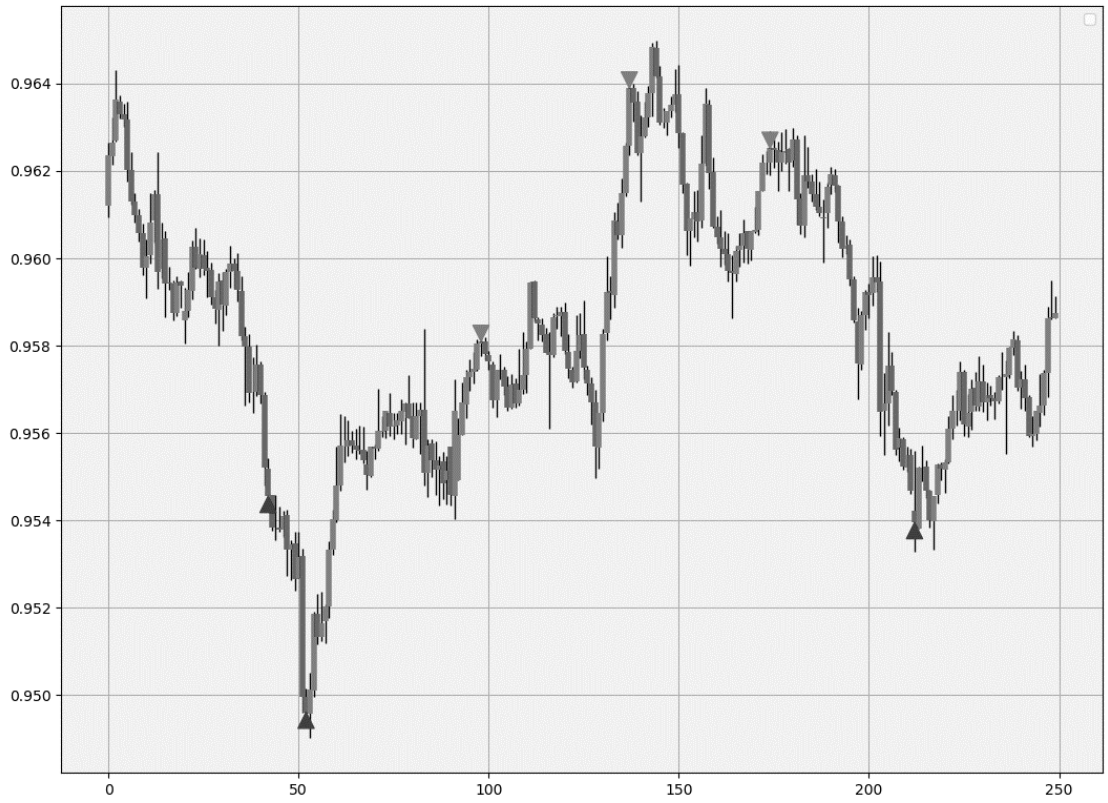
my_data = fibonacci_timing_pattern(my_data, count, step, step_two, step_three, 3, 6, 7)

The plot above shows the signals generated on the AUDUSD hourly data. Sometimes, when markets are very choppy, the signals can be less common but during these times, the quality goes up and it tends to be around tops and bottoms. Surely, this is a subjective statement built on personal experience but the visual component of the pattern when applied on the chart is satisfactory. We will see later a simple back-test using some special risk management conditions.



On some pairs, it may appear to be more common, and this is due to the market's properties and volatility. Sometimes, one move is enough to invalidate all the pattern. The above plot shows the USDCAD. When the market is trending in such a steep angle, it is unlikely that the signals work. However, we can trade them profitably in this environment. Evidently, this reversal pattern can also be used as a trend-following pattern in the following way:

- The bullish pattern is invalidated whenever the market continues below the bullish signal. This condition is valid when the move below the signal equals at least twice the body of the signal candle.
- The bearish pattern is invalidated whenever the market continues above the bearish signal. This condition is valid when the move above the signal equals at least twice the body of the signal candle.

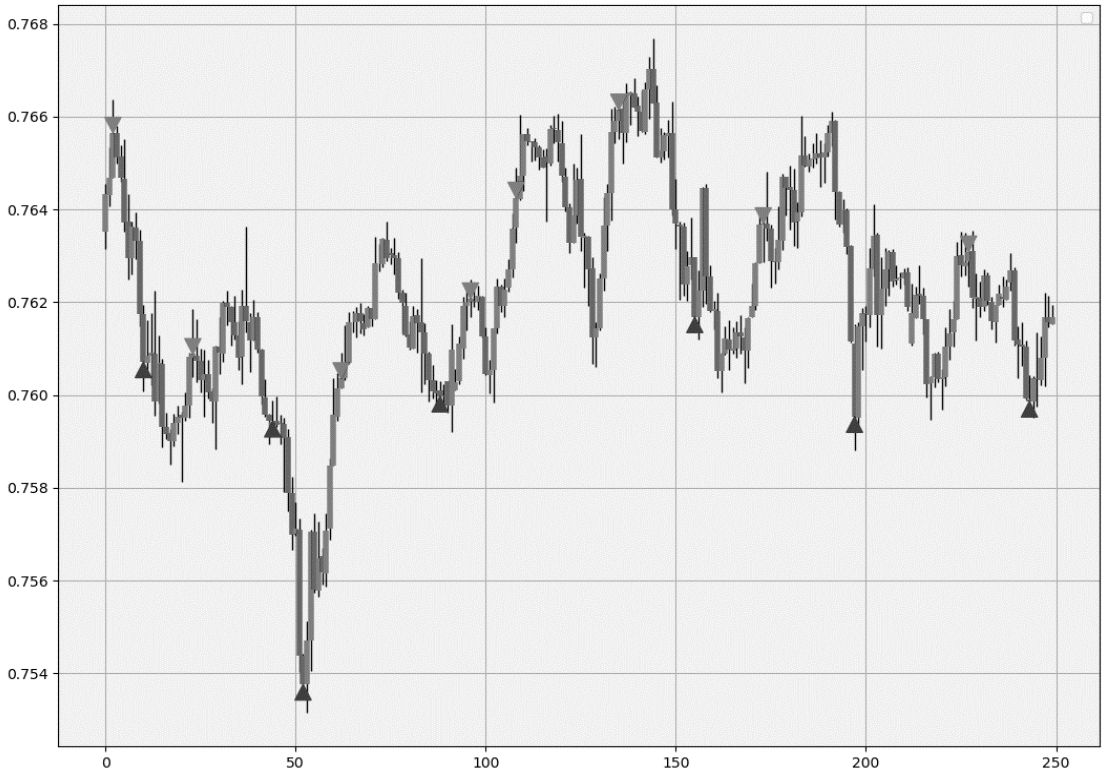


The last plot shows the AUDCAD in trending and ranging mode. We can notice the difference in the quality of signals between the two regimes. Notice how the first bullish signal actually provided an opportunity to short after it has been invalidated. But how do we choose our target? The target is simply the apparition of a new bullish signal while the stop should be the high of the signal candle.

FAST FIBONACCI VARIATION TIMING PATTERN

"For frequent signals."

I have found that to increase signals while not necessarily (severely) hurting the predictive power of the pattern, we can tweak the pattern so that it becomes {5, 3, 2, 1}. This means that each time we have five bars where each one is higher than the one 3 periods ago as well as higher than the one 2 periods ago and 1 period ago, we can have a bearish signal. Similarly, each time we have five bars where each one is lower than the one 3 periods ago as well as lower than the one 2 periods ago and 1 period ago, we can have a bullish signal.

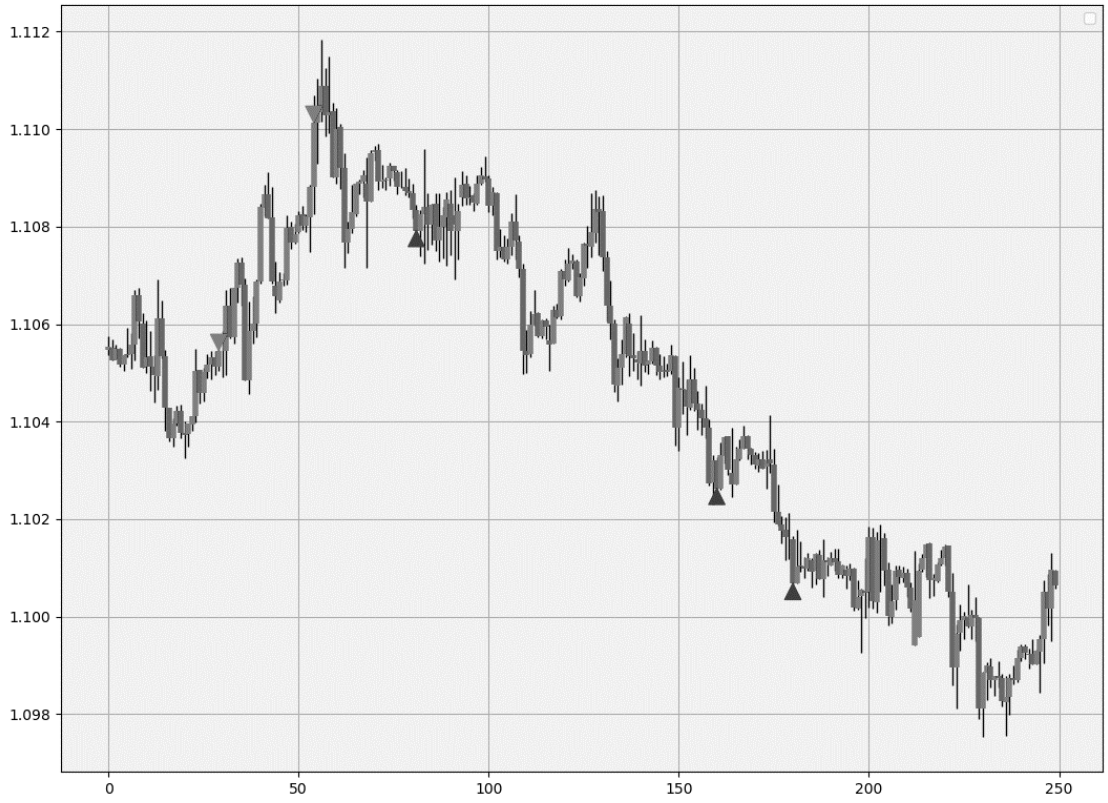


count = 5

step = 3

step_two = 2

step_three = 1



This concludes out pattern recognition chapter and the overall sections dealing with just one indicator. The next part will discuss structured strategies which are a combination of two or more indicators acting together to deliver a unified signal with a greater conviction that we would have gotten if we had used just one indicator. As usual, the back-testing results will be left to you to experiment with and if you need a full framework on how to get to trading results from the presented strategies, refer to chapter 1 to understand how to calculate performance metrics. Also, for the code and for replicability, make sure to refer to the GitHub link[8].

[8] https://github.com/sofienkaabar/the-book-of-more-back-tests

PART 5

STRUCTURED STRATEGIES

Structured strategies are the first step towards a robust system. It is unlikely that one indicator can capture the complexity of the markets and provide consistently profitable signals. This is why we need a confirmation factor, and this can be done through structured strategies which are simply a combination of two or more technical indicators. They are based on the fact that two opinions are better than one. This can act as a filter to improve the good signals and eliminate the bad signals.

Trading is based on a disciplined framework composed of the following areas:

- A solid strategy based on at least two or three indicators.
- A numerical way to determine the position sizing such as the Kelly Criterion.
- A numerical way to determine risk and reward such as the Average True Range.

The three steps must imperatively be of high quality as the whole trading process will not work if one of the above elements is obsolete. In this chapter, we will take a look at some structured strategies that should give us an idea on how the signal generation process can be. As usual, the results are omitted for the reasons cited in the beginning of the book. The best thing to take from this chapter is the idea generation phase and understanding how to properly combine the indicators so that intuitively, we have better signals.

