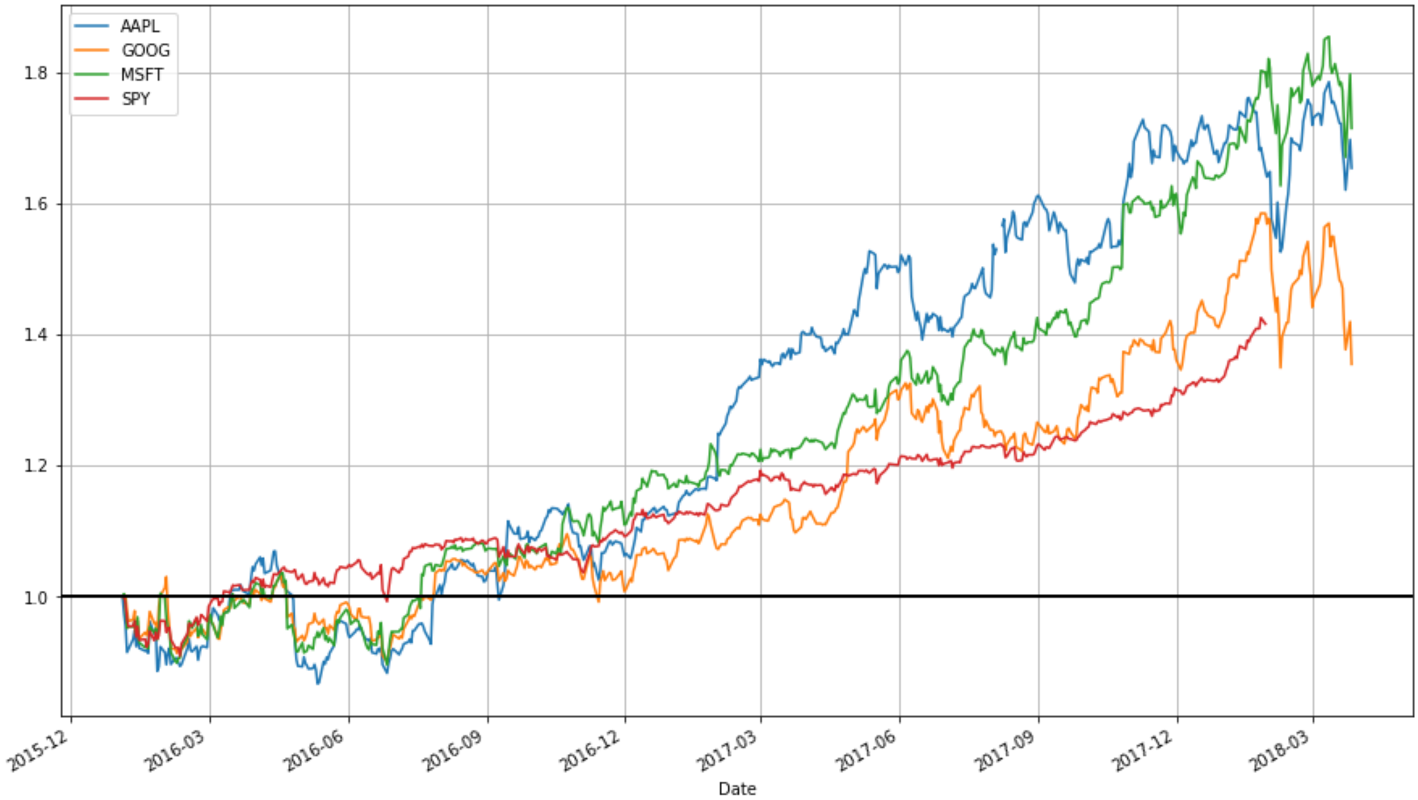
Curtis Miller's Personal Website



Posted on July 17, 2018July 16, 2018 | Economics and Finance, Python, Statistics and Data Science

Stock Data Analysis with Python (Second Edition)

Introduction

This is a lecture for MATH 4100/CS 5160: Introduction to Data Science (http://datasciencecourse.net/), offered at the University of Utah, introducing time series data analysis applied to finance. This is also an update to my earlier blog (https://ntguardian.wordpress.com/2016/09/19/introduction-stock-market-data-python-1/) posts (https://ntguardian.wordpress.com/2016/09/26/introduction-stock-market-data-python-2/) on the same topic (this one combining them together). I strongly advise referring to this blog post instead of the previous ones (which I am not altering for the sake of preserving a record). The code should work as of July 7th, 2018. (And sorry for some of the formatting; WordPress.com's free version doesn't play nice with code or tables.)

Advanced mathematics and statistics have been present in finance for some time. Prior to the 1980s, banking and finance were well-known for being “boring”; investment banking was distinct from commercial banking and the primary role of the industry was handling “simple” (at least in comparison to today) financial instruments, such as loans. Deregulation under the Regan administration, coupled with an influx of mathematical talent, transformed the industry from the “boring” business of banking to what it is today, and since then, finance has joined the other sciences as a motivation for mathematical research and advancement. For example one of the biggest recent achievements of mathematics was the derivation of the Black-Scholes formula (https://en.wikipedia.org/wiki/Black%E2%80%93Scholes_model), which facilitated the pricing of stock options (a contract giving the holder the right to purchase or sell a stock at a particular price to the issuer of the option). That said, bad statistical models, including the Black-Scholes formula, hold part of the blame for the 2008 financial crisis (https://www.theguardian.com/science/2012/feb/12/black-scholes-equation-credit-crunch).

In recent years, computer science has joined advanced mathematics in revolutionizing finance and **trading**, the practice of buying and selling of financial assets for the purpose of making a profit. In recent years, trading has become dominated by computers; algorithms are responsible for making rapid split-second trading decisions faster than humans could make (so rapidly, the speed at which light travels is a limitation when designing systems (http://www.nature.com/news/physics-in-finance-trading-at-the-speed-of-light-1.16872)). Additionally, machine learning and data mining techniques are growing in popularity (http://www.ft.com/cms/s/0/9278d1b6-1e02-11e6-b286-cddde55ca122.html#axzz4G8daZxcl) in the financial sector, and likely will continue to do so. For example, **high-frequency trading (HFT)** is a branch of algorithmic trading where computers make thousands of trades in short periods of time, engaging in complex strategies such as statistical arbitrage and market making. While algorithms may outperform humans, the technology is still new and playing an increasing role in a famously turbulent, high-stakes arena. HFT was responsible for phenomena such as the 2010 flash crash (https://en.wikipedia.org/wiki/2010_Flash_Crash) and a 2013 flash crash (http://money.cnn.com/2013/04/24/investing/twitter-flash-crash/) prompted by a hacked Associated Press tweet (http://money.cnn.com/2013/04/23/technology/security/ap-twitter-hacked/index.html?iid=EL) about an attack on the White House.

This lecture, however, will not be about how to crash the stock market with bad mathematical models or trading algorithms. Instead, I intend to provide you with basic tools for handling and analyzing stock market data with Python. We will be using stock data as a first exposure to **time series data**, which is data considered dependent on the time it was observed (other examples of time series include temperature data, demand for energy on a power grid, Internet server load, and many, many others). I will also discuss moving averages, how to construct trading strategies using moving averages, how to formulate exit strategies upon entering a position, and how to evaluate a strategy with backtesting.

DISCLAIMER: THIS IS NOT FINANCIAL ADVICE!!! Furthermore, I have ZERO experience as a trader (a lot of this knowledge comes from a one-semester course on stock trading I took at Salt Lake Community College)! This is purely introductory knowledge, not enough to make a living trading stocks. People can and do lose money trading stocks, and you do so at your own risk!

Preliminaries

I will be using two packages, **quandl** and **pandas_datareader**, which are not installed with Anaconda (https://www.anaconda.com/) if you are using it. To install these packages, run the following at the appropriate command prompt:

```
conda install quandl
conda install pandas-datareader
```

Getting and Visualizing Stock Data

Getting Data from Quandl

Before we analyze stock data, we need to get it into some workable format. Stock data can be obtained from Yahoo! Finance (http://finance.yahoo.com), Google Finance (http://finance.google.com), or a number of other sources. These days I recommend getting data from Quandl (https://www.quandl.com/), a provider of community-maintained financial and economic data. (Yahoo! Finance used to be the go-to source for good quality stock data, but the API was discontinued in 2017 and reliable data can no longer be obtained: see this question/answer on StackExchange (https://quant.stackexchange.com/questions/35019/is-yahoo-finance-data-good-or-bad-now) for more details.)

By default the `get()` function in **quandl** will return a **pandas** `DataFrame` containing the fetched data.

```
import pandas as pd
import quandl
import datetime

# We will look at stock prices over the past year, starting at January 1,
start = datetime.datetime(2016,1,1)
end = datetime.date.today()

# Let's get Apple stock data; Apple's ticker symbol is AAPL
# First argument is the series we want, second is the source ("yahoo" for
s = "AAPL"
apple = quandl.get("WIKI/" + s, start_date=start, end_date=end)

type(apple)
```

```
pandas.core.frame.DataFrame
```

```
apple.head()
```

	Open	High	Low	Close	Volume
Date					
2016-01-04	102.61	105.368	102.00	105.35	67649387.0
2016-01-05	105.75	105.850	102.41	102.71	55790992.0
2016-01-06	100.56	102.370	99.87	100.70	68457388.0
2016-01-07	98.68	100.130	96.43	96.45	81094428.0
2016-01-08	98.55	99.110	96.76	96.96	70798016.0

Ex-Dividend	Split Ratio	Adj. Open	Adj. High	Adj. Low	Adj. Close	Adj. Volume
0.0	1.0	99.136516	101.801154	98.547165	101.783763	67649387.0
0.0	1.0	102.170223	102.266838	98.943286	99.233131	55790992.0
0.0	1.0	97.155911	98.904640	96.489269	97.291172	68457388.0
0.0	1.0	95.339552	96.740467	93.165717	93.185040	81094428.0
0.0	1.0	95.213952	95.754996	93.484546	93.677776	70798016.0

Let's briefly discuss this. **Open** is the price of the stock at the beginning of the trading day (it need not be the closing price of the previous trading day), **high** is the highest price of the stock on that trading day, **low** the lowest price of the stock on that trading day, and **close** the price of the stock at closing time. **Volume** indicates how many stocks were traded. **Adjusted** prices (such as the adjusted close) is the price of the stock that adjusts the price for corporate actions. While stock prices are considered to be set mostly by traders, **stock splits** (when the company makes each extant stock worth two and halves the price) and **dividends** (payout of company profits per share) also affect the price of a stock and should be accounted for.

Visualizing Stock Data

Now that we have stock data we would like to visualize it. I first demonstrate how to do so using the **matplotlib** package. Notice that the `apple` `DataFrame` object has a convenience method, `plot()`, which makes creating plots easier.

```
import matplotlib.pyplot as plt   # Import matplotlib
# This line is necessary for the plot to appear in a Jupyter notebook
%matplotlib inline
# Control the default size of figures in this Jupyter notebook
%pylab inline
pylab.rcParams['figure.figsize'] = (15, 9)   # Change the size of plots

apple["Adj. Close"].plot(grid = True) # Plot the adjusted closing price of
```

```
Populating the interactive namespace from numpy and matplotlib
```

A linechart is fine, but there are at least four variables involved for each date (open, high, low, and close), and we would like to have some visual way to see all four variables that does not require plotting four separate lines. Financial data is often plotted with a **Japanese candlestick plot**, so named because it was first created by 18th century Japanese rice traders. Such a chart can be created with **matplotlib**, though it requires considerable effort.

I have made a function you are welcome to use to more easily create candlestick charts from **pandas** data frames, and use it to plot our stock data. (Code is based off this example (http://matplotlib.org/examples/pylab_examples/finance_demo.html), and you can read the documentation for the functions involved here (http://matplotlib.org/api/finance_api.html).)

```
from matplotlib.dates import DateFormatter, WeekdayLocator,\
    DayLocator, MONDAY
from mpl_finance import candlestick_ohlc

def pandas_candlestick_ohlc(dat, stick = "day", adj = False, otherseries
    """
    :param dat: pandas DataFrame object with datetime64 index, and float
    :param stick: A string or number indicating the period of time covere
    :param adj: A boolean indicating whether to use adjusted prices
    :param otherseries: An iterable that will be coerced into a list, con

    This will show a Japanese candlestick plot for stock data stored in d
    """
    mondays = WeekdayLocator(MONDAY)        # major ticks on the mondays
    alldays = DayLocator()              # minor ticks on the days
    dayFormatter = DateFormatter('%d')      # e.g., 12

    # Create a new DataFrame which includes OHLC data for each period spe
    fields = ["Open", "High", "Low", "Close"]
    if adj:
        fields = ["Adj. " + s for s in fields]
    transdat = dat.loc[:,fields]
    transdat.columns = pd.Index(["Open", "High", "Low", "Close"])
    if (type(stick) == str):
        if stick == "day":
            plotdat = transdat
            stick = 1 # Used for plotting
        elif stick in ["week", "month", "year"]:
            if stick == "week":
                transdat["week"] = pd.to_datetime(transdat.index).map(lam
            elif stick == "month":
                transdat["month"] = pd.to_datetime(transdat.index).map(la
            transdat["year"] = pd.to_datetime(transdat.index).map(lambda
            grouped = transdat.groupby(list(set(["year",stick]))) # Group
            plotdat = pd.DataFrame({"Open": [], "High": [], "Low": [], "C
            for name, group in grouped:
                plotdat = plotdat.append(pd.DataFrame({"Open": group.iloc
                                            "High": max(group.High),
                                            "Low": min(group.Low),
                                            "Close": group.iloc[-1,3]},
                                           index = [group.index[0]]))
            if stick == "week": stick = 5
            elif stick == "month": stick = 30
            elif stick == "year": stick = 365

    elif (type(stick) == int and stick >= 1):
        transdat["stick"] = [np.floor(i / stick) for i in range(len(trans
        grouped = transdat.groupby("stick")
        plotdat = pd.DataFrame({"Open": [], "High": [], "Low": [], "Close
        for name, group in grouped:
            plotdat = plotdat.append(pd.DataFrame({"Open": group.iloc[0,0
                                        "High": max(group.High),
                                        "Low": min(group.Low),
                                        "Close": group.iloc[-1,3]},
                                       index = [group.index[0]]))

    else:
        raise ValueError('Valid inputs to argument "stick" include the st
```

```

    # Set plot parameters, including the axis object ax used for plotting
    fig, ax = plt.subplots()
    fig.subplots_adjust(bottom=0.2)
    if plotdat.index[-1] - plotdat.index[0] < pd.Timedelta('730 days'):
        weekFormatter = DateFormatter('%b %d')  # e.g., Jan 12
        ax.xaxis.set_major_locator(mondays)
        ax.xaxis.set_minor_locator(alldays)
    else:
        weekFormatter = DateFormatter('%b %d, %Y')
    ax.xaxis.set_major_formatter(weekFormatter)

    ax.grid(True)

    # Create the candelstick chart
    candlestick_ohlc(ax, list(zip(list(date2num(plotdat.index.tolist())),
                         plotdat["Low"].tolist(), plotdat["Close"].tolist())
                         colorup = "black", colordown = "red", width = stick

    # Plot other series (such as moving averages) as lines
    if otherseries != None:
        if type(otherseries) != list:
            otherseries = [otherseries]
        dat.loc[:,otherseries].plot(ax = ax, lw = 1.3, grid = True)

    ax.xaxis_date()
    ax.autoscale_view()
    plt.setp(plt.gca().get_xticklabels(), rotation=45, horizontalalignmen

    plt.show()

pandas_candlestick_ohlc(apple, adj=True, stick="month")
```

With a candlestick chart, a black candlestick indicates a day where the closing price was higher than the open (a gain), while a red candlestick indicates a day where the open was higher than the close (a loss). The wicks indicate the high and the low, and the body the open and close (hue is used to determine which end of the body is the open and which the close). Candlestick charts are popular in finance and some strategies in technical analysis (https://en.wikipedia.org/wiki/Technical_analysis) use them to make trading decisions, depending on the shape, color, and position of the candles. I will not cover such strategies today.

