Python in algotrading lectures

Hugo E. Ramirez
hugoedu.ramirez@urosario.edu.co

Facultad de Economía. Universidad del Rosario
Scipy Latam

2019

Outline

Why Python? I

- Free and open source
- Widely used in Quant community. `Python` has an increasing use among scientist and trading communities.
- High-level and general purpose programming language.
- Extensible and interoperable with other languages (`Gnuplot`).
- Has standard free libraries for finance (Ready to use!).
- Since it is not as complex as more structured languages (C++ or Java) enables us to focus on the specific topic rather than the programming. Learning curve is much faster than other used languages.
- Allows Object-Oriented, procedural and functional programming.
- It offers the right balance between programming complexity and computational speed.
- Can be thought for many levels of computational knowledg

Universidad del Rosario

Some definitions I

Algorithmic trading: Means turning a trading idea into a trading strategy via an algorithm. The algorithmic trading strategy can be executed either manually or in an automated way.
Backtesting: Test a trading strategy with historical data to check whether it will give good returns in real markets.
Quantitative trading: This involves using advanced mathematical and statistical models for creating and executing trading strategies.
Automated trading: Completely automating the order generation, submission, and the order execution process.
HFT or high-frequency trading: Trading strategies can be categorized as low-frequency, medium-frequency and high-frequency strategies as per the holding time of the trades. High-frequency strategies are algorithmic strategies which get executed in an automated way in quick time, usually on a sub-second time scale.

Some definitions II

In Algorithmic trading, we follow four "simple" steps:

1. Hypothesis. Market insight.
2. Model. Mathematics.
3. Validation. Backtesting and calibration.
4. Analysis. Why it works

There are some known strategies: Portfolio optimisation, Pairs trading, Market making, trend following, Moving average crossover, HJB's, Kalman filter signalling, and else.
Some known algorithms to "make" money: Moving averages, Knight-Satchell-Tran process, Bull/Bear, Co-integration, Mean reverting, factor models, HJB mean reverting, stochastic volatility, and else.

Financial/Computational motivation I

- Capability to process Data: Nowadays it is not enough to consider the end-of-day quotes for financial instruments and there is a lot of information during the day (High-frequency trading)
- Speed in analytics: The decisions for investors have to be made very fast and thus large amounts of data have to be analysed in short periods of time.
- Numerical solutions instead of analytical solutions: Since are more readable and sometimes more accessible.
- Difficult or not possible analytical solutions to problems:

$$\int_a^b e^{-x^2} dx, \quad \text{for } a < b \in \mathbb{R}$$

- Trading is still very much an art rather than a science.

Financial/Computational motivation II

- Electronic trading has transformed the markets (more control and accessibility).
- A computerised system is now responsible for executing orders, by following a set of rules.
- The famous Capital Asset Pricing Model that measures a stock's beta is really just a linear regression equation. No reason for human power to do this.
- In trading, behavioural variables that can affect security values. Such as herding, overconfidence, overreaction to earnings announcements, and momentum trading can all lead to (human) errors.

Universidad del Rosario

Use in the classroom I

- Anaconda + Spyder. Must be installed in your computer. Standard programming interaction.
- Anaconda + Jupyter. Must be installed in your computer. Allows to create documents executable with text/equations, code and graphs.
- Jupyter notebooks (`jupyter.org`, `cocalc.com`). Open-source web application that allows to create and share documents that contain live code. You can also share and publish online your documents, for public to view.
- kaggle (`www.kaggle.com`). Collaboration community with customisable Jupyter Notebooks environment that access free online GPU's. Contains public notebooks with codes and data. Courses and even competitions.

Universidad del Rosario

Use in the classroom II

- google Colaboratory (`colab.research.google.com`). Free environment of Jupyter Notebook. Does not require much configuration. Executed on the cloud with powerful computational resources (GPU and TPU).

It is always important to import the desired libraries, we mostly use `pandas` ("PANel DAta") and `numpy` ("NUMerical PYthon") In finance data comes in forms of time series and/or tables (Similar to DB tables or excel spreadsheets). In `Python pandas`:

- The first corresponds to `Series`. The primary building block of pandas and represents a one-dimensional array, which may be indexed.
- The second is `DataFrame`. Usually in finance these are indexed by dates.