THE RELATIVE STRENGTH INDEX & MOVING AVERAGES

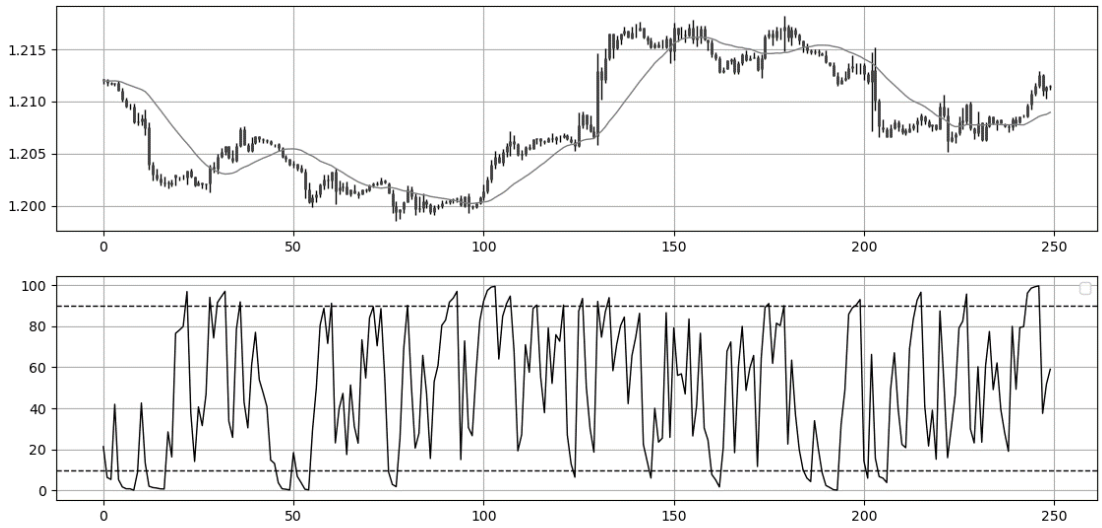
"What if trend-following and contrarian techniques become the same?"

Combining indicators into more elaborate strategies has two advantages:

- A filter for the generated signals: When an extra layer of condition is added, we call this filtering. It serves as a last sanity check of the signal before it is taken.
- An enhancer of returns: When the added condition has predictive power, it serves as an enhancer of returns as we will see later in the back-test.

The combination we will try out is a sort of contrarian trend-following strategy. Paradoxically, this can also be an even stronger strategy than a pure contrarian or a pure trend-follower. The way we do this is by identifying the trend using a short-term moving average. Once we determine whether we are trending upwards or downwards, we look at the short-term Relative Strength Index, if it is providing a contrarian signal in tandem with the trend, then we have a filtered signal. Here are two examples:

- The market is trending above its 13-period moving average and is slightly approaching it from the above. We notice that the 2-period RSI is at or below the 20% level. This is a filtered bullish signal where a bullish trend-following sub-signal is generated with the market price being above its moving average and a bullish contrarian sub-signal is generated with the market going to its support on the Relative Strength Index. This is visually seen as a market dropping to the direction of its bullish moving average, a form of pull-back, if you will.
- The market is trending below its 13-period moving average and is slightly approaching it from the below. We notice that the 2-period RSI is at or above the 80% level. This is a filtered bearish signal where a bearish trend-following sub-signal is generated with the market price being below its moving average and a bearish contrarian sub-signal is generated with the market going to its resistance on the Relative Strength Index. This is visually seen as a market going up to the direction of its bearish moving average.



Now, it is time to consult the conditions:

- Go long whenever the 2-period RSI is below 10 with the previous 4 readings above 10 while simultaneously, the current price closes above the 21-period moving average. Hold this position until getting a new signal or getting stopped by the risk management algorithm.
- Go short whenever the 2-period RSI is above 90 with the previous 4 readings below 90 while simultaneously, the current price closes below the 21-period moving average. Hold this position until getting a new signal or getting stopped by the risk management algorithm.

```
def signal(Data, what, ma_col, buy, sell):

    for i in range(len(Data)):

        if Data[i, what] < lower_barrier and Data[i - 1, what] > lower_barrier and Data[i - 2, what] >
lower_barrier and Data[i - 3, what] > lower_barrier and Data[i - 4, what] > lower_barrier and Data[i,
2] > Data[i, ma_col]:

            Data[i, buy] = 1

        if Data[i, what] > upper_barrier and Data[i - 1, what] < upper_barrier and Data[i - 2, what] <
upper_barrier and Data[i - 3, what] < upper_barrier and Data[i - 4, what] < upper_barrier and Data[i,
3] < Data[i, ma_col]:

            Data[i, sell] = -1

    return Data
```

THE BOLLINGER BANDS & THE KELTNER CHANNEL

"Basically, a Bollinger strategy made worse by the Keltner Channel."

The main idea is to combine these two volatility bands in order to create a strategy based on their signals. First things first, we have to introduce the Keltner Channel.

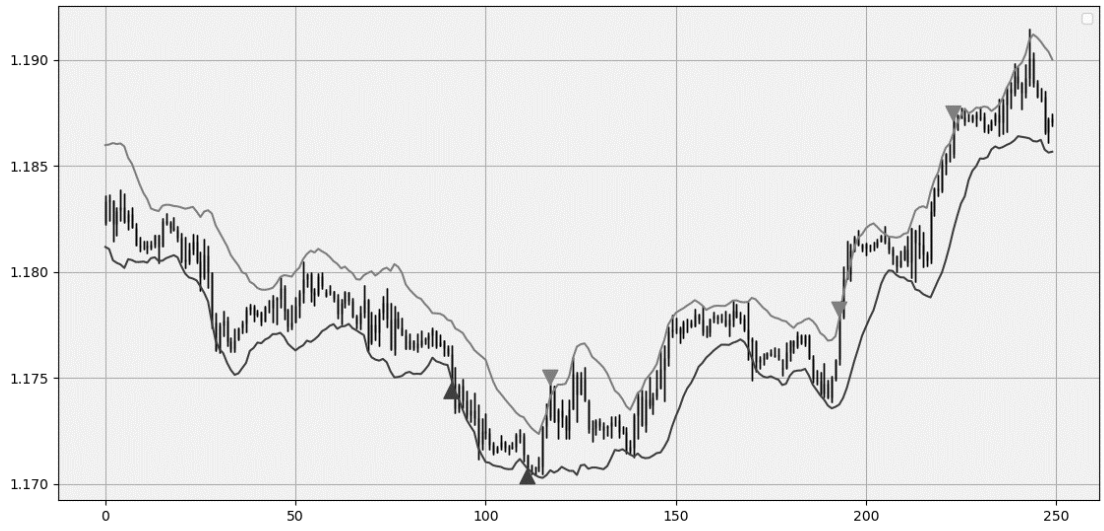
The Keltner Channel is a volatility-based technical indicator that resembles the Bollinger Bands, only it uses an exponential moving average as the mean calculation and the Average True Range as a volatility proxy. Hence, here is the main two differences between the two:

- The Bollinger Bands: A simple moving average with bands based on historical Standard Deviation.
- The Keltner Channel: An exponential moving average with bands based on the Average True Range.

A full chapter on the Average True Range can be found in Appendix II. Make sure to read it before continuing this section. Now, we calculate the Keltner Channel using an exponential moving average with the ATR of the price. Here is the formula:

$$Upper\ Band = Exponential\ Moving\ Average + (Constant\ .ATR)$$

$$Lower\ Band = Exponential\ Moving\ Average - (Constant\ .ATR)$$

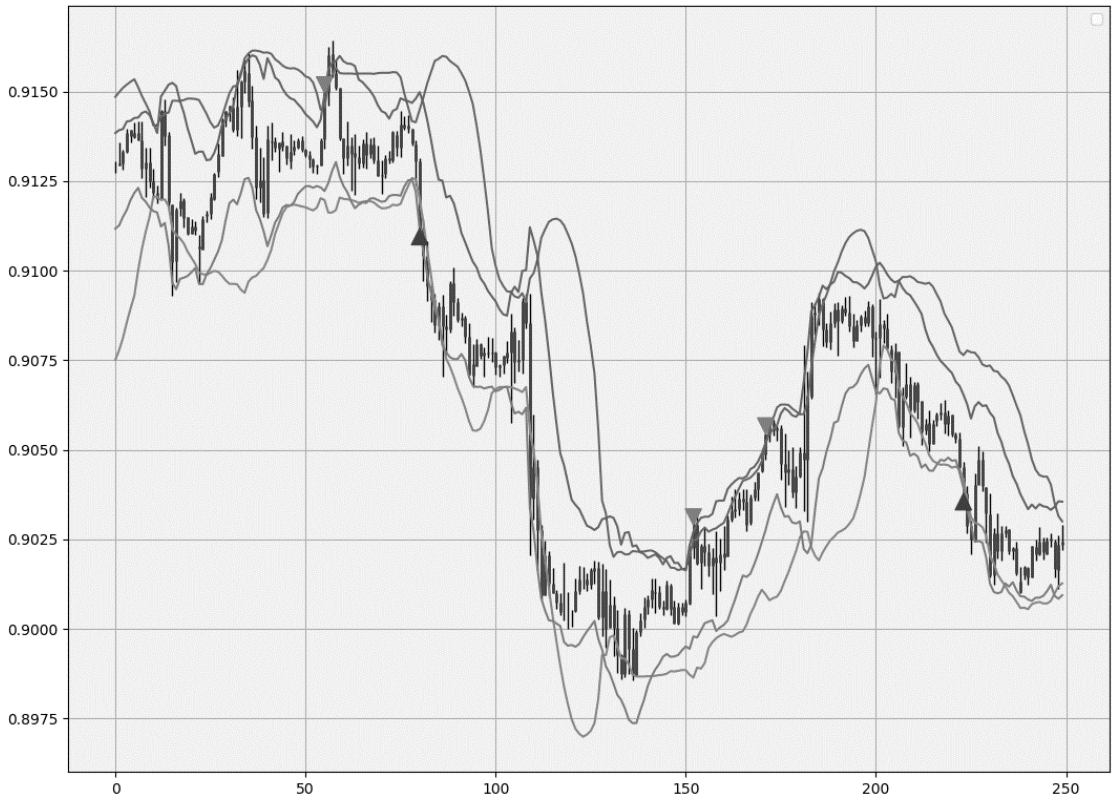


With the Python code to output the Keltner Channel:

```
def keltner_channel(Data, ma_lookback, atr_lookback, multiplier, what, where):
  Data = ema(Data, 2, ma_lookback, what, where)
  Data = atr(Data, atr_lookback, 2, 1, 3, where + 1)
  Data[:, where + 2] = Data[:, where] + (Data[:, where + 1] * multiplier)
  Data[:, where + 3] = Data[:, where] - (Data[:, where + 1] * multiplier)
  return Data
```

The concept of the idea is simple. If we like Bollinger Bands and the Keltner Channel, what would we get if we combine their signals? What if we get two signals at the same time? Would that provide a winning strategy? Financial markets are much more complicated than just combining two famous technical indicators together in the hopes of becoming rich without an effort.