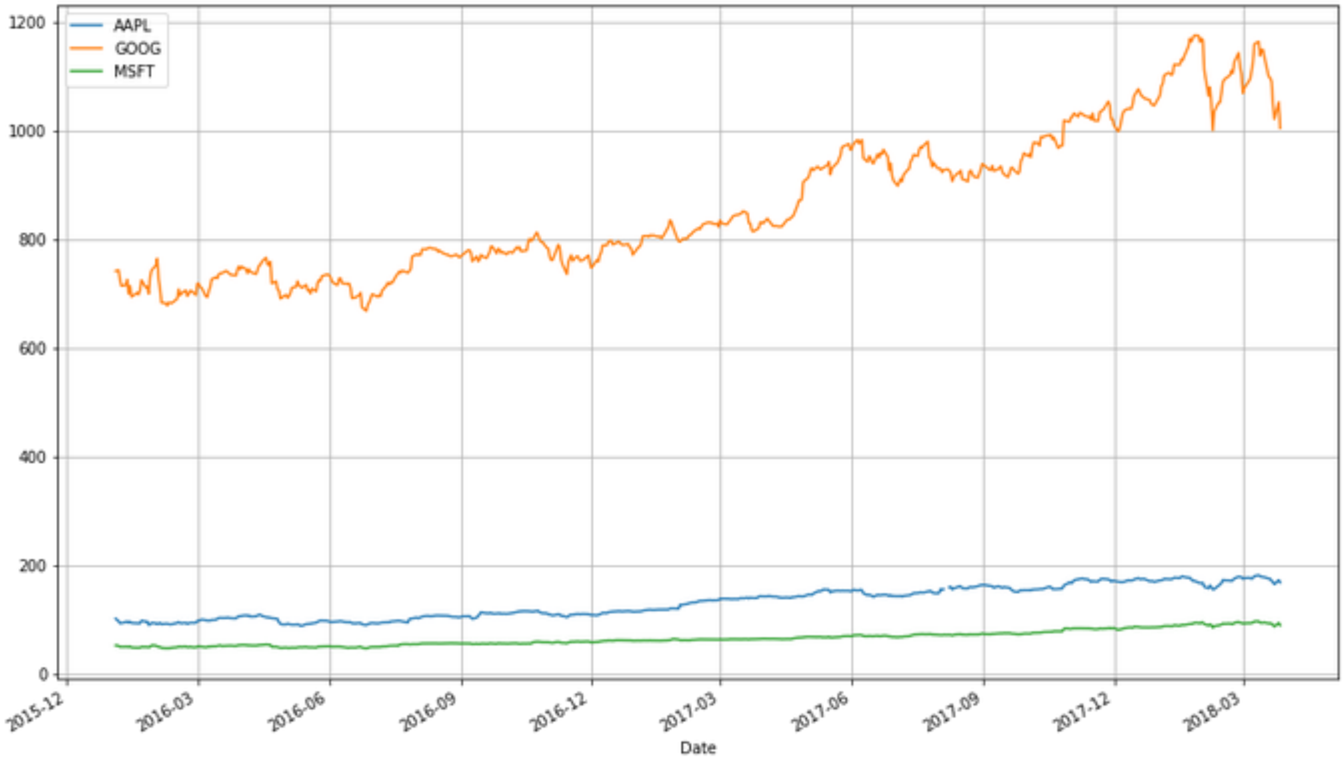
We may wish to plot multiple financial instruments together; we may want to compare stocks, compare them to the market, or look at other securities such as exchange-traded funds (ETFs) (https://en.wikipedia.org/wiki/Exchange-traded_fund). Later, we will also want to see how to plot a financial instrument against some indicator, like a moving average. For this you would rather use a line chart than a candlestick chart. (How would you plot multiple candlestick charts on top of one another without cluttering the chart?)

Below, I get stock data for some other tech companies and plot their adjusted close together.

```
microsoft, google = (quandl.get("WIKI/" + s, start_date=start, end_date=en

# Below I create a DataFrame consisting of the adjusted closing price of t
stocks = pd.DataFrame({"AAPL": apple["Adj. Close"],
                      "MSFT": microsoft["Adj. Close"],
                      "GOOG": google["Adj. Close"]})

stocks.head()
```

	AAPL	GOOG	MSFT
Date			
2016-01-04	101.783763	741.84	52.181598
2016-01-05	99.233131	742.58	52.419653
2016-01-06	97.291172	743.62	51.467434
2016-01-07	93.185040	726.39	49.677262
2016-01-08	93.677776	714.47	49.829617

```
stocks.plot(grid = True)
```



What's wrong with this chart? While absolute price is important (pricy stocks are difficult to purchase, which affects not only their volatility but *your* ability to trade that stock), when trading, we are more concerned about the relative change of an asset rather than its absolute price. Google's stocks are much more expensive than Apple's or Microsoft's, and this difference makes Apple's and Microsoft's stocks appear much less volatile than they truly are (that is, their price appears to not deviate much).

One solution would be to use two different scales when plotting the data; one scale will be used by Apple and Microsoft stocks, and the other by Google.

```
stocks.plot(secondary_y = ["AAPL", "MSFT"], grid = True)
```

A "better" solution, though, would be to plot the information we actually want: the stock's returns. This involves transforming the data into something more useful for our purposes. There are multiple transformations we could apply.

One transformation would be to consider the stock's return since the beginning of the period of interest. In other words, we plot:

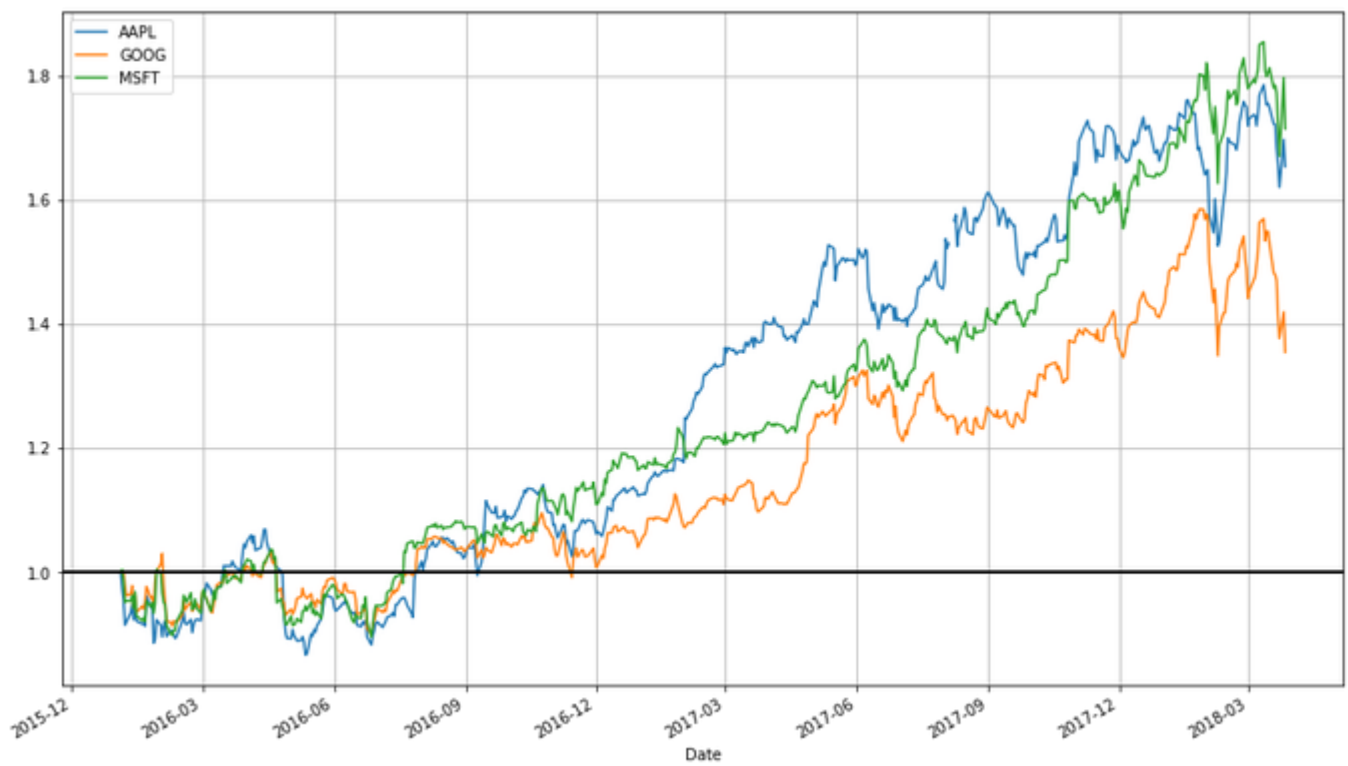
$$\text{return}_{t,0} = \frac{\text{price}_t}{\text{price}_0}$$

This will require transforming the data in the `stocks` object, which I do next. Notice that I am using a **lambda function**, which allows me to pass a small function defined quickly as a parameter to another function or method (you can read more about lambda functions here (https://docs.python.org/3/reference/expressions.html#lambda)).

```
python
# df.apply(arg) will apply the function arg to each column in df, and retu
# Recall that lambda x is an anonymous function accepting parameter x; in
stock_return = stocks.apply(lambda x: x / x[0])
stock_return.head() - 1
```

	AAPL	GOOG	MSFT
Date			
2016-01-04	0.000000	0.000000	0.000000
2016-01-05	-0.025059	0.000998	0.004562
2016-01-06	-0.044139	0.002399	-0.013686
2016-01-07	-0.084480	-0.020827	-0.047993
2016-01-08	-0.079639	-0.036895	-0.045073

```
stock_return.plot(grid = True).axhline(y = 1, color = "black", lw = 2)
```



This is a much more useful plot. We can now see how profitable each stock was since the beginning of the period. Furthermore, we see that these stocks are highly correlated; they generally move in the same direction, a fact that was difficult to see in the other charts.

Alternatively, we could plot the change of each stock per day. One way to do so would be to plot the percentage increase of a stock when comparing day t to day $t + 1$, with the formula:

$$\text{growth}_t = \frac{\text{price}_{t+1} - \text{price}_t}{\text{price}_t}$$

But change could be thought of differently as:

$$\text{increase}_t = \frac{\text{price}_t - \text{price}_{t-1}}{\text{price}_t}$$

These formulas are not the same and can lead to differing conclusions, but there is another way to model the growth of a stock: with log differences.

$$\text{change}_t = \log(\text{price}_t) - \log(\text{price}_{t-1})$$

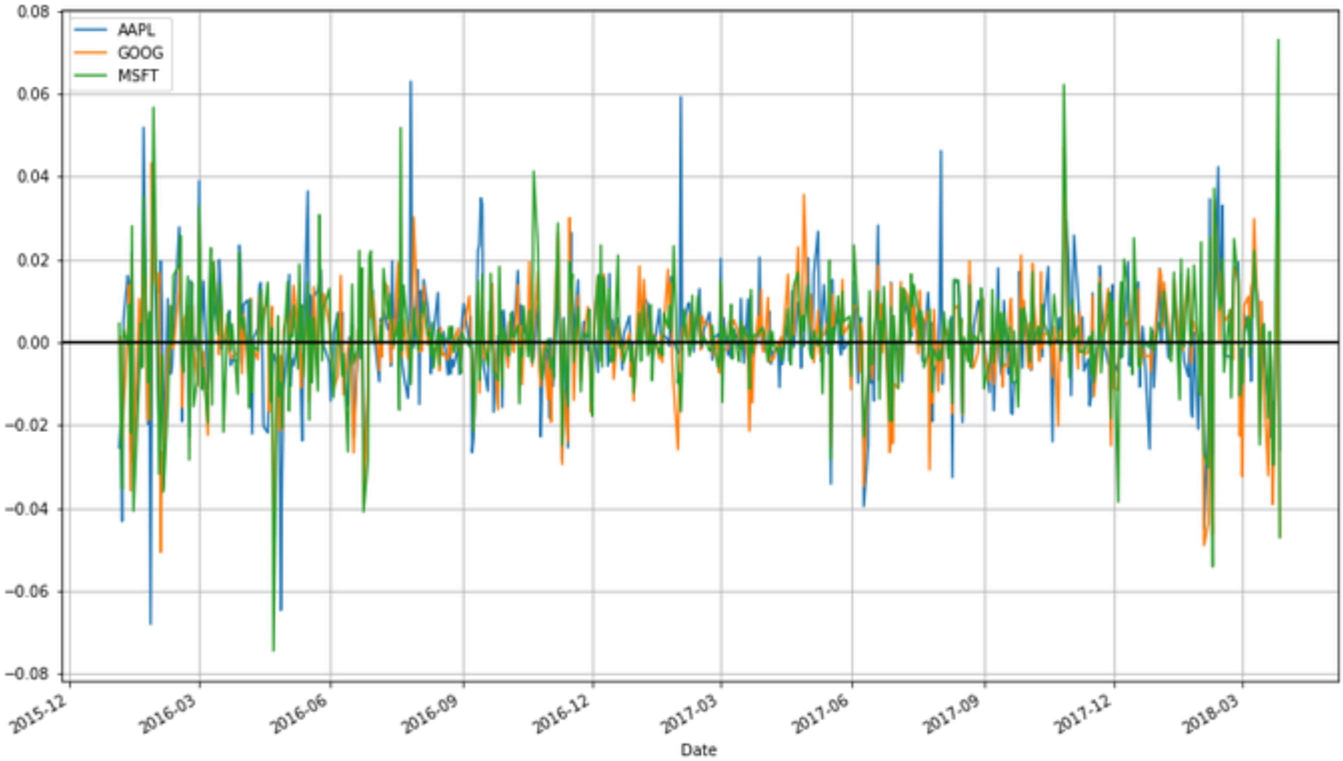
(Here, $\log$ is the natural log, and our definition does not depend as strongly on whether we use $\log(\text{price}_t) - \log(\text{price}_{t-1})$ or $\log(\text{price}_{t+1}) - \log(\text{price}_t)$.) The advantage of using log differences is that this difference can be interpreted as the percentage change in a stock but does not depend on the denominator of a fraction. Additionally, log differences have a desirable property: the sum of the log differences can be interpreted as the total change (as a percentage) over the period summed (which is not a property of the other formulations; they will overestimate growth). Log differences also more cleanly correspond to how stock prices are modeled in continuous time.

