

Basics of financial analysis and quantitative trading with Python.

Finance represents a system of capital, business models, investments, and other financial instruments. A very important sector of finance is trading. You can trade financial securities, equities, or tangible products like gold or oil.

Algorithmic or Quantitative trading is the process of designing and developing trading strategies based on mathematical and statistical analyses. It is an immensely sophisticated area of finance.

This tutorial serves as the beginner's guide to quantitative trading with Python. You'll find this post very helpful if you are:

1. A student or someone aiming to become a quantitative analyst (quant) at a fund or bank.

2. Someone who is planning to start your own quantitative trading business.

We'll go over the following topics in this post:

link to external sites

1. [Basics of stocks and trading](#)
2. [Extracting data from Quandl API](#)
3. [Exploratory data analysis on stock pricing data](#)
4. [Moving averages](#)
5. [Formulating a trading strategy with Python](#)
6. [Visualizing the performance of the strategy](#)

Before we deep dive into the details and dynamics of stock pricing data, we must first understand the basics of finance. If you are someone who is familiar with finance and how trading works, you can skip this section and [click here to go to the next one](#).

What Are Stocks? What is Stock Trading?

Stocks

A stock is a representation of a share in the ownership of a corporation, which is issued at a certain amount. It is a type of financial security that establishes your claim on a company's assets and performance. An organization or company issues stocks to raise more funds/capital in order to scale and engage in more projects. These stocks are then publicly available and are sold and bought.

Stock Trading and Trading Strategy

The process of buying and selling existing and previously issued stocks is called stock trading. There is a price at which a stock can be bought and sold, and this keeps on fluctuating depending upon the demand and the supply in the share market. Depending on the company's performance and actions, stock prices may move up and down, but the stock price movement is not limited to the company's performance.

Traders pay money in return for ownership within a company in hopes to make some profitable trades and sell the stocks at a higher price.

Another important technique that traders follow is short selling — borrowing shares and immediately selling them in the hope of buying them up later at a lower price, return them to the lender, and make the margin. So, most traders follow a plan and model to trade. This is known as a trading strategy.

Quantitative traders at hedge funds and investment banks design and develop these trading strategies and frameworks to test them. It requires profound programming expertise and an understanding of the languages needed to build your own strategy.

Python is one of the most [popular programming languages](#) used, among the likes of C++, Java, R, and MATLAB. It is being adopted widely across all domains, especially in data science, because of its easy syntax, huge community, and third-party support.

You'll need familiarity with Python and statistics in order to make the most of this tutorial. Make sure to brush up on your Python and check out the [fundamentals of statistics](#).

Extracting data from the Quandl API

In order to extract stock pricing data, we'll be using the [Quandl API](#).

But before that, let's set up the work environment. Here's how:

1. In your terminal, create a new directory for the project (name it however you want):

```
mkdir <directory_name>
```

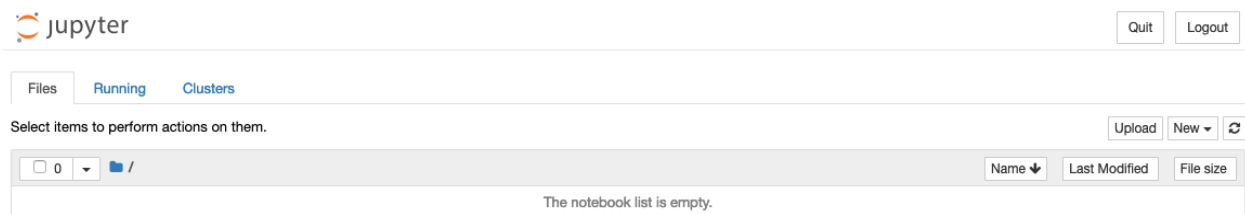
1. Make sure you have [Python 3](#) and [virtualenv](#) installed on your machine.
2. Create a new Python 3 virtualenv using `virtualenv`

`<env_name>` and activate it using `source`

```
<env_name>/bin/activate.
```

3. Now, install jupyter-notebook using [pip](#), and type in `pip install jupyter-notebook` in the terminal.
4. Similarly, install the `pandas`, `quandl`, and `numpy` packages.
5. Run your `jupyter-notebook` from the terminal.