- Go long (Buy) whenever the market price is below its lower 20-period Bollinger Band and below its lower 10-period Keltner Channel. Hold this position until getting another signal or getting stopped out by the risk management algorithm.
- Go short (Sell) whenever the market price is above its upper 20-period Bollinger Band and above its upper 10-period Keltner Channel. Hold this position until getting another signal or getting stopped out by the risk management algorithm.
- The standard deviation used on the Bollinger Bands is by default 2.0 and the multiplier used on the Keltner Channel is also by default 2.0




```
def signal(Data):
    for i in range(len(Data)):
        if Data[i, 3] <= Data[i, 6] and Data[i, 3] <= Data[i, 8] and Data[i - 1, 9] == 0 and \
            Data[i - 2, 9] == 0 and Data[i - 3, 9] == 0 and Data[i - 4, 9] == 0:
            Data[i, 9] = 1
        if Data[i, 3] >= Data[i, 5] and Data[i, 3] >= Data[i, 7] and Data[i - 1, 10] == 0 and \
            Data[i - 2, 10] == 0 and Data[i - 3, 10] == 0 and Data[i - 4, 10] == 0:
            Data[i, 10] = -1
    return Data
```

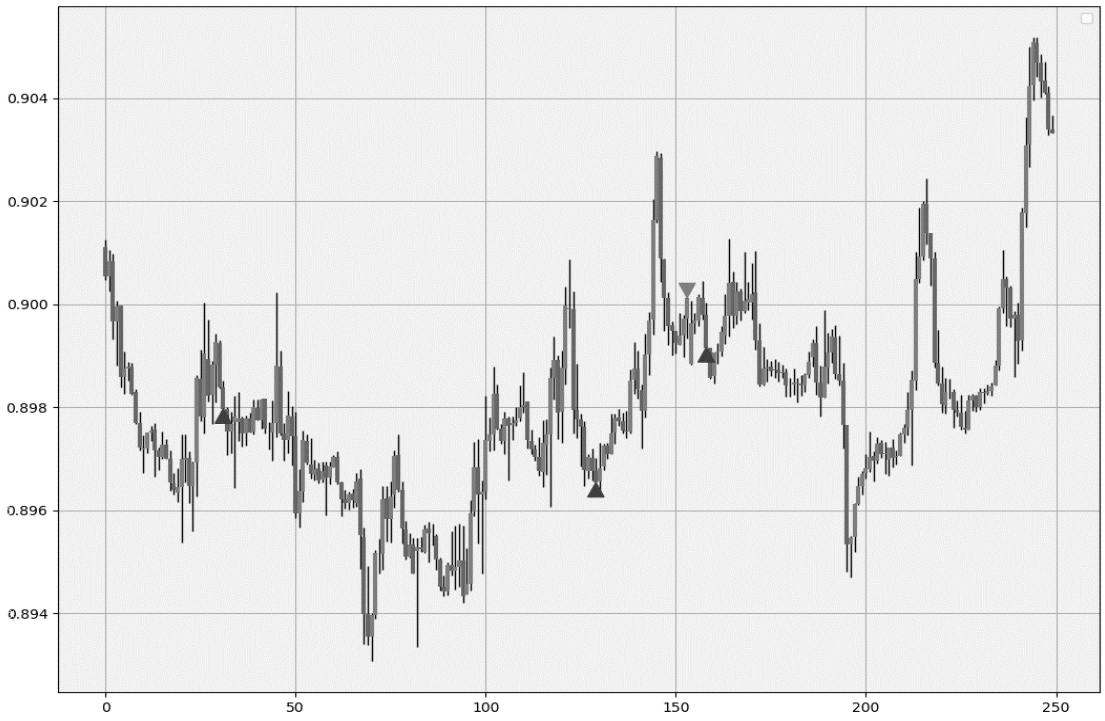
THE STOCHASTIC OSCILLATOR & THE RELATIVE STRENGTH INDEX

"Two behemoths in the world of trading coming together."

The main idea is to try to combine two known indicators in order to find the times when they provide a signal at the same time. Surely this violates the uncorrelation rule where we prefer to have two indicators that generally do not give signals at the same time, but by combining the RSI and the Stochastic Oscillator, we can benefit from an enhanced conviction from two well-known indicators.

The conditions required for the moving average cross

- Go long (Buy) whenever the 2-period Simple RSI and the 2-period Stochastic Oscillator are below 1 simultaneously. Hold the position until getting a new signal or getting stopped out by the risk management system.
- Go short (Sell) whenever the 2-period Simple RSI and the 2-period Stochastic Oscillator are above 99 simultaneously. Hold the position until getting a new signal or getting stopped out by the risk management system.



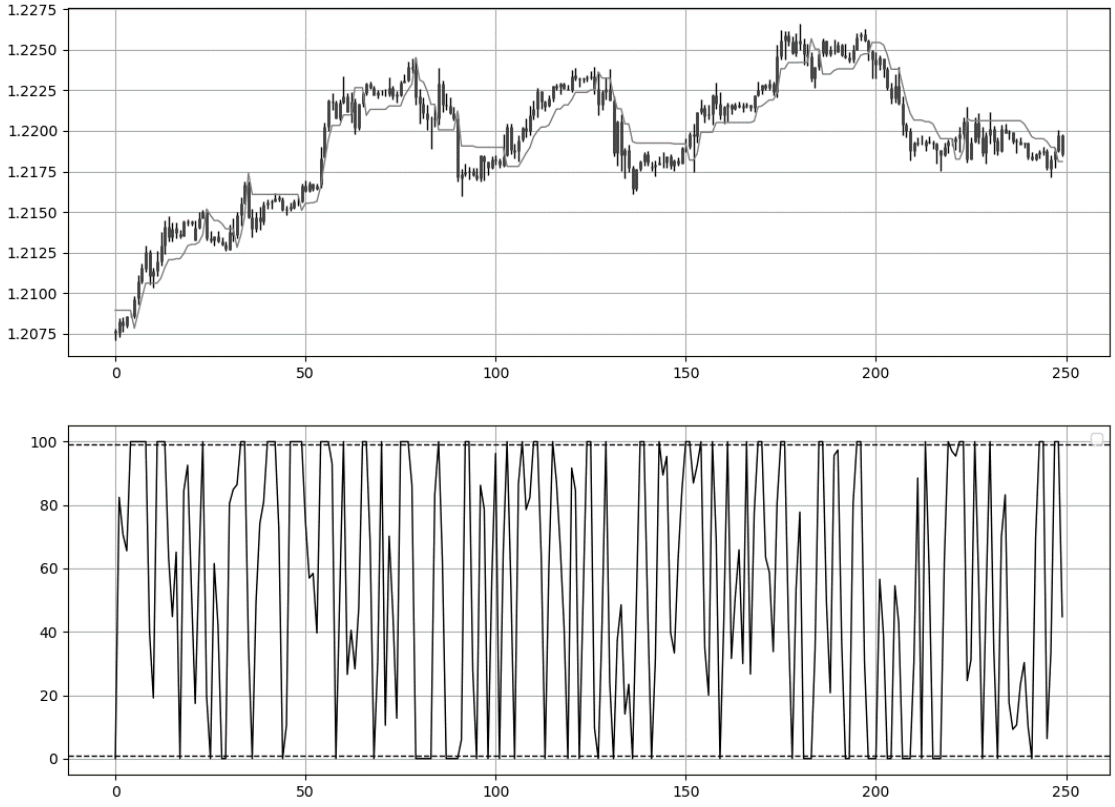
```
upper_barrier    = 99
lower_barrier    = 1
def signal(Data, rsi_col, stoch_col, buy, sell):
   Data = adder(Data, 10)
   Data = rounding(Data, 5)
   for i in range(len(Data)):
      if Data[i, rsi_col] < lower_barrier and Data[i, stoch_col] < lower_barrier:
         Data[i, buy] = 1
      elif Data[i, rsi_col] > upper_barrier and Data[i, stoch_col] > upper_barrier:
         Data[i, sell] = -1
   return Data
```

The Simple RSI is a simple change in the way the moving average is calculated inside the formula of the standard RSI. Instead of using a smoothed moving average as recommended by Wilder, we will use a simple moving average which is found in the function presented in this book, we have to just set the genre to simple.

THE SUPERTREND INDICATOR & THE RELATIVE STRENGTH INDEX

"The undervalued indicator and the overvalued indicator."

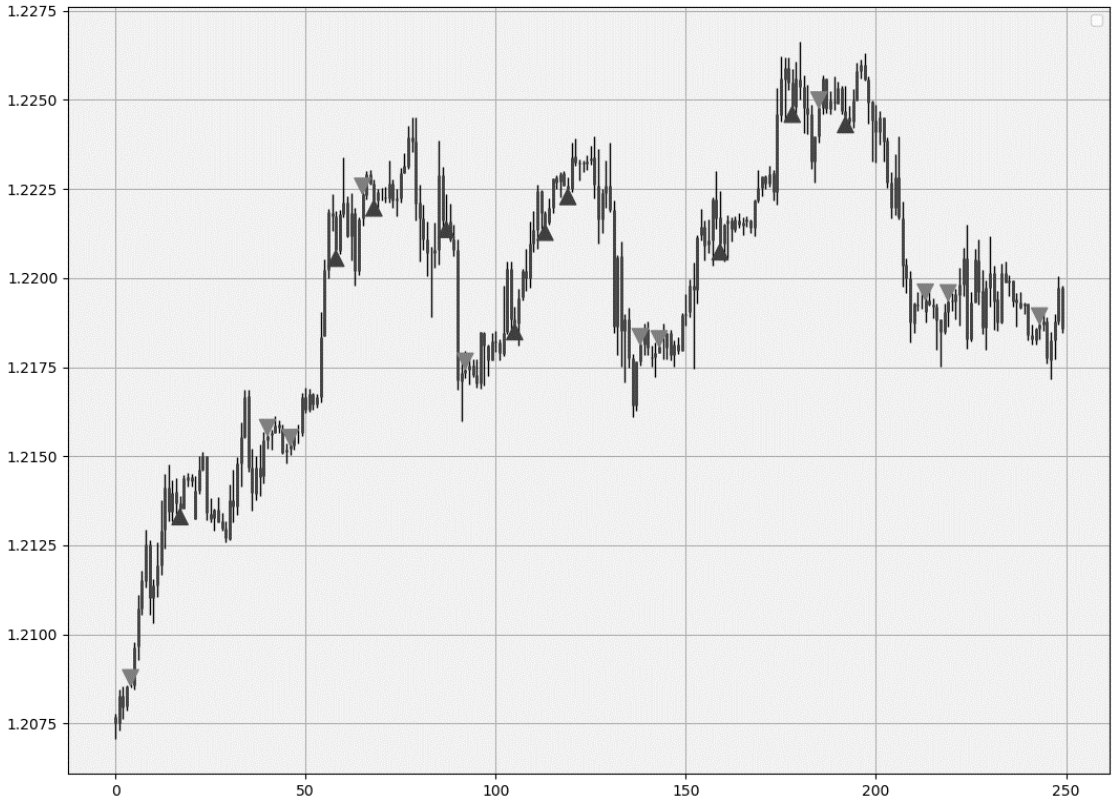
The main idea is when the market is above the SuperTrend, we are ought to be bullish as it is providing support and upside pressure. To help confirm this view and generate a signal, we can use a short-term Simple RSI to initiate a buy in case it is oversold. Similarly, when the market is below the SuperTrend, we are ought to be bearish as it is providing resistance and downside pressure. To help confirm this view and generate a signal, we can use a short-term Simple RSI to initiate a short sell trade in case it is overbought.



The conditions required for the moving average cross

- Go long (Buy) whenever the market is above the 15-period SuperTrend (with 1.25 multiplier) while the 2-period Simple RSI just hit 1.

- Go short (Sell) whenever the market is below the 15-period SuperTrend (with 1.25 multiplier) while the 2-period Simple RSI just hit 99.