We can obtain and plot the log differences of the data in `stocks` as follows:

```
# Let's use NumPy's log function, though math's log function would work ju
import numpy as np

stock_change = stocks.apply(lambda x: np.log(x) - np.log(x.shift(1))) # sh
stock_change.head()
```

	AAPL	GOOG	MSFT
Date			
2016-01-04	NaN	NaN	NaN
2016-01-05	-0.025379	0.000997	0.004552
2016-01-06	-0.019764	0.001400	-0.018332
2016-01-07	-0.043121	-0.023443	-0.035402
2016-01-08	0.005274	-0.016546	0.003062

```
stock_change.plot(grid = True).axhline(y = 0, color = "black", lw = 2)
```



Which transformation do you prefer? Looking at returns since the beginning of the period make the overall trend of the securities in question much more apparent. Changes between days, though, are what more advanced methods actually consider when modelling the behavior of a stock. so they should not be ignored.

We often want to compare the performance of stocks to the performance of the overall market. SPY (https://finance.yahoo.com/quote/SPY/), which is the ticker symbol for the SPDR S&P 500 exchange-traded mutual fund (ETF), is a fund that attempts only to imitate the composition of the S&P 500 stock index (https://finance.yahoo.com/quote/%5EGSPC?p=^GSPC), and thus represents the value in "the market."

SPY data is not available for free from Quandl, so I will get this data from Yahoo! Finance. (I don't have a choice.)

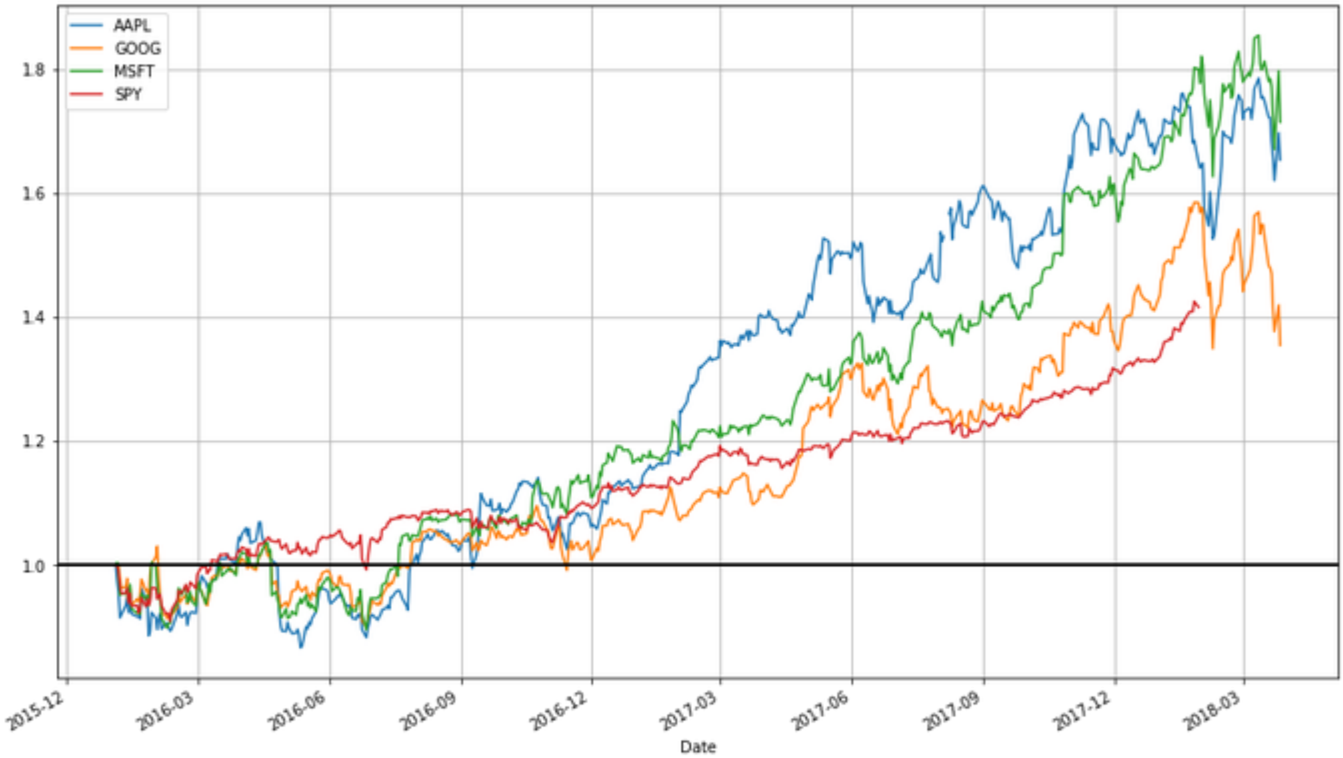
Below I get data for SPY and compare its performance to the performance of our stocks.

```
#import pandas_datareader.data as web    # Going to get SPY from Yahoo! (
#spyder = web.DataReader("SPY", "yahoo", start, end)    # Didn't work
#spyder = web.DataReader("SPY", "google", start, end)    # Didn't work ei
# If all else fails, read from a file, obtained from here: http://www.nas
spyderdat = pd.read_csv("/home/curtis/Downloads/HistoricalQuotes.csv")

spyderdat = pd.DataFrame(spyderdat.loc[:, ["open", "high", "low", "close"
                         index=pd.DatetimeIndex(spyderdat.iloc[1:, 0]),
                         columns=["Open", "High", "Low", "Close", "Adj Cl

spyder = spyderdat.loc[start:end]

stocks = stocks.join(spyder.loc[:, "Adj Close"]).rename(columns={"Adj Clo
stocks.head()
```

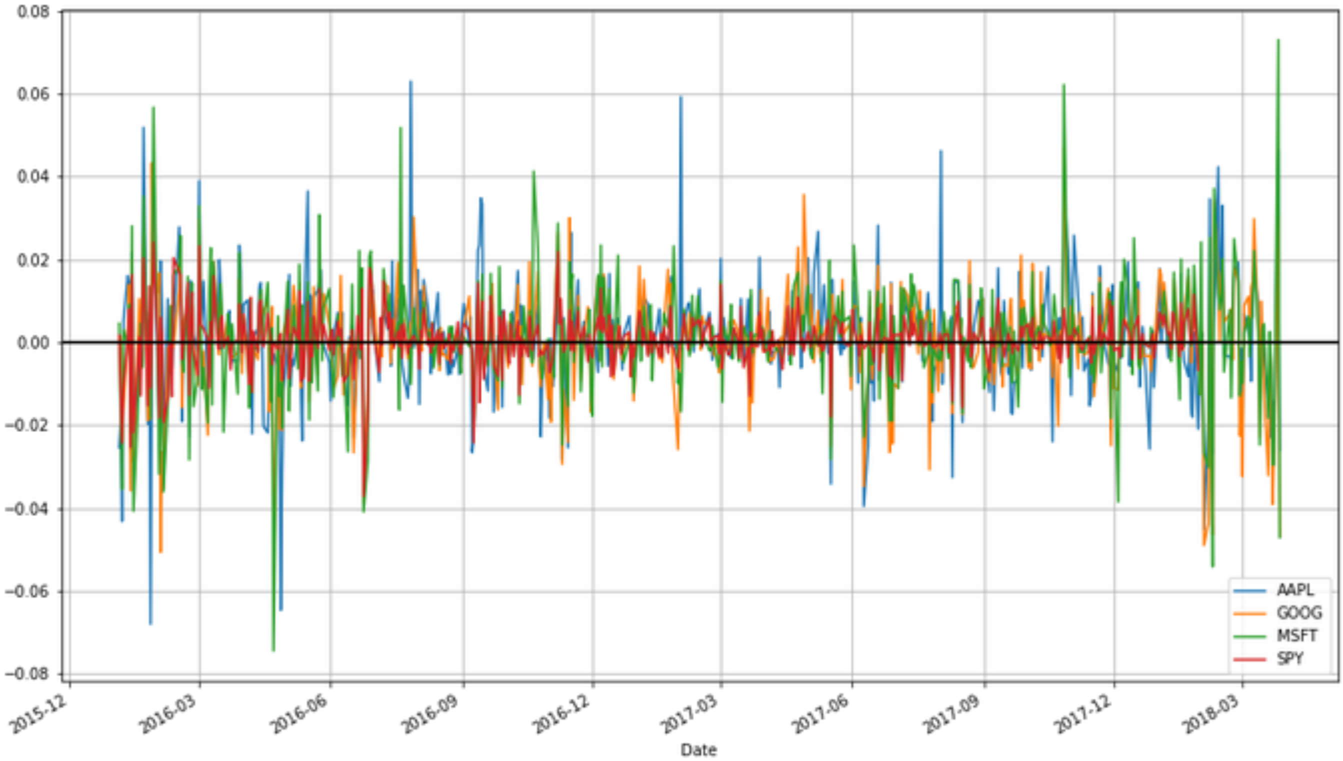
	AAPL	GOOG	MSFT	SPY
Date				

	AAPL	GOOG	MSFT	SPY
Date				
2016-01-04	101.783763	741.84	52.181598	201.0192
2016-01-05	99.233131	742.58	52.419653	201.3600
2016-01-06	97.291172	743.62	51.467434	198.8200
2016-01-07	93.185040	726.39	49.677262	194.0500
2016-01-08	93.677776	714.47	49.829617	191.9230

```
stock_return = stocks.apply(lambda x: x / x[0])
stock_return.plot(grid = True).axhline(y = 1, color = "black", lw = 2)
```




```
stock_change = stocks.apply(lambda x: np.log(x) - np.log(x.shift(1)))
stock_change.plot(grid=True).axhline(y = 0, color = "black", lw = 2)
```



Classical Risk Metrics

From what we have so far we can already compute informative metrics for our stocks, which can be considered some measure of risk.

First, we will want to **annualize** our returns, thus computing the **annual percentage rate (APR)**. This helps us keep returns on a common time scale.

```
stock_change_apr = stock_change * 252 * 100    # There are 252 trading day
stock_change_apr.tail()
```

	AAPL	GOOG	MSFT	SPY
Date				
2018-03-21	-577.463148	-157.285338	-176.499833	NaN
2018-03-22	-359.355133	-984.592233	-743.873619	NaN
2018-03-23	-589.663945	-669.637836	-743.366326	NaN
2018-03-26	1168.762361	768.649993	1839.012005	NaN
2018-03-27	-654.582257	-1178.241231	-1185.615651	NaN

Some of these numbers look initially like nonsense, but that's okay for now.

The metrics I want are:

- The average return
- Volatility (the standard deviation of returns)
- α and β

- The Sharpe ratio

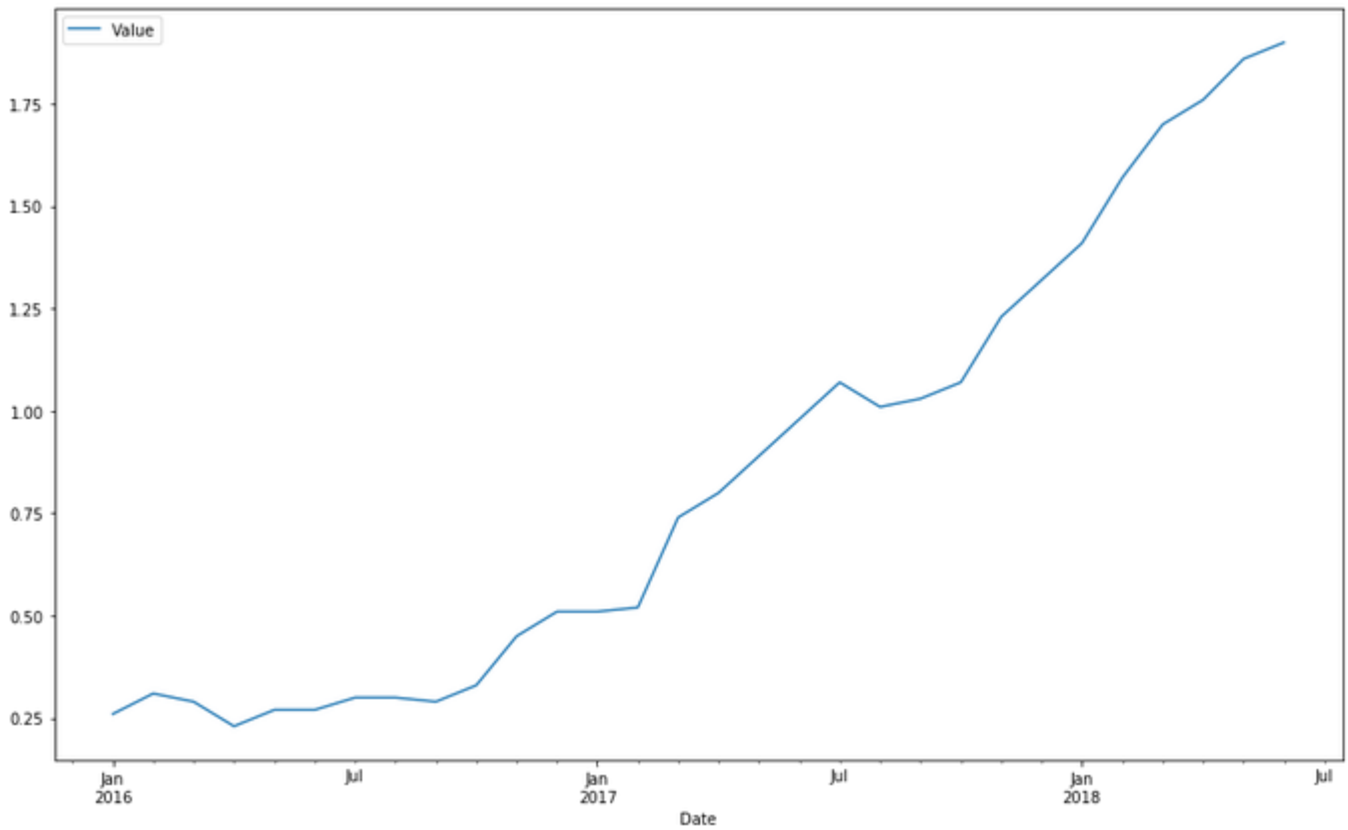
The first two metrics are largely self-explanatory, but the latter two need explaining.

First, the **risk-free rate**, which I denote by r_{RF}, is the rate of return on a risk-free financial asset. This asset exists only in theory but often yields on low-risk instruments like 3-month U.S. Treasury Bills can be viewed as being virtually risk-free and thus their yields can be used to approximate the risk-free rate. I get the data for these instruments below.

```
tbill = quandl.get("FRED/TB3MS", start_date=start, end_date=end)
tbill.tail()
```

	Value
Date	
2018-02-01	1.57
2018-03-01	1.70
2018-04-01	1.76
2018-05-01	1.86
2018-06-01	1.90

```
tbill.plot()
```




```
rrf = tbill.iloc[-1, 0]    # Get the most recent Treasury Bill rate
rrf
```

```
1.8999999999999999
```

Now, a **linear regression model** is a model of the following form:

$y_i = \alpha + \beta x_i + \epsilon_i$

ϵ_i is an error process. Another way to think of this process model is:

$\hat{y}_i = \alpha + \beta x_i$

$\hat{y}_i$ is the **predicted value** of y_i given x_i. In other words, a linear regression model tells you how x_i and y_i are related, and how values of x_i can be used to predict values of y_i. α is the **intercept** of the model and β is the **slope**. In particular, α would be the predicted value of y if x were zero, and β gives how much y changes when x changes by one unit.