Now, your notebook should be running on localhost like the screenshot below:



You can create your first notebook by clicking on the `New` dropdown on the right. Make sure you have created an account on [Quandl](#). Follow the steps mentioned [here](#) to create your API key.

Once you're all set, let's dive right in:

```
# importing required packagesimport pandas as pd
```

```
import quandl as q
```

Pandas is going to be the most rigorously used package in this tutorial as we'll be doing a lot of data manipulation and plotting.

After the packages are imported, we will make requests to the Quandl API by using the Quandl package:

```
# set the API key
```

```
q.ApiConfig.api_key = "<API key>"#send a get request to query Microsoft's  
end of day stock prices from 1st Jan, 2010 to 1st Jan, 2019
```

```
msft_data = q.get("EOD/MSFT", start_date="2010-01-01",  
end_date="2019-01-01")# look at the first 5 rows of the dataframe
```

```
msft_data.head()
```

Out[3]:

	Open	High	Low	Close	Volume	Dividend	Split	Adj_Open	Adj_High	Adj_Low	Adj_Close	Adj_Volume
Date												
2010-01-04	30.62	31.10	30.59	30.950	38409100.0	0.0	1.0	24.356481	24.738294	24.332618	24.618978	38409100.0
2010-01-05	30.85	31.10	30.64	30.960	49749600.0	0.0	1.0	24.539433	24.738294	24.372390	24.626932	49749600.0
2010-01-06	30.88	31.08	30.52	30.770	58182400.0	0.0	1.0	24.563297	24.722385	24.276937	24.475798	58182400.0
2010-01-07	30.63	30.70	30.19	30.452	50559700.0	0.0	1.0	24.364436	24.420117	24.014440	24.222847	50559700.0
2010-01-08	30.28	30.88	30.24	30.660	51197400.0	0.0	1.0	24.086030	24.563297	24.054213	24.388299	51197400.0

Here we have Microsoft's EOD stock pricing data of the last 9 years.

All you had to do was call the `get` method from the Quandl package and supply the stock symbol, MSFT, and the timeframe for the data you need.

This was really simple, right? Let's move ahead to understand and explore this data further.

Exploratory Data Analysis on Stock Pricing Data

With the data in our hands, the first thing we should do is understand what it represents and what kind of information does it encapsulate.

Printing the DataFrame's info, we can see all that it contains:

```
In [5]: msft_data.info()

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2264 entries, 2010-01-04 to 2018-12-31
Data columns (total 12 columns):
Open                2264 non-null float64
High                2264 non-null float64
Low                 2264 non-null float64
Close               2264 non-null float64
Volume              2264 non-null float64
Dividend            2264 non-null float64
Split               2264 non-null float64
Adj_Open            2264 non-null float64
Adj_High            2264 non-null float64
Adj_Low             2264 non-null float64
Adj_Close           2264 non-null float64
Adj_Volume          2264 non-null float64
dtypes: float64(12)
memory usage: 229.9 KB
```

As seen in the screenshot above, the DataFrame contains DatetimeIndex, which means we're dealing with time-series data. An index can be thought of as a data structure that helps to modify or reference the data. Time-series data is a sequence of snapshots of prices taken at consecutive, equally spaced intervals of time.

In trading, EOD stock pricing data captures the movement of certain parameters about a stock, such as the stock price, over a specified period of time with data points recorded at regular intervals.