The full Python code to get signals from the moving average cross can be found in the next code snippet. Added conditions include the obligation that the previous buy and sell columns be empty so that we have no duplicate and successive orders.

```
def signal(Data, rsi_col, supertrend_col, buy, sell):
    Data = adder(Data, 10)
    Data = rounding(Data, 5)
    for i in range(len(Data)):
        if Data[i, rsi_col] < lower_barrier and Data[i - 1, buy] == 0 and \
            Data[i - 2, buy] == 0 and Data[i - 3, buy] == 0 and Data[i - 4, buy] == 0 and Data[i, 2] > Data[i,
supertrend_col]:
            Data[i, buy] = 1
        elif Data[i, rsi_col] > upper_barrier and Data[i - 1, sell] == 0 and \
            Data[i - 2, sell] == 0 and Data[i - 3, sell] == 0 and Data[i - 4, sell] == 0  and Data[i, 1] < Data[i,
supertrend_col]:
            Data[i, sell] = -1
    return Data
```

THE TREND INTENSITY INDEX & MOVING AVERAGES

"The beauty and the beast."

The main idea is to use moving averages as trend identifiers while getting the signals from an indicator called the Trend Intensity Index described below. Generally, the latter is used to follow the trend, however, in this strategy we will use it as an indicator that detects a reversal in the direction of the overall trend identified by the moving average. The Trend Intensity Index is a measure of the strength of the trend. It is a relatively simple calculation to make. The indicator is created by a 10-period moving average and price deviations around it, then, we will count the number of the up periods relative to the total number of periods. Let us build the indicator step-by-step:

- Define the lookback period which is set at 10 and calculate the moving average on the market price using this lookback period.

```
lookback = 10
# Calculating the Moving Average
Data = ma(my_data, lookback, 3, 4)
```

- Calculate the deviations of the market price from the moving average. This is done by doing so on two columns. If the current market price is greater than its 10-period moving average, then the first column will be filled by the difference between the two (market price minus moving average). If the current market price is lower than its 10-period moving average, then the second column will be filled by the difference between the two (Moving average minus market price).

```
# Deviations
for i in range(len(my_data)):
  if my_data[i, what] > my_data[i, where]:
    my_data[i, where + 1] = my_data[i, what] - my_data[i, where]
  if my_data[i, what] < my_data[i, where]:
    my_data[i, where + 2] = my_data[i, where] - my_data[i, what]
```

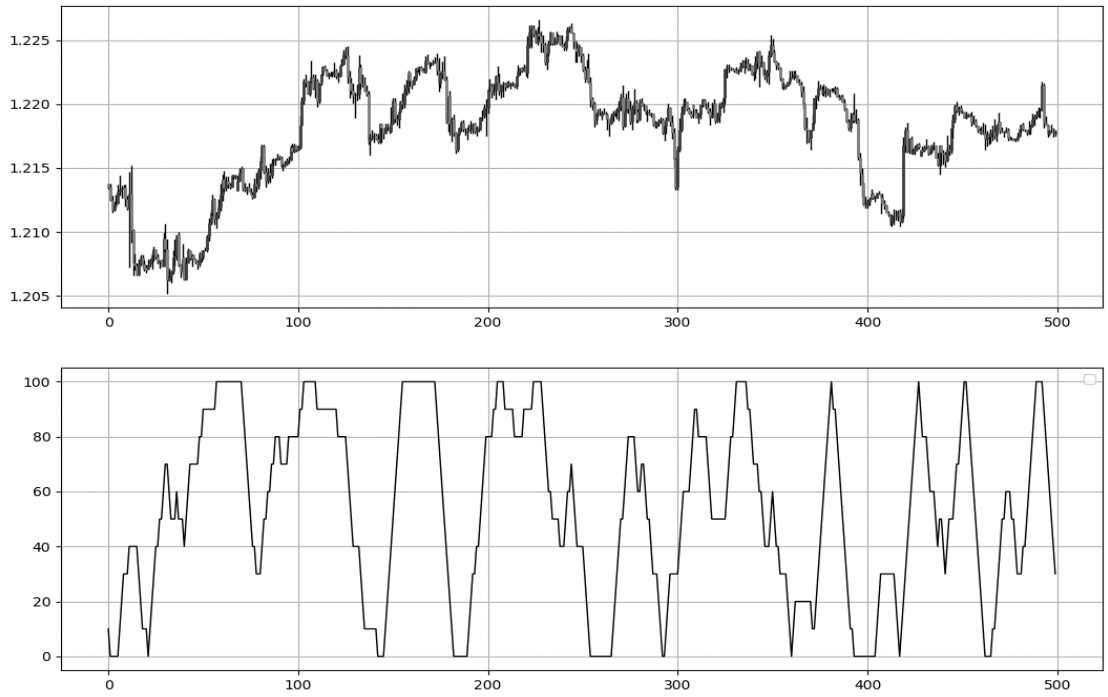
Now, we want to count the values where the market was above its 10-period moving average and where it was below it. This can be done using the numpy function count_nonzero():

```
for i in range(len(Data)):
  Data[i, where + 3] = np.count_nonzero(Data[i - lookback + 1:i + 1, where + 1])
for i in range(len(Data)):
  Data[i, where + 4] = np.count_nonzero(Data[i - lookback + 1:i + 1, where + 2])
```

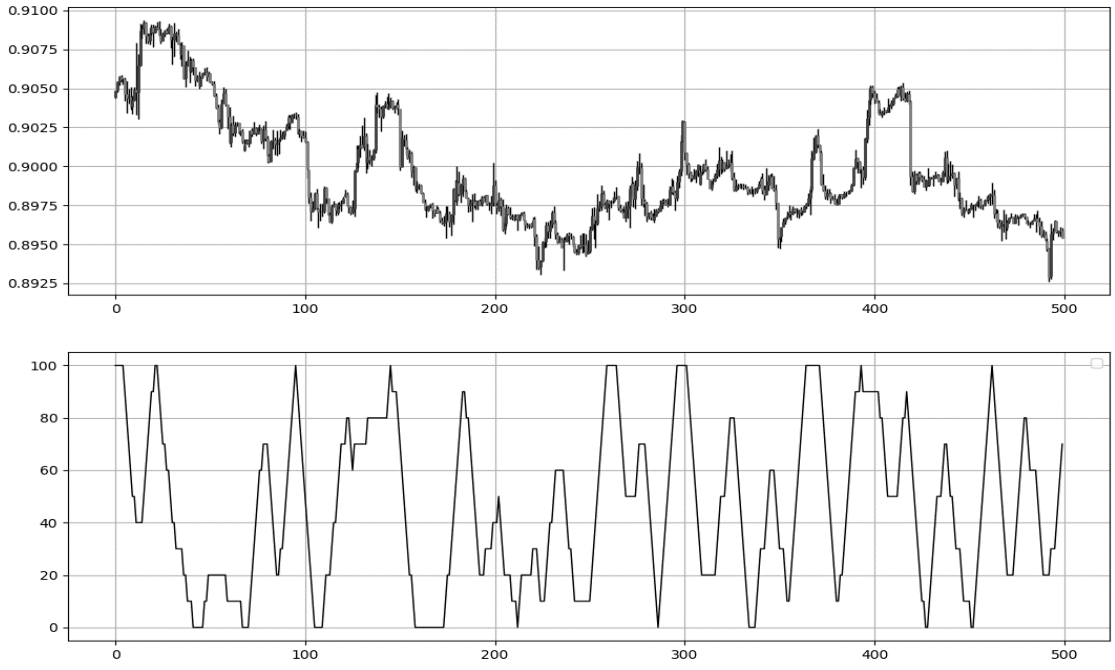
And finally, we can calculate the Trend Intensity Index using the below formula with the Python code below it:

$$Trend\ Intensity\ Indicator = \left(\frac{Total\ Up}{(Total\ Up + Total\ Down)}\right) x\ 100$$

```
# Trend Intensity Index
for i in range(len(Data)):
  Data[i, where + 5] = ((Data[i, where + 3]) / (Data[i, where + 3] + Data[i, where + 4])) * 100
```



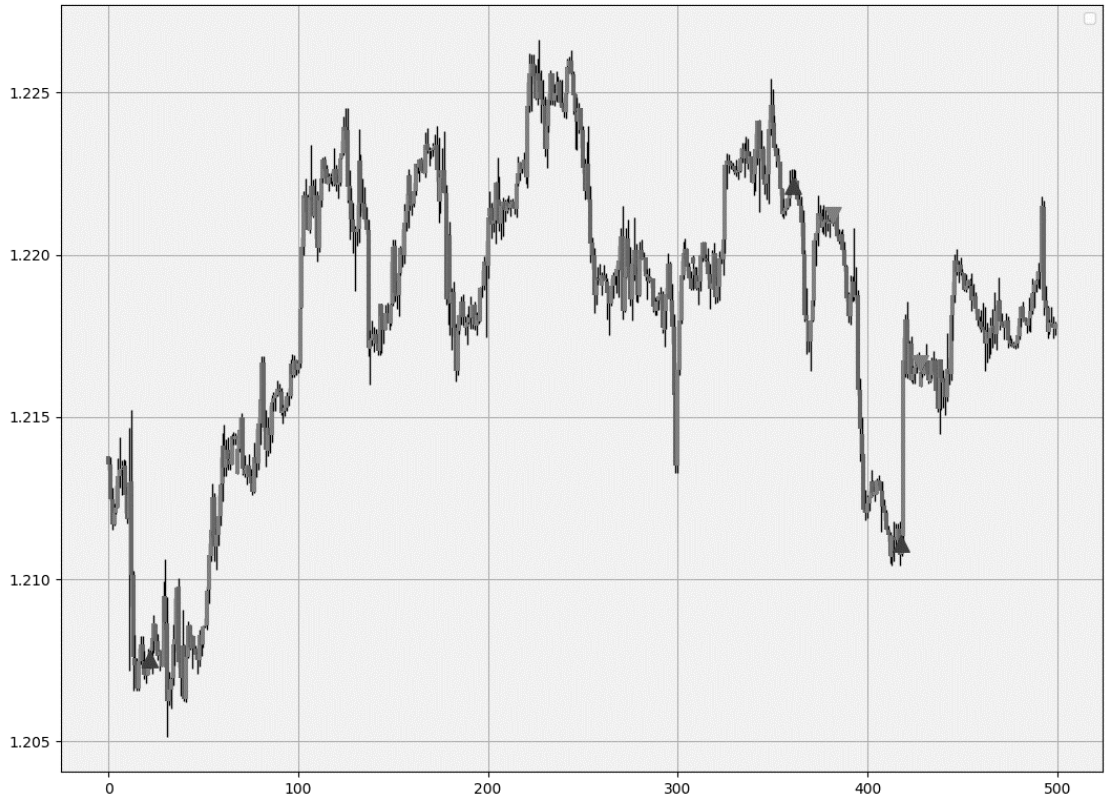
```
def trend_intensity_index(Data, lookback, what, where):
    # Calculating the Moving Average
    Data = ma(Data, lookback, what, where)
    # Deviations
    for i in range(len(Data)):
        if Data[i, what] > Data[i, where]:
            Data[i, where + 1] = Data[i, what] - Data[i, where]
        if Data[i, what] < Data[i, where]:
            Data[i, where + 2] = Data[i, where] - Data[i, what]
    # Trend Intensity Index
    for i in range(len(Data)):
        Data[i, where + 3] = np.count_nonzero(Data[i - lookback + 1:i + 1, where + 1])
    for i in range(len(Data)):
        Data[i, where + 4] = np.count_nonzero(Data[i - lookback + 1:i + 1, where + 2])
    for i in range(len(Data)):
        Data[i, where + 5] = ((Data[i, where + 3]) / (Data[i, where + 3] + Data[i, where + 4])) * 100
    Data = deleter(Data, where, 5)
    return Data
```



As with any proper research method, the aim is to test the strategy and to be able to see for ourselves whether it is worth having as an add-on to our pre-existing trading framework or not.