There is an easy way to compute α and β given the sample means $\bar{x}$ and $\bar{y}$ and sample standard deviations s_x and s_y and the correlation between x and $y, denoted with r:

$\beta = r\frac{s_y}{s_x}$
$\alpha = \bar{y} - \beta\bar{x}$

In finance, we use α and β like so:

$R_t - r_{RF} = \alpha + \beta(R_{Mt} - r_{RF}) + \epsilon_t$

R_t is the return of a financial asset (a stock) and $R_t - r_{RF}$ is the **excess return**, or return exceeding the risk-free rate of return. R_{Mt} is the return of the *market* at time t. Then α and β can be interpreted like so:

- α is average excess return over the market.
- β is how much a stock moves in relation to the market. If $\beta > 0$ then the stock generally moves in the same direction as the market, while when $\beta 1$ the stock moves strongly in response to the market $|\beta| < 1$ the stock is less responsive to the market.

Below I get a **pandas** `Series` that contains how much each stock is correlated with SPY (our approximation of the market).

```
smcorr = stock_change_apr.drop("SPY", 1).corrwith(stock_change_apr.SPY)


smcorr
```

```
AAPL    0.547184
GOOG    0.592740
MSFT    0.671356
dtype: float64
```

Then I compute α and β.

```
sy = stock_change_apr.drop("SPY", 1).std()
sx = stock_change_apr.SPY.std()
sy
```

```
AAPL    339.921782
GOOG    312.319468
MSFT    329.308164
dtype: float64
```

```
sx    # Standard deviation for x
```

```
164.60477271861888
```

```
ybar = stock_change_apr.drop("SPY", 1).mean() - rrf
xbar = stock_change_apr.SPY.mean() - rrf
ybar
```

```
AAPL    19.769035
GOOG    11.766893
MSFT    22.362806
dtype: float64
```

```
xbar
```

```
14.962934571070926
```

```
beta = smcorr * sy / sx
alpha = ybar - beta * xbar
beta
```

```
AAPL    1.129978
GOOG    1.124658
MSFT    1.343114
dtype: float64
```

```
alpha
```

```
AAPL    2.861252
GOOG   -5.061295
MSFT    2.265881
dtype: float64
```

The **Sharpe ratio** is another popular risk metric, defined below:

$$\text{Sharpe ratio} = \frac{\bar{R}_t - r_{RF}}{s}$$

Here s is the volatility of the stock. We want the sharpe ratio to be large. A large Sharpe ratio indicates that the stock's excess returns are large relative to the stock's volatilitly. Additionally, the Sharpe ratio is tied to a statistical test (the t-test) to determine if a stock earns more on average than the risk-free rate; the larger this ratio, the more likely this is to be the case.

Your challenge now is to compute the Sharpe ratio for each stock listed here, and interpret it. Which stock seems to be the better investment according to the Sharpe ratio?

```
sharpe = (ybar - rrf)/sy
sharpe
```

```
AAPL    0.052568
GOOG    0.031592
MSFT    0.062139
dtype: float64
```

```
(xbar - rrf)/sx
```

```
0.079359391318507888
```

Moving Averages

Charts are very useful. In fact, some traders base their strategies almost entirely off charts (these are the "technicians", since trading strategies based off finding patterns in charts is a part of the trading doctrine known as **technical analysis**). Let's now consider how we can find trends in stocks.

A q**-day moving average** is, for a series x_t and a point in time t, the average of the past q days: that is, if MA^q_t denotes a moving average process, then:

$$MA^q_t = \frac{1}{q} \sum_{i=0}^{q-1} x_{t-i}$$

Moving averages smooth a series and helps identify trends. The larger q is, the less responsive a moving average process is to short-term fluctuations in the series x_t. The idea is that moving average processes help identify trends from "noise". **Fast** moving averages have smaller q and more closely follow the stock, while **slow** moving averages have larger q, resulting in them responding less to the fluctuations of the stock and being more stable.

pandas provides functionality for easily computing moving averages. I demonstrate its use by creating a 20-day (one month) moving average for the Apple data, and plotting it alongside the stock.

```
apple["20d"] = np.round(apple["Adj. Close"].rolling(window = 20, center =
pandas_candlestick_ohlc(apple.loc['2016-01-04':'2016-12-31',:], otherserie
```

Notice how late the rolling average begins. It cannot be computed until 20 days have passed. This limitation becomes more severe for longer moving averages. Because I would like to be able to compute 200-day moving averages, I'm going to extend out how much AAPL data we have. That said, we will still largely focus on 2016.

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

You will notice that a moving average is much smoother than the actua stock data. Additionally, it's a stubborn indicator; a stock needs to be above or below the moving average line in order for the line to change direction. Thus, crossing a moving average signals a possible change in trend, and should draw attention.

Traders are usually interested in multiple moving averages, such as the 20-day, 50-day, and 200-day moving averages. It's easy to examine multiple moving averages at once.

```
apple["50d"] = np.round(apple["Adj. Close"].rolling(window = 50, center =
apple["200d"] = np.round(apple["Adj. Close"].rolling(window = 200, center

pandas_candlestick_ohlc(apple.loc['2016-01-04':'2016-12-31',:], otherserie
```

The 20-day moving average is the most sensitive to local changes, and the 200-day moving average the least. Here, the 200-day moving average indicates an overall **bearish** trend: the stock is trending downward over time. The 20-day moving average is at times bearish and at other times **bullish**, where a positive swing is expected. You can also see that the crossing of moving average lines indicate changes in trend. These crossings are what we can use as **trading signals**, or indications that a financial security is changind direction and a profitable trade might be made.

Trading Strategy

Our concern now is to design and evaluate trading strategies.

Any trader must have a set of rules that determine how much of her money she is willing to bet on any single trade. For example, a trader may decide that under no circumstances will she risk more than 10% of her portfolio on a trade. Additionally, in any trade, a trader must have an **exit strategy**, a set of conditions determining when she will exit the position, for either profit or loss. A trader may set a **target**, which is the minimum profit that will induce the trader to leave the position. Likewise, a trader may have a maximum loss she is willing to tolerate; if potential losses go beyond this amount, the trader will exit the position in order to prevent any further loss. We will suppose that the amount of money in the portfolio involved in any particular trade is a fixed proportion; 10% seems like a good number.

Here, I will be demonstrating a moving average crossover strategy (http://www.investopedia.com/university/movingaverage/movingaverages4.asp). We will use two moving averages, one we consider "fast", and the other "slow". The strategy is:

- Trade the asset when the fast moving average crosses over the slow moving average.
- Exit the trade when the fast moving average crosses over the slow moving average again.

A trade will be prompted when the fast moving average crosses from below to above the slow moving average, and the trade will be exited when the fast moving average crosses below the slow moving average later.

We now have a complete strategy. But before we decide we want to use it, we should try to evaluate the quality of the strategy first. The usual means for doing so is **backtesting**, which is looking at how profitable the strategy is on historical data. For example, looking at the above chart's performance on Apple stock, if the 20-day moving average is the fast moving average and the 50-day moving average the slow, this strategy does not appear to be very profitable, at least not if you are always taking long positions.

Let's see if we can automate the backtesting task. We first identify when the 20-day average is below the 50-day average, and vice versa.

```
apple['20d-50d'] = apple['20d'] - apple['50d']
apple.tail()
```

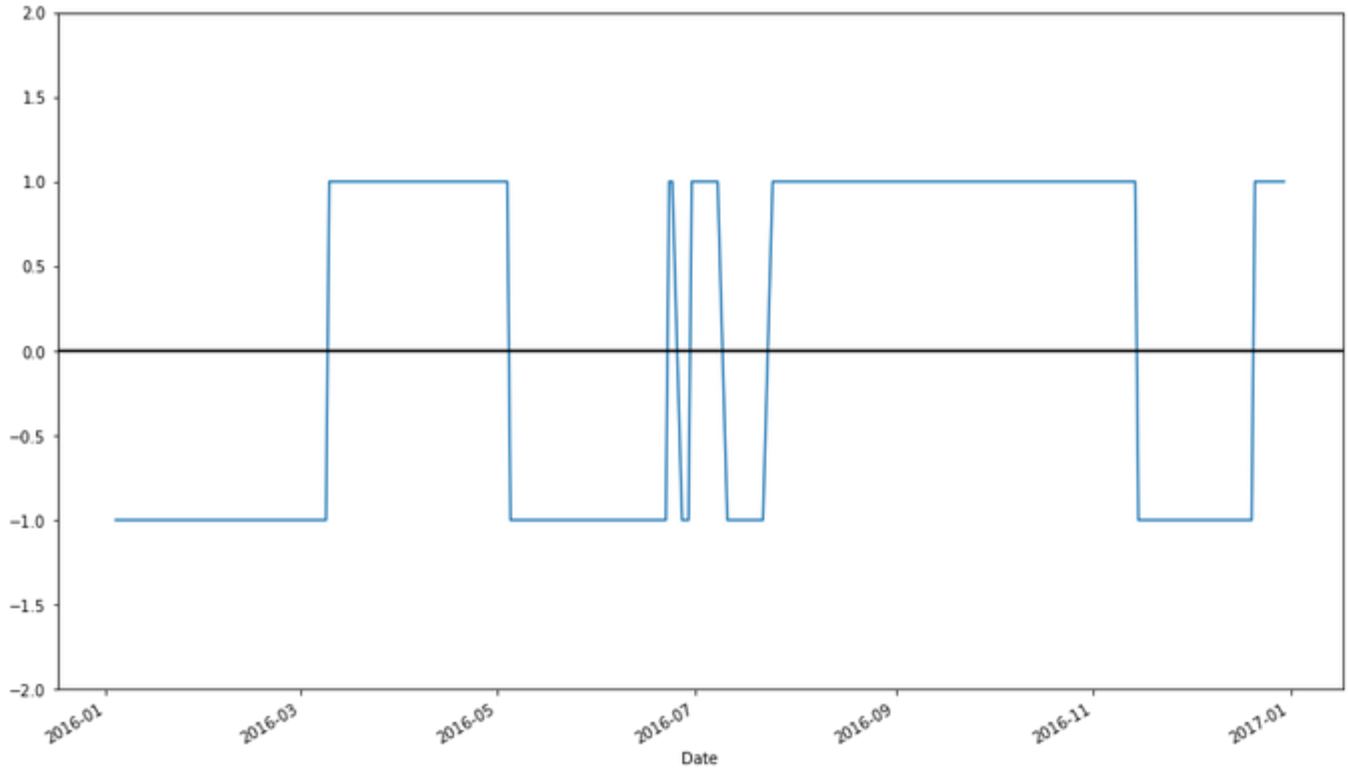
	Open	High	Low	Close	Volume
Date					
2018-03-21	175.04	175.09	171.26	171.270	35247358.0
2018-03-22	170.00	172.68	168.60	168.845	41051076.0
2018-03-23	168.39	169.92	164.94	164.940	40248954.0
2018-03-26	168.07	173.10	166.44	172.770	36272617.0
2018-03-27	173.68	175.15	166.92	168.340	38962839.0

Ex-Dividend	Split Ratio	Adj. Open	Adj. High	Adj. Low	Adj. Close
0.0	1.0	175.04	175.09	171.26	171.270
0.0	1.0	170.00	172.68	168.60	168.845
0.0	1.0	168.39	169.92	164.94	164.940
0.0	1.0	168.07	173.10	166.44	172.770
0.0	1.0	173.68	175.15	166.92	168.340

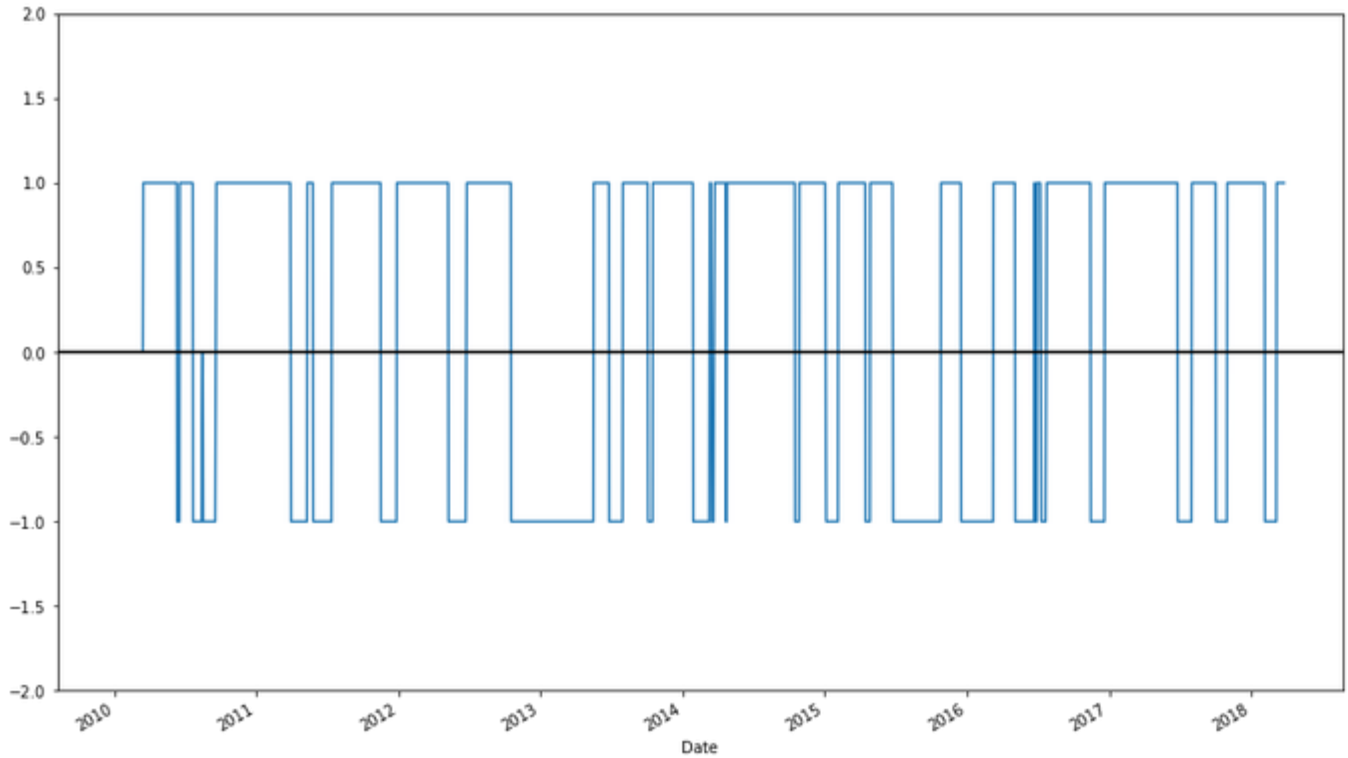
Adj. Volume	20d	50d	200d	20d-50d
35247358.0	176.94	172.57	162.68	4.37
41051076.0	176.76	172.46	162.75	4.30
40248954.0	176.23	172.27	162.81	3.96
36272617.0	175.92	172.22	162.91	3.70
38962839.0	175.41	172.05	162.98	3.36

We will refer to the sign of this difference as the **regime**; that is, if the fast moving average is above the slow moving average, this is a bullish regime (the bulls rule), and a bearish regime (the bears rule) holds when the fast moving average is below the slow moving average. I identify regimes with the following code.

```
# np.where() is a vectorized if-else function, where a condition is checke
```




```
apple["Regime"].plot(ylim = (-2,2)).axhline(y = 0, color = "black", lw = 2
```




```
apple["Regime"].value_counts()
```

```
 1    1323
-1     694
 0      53
```

The last line above indicates that for 1005 days the market was bearish on Apple, while for 600 days the market was bullish, and it was neutral for 54 days.