The to create an object of type `Series` we may use an `array`, `dictionary` or else, or create it as an empty object. For example:

Pandas II

```
1 import pandas as pd
2 import numpy as np
3 # Create a Series with default index system
4 s = pd.Series(np.random.randn(20))
5 # Create a Series with different index than default
6 s2 = pd.Series([3.2,2.7,4.5], index=['Juan','Pedro','
      Maria'])
7 #Or could be created as dictionary
8 s3 = pd.Series({'Juan':3.5, 'Andrea':3.5,'Pedro':4.0,'
      Maria':4.3, 'default':np.nan})
```

These objects are always indexed. and operations are performed over indices.

`Series` can only associate **a single** value with any given index label, it has only limited ability to model multiple variables.

`DataFrame` may be compared to a relational database table. An example of a DataFrame is:

	Name	Age	Height	···	Salary
index1	0.8	0.8	0.0	···	0.9
index2	0.4	0.8	0.3	···	0.1
index3	0.7	0.3	0.7	···	0.2
⋮	⋮	⋮	⋮	···	⋮

and the next code creates some random `Dataframe` with five row and four columns A, B, C, D

```
1 #Create a DataFrame
2 np.random.seed(456)
3 df = pd.DataFrame(np.random.randn(5, 4), columns=['A',
      'B', 'C', 'D'])
4 print df
```

Arithmetic operations may be made row-by-row or column-by-column, and also many database operations such as SubFrames, re-indexation, group by, join, merge, progressive and default filling.

Data from .csv I

It is possible to read the data from a .csv file (with columns: Contract_Name, Last Trade Date, Strike, Last Price, Bid, Ask, Change, % Change, Volume, Open Interest, Implied Volatility)

```
1 sp500options = pd.read_csv("sp500_OP.csv", index_col='
    Contract_Name')
2 print sp500options[['Last Price','Bid','Ask', 'Strike'
    ]].head(3)
```

Finally we show some useful examples on how to obtain some data from the `Dataframe`. Specifically, note the use of `loc, iloc, ix, at, iat`.

Data from .csv II

```
## A specific row By name
sp500options.loc['SPX180216C01840000']
#By index
sp500options.iloc[[0,2]]
sp500options.at['SPX180216C02590000','Volume']
sp500options.iat[10,2]
# They also accept boolean selection sentences.
sp500options.Volume>100
# or filtrate by the expresion
sp500options[sp500options.Volume>100]
# More complex expresions also hold
sp500options[(sp500options.Volume>100) & (sp500options.
    Ask<500)][['Volume','Last Price','Implied
    Volatility']]
```

Online data I

Among the various ways to do so, we will use a framework created for python. Specifically, we use Pandas-DataReader because of its simplicity, and because is similar to others. Our main source of data will be yahoo finance. Although there are many others, even Bloomberg and Eikon provide API's to connect with python.
In our setting the library we use to retrieve information is the pandas-Datareader. This library does not come included in `python`'s standard installation, so you may need to install it. You can install it via the command line or terminal by typing

```
>> pip install pandas-datareader
```

Or if you have anaconda installed, in the command line or terminal type:

```
>> conda install -c anaconda pandas-reader
```

Online data II

Remark. If you use codes from internet please be aware that there is a deprecated pandas-datareader, i.e. `import pandas.io.data as web`. For more information go to the url: `https://pydata.github.io/pandas-datareader/devel/remote_data.html`. So the next thing to do is import the datareader library. There is a very useful library when it comes to manipulating dates and is the `datetime`.

```
1 import pandas_datareader.data as data
2 import datetime
3 # We set an initial an terminal date for our
      information
4 start = datetime.datetime(2015, 1, 1)
5 end = datetime.datetime(2015, 12, 30)
```

So to retrieve the financial information of Microsoft between the dates `start` and `end`, we type

Online data III

```
msft = data.DataReader("MSFT", 'yahoo', start, end)
print msft[:3]
```

Similarly for Apple

```
aapl = data.DataReader("AAPL", 'yahoo', start, end)
print aapl[:3]
```