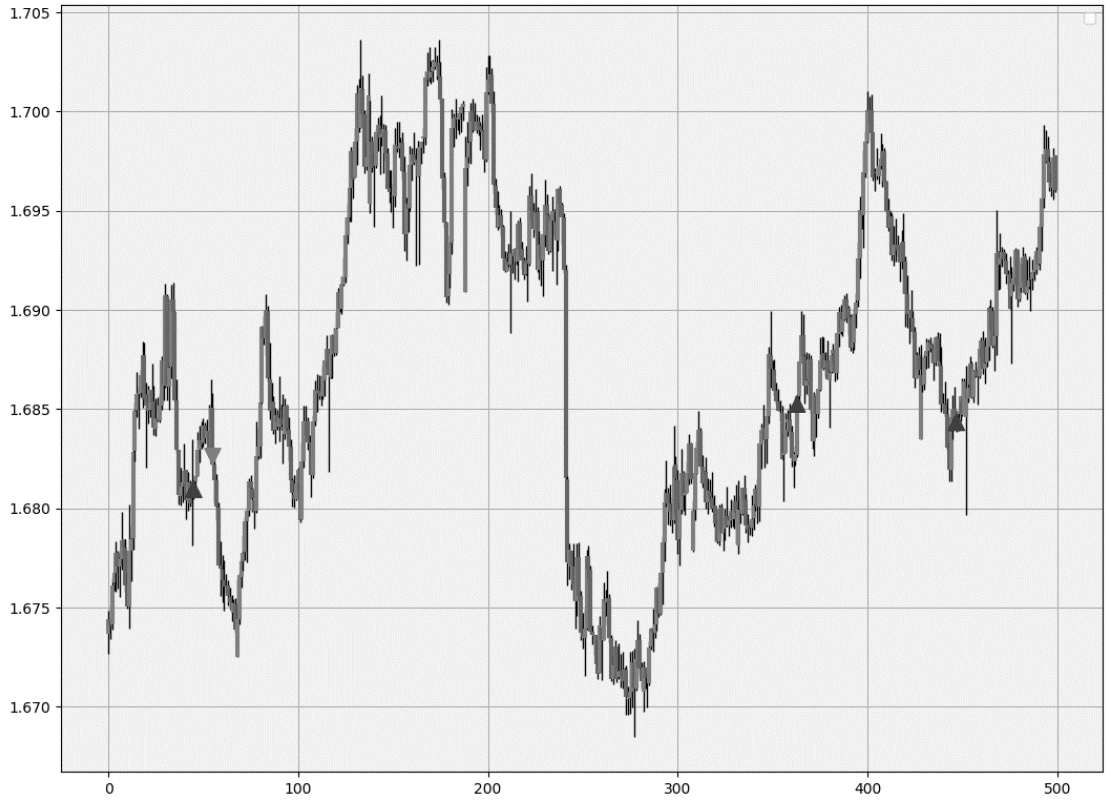
The first step is creating the trading rules. When will the system buy and when will it go short? In other words, when is the signal given that tells the system that the current market will go up or down? The trading conditions we can choose are:

- Go long (Buy) whenever the Trend Intensity Index shapes a V formation, meaning the current reading is above zero, the previous reading equals zero, and the one prior to greater than zero. Simultaneously, the 200-period moving average must be below the market price.
- Go short (Sell) whenever the Trend Intensity Index shapes an inverted V formation, meaning the current reading is below 100, the previous reading equals 100, and the one prior to below than 100. Simultaneously, the 200-period moving average must be above the market price.



The above chart shows the signals generated from the system. We have to keep in mind the frequency of the signals when we are developing a trading algorithm. The signal function used to generate the triggers based on the conditions mentioned above can be found in this snippet:

```
def signal(Data, close, tii, ma_col, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, tii] > 0 and Data[i - 1, tii] == 0 and Data[i - 2, tii] > 0 and Data[i, close] > Data[i, ma_col]:
            Data[i, buy] = 1
        elif Data[i, tii] < 100 and Data[i - 1, tii] == 100 and Data[i - 2, tii] < 100 and Data[i, close] < Data[i, ma_col]:
            Data[i, sell] = -1
    return Data
```



The above shows the same technique applied onto hourly values of the GBPUSD. Without a doubt, lagging indicators will have a tough time detecting the trend at its beginning, which is why we sometimes see some bullish signals around the tops and bearish signals around the bottoms.

THE FISHER AGGREGATE STRATEGY

"A complex but simple strategy."

The Fisher Aggregate Index seeks to gather as much extremes as possible from a range of lookback periods. This means that we will calculate the Modified Fisher Transformation using a set of different lookback periods and then average them all out. We will try out a range between a 3-period Fisher and a 29-period Fisher. This can be coded as the below:

```
where = 4
for i in range(3, 30):
   my_data = fisher_transform(my_data, i, 3, where)
   where = where + 1
my_data= adder(my_data, 1)
for i in range(len(my_data)):
   my_data[i, -1] = np.sum(my_data[i, 4:4+ 30 - 3])
   my_data[i, -1] = my_data[i, - 1] / (30 - 3)
my_data = deleter(my_data, 4, 27)
```

The next plot shows the signals generated using the index. The conditions can be found right after.

If we want to follow the barriers technique where we buy around support and go short around resistance, then we can try out the following conditions:

- Go long (Buy) whenever the Fisher Aggregate Index reaches the lower barrier with the two previous readings higher than the lower barrier. Hold this position until getting a new signal or getting stopped out by the risk management system.
- Go short (Sell) whenever the Fisher Aggregate Index reaches the upper barrier with the two previous readings lower than the upper barrier. Hold this position until getting a new signal or getting stopped out by the risk management system.

```
upper_barrier = 2.24

lower_barrier = -2.24

def signal(Data, what, buy, sell):

  for i in range(len(Data)):

    if Data[i, what] < lower_barrier and Data[i - 1, what] > lower_barrier and Data[i - 2, what] >
lower_barrier :

      Data[i, buy] = 1

    if Data[i, what] > upper_barrier and Data[i - 1, what] < upper_barrier and Data[i - 2, what] <
upper_barrier :

      Data[i, sell] = -1

  return Data
```

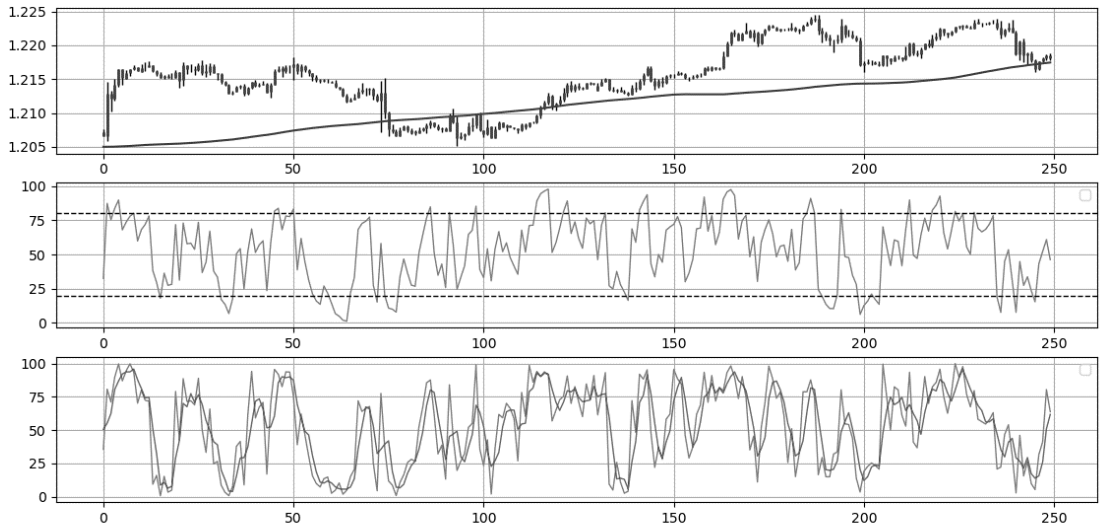
The choice of the barriers was to make a zone around the 2.00 graphical level. 2.24 equals the sum of the golden ratio and its reciprocal(1.618 and 0.618). Remember to always do your back-tests. Even though I supply the indicator's function (as opposed to just brag about it and say it is the holy grail and its function is a secret), you should always believe that other people are wrong.

THE RELATIVE STRENGTH INDEX, THE STOCHASTIC OSCILLATOR, & MOVING AVERAGES

"Things are getting complicated."

The main idea is to combine the three indicators into one trading system. By now, we already know how to code the RSI, the Stochastic Oscillator, and moving averages. All we need is objective conditions:

- For a long (Buy) position, the 3-period Relative Strength Index must be below 20 while the 6-period Stochastic Oscillator is above its 3-period Moving Average, and the market price must be above its 150-period Moving Average.
- For a short (Sell) position, the 3-period Relative Strength Index must be above 80 while the 6-period Stochastic Oscillator is below its 3-period Moving Average, and the market price must be below its 150-period Moving Average.



```
def signal(Data, rsi_column, stochastic_column, stochastic_ma_column, ma_column, buy, sell):
   for i in range(len(Data)):
      if Data[i, rsi_column] < lower_barrier and Data[i,      stochastic_column] > Data[i, stochastic_ma_column] and Data[i - 1, stochastic_column] < Data[i - 1, stochastic_ma_column] and Data[i, 3] > Data[i, ma_column]:
            Data[i, buy] = 1
      elif Data[i, rsi_column] > upper_barrier and Data[i, stochastic_column] < Data[i, stochastic_ma_column] and Data[i - 1, stochastic_column] > Data[i - 1, stochastic_ma_column] and Data[i, 3] < Data[i, ma_column]:
         Data[i, sell] = -1
   return Data
```

 # The Data variable refers to the OHLC data with the calculated indicators

 # The rsi_column variable is the index of the column where the RSI is stored

 # The stochastic_column variable is the index of the column where the Stochastic Oscillator is stored

 # The stochastic_ma_column variable is the index of the column where the Moving Average of the Stochastic Oscillator is stored

 # The Closing price is indexed at 3

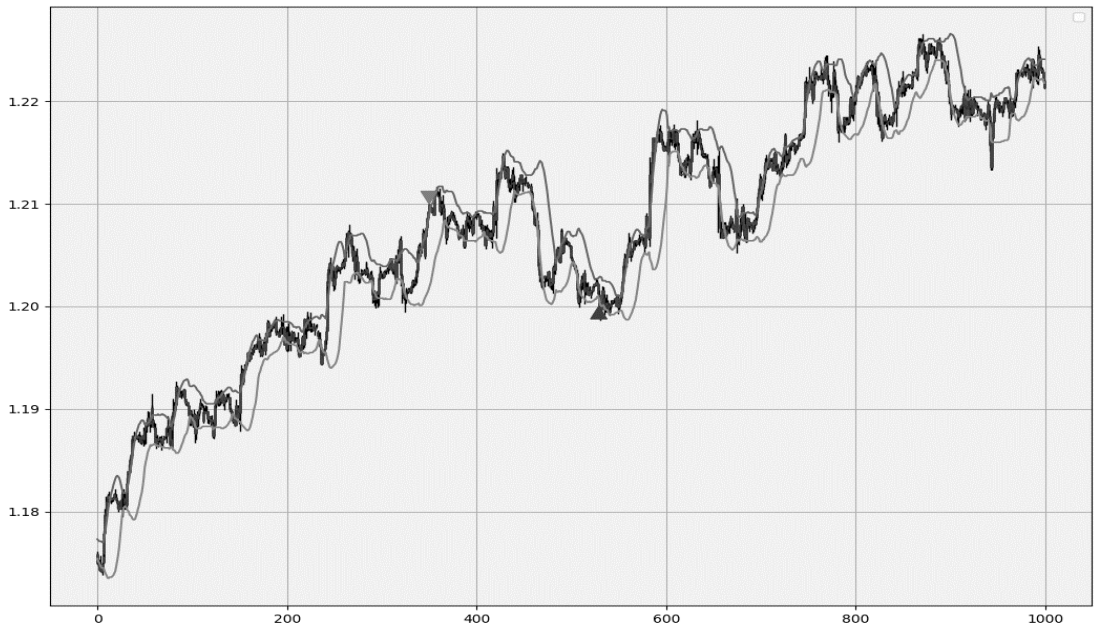
THE BOLLINGER BANDS & PSYCHOLOGICAL LEVELS

"Psychologically and Bollingerly insane."