Trading signals appear at regime changes. When a bullish regime begins, a buy signal is triggered, and when it ends, a sell signal is triggered. Likewise, when a bearish regime begins, a sell signal is triggered, and when the regime ends, a buy signal is triggered (this is of interest only if you ever will short the stock, or use some derivative like a stock option to bet against the market).

It's simple to obtain signals. Let r_t indicate the regime at time t, and s_t the signal at time t. Then:

$$s_t = \text{sign}(r_t - r_{t-1})$$

$s_t \in \{-1, 0, 1\}$, with -1 indicating "sell", 1 indicating "buy", and 0 no action. We can obtain signals like so:

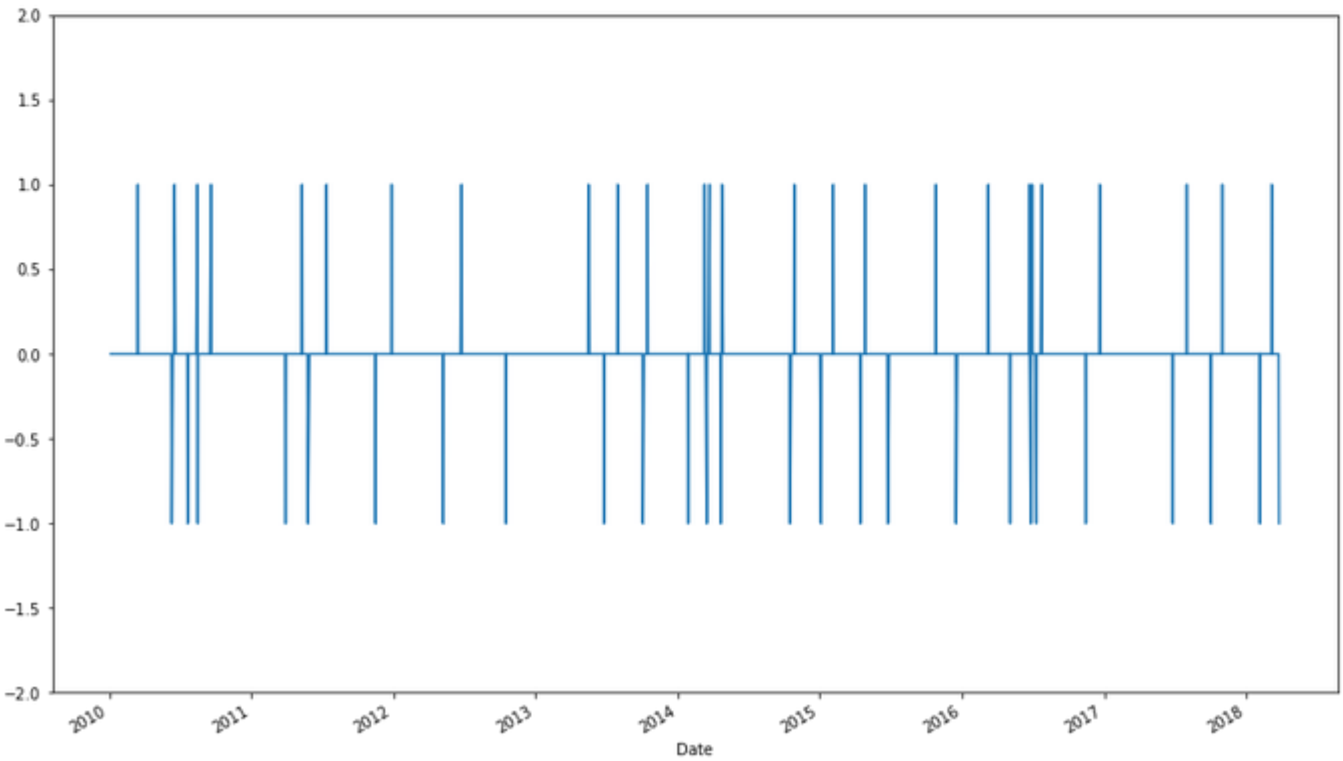
```
# To ensure that all trades close out, I temporarily change the regime of
regime_orig = apple.loc[:, "Regime"].iloc[-1]
apple.loc[:, "Regime"].iloc[-1] = 0
apple["Signal"] = np.sign(apple["Regime"] - apple["Regime"].shift(1))
# Restore original regime data
apple.loc[:, "Regime"].iloc[-1] = regime_orig
apple.tail()
```

```
/home/curtis/anaconda3/lib/python3.6/site-packages/pandas/core/indexing.py
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs
      self._setitem_with_indexer(indexer, value)
```

	Open	High	Low	Close	Volume	Ex-Dividend	Split Ratio	Adj. Open	Adj. High	Adj. Low	Adj. Close
Date											
2018-03-21	175.04	175.09	171.26	171.270	35247358.0	0.0	1.0	175.04	175.09	171.26	171.270
2018-03-22	170.00	172.68	168.60	168.845	41051076.0	0.0	1.0	170.00	172.68	168.60	168.845
2018-03-23	168.39	169.92	164.94	164.940	40248954.0	0.0	1.0	168.39	169.92	164.94	164.940
2018-03-26	168.07	173.10	166.44	172.770	36272617.0	0.0	1.0	168.07	173.10	166.44	172.770
2018-03-27	173.68	175.15	166.92	168.340	38962839.0	0.0	1.0	173.68	175.15	166.92	168.340

Adj. Volume	20d	50d	200d	20d-50d	Regime	Signal
35247358.0	176.94	172.57	162.68	4.37	1	0.0
41051076.0	176.76	172.46	162.75	4.30	1	0.0
40248954.0	176.23	172.27	162.81	3.96	1	0.0
36272617.0	175.92	172.22	162.91	3.70	1	0.0
38962839.0	175.41	172.05	162.98	3.36	1	-1.0

```
apple["Signal"].plot(ylim = (-2, 2))
```




```
apple["Signal"].value_counts()
```

```
0.0     2014
-1.0      28
 1.0      27
Name: Signal, dtype: int64
```

We would buy Apple stock 23 times and sell Apple stock 23 times. If we only go long on Apple stock, only 23 trades will be engaged in over the 6-year period, while if we pivot from a long to a short position every time a long position is terminated, we would engage in 23 trades total. (Bear in mind that trading more frequently isn't necessarily good; trades are never free.)

You may notice that the system as it currently stands isn't very robust, since even a fleeting moment when the fast moving average is above the slow moving average triggers a trade, resulting in trades that end immediately (which is bad if not simply because realistically every trade is accompanied by a fee that can quickly erode earnings). Additionally, every bullish regime immediately transitions into a bearish regime, and if you were constructing trading systems that allow both bullish and bearish bets, this would lead to the end of one trade immediately triggering a new trade that bets on the market in the opposite direction, which again seems finnicky. A better system would require more evidence that the market is moving in some particular direction. But we will not concern ourselves with these details for now.

Let's now try to identify what the prices of the stock is at every buy and every sell.

```
apple.loc[apple["Signal"] == 1, "Close"]
```

```
Date
2010-03-16    224.450
2010-06-18    274.074
2010-08-16    247.640
2010-09-20    283.230
2011-05-12    346.570
2011-07-14    357.770
2011-12-28    402.640
2012-06-25    570.765
2013-05-17    433.260
2013-07-31    452.530
2013-10-16    501.114
2014-03-11    536.090
2014-03-12    536.610
2014-03-24    539.190
2014-04-25    571.940
2014-10-28    106.740
2015-02-05    119.940
2015-04-28    130.560
2015-10-27    114.550
2016-03-10    101.170
2016-06-23     96.100
2016-06-30     95.600
2016-07-25     97.340
2016-12-21    117.060
2017-08-02    157.140
2017-11-01    166.890
2018-03-08    176.940
Name: Close, dtype: float64
```

```
apple.loc[apple["Signal"] == -1, "Close"]
```

```
Date
2010-06-11    253.5100
2010-07-22    259.0240
2010-08-17    251.9700
2011-03-30    348.6300
2011-03-31    348.5075
2011-05-27    337.4100
2011-11-17    377.4100
2012-05-09    569.1800
2012-10-17    644.6136
2013-06-26    398.0700
2013-10-04    483.0300
2014-01-28    506.5000
2014-03-17    526.7400
2014-04-22    531.6990
2014-10-17     97.6700
2015-01-05    106.2500
2015-04-16    126.1700
2015-06-25    127.5000
2015-06-26    126.7500
2015-12-18    106.0300
```

```
# Create a DataFrame with trades, including the price at the trade and th
apple_signals = pd.concat([
        pd.DataFrame({"Price": apple.loc[apple["Signal"] == 1, "Adj. Clos
                     "Regime": apple.loc[apple["Signal"] == 1, "Regime"],
                     "Signal": "Buy"}),
        pd.DataFrame({"Price": apple.loc[apple["Signal"] == -1, "Adj. Clo
                     "Regime": apple.loc[apple["Signal"] == -1, "Regime"]
                     "Signal": "Sell"}),
    ])
apple_signals.sort_index(inplace = True)
apple_signals
```

	Price	Regime	Signal
Date			
2010-03-16	28.844953	1	Buy
2010-06-11	32.579568	-1	Sell
2010-06-18	35.222329	1	Buy
2010-07-22	33.288194	-1	Sell
2010-08-16	31.825192	0	Buy
2010-08-17	32.381657	-1	Sell
2010-09-20	36.399003	1	Buy

	Price	Regime	Signal
Date			
2011-03-30	44.803814	0	Sell
2011-03-31	44.788071	-1	Sell
2011-05-12	44.539075	1	Buy
2011-05-27	43.361888	-1	Sell
2011-07-14	45.978431	1	Buy
2011-11-17	48.502445	-1	Sell
2011-12-28	51.744852	1	Buy
2012-05-09	73.147563	-1	Sell
2012-06-25	73.351258	1	Buy
2012-10-17	83.195498	-1	Sell
2013-05-17	56.878472	1	Buy
2013-06-26	52.258721	-1	Sell
2013-07-31	59.408242	1	Buy
2013-10-04	63.831819	-1	Sell
2013-10-16	66.221597	1	Buy
2014-01-28	67.325247	-1	Sell
2014-03-11	71.682490	0	Buy
2014-03-12	71.752021	1	Buy
2014-03-17	70.432269	-1	Sell
2014-03-24	72.097002	1	Buy
2014-04-22	71.095354	-1	Sell
2014-04-25	76.476120	1	Buy
2014-10-17	92.387441	-1	Sell
2014-10-28	100.966883	1	Buy
2015-01-05	100.937944	-1	Sell
2015-02-05	114.390004	1	Buy
2015-04-16	120.331722	-1	Sell
2015-04-28	124.518583	1	Buy
2015-06-25	122.104986	0	Sell
2015-06-26	121.386721	-1	Sell
2015-10-27	110.198438	1	Buy
2015-12-18	102.440744	-1	Sell
2016-03-10	98.271427	1	Buy

	Price	Regime	Signal
Date			
2016-05-05	91.122295	-1	Sell
2016-06-23	93.917337	1	Buy
2016-06-27	89.949550	-1	Sell
2016-06-30	93.428693	1	Buy
2016-07-11	94.777350	-1	Sell
2016-07-25	95.129174	1	Buy
2016-11-15	105.787035	-1	Sell
2016-12-21	115.614138	1	Buy
2017-06-27	143.159139	-1	Sell
2017-08-02	156.504989	1	Buy
2017-10-03	154.480000	-1	Sell
2017-11-01	166.890000	1	Buy
2018-02-06	163.030000	-1	Sell
2018-03-08	176.940000	1	Buy
2018-03-27	168.340000	1	Sell

```python
# Let's see the profitability of long trades
apple_long_profits = pd.DataFrame({
        "Price": apple_signals.loc[(apple_signals["Signal"] == "Buy") &
                                  apple_signals["Regime"] == 1, "Price"],
        "Profit": pd.Series(apple_signals["Price"] - apple_signals["Price
            apple_signals.loc[(apple_signals["Signal"].shift(1) == "Buy")
        ].tolist(),
        "End Date": apple_signals["Price"].loc[
            apple_signals.loc[(apple_signals["Signal"].shift(1) == "Buy")
        ].index
    })
apple_long_profits
```

	End Date	Price	Profit
Date			
2010-03-16	2010-06-11	28.844953	3.734615
2010-06-18	2010-07-22	35.222329	-1.934135
2010-09-20	2011-03-30	36.399003	8.404812
2011-05-12	2011-05-27	44.539075	-1.177188
2011-07-14	2011-11-17	45.978431	2.524014
2011-12-28	2012-05-09	51.744852	21.402711
2012-06-25	2012-10-17	73.351258	9.844240
2013-05-17	2013-06-26	56.878472	-4.619751

	End Date	Price	Profit
Date			
2013-07-31	2013-10-04	59.408242	4.423577
2013-10-16	2014-01-28	66.221597	1.103650
2014-03-12	2014-03-17	71.752021	-1.319753
2014-03-24	2014-04-22	72.097002	-1.001648
2014-04-25	2014-10-17	76.476120	15.911321
2014-10-28	2015-01-05	100.966883	-0.028939
2015-02-05	2015-04-16	114.390004	5.941719
2015-04-28	2015-06-25	124.518583	-2.413598
2015-10-27	2015-12-18	110.198438	-7.757693
2016-03-10	2016-05-05	98.271427	-7.149132
2016-06-23	2016-06-27	93.917337	-3.967788
2016-06-30	2016-07-11	93.428693	1.348657
2016-07-25	2016-11-15	95.129174	10.657861
2016-12-21	2017-06-27	115.614138	27.545001
2017-08-02	2017-10-03	156.504989	-2.024989
2017-11-01	2018-02-06	166.890000	-3.860000
2018-03-08	2018-03-27	176.940000	-8.600000

Let's now create a simulated portfolio of $1,000,000, and see how it would behave, according to the rules we have established. This includes:

- Investing only 10% of the portfolio in any trade
- Exiting the position if losses exceed 20% of the value of the trade.

When simulating, bear in mind that:

- Trades are done in batches of 100 stocks.
- Our stop-loss rule involves placing an order to sell the stock the moment the price drops below the specified level. Thus we need to check whether the lows during this period ever go low enough to trigger the stop-loss. Realistically, unless we buy a put option, we cannot guarantee that we will sell the stock at the price we set at the stop-loss, but we will use this as the selling price anyway for the sake of simplicity.
- Every trade is accompanied by a commission to the broker, which should be accounted for. I do not do so here.