Important Terminology

Looking at other columns, let's try to understand what each column represents:

- Open/Close — Captures the opening/closing price of the stock
- Adj_Open/Adj_Close — An adjusted opening/closing price is a stock's price on any given day of trading that has been revised to include any dividend distributions, stock splits, and other corporate actions that occurred at any time before the next day's open.
- Volume — It records the number of shares that are being traded on any given day of trading.
- High/Low — It tracks the highest and the lowest price of the stock during a particular day of trading.

These are the important columns that we will focus on at this point in time.

We can learn about the summary statistics of the data, which shows us the number of rows, mean, max, standard deviations, etc. Try running the following line of code in the Ipython cell:

```
msft_data.describe()
```

In [22]:

msft_data.describe()

Out[22]:

	Open	High	Low	Close	Volume	Dividend	Split	Adj_Open	Adj_High	Adj_Low	Adj_Close	Adj_Vol
count	2264.000000	2264.000000	2264.000000	2264.000000	2.264000e+03	2264.000000	2264.0	2264.000000	2264.000000	2264.000000	2264.000000	2.264000e+03
mean	48.069969	48.480070	47.635572	48.078491	4.163288e+07	0.004527	1.0	44.361259	44.739290	43.959705	44.368526	4.163288e+07
std	23.917687	24.123013	23.658271	23.896640	2.349079e+07	0.037723	0.0	25.105301	25.321946	24.834611	25.084643	2.349079e+07
min	23.090000	23.320000	22.730000	23.010000	7.425603e+06	0.000000	1.0	18.534889	18.719515	18.245908	18.470671	7.425603e+06
25%	28.887500	29.190000	28.615000	28.860000	2.586054e+07	0.000000	1.0	24.083973	24.280747	23.902204	24.075855	2.586054e+07
50%	41.370000	41.675000	41.040000	41.470000	3.620061e+07	0.000000	1.0	37.279791	37.622506	37.058864	37.375225	3.620061e+07
75%	57.872500	58.066250	57.427500	57.890000	5.172612e+07	0.000000	1.0	54.962572	55.172362	54.608796	54.924585	5.172612e+07
max	115.420000	116.180000	114.930000	115.610000	3.193179e+08	0.460000	1.0	114.425180	115.178630	113.939403	114.613542	3.193179e+08

resample()

Pandas' resample() method is used to facilitate control and flexibility on the frequency conversion of the time series data. We can specify the time intervals to resample the data to monthly, quarterly, or yearly, and perform the required operation over it.

```
msft_data.resample('M').mean()
```

```
In [7]: msft_data.resample('M').mean().head()
```

```
Out[7]:
```

Date	Open	High	Low	Close	Volume	Dividend	Split	Adj_Open	Adj_High	Adj_Low	Adj_Close	Adj_Volume
2010-01-31	30.287895	30.584211	29.830526	30.146684	7.156057e+07	0.000000	1.0	24.092310	24.328013	23.728499	23.979985	7.156057e+07
2010-02-28	28.323684	28.563158	28.064737	28.356263	5.656017e+07	0.006842	1.0	22.579229	22.770107	22.372912	22.605267	5.656017e+07
2010-03-31	29.254674	29.477391	29.044348	29.258478	4.827118e+07	0.000000	1.0	23.377149	23.555120	23.209079	23.380189	4.827118e+07
2010-04-30	30.438571	30.749919	30.255205	30.522571	6.281093e+07	0.000000	1.0	24.323191	24.571986	24.176664	24.390314	6.281093e+07
2010-05-31	28.285000	28.616280	27.748750	28.116875	8.600651e+07	0.006500	1.0	22.646239	22.911497	22.216871	22.511523	8.600651e+07

This is an interesting way to analyze stock performance in different timeframes.

[Introduction to Time Series Forecasting of Stock Prices with Python | Data Driven Investor](#)

[In this simple tutorial, we will have a look at applying a time series model to stock prices. More specifically, a...](#)

www.datadriveninvestor.com

Calculating returns

A financial return is simply the money made or lost on an investment.

A return can be expressed nominally as the change in the amount of

an investment over time. A return can be calculated as the percentage derived from the ratio of profit to investment.