The main idea is to combine statistical extreme with psychological round levels so that the conviction is increased by incorporating two unrelated fields together. Bollinger Bands are known to be widely used while psychological levels are known to be very useful to scalpers and market makers as well as other traders that place their stops/targets at memorable places.

The conditions of the strategy are as follows:

- Go long (Buy) whenever the market is at or below its lower Bollinger Band while simultaneously being on a psychological level (i.e. round level). Hold the position until getting a contrarian signal or until getting stopped out by the risk management system.
- Go short (Sell) whenever the market is at or above its upper Bollinger Band while simultaneously being on a psychological level (i.e. round level). Hold the position until getting a contrarian signal or until getting stopped out by the risk management system.



The previous chart shows the 20-period default Bollinger Bands and the signals generated close to psychological levels.

```
def signal(Data, upper_bollinger_column, lower_bollinger_column,
psychological_level_signal_column, buy, sell):
    Data = adder(Data, 10)
    for i in range(len(Data)):
        if Data[i, 3] <= Data[i, lower_bollinger_column] and Data[i,
psychological_level_signal_column] == 1:
            Data[i, buy] = 1
        elif Data[i, 3] >= Data[i, upper_bollinger_column] and Data[i,
psychological_level_signal_column] == 1:
            Data[i, sell] = -1
    return Data
```

The conclusion of this is that we can tweak the Bollinger Bands so that they add a confirmation factor to the psychological levels technique. This is helpful in knowing whether to initiate a long or short position when encountering such levels.

THE PARABOLIC SAR & MOVING AVERAGES

"The Parabolic what now?"

The Parabolic stop-and-reverse is an interesting indicator created by Welles Wilder who is also the creator of the famous RSI. This indicator is mostly used as a trailing stop that tracks the trend as it develops but there is no harm in testing it as a trading strategy. It is worth noting that it performs relatively well in steady trends but just as any other indicator, it has its weakness, in this case, ranging markets.

I will refer to a python library called talib from where the user could import the sar function that uses a data frame and calculates the values. Having modified that function, you can refer to the one just after (I do not take credit for it as I merely just changed some lines as opposed to the other functions which have been coded by me):

```
def sar(s, af = 0.02, amax = 0.2):
    high, low = s.high, s.low
    # Starting values
    sig0, xpt0, af0 = True, high[0], af
    sar = [low[0] - (high - low).std()]
    for i in range(1, len(s)):
        sig1, xpt1, af1 = sig0, xpt0, af0
        lmin = min(low[i - 1], low[i])
        lmax = max(high[i - 1], high[i])
        if sig1:
            sig0 = low[i] > sar[-1]
            xpt0 = max(lmax, xpt1)
        else:
            sig0 = high[i] >= sar[-1]
            xpt0 = min(lmin, xpt1)
        if sig0 == sig1:
            sari = sar[-1] + (xpt1 - sar[-1])*af1
            af0 = min(amax, af1 + af)
        if sig0:
                af0 = af0 if xpt0 > xpt1 else af1
                sari = min(sari, lmin)
            else:
                af0 = af0 if xpt0 < xpt1 else af1
                sari = max(sari, lmax)
        else:
            af0 = af
            sari = xpt0sar.append(sari)
        return sar
```

The basic understanding is that when the Parabolic SAR (the dots around the market price) is under the current price, then the outlook is bullish and when it is above the current price, then the outlook is bearish.

To add the Parabolic SAR to your OHLC array (preferably numpy), use the following steps:

```
importing pandas as pd
# Converting to a pandas Data frame
my_data = pd.DataFrame(my_data)
# Renaming columns to fit the function
my_data.columns = ['open','high','low','close']
# Calculating the Parabolic SAR
Parabolic = sar(my_data, 0.02, 0.2)
# Converting the Parabolic values back to an array
Parabolic = np.array(Parabolic)
# Reshaping
Parabolic = np.reshape(Parabolic, (-1, 1))
# Concatenating with the OHLC Data
my_data = np.concatenate((my_data, Parabolic), axis = 1)
```

The conditions we will be using for this strategy are:

- Go long (Buy) whenever the Parabolic SAR turns below the market price while simultaneously the market price is above and close to the 300-period moving average.
- Go short (Buy) whenever the Parabolic SAR turns above the market price while simultaneously the market price is below and close to the 300-period moving average.

Let us first be sure that we have the necessary array with the two indicators already calculated before we introduce the signal function that initiates the buy and sell orders based on the above conditions.

```
# The Parabolic SAR has been calculated in the above part which should give us an array of OHLC data and the indicator

# Adding three columns for the moving average and the buy/sell columns

my_data = adder(my_data, 3)

# Calculating the 300-period moving average

my_data = ma(my_data, 300, 3, 5)
```

Now, we are ready to write the signal function as mentioned above:

```
def signal(Data, close, psar, ma_column, threshold, buy, sell):

  for i in range(len(Data)):

    if Data[i, close] > Data[i, psar] and Data[i - 1, close] < Data[i - 1, psar] and Data[i, close] > Data[i, ma_column] and (Data[i, close] - Data[i, ma_column]) < threshold:

Data[i, buy] = 1

    if Data[i, close] < Data[i, psar] and Data[i - 1, close] > Data[i - 1, psar] and Data[i, close] < Data[i, ma_column] and (Data[i, ma_column] - Data[i, close]) < threshold:

Data[i, sell] = -1

# Defining the variables

threshold  = 0.0040

# Using the function to generate the trades

signal(my_data, 3, 4, 5, threshold, 6, 7)
```

The threshold variable is the minimum distance of the market price to its moving average to be able to meet the condition. This means that if we have a bullish flip on the Parabolic SAR with a market price that is close to its moving average and still above it (distance must be less than 40 pips), then the bullish signal is generated.

The above shows the EURGBP chart with the 300-period moving average alongside the Parabolic SAR. Notice that in a downtrend, the market remains below its moving average and while approaching it, we await the signal from the Parabolic SAR to enter into a short trade as shown by the red arrows.

The strategy uses insights from reversal trading to initiate trend-following signals which can provide extended moves. The risk management used should be based on the trader's choice as it depends on many subjective factors like the risk appetite and tolerance.


Parabolic SAR
300-period Moving Average
0.95
0.94
0.93
0.92
0.91
0.90
0.89
0
200
400
600
800
1000

CONCLUSION

Once again, the aim of the book was not to give you a golden egg to be applied directly and make you rich. Giant hedge funds with extremely smart people are underperforming the markets, therefore, you should not expect to never lose money while entering the trading realm. This is the most chaotic and random environment you will experience as irrationality looms around every corner. Whenever you think you are sure of a certain direction, the market will make sure you know your place. What to do then? The best thing is to develop a robust strategy based on solid back-testing results, combine it with superior risk management techniques that will make sure you remain in the game, and of course, practice the lost art of position sizing which can be based either on convictions or on historical success rates. Only then, will you have a chance to survive and live to fight another day. Do not fall into the trap of scammers, this is one of the biggest pitfalls new traders fall into.

A scam is a fraudulent attempt to steal your money without giving you something in return of equal money. For example, I come to you asking for $1 USD in exchange of giving it back 2$ next week only to realize seven days later that I have vanished with your Dollar. Scams come in many shapes and forms which are discussed below. These should form a basic barrier before deciding to trade.

Important Disclaimer: There is a huge number of honest and talented people in the trading industry which are role models. The below is just a warning from other people who are less honest. Consider the below as warning points that you should think about before following or trusting a manager with your money.

- **The Guaranteed Returns Scam**

Nothing is guaranteed in life and especially not in finance. Trading and investing are a complex game in a highly complex and chaotic environment. Even risk-free securities are not risk-free, they are just virtually and theoretically risk-free. Remember, this is a place where Oil futures prices went negative for a while, therefore, even when you buy

at $0.00, you must not be sure that the asset's price will go up. This is an outstretched example, but it should illustrate the wilderness that is financial markets.

The guaranteed returns scam is used to access other people's funds easily by selling them dreams. Basically, someone will contact you stating that he has a guaranteed win strategy and wants you to invest your own funds so that you become rich just like the ones that have already invested with him. What you should know here is that the truth is very far from all of this. There is no strategy, there are no returns, and above all, there is no integrity. To detect this kind of specimen, look out for the following:

- They generally do not have a profile picture and their job record says Forex Trader since 2001.
- They post the same comment everywhere with a link to something you should not open.
- They promise you either to double your money or to give you 1% increase per day. Both are impossible and if they were possible, you would be a billionaire in a very short time.
- They insist. A LOT.

Rocket scientists with as many PhD's as Taylor Swift's ex-boyfriends cannot reach the accuracies claimed by these people who have magically chosen you to invest your money with them. Think about that before falling into their trap.

- **The Signals Scam**

This is where you must pay a subscription to receive random signals that will have a track record very different from the one you have given prior to your subscription. It starts with the scammer posting fake track records and false profit statements that a 5-year-old can photoshop. Then, when people start subscribing to this signal provider, suddenly, the market becomes too random to be predicted and the results go south.

- **The Profiles that Publish Profit/Gain Statement on Social Media**

  Who among us has not scrolled on LinkedIn or Twitter and seen meaningless screenshots of trading results with generally immature captions like "easy day" or "subscribe for only 4$ / month to become rich"? It is extremely easy to fake a profitable MetaTrader profit and loss statements. There are two ways:

  - Simply photoshopping and aligning the good numbers.
  - Opening a demo account where the spreads are 0.1, then buying for a few minutes and closing out with an easy profit as soon as the price slightly moves above the buying price.

  The second choice is so easy that it is ridiculous. I have managed to get 25 profitable trades in a row. This is of course not possible with real life accounts due to many factors especially high costs and latency.

  To detect this kind of scams, look at the opening and closing times of the trades, also look for the lack of diversity between the trades to get a clue on the fake statement. Weird position sizing should give you a hint as well.

  "Your hard-earned money deserves to stay in your pocket or in a valid investment that you deem worthy."

- **Your Broker, The Back-Stabbing Friend**

  Here is a simple way of putting it. Most brokers are using a shady technique called B-Booking. This is where they bet against you. The quick definition is:

  - A-Booking: These are brokers that send your orders to the market. They make money based on commissions or spreads. They do not care whether you win or lose. In other words, they are honest brokers.
  - B-Booking: These are brokers that keep your orders internally. They trade against you and control everything; thus, they make a lot of money betting against you. Your orders never even reach the market.

Your broker is not your friend. The broker's job is to make money on commissions and on bid/ask spreads and since the FX market is known to be commission-free, almost all the profits come from the bid-ask spread, but also one more thing; your broker bets against you because when you take one position, your broker is taking the other one and therefore, will do everything for you to lose. This is sadly the case for most brokers.

Although not technically a scam, but a broker that provides you with research even though he is rooting against you, is a giant red flag. Do not trust anyone but your judgement. Also, Notice the irony when a B-Book broker is giving you free trading signals.

- **The Free Trading Scam**

  Have you ever seen those big flashy ads with 0% commissions and fees? They generally have two flashy buy and sell buttons as well to make you think you are all set to trade risk-free and fee-free. Well, this is another deception made by shady unregulated brokers.

  Understand this phrase: An unregulated broker is a scammer, and you will not get your money back if you create an account with them. Therefore, anything based outside of compliance protected zones such as the United States and Australia is a major no-no. Even if by miracle, you manage to get your money back from them, there will be enormous costs and banking problems such as delays.

  Generally, good respectable brokers are regulated by government entities such as the SEC in the United States and the FSA in the United Kingdom as well as being A-Booking brokers.