You may check the datareader documentation:

- Google finance does not work, it seems an update of the url, xml or cvs structure is needed.
- Quandl. The symbol names consist of two parts: DB name and symbol name

```
symbol = 'WIKI/AAPL' # or 'AAPL.US'
df = data.DataReader(symbol, 'quandl', "2015-01-01"
    , "2015-01-05")
print df.loc['2015-01-02']
```

Online data IV

- FRED (Federal Reserve Economic Data). We check the Gross Domestic Product

```
gdp = data.DataReader("GDP", "fred", start, end)
print gdp.ix['2013-01-01']
# Multiple series involving Consumer Price Index (All Items, Less Food and Energy):
inflation = data.DataReader(["CPIAUCSL", "CPILFESL"], "fred", start, end)
inflation.head()
```

More information and specific API's may be found in `https://pydata.github.io/pandas-datareader/devel/readers/`
There are also alternative ways to fetch data, such as:

- googlefinance.client, a python client for google finance api.
- Google Finance GET. Uses googlefinance.client.
- Quandl. Directly using quandl api (`quandl.get(...)`).
- IEX Finance.

Online data V

- Alpha vantage. Minute level data
- rsvp/fecon235. Searches in yahoo finance, and backups with google finance.

Graphical analysis I

In finance we are often interested in the prices of assets and its returns. One way to see the data is graphically. We crate a function oget that retrieves all the tickers in a list and returns a dataframe with the data indexed by (tckr,date).

```
tckrs = ['AAPL','MSFT','IBM','F','GM','TM', 'PEP', 'KO']
tck_data = oget(tckrs,'2016-1-1','2018-1-1')
```

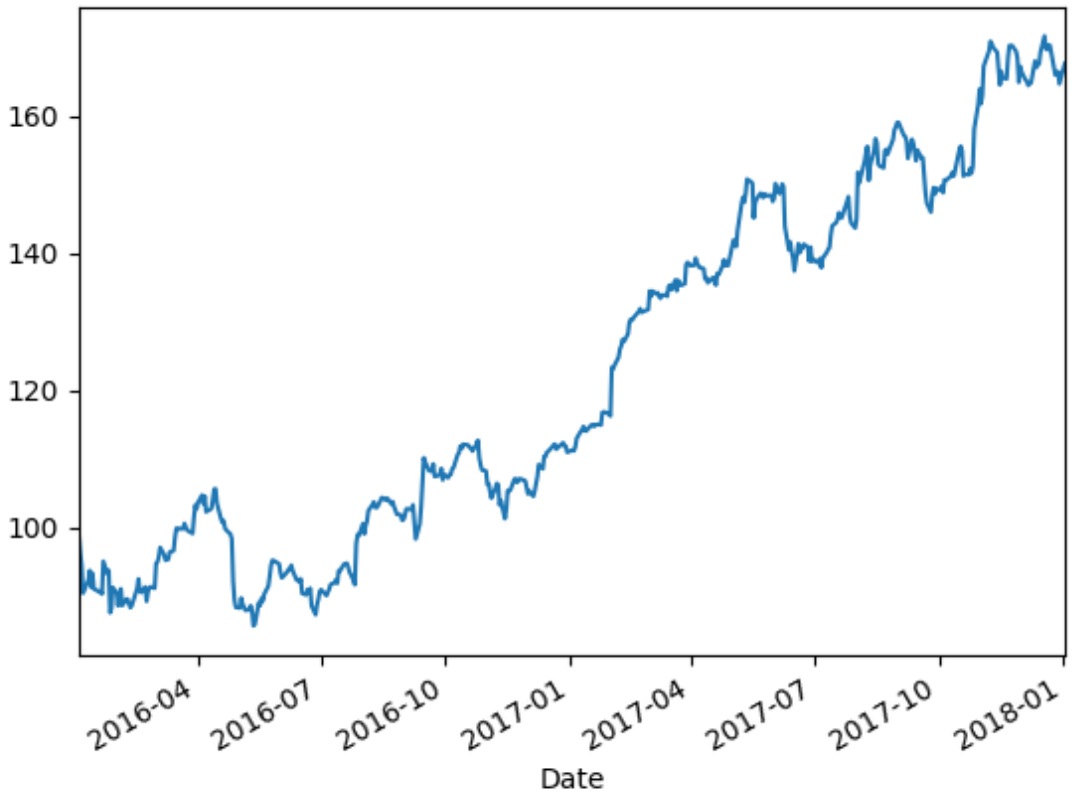
For example, to visualize the data using (`import matplotlib.pyplot as plt`):