We have the `pct_change()` at our disposal for this purpose. Here is how you can calculate returns:

```
# Import numpy package

import numpy as np# assign `Adj Close` to `daily_close`

daily_close = msft_data[['Adj Close']]# returns as fractional change

daily_return = daily_close.pct_change()# replacing NA values with 0

daily_return.fillna(0, inplace=True)print(daily_return)
```

This will print the returns that the stock has been generating on daily basis. Multiplying the number by 100 will give you the percentage change.

The formula used in `pct_change()` is:

$$\text{Return} = \{(\text{Price at } t) - (\text{Price at } t-1)\} / \{\text{Price at } t-1\}$$

Now, to calculate monthly returns, all you need to do is:

```
mdata = msft_data.resample('M').apply(lambda x: x[-1])
```

```
monthly_return = mdata.pct_change()
```

After resampling the data to months (for business days), we can get the last day of trading in the month using the `apply()` function. `apply()` takes in a function and applies it to each and every row of the Pandas series. `lambda` function is an anonymous function in Python which can be defined without a name, and only takes expressions in the following format:

```
Lambda: expression
```

For example, `lambda x: x * 2` is a lambda function. Here, `x` is the argument and `x * 2` is the expression that gets evaluated and returned.

Moving Averages in Trading

The concept of moving averages is going to build the base for our momentum-based trading strategy. In finance, analysts often have to evaluate statistical metrics continually over a sliding window of time, which is called moving window calculations. Let's see how we can calculate the rolling mean over a window of 50 days, and slide the window by 1 day.

rolling()

This is the magical function which does the tricks for us:

```
# assigning adjusted closing prices to adj_prices
```

```
adj_price = msft_data['Adj_Close']# calculate the moving average
```

```
mav = adj_price.rolling(window=50).mean()# print the result
```

```
print(mav[-10:])
```

```
Date
2018-12-17    106.967802
2018-12-18    106.815042
2018-12-19    106.682086
2018-12-20    106.477759
2018-12-21    106.329061
2018-12-24    106.103671
2018-12-26    105.933761
2018-12-27    105.815260
2018-12-28    105.613611
2018-12-31    105.441212
Name: Adj_Close, dtype: float64
```

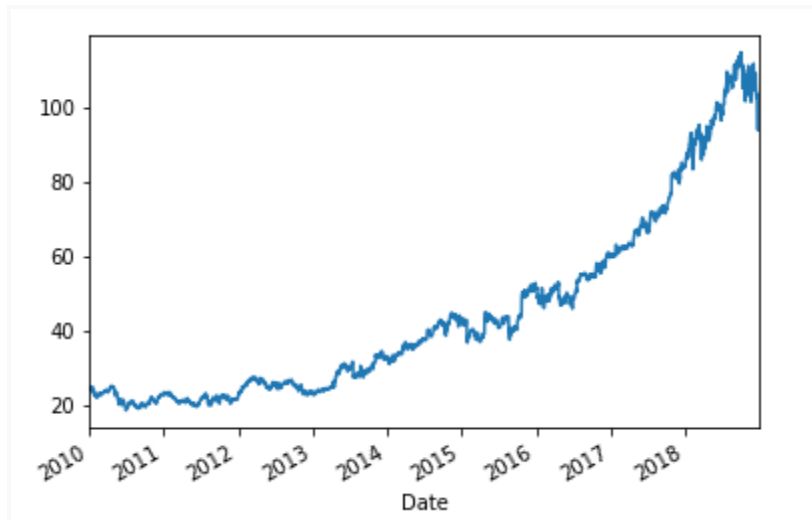
You'll see the rolling mean over a window of 50 days (approx. 2 months). Moving averages help smooth out any fluctuations or spikes in the data, and gives you a smoother curve for the performance of the company.