  One more thing; I am not really referring to Robinhood when I say free trading. Robinhood is advertising a commission-free (but you still must pay bid-ask spreads) trading which is of course a little misleading. However, some brokers will not even mention their huge spreads hidden behind the commission-free trading.

- **The Holy Grail Strategy**

  Many aspiring traders get scammed because they are trying to find someone who offers them infallible money management where they use some sort of perfect strategy. You will sometimes see 100% hit ratio strategies published on forums and websites. It makes you wonder, how are these people living with us in this world?

  Let us get one thing clear; financial markets are random-like chaotic environment. This means that there is no one strategy that is able to capture such complexity. Every time you see one of those holy grail strategies, you have to disregard them. No one will put this hypothetically perfect strategy online for everyone to use. Personally, if I had one, I would use it myself and never tell anyone about it.

  I would also not brag about it on social media as that kind of behavior is destroying our society. I have once seen a post of a guy selling a robo-trader with the following comment:

  "I was told it is good but have not tested it, anyway, selling it for $300"

  You read that right, he was told. He is selling something he has absolutely no idea about. The robo-trader had an image with the word Accuracy 98% taking up most of the space.

  To be profitable in trading, you can do it with 30% hit ratio or what they call accuracy. However, let me show you something. What if I told you that 90–98% accuracy is possible? How would that sound considering all the crusade I have led earlier against this kind of stupid statements?

  Well, if I do a simple back-test of an average strategy but tweak the risk parameter so that it takes profit at a fraction of the stop I place? In other word, I use a risk-reward ratio of say 0.01. This will give me a whopping accuracy of above 90% but I will still lose money because the magnitude of the rare losses will outweigh the frequent small gains. This is like picking up pennies in front of a steamroller, only, you have no way to escape the tracks and there are many steamrollers coming.

- **Email Scams**

  Some scammers like to send emails to appear professional. On multiple occasions, I have received mails with professional signatures stating that they have chosen me to be one of the first investors in their robo-advisor that has solved the mystery of the FX market. All I had to do is wire them money. Simple, right?

  And be careful of the "This man made $1,000,000 trading FX overnight" advertisements. Are they really taking you for a fool?

- **Instagram Traders, The Picture Billionaires**

  Have you ever thought that if they did have a trading strategy that is that accurate, why would they spend their time trying to sell it when they can just put it to use and make themselves rich? The short answer is that they are trying to scam you into believing that it works, and they will provide all sort of "proofs" and "reliable" track records to get to your money.

  Now, what about the ones that always pose next to a Lamborghini or a Ferrari and imply that it was their trading that made them this amount of money?

  I used to know a guy who was extremely unlucky with his trading and spent years with zero to negative return. One time, I was checking his Instagram profile and could not recognize him. The number of fancy cars and fancy places that were implied to be his was enormous. When I asked him where did that come from? He said, nowhere, they were cars in the streets he came across with and just posed next to them. Now, I know that some guys need to satisfy their egos by showing superiority and impressing people, but this is called deceiving people.

  One last thing about Instagram. Each time I publish a post even unrelated to trading, I get comments such as "I owe it to Mr. X for making me $65,000 in Bitcoin, here is his profile link". This is why I deleted the app.

- **How to avoid the Noise?**

  Trading has the potential to be a valuable source of income as long as we are careful. Here is how to avoid scams:

  Trust that you are the only one who is in charge of making you money. No one will be willing to do that for you. Understand that the FX market is too complicated for someone to offer you the holy grail. Know the limitations of gains and losses and practice good risk management. This means, never risk more than what you expect to gain on a trade. Make sure that everyone parading around saying he got rich from trading is overcompensating for something by lying. At the end, you can extract a lot of potential from trading and investing but only by being patient, rigorous, disciplined, and diversified.

APPENDIX I

PLOTTING CANDLESTICKS CHARTS IN PYTHON

Candlestick charts are among the most famous ways to analyze the time series visually. They contain more information than a simple line chart and have more visual interpretability than bar charts. Many libraries in Python offer charting functions but being someone who suffers from malfunctioning import of libraries and functions alongside their fogginess, I have created my own simple function that charts candlesticks manually with no exogenous help needed.

OHLC data is an abbreviation for Open, High, Low, and Close price. They are the four main ingredients for a timestamp. It is always better to have these four values together so that our analysis reflects more the reality. Here is a table that summarizes the OHLC data of hypothetical security:

Time Stamp	Open	High	Low	Close
01/01/2021	1.4294	1.4297	1.4272	1.4285
02/01/2021	1.4285	1.4299	1.4282	1.4290
03/01/2021	1.4289	1.4308	1.4287	1.4300
04/01/2021	1.4300	1.4309	1.4280	1.4295
05/01/2021	1.4295	1.4307	1.4269	1.4303
06/01/2021	1.4303	1.4338	1.4300	1.4337
07/01/2021	1.4337	1.4395	1.4336	1.4395

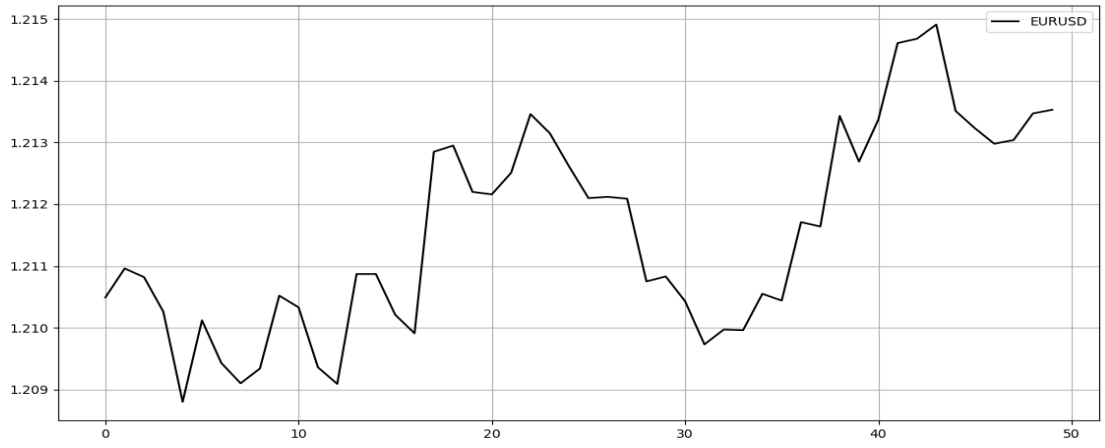
Our job now is to plot the data so that we can visually interpret what kind of trend is the price following. We will start with the basic line plot before we move on to candlestick plotting.

Note that you can download the data manually or using Python. In case you have an excel file that has OHLC only data starting from the first row and column, you can import it using the below code snippet:

```
import numpy as np
import pandas as pd
# Importing the Data
my_ohlc_data = pd.read_excel('my_ohlc_data.xlsx')
# Converting to Array
my_ohlc_data = np.array(my_ohlc_data)
```

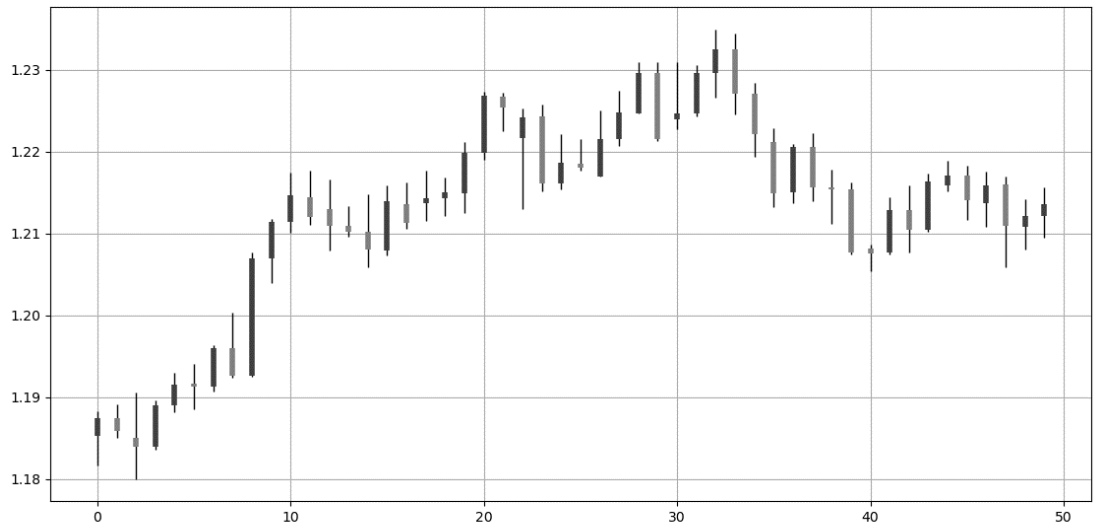
Plotting basic line plots is extremely easy in Python and requires only one line of code. We have to make sure that we have imported a library called matplotlib and then we will call a function that plots the data for us.

```
# Importing the necessary charting library
import matplotlib.pyplot as plt
# The syntax to plot a line chart
plt.plot(my_ohlc_data, color = 'black', label = 'EURUSD')
# The syntax to add the label created above
plt.legend()
# The syntax to add a grid
plt.grid()
```

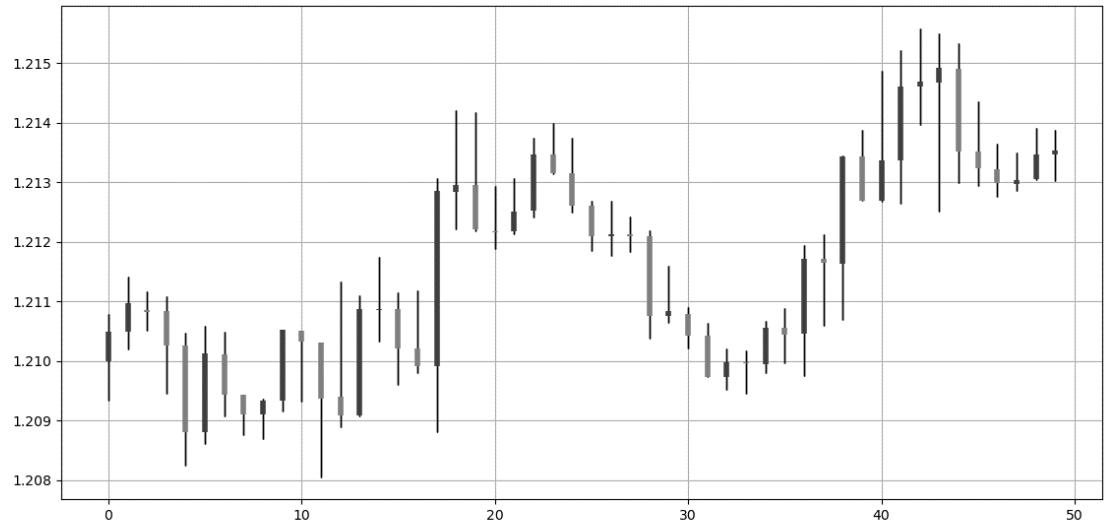


Now that we have seen how to create normal line charts, it is time to take it to the next level with candlestick charts. The way to do this with no complications is to think about vertical lines. Here is the intuition (followed by an application of the function below):