Here's how a backtest may look:

```
# We need to get the low of the price during each trade.
tradeperiods = pd.DataFrame({"Start": apple_long_profits.index,
                            "End": apple_long_profits["End Date"]})
apple_long_profits["Low"] = tradeperiods.apply(lambda x: min(apple.loc[x["
apple_long_profits
```

	End Date	Price	Profit	Low
Date				
2010-03-16	2010-06-11	28.844953	3.734615	25.606402
2010-06-18	2010-07-22	35.222329	-1.934135	30.791939
2010-09-20	2011-03-30	36.399003	8.404812	35.341333
2011-05-12	2011-05-27	44.539075	-1.177188	42.335061
2011-07-14	2011-11-17	45.978431	2.524014	45.367990
2011-12-28	2012-05-09	51.744852	21.402711	51.471117
2012-06-25	2012-10-17	73.351258	9.844240	72.688768
2013-05-17	2013-06-26	56.878472	-4.619751	51.942335
2013-07-31	2013-10-04	59.408242	4.423577	59.001273
2013-10-16	2014-01-28	66.221597	1.103650	65.972629
2014-03-12	2014-03-17	71.752021	-1.319753	69.932180
2014-03-24	2014-04-22	72.097002	-1.001648	68.371743
2014-04-25	2014-10-17	76.476120	15.911321	75.409086
2014-10-28	2015-01-05	100.966883	-0.028939	99.652062
2015-02-05	2015-04-16	114.390004	5.941719	112.949876
2015-04-28	2015-06-25	124.518583	-2.413598	117.651750
2015-10-27	2015-12-18	110.198438	-7.757693	102.228192
2016-03-10	2016-05-05	98.271427	-7.149132	89.752692
2016-06-23	2016-06-27	93.917337	-3.967788	89.421814
2016-06-30	2016-07-11	93.428693	1.348657	92.158220
2016-07-25	2016-11-15	95.129174	10.657861	94.230069
2016-12-21	2017-06-27	115.614138	27.545001	113.342546
2017-08-02	2017-10-03	156.504989	-2.024989	149.160000
2017-11-01	2018-02-06	166.890000	-3.860000	154.000000
2018-03-08	2018-03-27	176.940000	-8.600000	164.940000

```python
# Now we have all the information needed to simulate this strategy in app
cash = 1000000
apple_backtest = pd.DataFrame({"Start Port. Value": [],
                               "End Port. Value": [],
                               "End Date": [],
                               "Shares": [],
                               "Share Price": [],
                               "Trade Value": [],
                               "Profit per Share": [],
                               "Total Profit": [],
                               "Stop-Loss Triggered": []})
port_value = .1  # Max proportion of portfolio bet on any trade
batch = 100      # Number of shares bought per batch
```

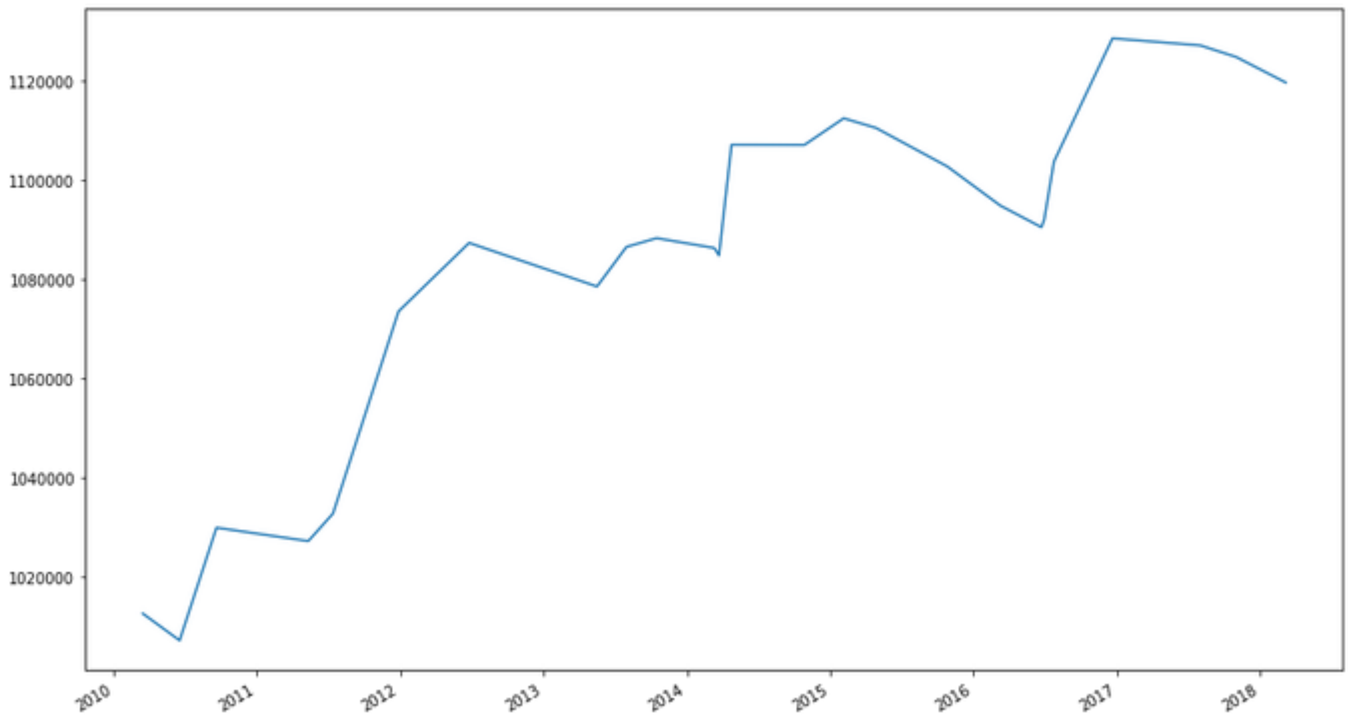
	End Date	End Port. Value	Profit per Share	Share Price	Shares
2010-03-16	2010-06-11	1.012698e+06	3.734615	28.844953	3400.0
2010-06-18	2010-07-22	1.007282e+06	-1.934135	35.222329	2800.0
2010-09-20	2011-03-30	1.029975e+06	8.404812	36.399003	2700.0
2011-05-12	2011-05-27	1.027268e+06	-1.177188	44.539075	2300.0
2011-07-14	2011-11-17	1.032820e+06	2.524014	45.978431	2200.0
2011-12-28	2012-05-09	1.073486e+06	21.402711	51.744852	1900.0
2012-06-25	2012-10-17	1.087267e+06	9.844240	73.351258	1400.0
2013-05-17	2013-06-26	1.078490e+06	-4.619751	56.878472	1900.0
2013-07-31	2013-10-04	1.086452e+06	4.423577	59.408242	1800.0
2013-10-16	2014-01-28	1.088218e+06	1.103650	66.221597	1600.0

	End Date	End Port. Value	Profit per Share	Share Price	Shares
2014-03-12	2014-03-17	1.086239e+06	-1.319753	71.752021	1500.0
2014-03-24	2014-04-22	1.084736e+06	-1.001648	72.097002	1500.0
2014-04-25	2014-10-17	1.107012e+06	15.911321	76.476120	1400.0
2014-10-28	2015-01-05	1.106983e+06	-0.028939	100.966883	1000.0
2015-02-05	2015-04-16	1.112331e+06	5.941719	114.390004	900.0
2015-04-28	2015-06-25	1.110400e+06	-2.413598	124.518583	800.0
2015-10-27	2015-12-18	1.102642e+06	-7.757693	110.198438	1000.0
2016-03-10	2016-05-05	1.094778e+06	-7.149132	98.271427	1100.0
2016-06-23	2016-06-27	1.090413e+06	-3.967788	93.917337	1100.0
2016-06-30	2016-07-11	1.091897e+06	1.348657	93.428693	1100.0
2016-07-25	2016-11-15	1.103621e+06	10.657861	95.129174	1100.0
2016-12-21	2017-06-27	1.128411e+06	27.545001	115.614138	900.0
2017-08-02	2017-10-03	1.126994e+06	-2.024989	156.504989	700.0
2017-11-01	2018-02-06	1.124678e+06	-3.860000	166.890000	600.0
2018-03-08	2018-03-27	1.119518e+06	-8.600000	176.940000	600.0

Start Port. Value	Stop-Loss Triggered	Total Profit	Trade Value
1.000000e+06	0.0	12697.691096	98072.841239
1.012698e+06	0.0	-5415.577333	98622.521053
1.007282e+06	0.0	22692.991110	98277.306914
1.029975e+06	0.0	-2707.531638	102439.873355
1.027268e+06	0.0	5552.830218	101152.549241
1.032820e+06	0.0	40665.151235	98315.218526
1.073486e+06	0.0	13781.935982	102691.760672
1.087267e+06	0.0	-8777.527400	108069.096937
1.078490e+06	0.0	7962.438409	106934.835757
1.086452e+06	0.0	1765.839598	105954.555657
1.088218e+06	0.0	-1979.628917	107628.031714
1.086239e+06	0.0	-1502.472160	108145.503103
1.084736e+06	0.0	22275.849051	107066.568572
1.107012e+06	0.0	-28.938709	100966.883069
1.106983e+06	0.0	5347.546691	102951.003221
1.112331e+06	0.0	-1930.878038	99614.866549
1.110400e+06	0.0	-7757.693367	110198.437846
1.102642e+06	0.0	-7864.045388	108098.569555

Start Port. Value	Stop-Loss Triggered	Total Profit	Trade Value
1.094778e+06	0.0	-4364.566368	103309.070918
1.090413e+06	0.0	1483.522558	102771.562745
1.091897e+06	0.0	11723.647322	104642.091188
1.103621e+06	0.0	24790.501098	104052.724175
1.128411e+06	0.0	-1417.492367	109553.492367
1.126994e+06	0.0	-2316.000000	100134.000000
1.124678e+06	0.0	-5160.000000	106164.000000

```
apple_backtest["End Port. Value"].plot()
```



Our portfolio's value grew by 13% in about six years. Considering that only 10% of the portfolio was ever involved in any single trade, this is not bad performance.

Notice that this strategy never lead to our rule of never allowing losses to exceed 20% of the trade's value being invoked. For the sake of simplicity, we will ignore this rule in backtesting.

A more realistic portfolio would not be betting 10% of its value on only one stock. A more realistic one would consider investing in multiple stocks. Multiple trades may be ongoing at any given time involving multiple companies, and most of the portfolio will be in stocks, not cash. Now that we will be investing in multiple stops and exiting only when moving averages cross (not because of a stop-loss), we will need to change our approach to backtesting. For example, we will be using one **pandas** `DataFrame` to contain all buy and sell orders for all stocks being considered, and our loop above will have to track more information.

I have written functions for creating order data for multiple stocks, and a function for performing the backtesting.

```
def ma_crossover_orders(stocks, fast, slow):
    """

    :param stocks: A list of tuples, the first argument in each tuple be
    :param fast: Integer for the number of days used in the fast moving
    :param slow: Integer for the number of days used in the slow moving

    :return: pandas DataFrame containing stock orders

    This function takes a list of stocks and determines when each stock
    """
    fast_str = str(fast) + 'd'
    slow_str = str(slow) + 'd'
    ma_diff_str = fast_str + '-' + slow_str

    trades = pd.DataFrame({"Price": [], "Regime": [], "Signal": []})
    for s in stocks:
        # Get the moving averages, both fast and slow, along with the di
        s[1][fast_str] = np.round(s[1]["Close"].rolling(window = fast, c
        s[1][slow_str] = np.round(s[1]["Close"].rolling(window = slow, c
        s[1][ma_diff_str] = s[1][fast_str] - s[1][slow_str]

        # np.where() is a vectorized if-else function, where a condition
        s[1]["Regime"] = np.where(s[1][ma_diff_str] > 0, 1, 0)
        # We have 1's for bullish regimes and 0's for everything else. B
        s[1]["Regime"] = np.where(s[1][ma_diff_str] < 0, -1, s[1]["Regim
        # To ensure that all trades close out, I temporarily change the
        regime_orig = s[1].loc[:, "Regime"].iloc[-1]
        s[1].loc[:, "Regime"].iloc[-1] = 0
        s[1]["Signal"] = np.sign(s[1]["Regime"] - s[1]["Regime"].shift(1
        # Restore original regime data
        s[1].loc[:, "Regime"].iloc[-1] = regime_orig

        # Get signals
        signals = pd.concat([
            pd.DataFrame({"Price": s[1].loc[s[1]["Signal"] == 1, "Adj. C
                          "Regime": s[1].loc[s[1]["Signal"] == 1, "Regime
                          "Signal": "Buy"}),
            pd.DataFrame({"Price": s[1].loc[s[1]["Signal"] == -1, "Adj.
                          "Regime": s[1].loc[s[1]["Signal"] == -1, "Regim
                          "Signal": "Sell"}),
        ])
        signals.index = pd.MultiIndex.from_product([signals.index, [s[0]
        trades = trades.append(signals)

    trades.sort_index(inplace = True)
    trades.index = pd.MultiIndex.from_tuples(trades.index, names = ["Dat

    return trades


def backtest(signals, cash, port_value = .1, batch = 100):
    """
    :param signals: pandas DataFrame containing buy and sell signals wit
    :param cash: integer for starting cash value
    :param port_value: maximum proportion of portfolio to risk on any si
    :param batch: Trading batch sizes

    :return: pandas DataFrame with backtesting results
```

```python

        This function backtests strategies, with the signals generated by th
        """

        SYMBOL = 1 # Constant for which element in index represents symbol
        portfolio = dict()    # Will contain how many stocks are in the port
        port_prices = dict()  # Tracks old trade prices for determining prof
        # Dataframe that will contain backtesting report
        results = pd.DataFrame({"Start Cash": [],
                                "End Cash": [],
                                "Portfolio Value": [],
                                "Type": [],
                                "Shares": [],
                                "Share Price": [],
                                "Trade Value": [],
                                "Profit per Share": [],
                                "Total Profit": []})

        for index, row in signals.iterrows():
            # These first few lines are done for any trade
            shares = portfolio.setdefault(index[SYMBOL], 0)
            trade_val = 0
            batches = 0
            cash_change = row["Price"] * shares   # Shares could potentially
            portfolio[index[SYMBOL]] = 0  # For a given symbol, a position i

            old_price = port_prices.setdefault(index[SYMBOL], row["Price"])
            portfolio_val = 0
            for key, val in portfolio.items():
                portfolio_val += val * port_prices[key]

            if row["Signal"] == "Buy" and row["Regime"] == 1:  # Entering a
                batches = np.floor((portfolio_val + cash) * port_value) // n
                trade_val = batches * batch * row["Price"] # How much money
                cash_change -= trade_val  # We are buying shares so cash wil
                portfolio[index[SYMBOL]] = batches * batch  # Recording how
                port_prices[index[SYMBOL]] = row["Price"]   # Record price
                old_price = row["Price"]
            elif row["Signal"] == "Sell" and row["Regime"] == -1: # Entering
                pass
                # Do nothing; can we provide a method for shorting the marke
            #else:
                #raise ValueError("I don't know what to do with signal " + r

            pprofit = row["Price"] - old_price   # Compute profit per share;

            # Update report
            results = results.append(pd.DataFrame({
                    "Start Cash": cash,
                    "End Cash": cash + cash_change,
                    "Portfolio Value": cash + cash_change + portfolio_val +
                    "Type": row["Signal"],
                    "Shares": batch * batches,
                    "Share Price": row["Price"],
                    "Trade Value": abs(cash_change),
                    "Profit per Share": pprofit,
                    "Total Profit": batches * batch * pprofit
                }, index = [index]))
            cash += cash_change  # Final change to cash balance
```

```python

        results.sort_index(inplace = True)
        results.index = pd.MultiIndex.from_tuples(results.index, names = ["D

        return results

    # Get more stocks
    (microsoft, google, facebook, twitter, netflix,
    amazon, yahoo, ge, qualcomm, ibm, hp) = (quandl.get("WIKI/" + s, start_d