We can plot and see the difference:

```
# import the matplotlib package to see the plot
```

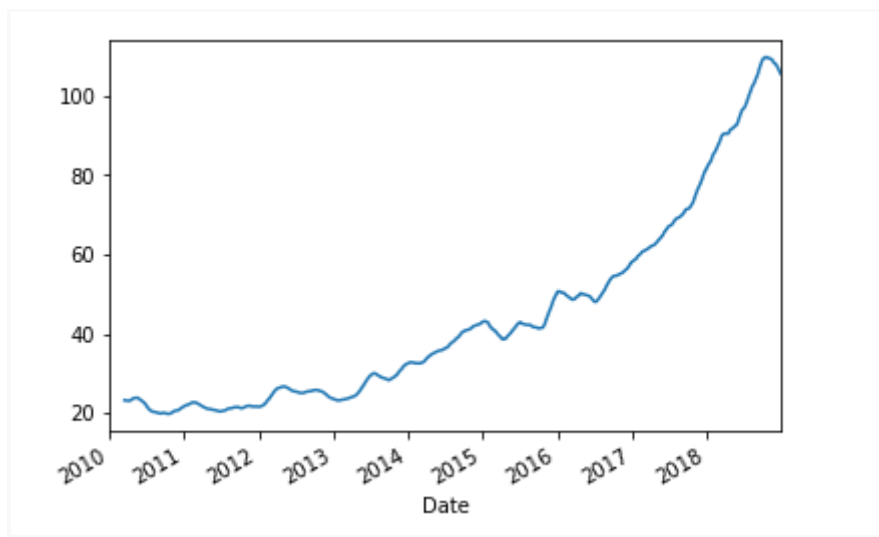
```
import matplotlib.pyplot as plt
```

```
adj_price.plot()
```



You can now plot the rolling mean()

```
mav.plot()
```



And you can see the difference for yourself, how the spikes in the data are consumed to give a general sentiment around the performance of the stock.

Formulating a Trading Strategy

Here comes the final and interesting part, designing and making the trading strategy. This will be a step-by-step guide to developing a momentum-based Simple Moving Average Crossover (SMAC) strategy.

Momentum-based strategies are based on a technical indicator that capitalizes on the continuance of the market trend. We purchase securities that show an upwards trend and short-sell securities which show a downward trend.

The SMAC strategy is a well-known schematic momentum strategy. It is a long-only strategy. Momentum, here, is the total return of stock

including the dividends over the last n months. This period of n months is called the lookback period.

There are 3 main types of lookback periods: short term, intermediate-term, and long term. We need to define 2 different lookback periods of a particular time series.

A buy signal is generated when the shorter lookback rolling means (or moving average) overshoots the longer lookback moving average. A sell signal occurs when the shorter lookback moving average dips below the longer moving average.

Now, let's see how the code for this strategy will look:

```
# step1: initialize the short and long lookback periods
```

```
short_lb = 50
```

```
long_lb = 120# step2: initialize a new DataFrame called signal_df with the  
signal column
```

```

signal_df = pd.DataFrame(index=msft_data.index)

signal_df['signal'] = 0.0# step3: create a short simple moving average
over the short lookback period

signal_df['short_mav'] = msft_data['Adj_Close'].rolling(window=short_lb,
min_periods=1, center=False).mean()# step4: create long simple moving
average over the long lookback period

signal_df['long_mav'] = msft_data['Adj_Close'].rolling(window=long_lb,
min_periods=1, center=False).mean()# step5: generate the signals based on
the conditional statement

signal_df['signal'][short_lb:] =
np.where(signal_df['short_mav'][short_lb:] >
signal_df['long_mav'][short_lb:], 1.0, 0.0) # step6: create the trading
orders based on the positions column

signal_df['positions'] = signal_df['signal'].diff()

signal_df[signal_df['positions'] == -1.0]

```

Let's see what's happening here. We have created 2 lookback periods, the short lookback period `short_lb` is of 50 days and the longer

lookback period for the long moving average is defined as `long_lb` of 120 days.