1. Close prices.

Graphical analysis II

```
#Get the close prices
all_clspx = tck_data[['Adj Close']].reset_index()
# Organize the data
daily_px = all_clspx.pivot('Date', 'Ticker','Adj
    Close')
#plot just one ticker
daily_px['AAPL'].plot()
plt.show()
```

Graphical analysis III

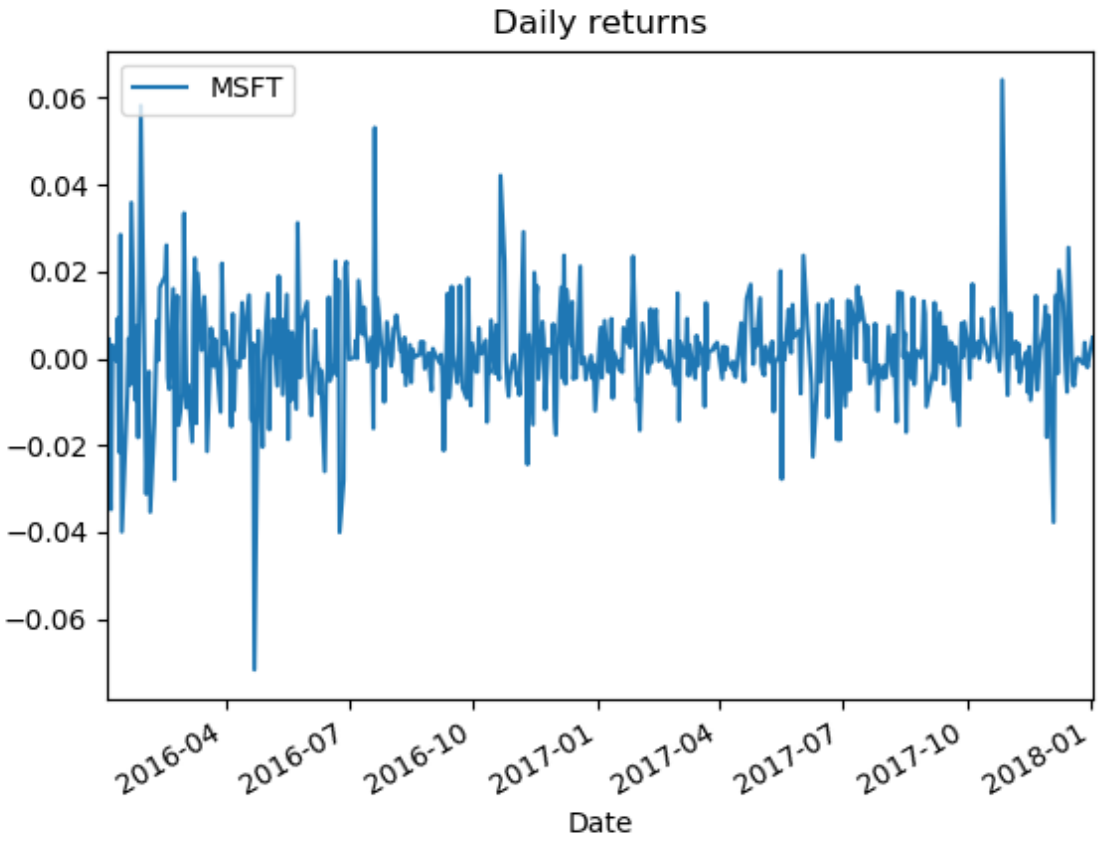


2. Returns. We calculate the return as the percentage change:

Graphical analysis IV

```
d_ret = daily_px.pct_change()
# Replace the NaN by 0
d_ret.fillna(0, inplace=True)
d_ret['MSFT'].plot()
plt.legend(loc=2)
plt.title('Daily returns')
plt.show()
```