```

```python
signals = ma_crossover_orders([("AAPL", apple),
                               ("MSFT",  microsoft),
                               ("GOOG",  google),
                               ("FB",    facebook),
                               ("TWTR",  twitter),
                               ("NFLX",  netflix),
                               ("AMZN",  amazon),
                               ("YHOO",  yahoo),
                               ("GE",    ge),
                               ("QCOM",  qualcomm),
                               ("IBM",   ibm),
                               ("HPQ",   hp)],
                              fast = 20, slow = 50)
signals
```

```
/home/curtis/anaconda3/lib/python3.6/site-packages/pandas/core/indexing.py
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs
self._setitem_with_indexer(indexer, value)
```

		Price	Regime	Signal
Date	**Symbol**			
2010-03-16	**AAPL**	28.844953	1.0	Buy
	AMZN	131.790000	1.0	Buy
	GE	14.129260	1.0	Buy
	HPQ	19.921951	1.0	Buy
	IBM	105.460506	1.0	Buy
	MSFT	23.978839	-1.0	Sell
	NFLX	10.090000	1.0	Buy
	QCOM	32.235226	-1.0	Sell
	YHOO	16.360000	-1.0	Sell
2010-03-17	**YHOO**	16.500000	1.0	Buy
2010-03-24	**MSFT**	24.207442	1.0	Buy
2010-04-01	**QCOM**	34.929069	1.0	Buy
2010-05-07	**QCOM**	30.161131	-1.0	Sell

		Price	Regime	Signal
Date	**Symbol**			
2010-05-10	**HPQ**	18.684203	-1.0	Sell
2010-05-17	**YHOO**	16.270000	-1.0	Sell
2010-05-19	**AMZN**	124.590000	-1.0	Sell
	GE	13.495907	-1.0	Sell
	MSFT	23.161072	-1.0	Sell
2010-05-20	**IBM**	102.001194	-1.0	Sell
2010-06-11	**AAPL**	32.579568	-1.0	Sell
2010-06-18	**AAPL**	35.222329	1.0	Buy
2010-06-29	**IBM**	103.064049	1.0	Buy
2010-06-30	**IBM**	101.737540	-1.0	Sell
2010-07-07	**IBM**	104.637735	1.0	Buy
2010-07-20	**IBM**	104.266971	-1.0	Sell
2010-07-22	**AAPL**	33.288194	-1.0	Sell
2010-07-27	**QCOM**	32.585294	1.0	Buy
2010-07-28	**IBM**	105.815940	1.0	Buy
2010-07-29	**NFLX**	14.002857	-1.0	Sell
2010-08-02	**HPQ**	18.129988	1.0	Buy
...	**...**	...	...	...
2017-11-01	**AAPL**	166.890000	1.0	Buy
2017-12-06	**NFLX**	185.300000	-1.0	Sell
2017-12-15	**HPQ**	20.920000	-1.0	Sell
2017-12-26	**FB**	175.990000	-1.0	Sell
2018-01-03	**FB**	184.670000	1.0	Buy
2018-01-09	**NFLX**	209.310000	1.0	Buy
2018-01-11	**HPQ**	22.410000	1.0	Buy
2018-01-18	**QCOM**	68.050000	-1.0	Sell
2018-01-19	**QCOM**	68.040000	1.0	Buy
2018-02-06	**AAPL**	163.030000	-1.0	Sell
2018-02-21	**IBM**	153.960000	-1.0	Sell
	QCOM	63.400000	-1.0	Sell
2018-02-22	**HPQ**	21.390000	-1.0	Sell
2018-02-23	**FB**	183.290000	-1.0	Sell
2018-02-27	**GOOG**	1118.290000	-1.0	Sell

		Price	Regime	Signal
Date	**Symbol**			
2018-03-08	**AAPL**	176.940000	1.0	Buy
2018-03-09	**HPQ**	24.650000	1.0	Buy
2018-03-14	**GOOG**	1149.490000	1.0	Buy
2018-03-23	**GOOG**	1021.570000	-1.0	Sell
2018-03-27	**AAPL**	168.340000	1.0	Sell
	AMZN	1497.050000	1.0	Sell
	FB	152.190000	-1.0	Buy
	GE	13.440000	-1.0	Buy
	GOOG	1005.100000	-1.0	Buy
	HPQ	21.770000	1.0	Sell
	IBM	151.910000	-1.0	Buy
	MSFT	89.470000	1.0	Sell
	NFLX	300.690000	1.0	Sell
	QCOM	54.840000	-1.0	Buy
	TWTR	28.070000	1.0	Sell

511 rows × 3 columns

```
bk = backtest(signals, 1000000)
bk
```

		End Cash	Portfolio Value	Profit per Share
Date	**Symbol**			
2010-03-16	**AAPL**	9.019272e+05	1.000000e+06	0.000000
	AMZN	8.096742e+05	1.000000e+06	0.000000
	GE	7.107693e+05	1.000000e+06	0.000000
	HPQ	6.111596e+05	1.000000e+06	0.000000
	IBM	5.162451e+05	1.000000e+06	0.000000
	MSFT	5.162451e+05	1.000000e+06	0.000000
	NFLX	4.163541e+05	1.000000e+06	0.000000
	QCOM	4.163541e+05	1.000000e+06	0.000000
	YHOO	4.163541e+05	1.000000e+06	0.000000
2010-03-17	**YHOO**	3.173541e+05	1.000000e+06	0.000000
2010-03-24	**MSFT**	2.181036e+05	1.000000e+06	0.000000
2010-04-01	**QCOM**	1.203022e+05	1.000000e+06	0.000000
2010-05-07	**QCOM**	2.047534e+05	9.866498e+05	-4.767938

		End Cash	Portfolio Value	Profit per Share
Date	**Symbol**			
2010-05-10	**HPQ**	2.981744e+05	9.804610e+05	-1.237749
2010-05-17	**YHOO**	3.957944e+05	9.790810e+05	-0.230000
2010-05-19	**AMZN**	4.830074e+05	9.740410e+05	-7.200000
	GE	5.774787e+05	9.696076e+05	-0.633354
	MSFT	6.724391e+05	9.653174e+05	-1.046370
2010-05-20	**IBM**	7.642402e+05	9.622041e+05	-3.459312
2010-06-11	**AAPL**	8.750107e+05	9.749017e+05	3.734615
2010-06-18	**AAPL**	7.799105e+05	9.749017e+05	0.000000
2010-06-29	**IBM**	6.871528e+05	9.749017e+05	0.000000
2010-06-30	**IBM**	7.787166e+05	9.737079e+05	-1.326510
2010-07-07	**IBM**	6.845426e+05	9.737079e+05	0.000000
2010-07-20	**IBM**	7.783829e+05	9.733742e+05	-0.370764
2010-07-22	**AAPL**	8.682610e+05	9.681520e+05	-1.934135
2010-07-27	**QCOM**	7.737637e+05	9.681520e+05	0.000000
2010-07-28	**IBM**	6.785293e+05	9.681520e+05	0.000000
2010-07-29	**NFLX**	8.171576e+05	1.006889e+06	3.912857
2010-08-02	**HPQ**	7.174427e+05	1.006889e+06	0.000000
...	**...**	...	...	...
2017-11-01	**AAPL**	1.297792e+05	2.153164e+06	0.000000
2017-12-06	**NFLX**	3.336092e+05	2.149600e+06	-3.240000
2017-12-15	**HPQ**	5.700052e+05	2.170395e+06	1.840267
2017-12-26	**FB**	8.339902e+05	2.244450e+06	49.370000
2018-01-03	**FB**	6.123862e+05	2.244450e+06	0.000000
2018-01-09	**NFLX**	4.030762e+05	2.244450e+06	0.000000
2018-01-11	**HPQ**	1.789762e+05	2.244450e+06	0.000000
2018-01-18	**QCOM**	4.511762e+05	2.301960e+06	14.377402
2018-01-19	**QCOM**	2.266442e+05	2.301960e+06	0.000000
2018-02-06	**AAPL**	4.222802e+05	2.297328e+06	-3.860000
2018-02-21	**IBM**	6.378242e+05	2.309716e+06	8.848221
	QCOM	8.470442e+05	2.294404e+06	-4.640000
2018-02-22	**HPQ**	1.060944e+06	2.284204e+06	-1.020000
2018-02-23	**FB**	1.280892e+06	2.282548e+06	-1.380000
2018-02-27	**GOOG**	1.504550e+06	2.316306e+06	168.790000

		End Cash	Portfolio Value	Profit per Share
Date	**Symbol**			
2018-03-08	**AAPL**	1.274528e+06	2.316306e+06	0.000000
2018-03-09	**HPQ**	1.045283e+06	2.316306e+06	0.000000
2018-03-14	**GOOG**	8.153852e+05	2.316306e+06	0.000000
2018-03-23	**GOOG**	1.019699e+06	2.290722e+06	-127.920000
2018-03-27	**AAPL**	1.238541e+06	2.279542e+06	-8.600000
	AMZN	1.537951e+06	2.377684e+06	490.710000
	FB	1.537951e+06	2.377684e+06	-32.480000
	GE	1.537951e+06	2.377684e+06	-16.213194
	GOOG	1.537951e+06	2.377684e+06	-144.390000
	HPQ	1.740412e+06	2.350900e+06	-2.880000
	IBM	1.740412e+06	2.350900e+06	6.798221
	MSFT	2.026716e+06	2.451672e+06	31.491454
	NFLX	2.327406e+06	2.543052e+06	91.380000
	QCOM	2.327406e+06	2.543052e+06	-13.200000
	TWTR	2.683895e+06	2.683895e+06	11.090000

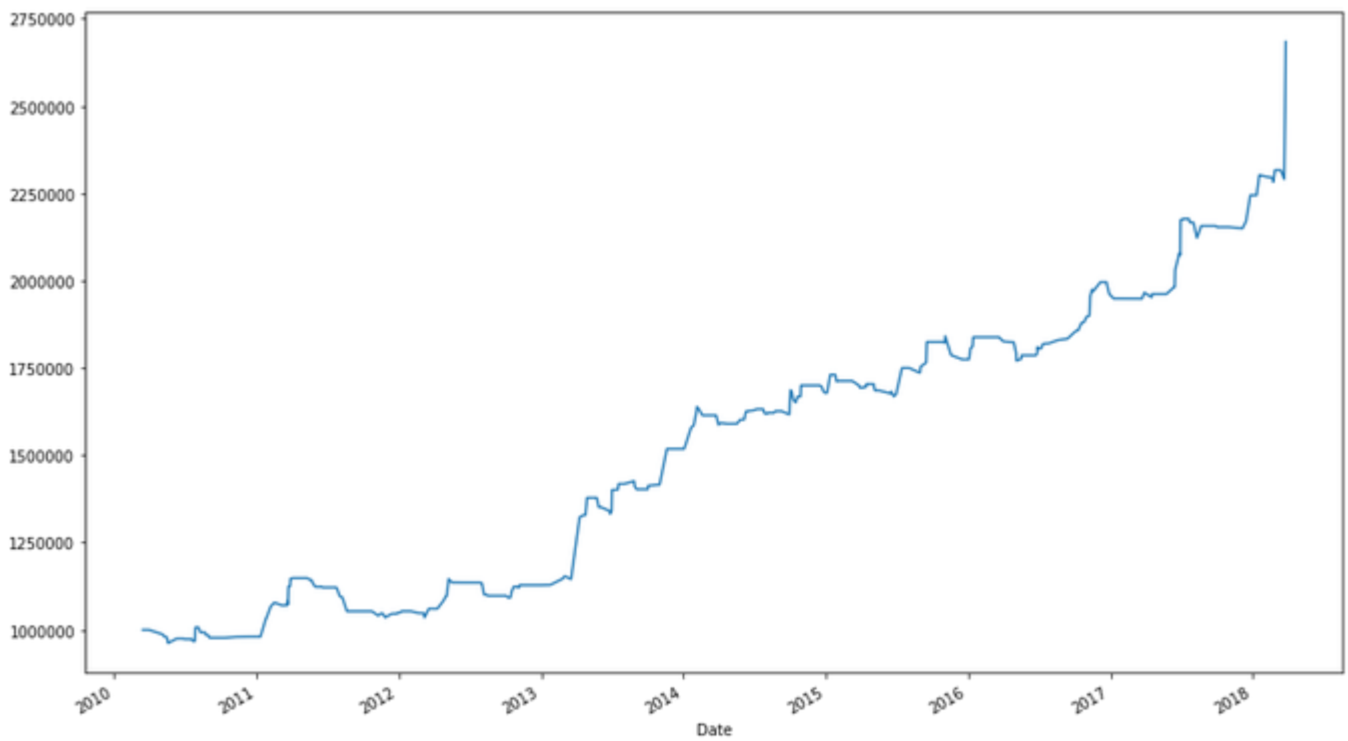
Share Price	Shares	Start Cash	Total Profit	Trade Value	Type
28.844953	3400.0	1.000000e+06	0.0	98072.841239	Buy
131.790000	700.0	9.019272e+05	0.0	92253.000000	Buy
14.129260	7000.0	8.096742e+05	0.0	98904.822860	Buy
19.921951	5000.0	7.107693e+05	0.0	99609.756314	Buy
105.460506	900.0	6.111596e+05	0.0	94914.455453	Buy
23.978839	0.0	5.162451e+05	0.0	0.000000	Sell
10.090000	9900.0	5.162451e+05	0.0	99891.000000	Buy
32.235226	0.0	4.163541e+05	0.0	0.000000	Sell
16.360000	0.0	4.163541e+05	0.0	0.000000	Sell
16.500000	6000.0	4.163541e+05	0.0	99000.000000	Buy
24.207442	4100.0	3.173541e+05	0.0	99250.512998	Buy
34.929069	2800.0	2.181036e+05	0.0	97801.393965	Buy
30.161131	0.0	1.203022e+05	-0.0	84451.168198	Sell
18.684203	0.0	2.047534e+05	-0.0	93421.012620	Sell
16.270000	0.0	2.981744e+05	-0.0	97620.000000	Sell
124.590000	0.0	3.957944e+05	-0.0	87213.000000	Sell