We have created a new DataFrame which is designed to capture the signals which are being generated whenever the short moving average crosses the long moving average using the `np.where` and assigning `1.0` for true and `0.0` if the condition comes out to be false.

The `positions` columns in the DataFrame tells us if there is a buy signal or a sell signal, or to stay put. We're basically calculating the difference in the signals column from the previous row using [`diff`](#).

And there we have our strategy implemented in just 6 steps using Pandas. Easy, isn't it?

Now, let's try to visualize this using Matplotlib. All we need to do is initialize a plot figure, add the adjusted closing prices, short and long

moving averages to the plot, and then plot the buy and sell signals

using the positions column in the `signal_df` above:

```
# initialize the plot using plt
```

```
fig = plt.figure()# Add a subplot and label for y-axis
```

```
plt1 = fig.add_subplot(111, ylabel='Price in  
$')msft_data['Adj_Close'].plot(ax=plt1, color='r', lw=2.)# plot the short  
and long lookback moving averages
```

```
signal_df[['short_mav', 'long_mav']].plot(ax=plt1, lw=2., figsize=(12,8))#  
plotting the sell signals
```

```
plt1.plot(signal_df.loc[signal_df.positions == -1.0].index,
```

```
        signal_df.short_mav[signal_df.positions == -1.0],
```

```
        'v', markersize=10, color='k')# plotting the buy signals
```

```
plt1.plot(signal_df.loc[signal_df.positions == 1.0].index,
```

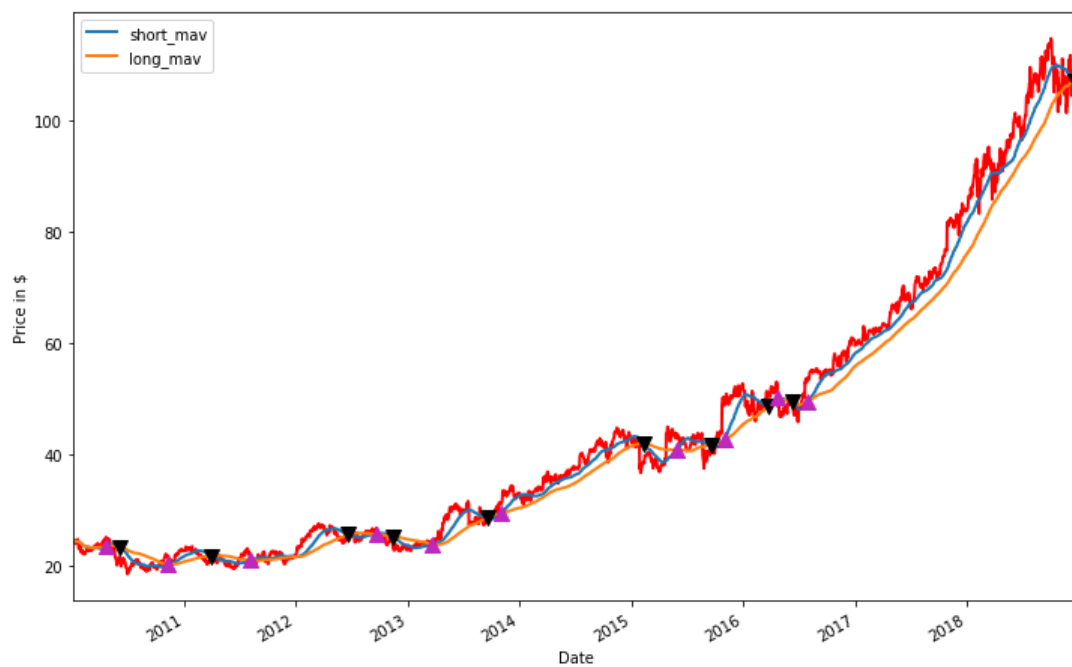
```
        signal_df.short_mav[signal_df.positions == 1.0],
```

```
'^', markersize=10, color='m')
```

```
# Show the plot
```

```
plt.show()
```