- Select a lookback period. This is the number of values you want to appear on the chart.
- Plot vertical lines for each row representing the highs and lows. For example, on OHLC data, we will use a matplotlib function called vlines which plots a vertical line on the chart using a minimum (low) value and a maximum (high value).
- Make a color condition which states that if the closing price is greater than the opening price, then execute the selected block of code (which naturally contains the color green). Do this with the color red (bearish candle) and the color black (Doji candle).
- Plot vertical lines using the conditions with the min and max values representing closing prices and opening prices. Make sure to make the line's width extra big so that the body of the candle appears sufficiently enough that the chart is deemed a candlestick chart.



```
def ohlc_plot(Data, window, name):
    Chosen = Data[-window:, ]
    for i in range(len(Chosen)):
      plt.vlines(x = i, ymin = Chosen[i, 2], ymax = Chosen[i, 1], color = 'black', linewidth = 1)
      if Chosen[i, 3] > Chosen[i, 0]:
          color_chosen = 'green'
          plt.vlines(x = i, ymin = Chosen[i, 0], ymax = Chosen[i, 3], color = color_chosen, linewidth = 4)
      if Chosen[i, 3] < Chosen[i, 0]:
          color_chosen = 'red'
          plt.vlines(x = i, ymin = Chosen[i, 3], ymax = Chosen[i, 0], color = color_chosen, linewidth = 4)
      if Chosen[i, 3] == Chosen[i, 0]:
          color_chosen = 'black'
          plt.vlines(x = i, ymin = Chosen[i, 3], ymax = Chosen[i, 0], color = color_chosen, linewidth = 4)
    plt.grid()
    plt.title(name)
# Using the function
ohlc_plot(my_ohlc_data, 50, '')
```



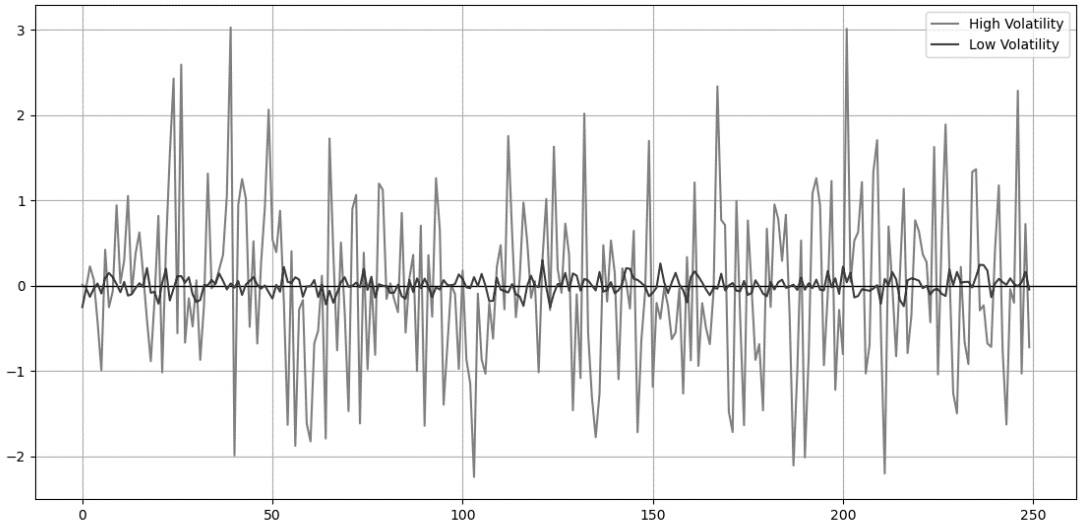
APPENDIX II

THE AVERAGE TRUE RANGE

Trading is a combination of four things, research, implementation, risk management, and post-trade evaluation. The bulk of what we spend our time doing is the first two, meaning that we spend the vast majority of the time searching for a profitable strategy and implementing it (i.e. trading). However, we forget that the pillar of trading is not losing money. It is even more important than gaining money because it is fine to spend time trading and still have the same capital or slightly less than to spend time trading and find yourself wiped out. We will discuss the third pillar as a way of enhancing returns and capital protection. Every trading strategy must be accompanied by its own personalized risk management protocol. One such protocol (or indicator) is the famous Average True Range.

To understand the Average True Range, we must first understand the concept of Volatility. It is a key concept in finance, whoever masters it holds a tremendous edge in the markets.

Unfortunately, we cannot always measure and predict it with accuracy. Even though the concept is more important in options trading, we need it pretty much everywhere else. Traders cannot trade without volatility nor manage their positions and risk. Quantitative analysts and risk managers require volatility to be able to do their work. Before we discuss the different types of volatility, why not look at a graph that sums up the concept?



You can code the above in Python yourself using the following snippet:

```
# Importing the necessary libraries
import numpy as np
import matplotlib.pyplot as plt
# Creating high volatility noise
hv_noise = np.random.normal(0, 1, 250)
# Creating low volatility noise
lv_noise = np.random.normal(0, 0.1, 250)
# Plotting
plt.plot(hv_noise, color = 'red', linewidth = 1.5, label = 'High Volatility')
plt.plot(lv_noise, color = 'green', linewidth = 1.5, label = 'Low Volatility')
plt.axhline(y = 0, color = 'black', linewidth = 1)
plt.grid()
plt.legend()
```

The different types of volatility around us can be summed up in the following:

- Historical volatility: It is the realized volatility over a certain period of time. Even though it is backward looking, historical volatility is used more often than not as an expectation of future volatility. One example of a historical measure is the standard deviation, which we will see later. Another example is the Average True Range..
- Implied volatility: In its simplest definition, implied volatility is the measure that when inputted into the Black-Scholes equation, gives out the option's market price. It is considered as the expected future actual volatility by market participants. It has one time scale, the option's expiration.
- Forward volatility: It is the volatility over a specific period in the future.
- Actual volatility: It is the amount of volatility at any given time. Also known as local volatility, this measure is hard to calculate and has no time scale.

The most basic type of volatility is our old friend "the Standard Deviation". It is one of the pillars of descriptive statistics and an important element in some technical indicators (such as the Bollinger Bands). But first let us define what variance is before we find Standard Deviation. Variance is the squared deviations from the mean (a dispersion measure), we take the square deviations so as to force the distance from the mean to be non-negative, finally we take the square root to make the measure have the same units as the mean, in a way we are comparing apples to apples (mean to standard deviation standard deviation). Variance is calculated through this formula:

$$\sigma^2 = \frac{\sum(x - \bar{x})^2}{n - 1}$$

Following our logic, standard deviation is therefore:

$$\sigma = \sqrt{\frac{\sum(x - \bar{x})^2}{n - 1}}$$

Therefore, if we want to understand the concept in layman's terms, we can say that Standard Deviation is the average distance away from the mean that we expect to find when we analyze the different components of the time series. Let us now keep the concept of being away from the mean in our heads and move away to the concept of the Average True Range.

In technical analysis, an indicator called the Average True Range -ATR- can be used as a gauge for historical volatility. Although it is considered as a lagging indicator, it gives some insights as to where volatility is now and where has it been last period (day, week, month, etc.). But first, we should understand how the True Range is calculated (the ATR is just the average of that calculation). Consider an OHLC data composed of an timely arrange Open, High, Low, and Close prices. For each time period (bar), the true range is simply the greatest of the three price differences:

- High - Low
- High - Previous close
- Previous close - Low

Once we have got the maximum out of the above three, we simply take a smoothed average of n periods of the true ranges to get the Average True Range. Generally, since in periods of panic and price depreciation we see volatility go up, the ATR will most likely trend higher during these periods, similarly in times of steady uptrends or downtrends, the ATR will tend to go lower. One should always remember that this indicator is very lagging and therefore has to be used with extreme caution.

Since it has been created by Wilder Wiles, also the creator of the Relative Strength Index, it uses Wilder's own type of moving average, the smoothed kind. To simplify things, the smoothed moving average can be found through a simple transformation of the exponential moving average.

$$Smoothed\ Moving\ Average = (Exponential\ Moving\ Average\ x\ 2) - 1$$

The above formula means that a 100 smoothed moving average is the same thing as (100 x 2) -1 = 199 exponential moving average. While we are on that, we can code the exponential moving average using this function:

```
def atr(Data, lookback, high, low, close, where):
    # Adding the required columns
    Data = adder(Data, 1)
    # True Range Calculation
    for i in range(len(Data)):
        try:
            Data[i, where] =   max(Data[i, high] - Data[i, low],
                                   abs(Data[i, high] - Data[i - 1, close]),
                                   abs(Data[i, low] - Data[i - 1, close]))
        except ValueError:
            pass
    Data[0, where] = 0
    # Average True Range Calculation
    Data = ema(Data, 2, lookback, where, where + 1)
    # Cleaning
    Data = deleter(Data, where, 1)
    Data = jump(Data, lookback)
    return Data
```

When I say I use ATR-based risk management system (Average True Range), it means that the algorithm will do the following steps with regards to the position it takes.

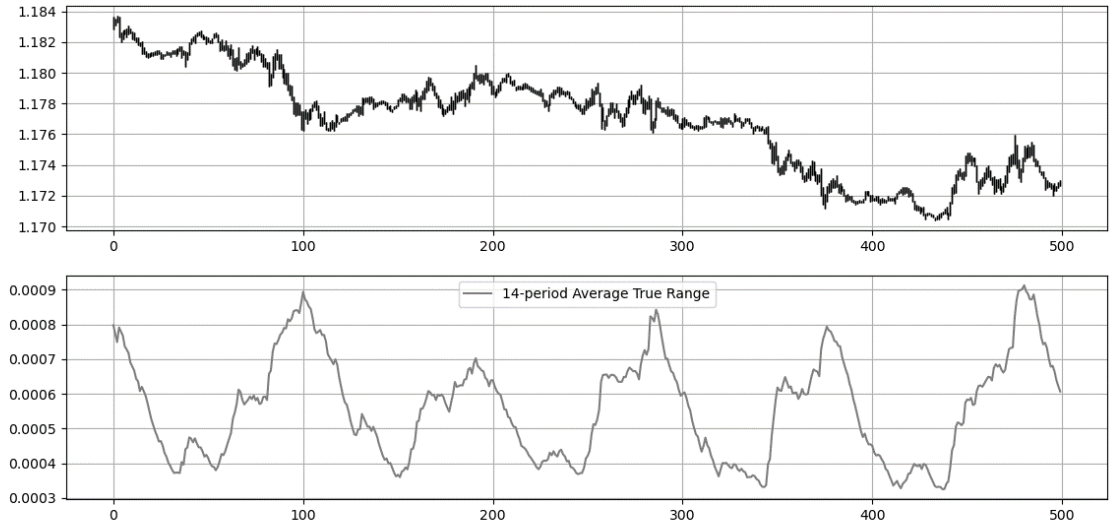
A long (Buy) position:

- The algorithm initiates a buy order after a signal has been generated following a certain strategy.
- Then, the algorithm will monitor the ticks and whenever the high equals a certain constant multiplied by ATR value at the time of the trade inception, an exit (at

profit) order is initiated. Simultaneously, if a low equals a certain constant multiplied by ATR value at the time of the trade inception is seen, an exit (at loss) is initiated. The exit encountered first is naturally the taken event.

A short (Sell) position:

- The algorithm initiates a short sell order after a signal has been generated following a certain strategy.
- Then, the algorithm will monitor the ticks and whenever the low equals a certain constant multiplied by ATR value at the time of the trade inception, an exit (at profit) order is initiated. Simultaneously, if a high equals a certain constant multiplied by ATR value at the time of the trade inception is seen, an exit (at loss) is initiated. The exit encountered first is naturally the taken event.



The plot above shows the Average True Range. Take a look at the latest value on the ATR. It is around 0.0006 (6 pips). If we initiate a buy order following a simple 2 risk-reward ratio (risking half of what we expect to gain), we can place an order this way:

- Buy at current market price.
- Take profit at current market price + (2 x 6 pips).
- Stop the position at current market price - (1 x 6 pips).

The Average True Range Indicator is a key pillar when dealing with volatility. It is one way to measure it and is followed by many market participants thus, giving it more strength. The biggest utility of the indicator is as described above, risk management and placing stops/targets. You can set your expected (theoretical) risk-reward ratio from your trades by setting the parameters on the indicator. For example, the multiplication example we have seen should give us a theoretical risk-reward ratio of 2.00.