Graphical analysis V

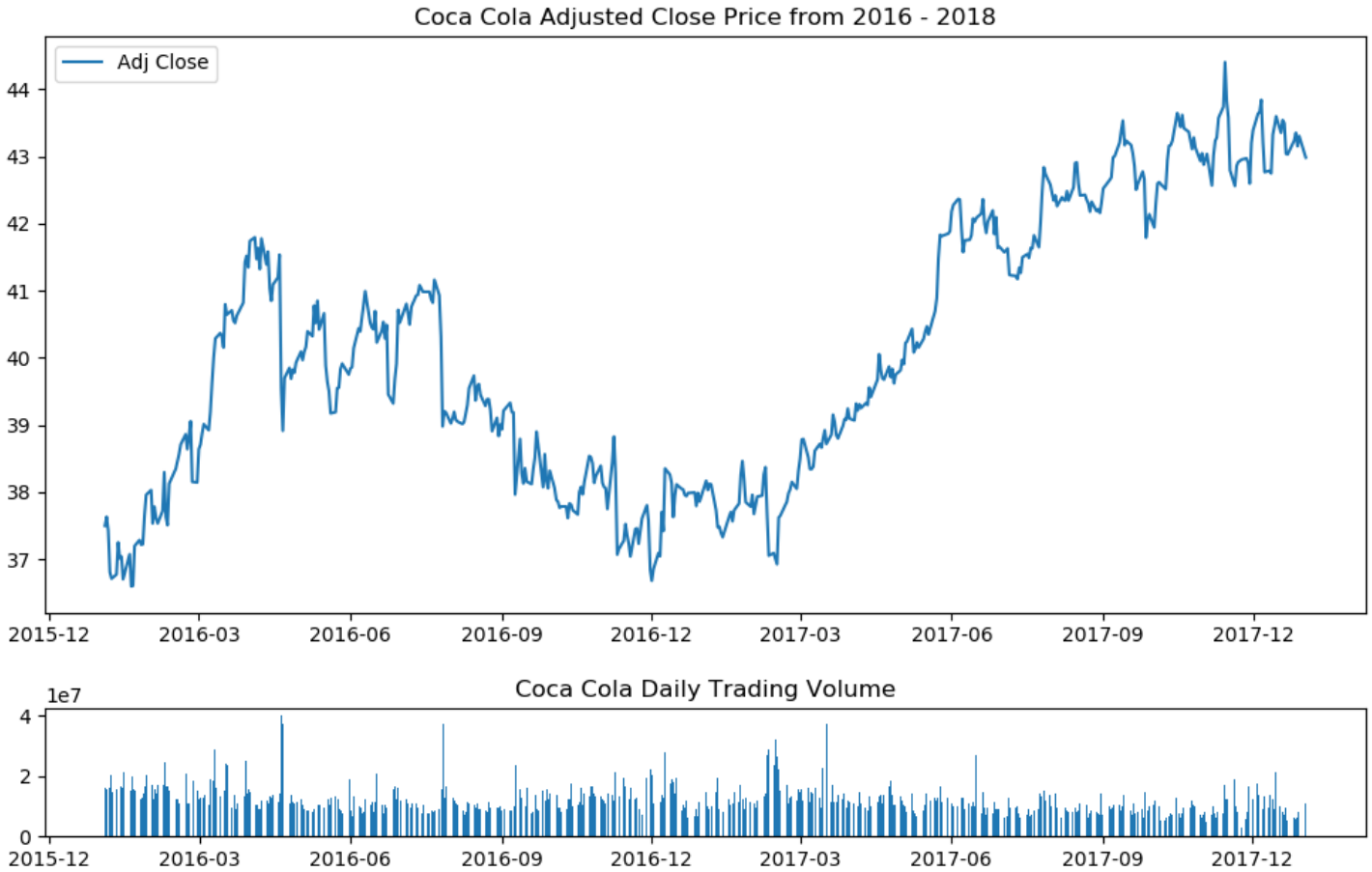


3. Volume. We plot the prices and volumes trades

Graphical analysis VI

```
top = plt.subplot2grid((4,1), (0, 0), rowspan=3,
    colspan=1)
top.plot(daily_px.index,daily_px['KO'], label='Adj
    Close')
plt.title('Coca Cola Adjusted Close Price from 2016
     - 2018')
plt.legend(loc=2)
bottom = plt.subplot2grid((4,1), (3,0), rowspan=1,
    colspan=1)
bottom.bar(KOvol.index, KOvol)
plt.title('Coca Cola Daily Trading Volume')
plt.gcf().set_size_inches(12,8)