Running the above cell in the Jupyter notebook would yield a plot like the one below:



Now, you can clearly see that whenever the blue line (short moving average) goes up and beyond the orange line (long moving average), there is a pink upward marker indicating a buy signal. A sell signal is denoted by a black downward marker where there's a fall of the

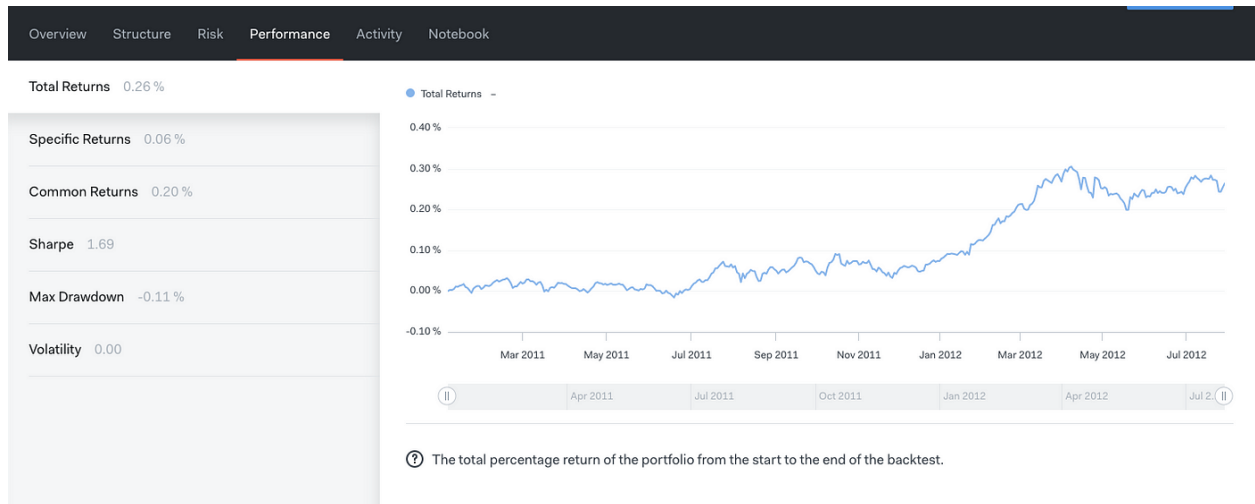
`short_mav` below `long_mav`.

Visualize the Performance of the Strategy on Quantopian

[Quantopian](#) is a Zipline powered platform that has manifold use cases.

You can write your own algorithms, access free data, backtest your strategy, contribute to the community, and collaborate with Quantopian if you need capital.

We have written an algorithm to backtest our SMA strategy, and here are the results:



Here is an explanation of the above metrics:

- **Total return:** The total percentage return of the portfolio from the start to the end of the backtest.
- **Specific return:** The difference between the portfolio's total returns and common returns.
- **Common return:** Returns that are attributable to common risk factors. There are 11 sector and 5 style risk factors that make up these returns. The Sector Exposure and Style Exposure charts in the Risk section provide more detail on these factors.

- Sharpe: The 6-month rolling Sharpe ratio. It is a measure of risk-adjusted investment. It is calculated by dividing the portfolio's excess returns over the risk-free rate by the portfolio's standard deviation.
- Max Drawdown: The largest drop of all the peak-to-trough movement in the portfolio's history.
- Volatility: Standard deviation of the portfolio's returns.

Pat yourself on the back as you have successfully implemented your quantitative trading strategy!

Where to go From Here?

Now that your algorithm is ready, you'll need to backtest the results and assess the [metrics mapping the risk](#) involved in the strategy and the stock. Again, you can use [BlueShift](#) and [Quantopian](#) to learn more about backtesting and trading strategies.