Share Price	Shares	Start Cash	Total Profit	Trade Value	Type
13.495907	0.0	4.830074e+05	-0.0	94471.347126	Sell
23.161072	0.0	5.774787e+05	-0.0	94960.396545	Sell
102.001194	0.0	6.724391e+05	-0.0	91801.074363	Sell
32.579568	0.0	7.642402e+05	0.0	110770.532335	Sell
35.222329	2700.0	8.750107e+05	0.0	95100.288159	Buy
103.064049	900.0	7.799105e+05	0.0	92757.644524	Buy
101.737540	0.0	6.871528e+05	-0.0	91563.785641	Sell
104.637735	900.0	7.787166e+05	0.0	94173.961584	Buy
104.266971	0.0	6.845426e+05	-0.0	93840.274319	Sell
33.288194	0.0	7.783829e+05	-0.0	89878.124302	Sell
32.585294	2900.0	8.682610e+05	0.0	94497.352787	Buy
105.815940	900.0	7.737637e+05	0.0	95234.345561	Buy
14.002857	0.0	6.785293e+05	0.0	138628.285714	Sell
18.129988	5500.0	8.171576e+05	0.0	99714.934419	Buy
…	…	…	…	…	…
166.890000	1200.0	3.300472e+05	0.0	200268.000000	Buy
185.300000	0.0	1.297792e+05	-0.0	203830.000000	Sell
20.920000	0.0	3.336092e+05	0.0	236396.000000	Sell
175.990000	0.0	5.700052e+05	0.0	263985.000000	Sell
184.670000	1200.0	8.339902e+05	0.0	221604.000000	Buy
209.310000	1000.0	6.123862e+05	0.0	209310.000000	Buy
22.410000	10000.0	4.030762e+05	0.0	224100.000000	Buy
68.050000	0.0	1.789762e+05	0.0	272200.000000	Sell
68.040000	3300.0	4.511762e+05	0.0	224532.000000	Buy
163.030000	0.0	2.266442e+05	-0.0	195636.000000	Sell
153.960000	0.0	4.222802e+05	0.0	215544.000000	Sell
63.400000	0.0	6.378242e+05	-0.0	209220.000000	Sell
21.390000	0.0	8.470442e+05	-0.0	213900.000000	Sell
183.290000	0.0	1.060944e+06	-0.0	219948.000000	Sell
1118.290000	0.0	1.280892e+06	0.0	223658.000000	Sell
176.940000	1300.0	1.504550e+06	0.0	230022.000000	Buy
24.650000	9300.0	1.274528e+06	0.0	229245.000000	Buy
1149.490000	200.0	1.045283e+06	0.0	229898.000000	Buy

Share Price	Shares	Start Cash	Total Profit	Trade Value	Type
1021.570000	0.0	8.153852e+05	-0.0	204314.000000	Sell
168.340000	0.0	1.019699e+06	-0.0	218842.000000	Sell
1497.050000	0.0	1.238541e+06	0.0	299410.000000	Sell
152.190000	0.0	1.537951e+06	-0.0	0.000000	Buy
13.440000	0.0	1.537951e+06	-0.0	0.000000	Buy
1005.100000	0.0	1.537951e+06	-0.0	0.000000	Buy
21.770000	0.0	1.537951e+06	-0.0	202461.000000	Sell
151.910000	0.0	1.740412e+06	0.0	0.000000	Buy
89.470000	0.0	1.740412e+06	0.0	286304.000000	Sell
300.690000	0.0	2.026716e+06	0.0	300690.000000	Sell
54.840000	0.0	2.327406e+06	-0.0	0.000000	Buy
28.070000	0.0	2.327406e+06	0.0	356489.000000	Sell

511 rows × 9 columns

```
bk["Portfolio Value"].groupby(level = 0).apply(lambda x: x[-1]).plot()
```



A more realistic portfolio that can invest in any in a list of twelve (tech) stocks has a final growth of about 100%. How good is this? While on the surface not bad, we will see we could have done better.

Benchmarking

Backtesting is only part of evaluating the efficacy of a trading strategy. We would like to **benchmark** the strategy, or compare it to other available (usually well-known) strategies in order to determine how well we have done.

Whenever you evaluate a trading system, there is one strategy that you should always check, one that beats all but a handful of managed mutual funds and investment managers: buy and hold SPY (https://finance.yahoo.com/quote/SPY). The **efficient market hypothesis** claims that it is all but impossible for anyone to beat the market. Thus, one should always buy an index fund that merely reflects the composition of the market.By buying and holding SPY, we are effectively trying to match our returns with the market rather than beat it.

I look at the profits for simply buying and holding SPY.

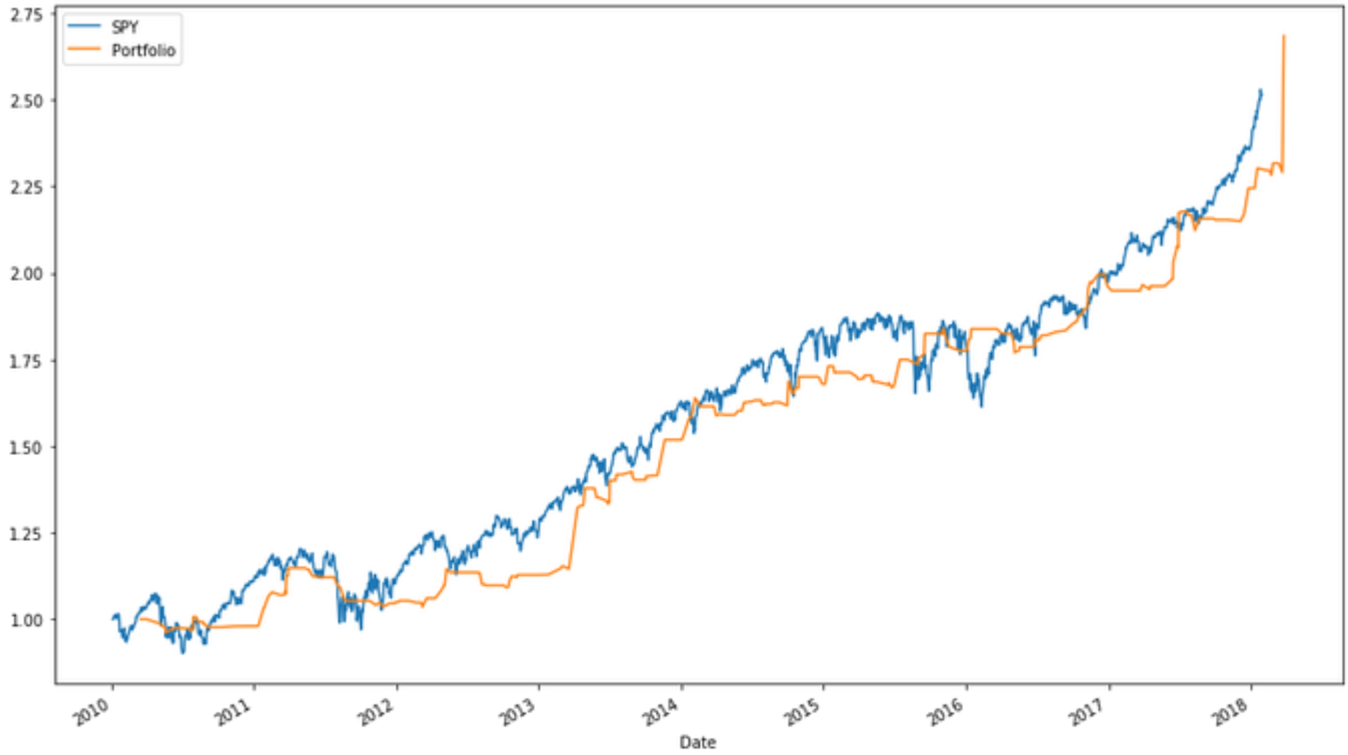
```
#spyder = web.DataReader("SPY", "yahoo", start, end)
spyder = spyderdat.loc[start:end]
spyder.iloc[[0,-1],:]
```

	Open	High	Low	Close	Adj Close
date					
2010-01-04	112.37	113.39	111.51	113.33	113.33
2018-01-29	285.93	286.43	284.50	284.68	284.68

```
batches = 1000000 // np.ceil(100 * spyder.loc[:,"Adj Close"].iloc[0]) # Ma
trade_val = batches * batch * spyder.loc[:,"Adj Close"].iloc[0] # How much
final_val = batches * batch * spyder.loc[:,"Adj Close"].iloc[-1] + (100000
final_val
```

```
2507880.0
```

```
# We see that the buy-and-hold strategy beats the strategy we developed ea
ax_bench = (spyder["Adj Close"] / spyder.loc[:, "Adj Close"].iloc[0]).plot
ax_bench = (bk["Portfolio Value"].groupby(level = 0).apply(lambda x: x[-1]
ax_bench.legend(ax_bench.get_lines(), [l.get_label() for l in ax_bench.get
ax_bench
```



Buying and holding SPY performs about as well as our trading system, at least how we currently set it up, and we haven't even accounted for how expensive our more complex strategy is in terms of fees. Given both the opportunity cost and the expense associated with the active strategy, we should not use it.

What could we do to improve the performance of our system? For starters, we could try diversifying. All the stocks we considered were tech companies, which means that if the tech industry is doing poorly, our portfolio will reflect that. We could try developing a system that can also short stocks or bet bearishly, so we can take advantage of movement in any direction. We could seek means for forecasting how high we expect a stock to move. Whatever we do, though, must beat this benchmark; otherwise there is an opportunity cost associated with our trading system.

Other benchmark strategies exist, and if our trading system beat the "buy and hold SPY" strategy, we may check against them. Some such strategies include:

- Buy SPY when its closing monthly price is aboves its ten-month moving average.
- Buy SPY when its ten-month momentum is positive. (**Momentum** is the first difference of a moving average process, or $MO_t^q = MA_t^q - MA_{t-1}^q$.)

(I first read of these strategies here (https://www.r-bloggers.com/are-r2s-useful-in-finance-hypothesis-driven-development-in-reverse/?utm_source=feedburner&utm_medium=email&utm_campaign=Feed%3A+RBloggers+%28R+bloggers%29).) The general lesson still holds: *don't use a complex trading system with lots of active trading when a simple strategy involving an index fund without frequent trading beats it.* This is actually a very difficult requirement to meet. (http://www.nytimes.com/2015/03/15/your-money/how-many-mutual-funds-routinely-rout-the-market-zero.html?_r=0)

As a final note, suppose that your trading system *did* manage to beat any baseline strategy thrown at it in backtesting. Does backtesting predict future performance? Not at all. Backtesting has a propensity for overfitting (http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2745220), so just because backtesting predicts high growth doesn't mean that growth will hold in the future. There are strategies for combatting overfitting, such as walk-forward analysis (https://ntguardian.wordpress.com/2017/06/19/walk-forward-analysis-demonstrationbacktrader/) and holding out a portion of a dataset (likely the most recent part) as a final test set to determine if a strategy is profitable, followed by "sitting on" a strategy that managed to survive these two filters and seeing if it remains profitable in current markets.

Conclusion

While this lecture ends on a depressing note, keep in mind that the efficient market hypothesis has many critics. (http://www.nytimes.com/2009/06/06/business/06nocera.html) My own opinion is that as trading becomes more algorithmic, beating the market will become more difficult. That said, it may be possible to beat the market, even though mutual funds seem incapable of doing so (bear in mind, though, that part of the reason mutual funds perform so poorly is because of fees, which is not a concern for index funds).

This lecture is very brief, covering only one type of strategy: strategies based on moving averages. Many other trading signals exist and employed. Additionally, we never discussed in depth shorting stocks, currency trading, or stock options. Stock options, in particular, are a rich subject that offer many different ways to bet on the direction of a stock. You can read more about derivatives (including stock options and other derivatives) in the book *Derivatives Analytics with Python: Data Analysis, Models, Simulation, Calibration and Hedging*, which is available from the University of Utah library. (http://proquest.safaribooksonline.com.ezproxy.lib.utah.edu/9781119037996)

Another resource (which I used as a reference while writing this lecture) is the O'Reilly book *Python for Finance*, also available from the University of Utah library. (http://proquest.safaribooksonline.com.ezproxy.lib.utah.edu/book/programming/python/9781491945360)

If you were interested in investigating algorithmic trading, where would you go from here? I would not recommend using the code I wrote above for backtesting; there are better packages for this task. Python has some libraries for algorithmic trading, such as **pyfolio** (https://quantopian.github.io/pyfolio/) (for analytics), **zipline** (http://www.zipline.io/beginner-tutorial.html) (for backtesting and algorithmic trading), and **backtrader** (https://www.backtrader.com/) (also for backtesting and trading). **zipline** seems to be popular likely because it is used and developed by **quantopian** (https://www.quantopian.com/), a "crowd-sourced hedge fund" that allows users to use their data for backtesting and even will license profitable strategies from their authors, giving them a cut of the profits. However, I prefer **backtrader** and have written blog posts (https://ntguardian.wordpress.com/tag/backtrader/) on using it. It is likely the more complicated between the two but that's the cost of greater power. I am a fan of its design. I also would suggest learning R (https://www.r-project.org/), since it has many packages for analyzing financial data (moreso than Python) and it's surprisingly easy to use R functions in Python (as I demonstrate in this post (https://ntguardian.wordpress.com/2017/06/28/stock-trading-analytics-and-optimization-in-python-with-pyfolio-rs-performanceanalytics-and-backtrader/)).

You can read more about using R and Python for finance on my blog (https://ntguardian.wordpress.com/category/economics-and-finance/).

Remember that it is possible (if not common) to lose money in the stock market. It's also true, though, that it's difficult to find returns like those found in stocks, and any investment strategy should take investing in it seriously. This lecture is intended to provide a starting point for evaluating stock trading and investments, and, more generally, analyzing temporal data, and I hope you continue to explore these ideas.

I have created a video course published by Packt Publishing (http://packtpub.com/) entitled *Training Your Systems with Python Statistical Modeling* (https://ntguardian.wordpress.com/video-courses/training-your-systems-with-python-statistical-modeling/), the third volume in a four-volume set of video courses entitled, *Taming Data with Python; Excelling as a Data Analyst*. This course discusses how to use Python for machine learning. The course covers classical statistical methods, supervised learning including classification and regression, clustering, dimensionality reduction, and more! The course is peppered with examples demonstrating the techniques and software on real-world data and visuals to explain the concepts presented. Viewers get a hands-on experience using Python for machine learning. If you are starting out using Python for data analysis or know someone who is, please consider buying my course (https://www.packtpub.com/big-data-and-business-intelligence/training-your-systems-python-statistical-modeling-video) or at least spreading the word about it. You can buy the course directly or purchase a subscription to Mapt (https://www.packtpub.com/mapt/) and watch it there.

If you like my blog and would like to support it, spread the word (if not get a copy yourself)! Also, stay tuned for future courses I publish with Packt at the Video Courses section of my site.

algorithmic trading alpha apple backtesting bear market benchmarking beta bull market candlestick chart cs 5160 efficient market hypothesis etf finance financial crisis financial sector flash crash google google finance hft math 4100 matplotlib microsoft moving average moving average crossover strategy numpy pandas quandl reagan s&p 500 scipy spdy stock market stocks visualization yahoo finance

4 thoughts on "Stock Data Analysis with Python (Second Edition)"

Pingback: An Introduction to Stock Market Data Analysis with Python (Part 1) | Curtis Miller's Personal Website

Pingback: An Introduction to Stock Market Data Analysis with Python (Part 2) | Curtis Miller's Personal Website

have you considered publishing your python code as a jupyter notebook?

ninjaz155 , August 1, 2018 at 5:02 pm
Reply

I may do that. Maybe.

ntguardian , August 1, 2018 at 5:27 pm
Reply