



Eryk Lewinson

Follow

Jul 28, 2019 · 10 min read · Member-only

Source: [pixabay](#)

Introduction to backtesting trading strategies

Learn how to build and backtest trading strategies using [zipline](#)

In this article, I would like to continue the series on quantitative finance. In the [first article](#), I described the stylized facts of asset returns. Now I would like to introduce the concept of backtesting trading strategies and how to do it using existing frameworks in Python.

What is backtesting?

Let's start with a trading strategy. It can be defined as a method (based on predefined rules) of buying and/or selling assets in markets. These rules can be based on, for example, technical analysis or machine learning models.

Backtesting is basically evaluating the performance of a trading strategy on historical data — if we used a given strategy on a set of assets in the past, how well/bad would it have performed. Of course, there is no guarantee that past performance is indicative of the future one, but we can still investigate!

There are a few available frameworks for backtesting in Python, in this article, I decided to use [zipline](#).

Why [zipline](#)?

Some of the nice features offered by the [zipline](#) environment include:

- ease of use — there is a clear structure of how to build a backtest and what outcome we can expect, so the majority of the time can be spent on developing state-of-the-art trading strategies :)
- realistic — includes transaction costs, slippage, order delays, etc.
- stream-based — processes each event individually, thus avoids look-ahead bias



- integration with PyData ecosystem — `zipline` uses Pandas DataFrames for storing input data, as well as performance metrics
- it is easy to integrate other libraries, such as `matplotlib`, `scipy`, `statsmodels` and `sklearn` into the workflow of building and evaluating strategies
- developed and updated by Quantopian which provides a web-interface for `zipline`, historical data and even live-trading capabilities

I believe these arguments speak for themselves. Let's start coding!

Setting up the virtual environment using `conda`

The most convenient way to install `zipline` is to use a virtual environment. In this article, I use `conda` to do so. I create a new environment with Python 3.5 (I experienced issues with using 3.6 or 3.7) and then install `zipline`. You can also `pip install` it.

```
# create new virtual environment
conda create -n env_zipline python=3.5

# activate it
conda activate env_zipline

# install zipline
conda install -c Quantopian zipline
```

For everything to be working properly you should also install `jupyter` and other packages used in this article (see the `watermark` printout below).

Importing libraries

First, we need to load IPython extensions using the `%load_ext` magic.

```
%load_ext watermark
%load_ext zipline
```

Then we import the rest of the libraries:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import zipline
from yahooфинанциалс import YahooFinancials
import warnings

plt.style.use('seaborn')
plt.rcParams['figure.figsize'] = [16, 9]
plt.rcParams['figure.dpi'] = 200
warnings.simplefilter(action='ignore', category=FutureWarning)
```

Below you can see the list of libraries used in this article, together with their versions.



```
%watermark --iversions
```

```
executed in 4ms, finished 20:13:34 2019-07-26
```

```
pandas      0.22.0
zipline      1.3.0
json         2.0.9
numpy        1.14.6
matplotlib  3.0.0
```

Import custom data

`zipline` comes ready with data downloaded from Quandl (the WIKI database). You can always inspect the already ingested data by running:

```
!zipline bundles
```

The issue with this approach is that in mid-2018 the data was discontinued, so there is no data for the last year. To overcome this, I show how to manually ingest data from any source. To do so I use the `yahoofinancials` library. In order to be loaded into `zipline`, the data must be in a CSV file and in a predefined format — like the one on the preview of the DataFrame.



```
df.to_csv('daily/aapl.csv', header=True, index=False)
```

In the next step, we need to modify the `extension.py` file located in the `zipline` directory. After the installation of `zipline` it is empty and we need to add the following:

We can also define and provide a custom calendar to the data-ingesting script — for example when working with European stocks. For details on how to do it please look at the [documentation](#).

In contrast to the data downloading function, we need to pass the exact range of dates of the downloaded data. In this example, we start with `2017-01-03`, as this is the first day for which we have pricing data.

Lastly, we run the following command:

```
!zipline ingest -b apple-prices-2017-2019
```

We can verify that the bundle was successfully ingested:

```
!zipline bundles
```



There is a [known issue](#) with downloading the benchmark data, so — for now — we stick to historical data in the default bundle. However, you now know how to ingest data using a custom CSV file.

For detailed information on how to load custom data using the `csvdir` bundle please refer to this [article](#), in which I show how to import European stocks data and run basic strategies on their basis.

Buy And Hold Strategy

We start with the most basic strategy — Buy and Hold. The idea is that we buy a certain asset and do not do anything for the entire duration of the investment horizon. This simple strategy can also be considered a benchmark for more advanced ones — because there is no point in using a very complex strategy that generates less money (for example due to transaction costs) than buying and doing nothing.

In this example, we consider Apple's stock and select years 2016–2017 as the duration of the backtest. We start with a capital of 1050 USD. I selected this number as I know how much more or less we need to have for the initial purchase and I like to keep this number as small as possible because we are only buying 10 shares, so no need for a starting balance of a couple of thousands. We assume the default transaction costs (0.001\$ per share, without a minimum cost per trade).

There are two approaches to using `zipline` — using the command line or Jupyter Notebook. To use the latter we have to write the algorithm within a Notebook cell and indicate that `zipline` is supposed to run it. This is done via the `%%zipline` IPython magic command. This magic takes the same arguments as the CLI mentioned above.

Also one important thing, all imports required for the algorithm to run (such as `numpy`, `sklearn`, etc.) must be specified in the algorithm cell, even if they were previously imported elsewhere.



Congrats, we have written our first backtest. So what actually happened?

Each `zipline` algorithm contains (at least) two functions we have to define:

```
* initialize(context)
* handle_data(context, data)
```

Before the algorithm starts, the `initialize()` function is called and a `context` variable is passed. `context` is a global variable in which we can store additional variables we need to access from one iteration of the algorithm to the next.

After the initialization of the algorithm, the `handle_data()` function is called once for each event. At every call, it passes the same `context` variable and an event frame called `data`. It contains the current trading bar with open, high, low, and close (OHLC) prices together with the volume.

We create an order by using `order(asset, number_of_units)`, where we specify what to buy and how many shares/units. A positive number indicates buying that many shares, 0 means selling everything we have, and a negative number is used for short-selling. Another useful type of order is `order_target`, which orders as many shares as needed to achieve the desired number in the portfolio.

In our Buy and Hold strategy, we check if we have already placed an order. If not, we order a given amount of shares and then do nothing for the rest of the backtest.

Let's analyze the performance of the strategy. First, we need to load the performance DataFrame from the pickle file.

```
# read the performance summary dataframe
buy_and_hold_results = pd.read_pickle('buy_and_hold.pkl')
```

And now we can plot some of the stored metrics:



From the first look, we see that the portfolio generated money over the investment horizon and was very much following the price of Apple (what makes sense as it is the only asset in the portfolio).

To view the transactions we need to transform the `transactions` column from the performance DataFrame.

```
pd.DataFrame.from_records([x[0] for x in  
buy_and_hold_results.transactions.values if x != []])
```

By inspecting the columns of the performance DataFrame we can see all the available metrics.

```
buy_and_hold_results.columns
```

Some of the noteworthy ones:

- starting/ending cash — inspecting the cash holding on a given day
- starting/ending value — inspecting the asset's value on a given day



when it is actually executed on the next trading day

- pnl — daily profit and loss

Simple Moving Average Strategy

The second strategy we consider is based on the simple moving average (SMA). The ‘mechanics’ of the strategy can be summarized by the following:

- when the price crosses the 20-day SMA upwards — buy x shares
- when the price crosses the 20-day SMA downwards — sell the shares
- we can only have a maximum of x shares at any given time
- there is no short-selling in the strategy (though it can be easily implemented)

The remaining components of the backtest like the considered asset, investment horizon or the starting capital are the same as in the Buy and Hold example.

The code for this algorithm is a little bit more complex, but we cover all the new aspects of the code. For simplicity, I marked the points of reference in the code snippet above (`sma_strategy.py`) and will refer to them by number below.

1. I show how to manually set the commission. In this case, I use the default value for comparison's sake.
2. The “warm-up period” — this is a trick used in order to make sure that the algorithm has enough days to calculate the moving average. If we are using multiple metrics with different window lengths, we should always take the longest one for the warm-up.
3. We access the historical (and current) data-points by using `data.history`. In this example, we access the last 20 days. The data (in case of a single asset) is stored as a `pandas.Series`, indexed by time.
4. The SMA is a very basic measure, so for calculation, I simply take the mean of the previously accessed data.
5. I encapsulate the logic of the trading strategy in an `if` statement. To access the current and previous data-points I use `price_history[-2]` and `price_history[-1]`, respectively. To see if there was a crossover, I compare the current and previous prices to the MA and determine which case I am dealing with (buy/sell signal). In the case where there is no signal, the algorithm does nothing.
6. You can use the `analyze(context, perf)` statement to carry out extra analysis (like plotting) when the backtest is finished. The `perf` object is simply the performance DataFrame we also store in a pickle file. But when used withing the algorithm, we should refer to it as `perf` and there is no need for loading it.

As compared to the Buy and Hold strategy, you might have noticed the periods where the portfolio value is flat. That is because when we sell the asset (and before buying again), we only hold cash.

In our case, the Simple Moving Average strategy outperformed the Buy and Hold one. The ending worth of the portfolio (including cash) is 1784.12 USD for the SMA strategy, while it is 1714.68 USD in the case of the simpler one.

Conclusions

In this article, I have shown how to use the `zipline` framework to carry out the backtesting of trading strategies. Once you get familiar with the library, it is easy to test out different strategies. In a future article, I will cover using more advanced trading strategies based on technical analysis.

As always, any constructive feedback is welcome. You can reach out to me on [Twitter](#) or in the comments. You can find the code used for this article on my [GitHub](#).

Liked the article? Become a Medium member to continue learning by reading without limits. If you use [this link](#) to become a member, you will support me at no extra cost to you. Thanks in advance and see you around!

Below you can find the next articles in the series:

[Sign In](#)[Get started](#)

- building algorithmic trading strategies based on Technical Analysis ([link](#))
- building algorithmic trading strategies based on the mean-variance analysis ([link](#))

• • •

I recently published a book on using Python for solving practical tasks in the financial domain. If you are interested, I posted [an article](#) introducing the contents of the book. You can get the book on [Amazon](#) or [Packt's website](#).




More from Towards Data Science

[Follow](#)


Your home for data science. A Medium publication sharing concepts, ideas and codes.

[Read more from Towards Data Science](#)


Recommended from Medium

 Antony He... in Towards Data Sci...
Generalizing data load processes with Airflow




 Adam Votava
Keeping Up With Data #68




 Mark Burgess
Universal Data Analytics as Semantic Spacetime




 James Shah in Byte Tales
Learning Data Analysis with Python—Introduction to Pandas




 Adrian
Master Data Management (MDM): Top 10 Definitions



 James Cerwi... in Towards Data Sc...
How to Accurately Calculate Age in BigQuery



 Shubhi Asthana in Code Like A Girl
Understanding p-values more closely



 Sharvil
Linear Discriminant Analysis

[About](#) [Help](#) [Terms](#) [Privacy](#)