plt.subplots_adjust(hspace=0.75)
plt.show()
```

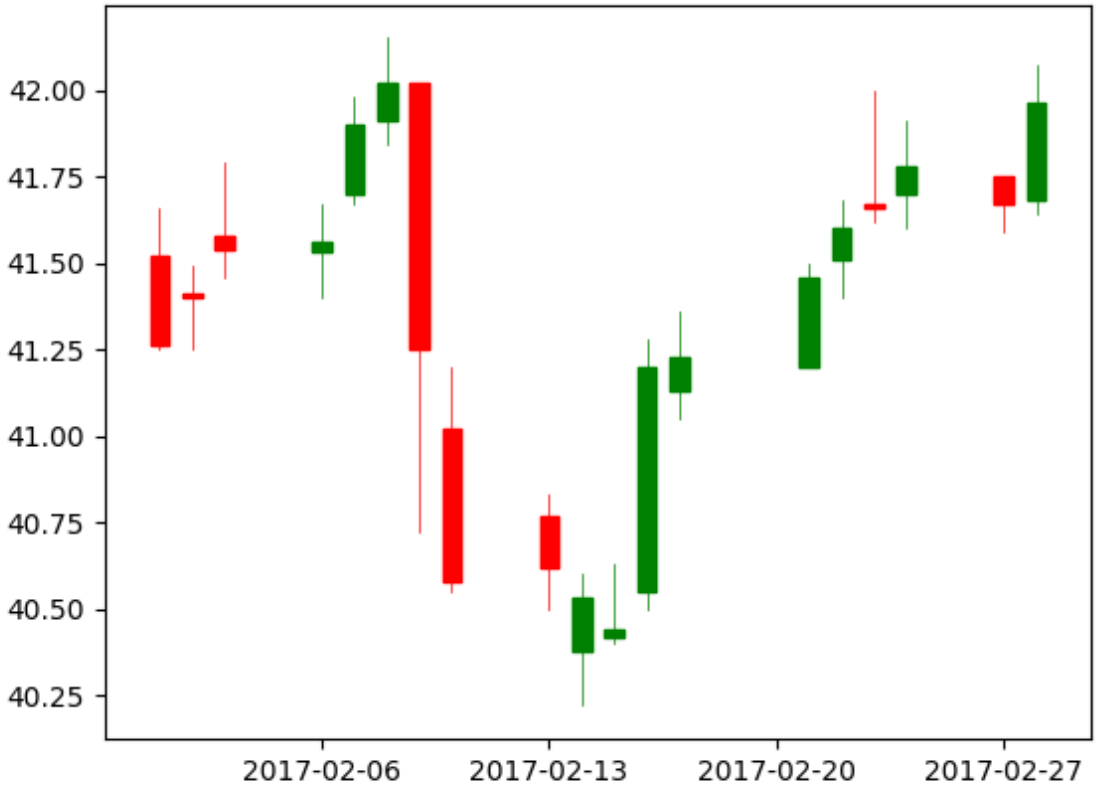
Graphical analysis VII



4. Candlesticks. The candlestick charts are useful and mostly used in finance to represent the open-high-low-close plots, and illustrates movements in the price over time.(`from matplotlib.finance import candlestick_ohlc`)

```
fig, ax = plt.subplots()
ax.xaxis.set_major_locator(mondays)
#ax.xaxis.set_minor_locator(alldays)
ax.xaxis.set_major_formatter(week_formatter)
#ax.xaxis.set_minor_formatter(dayFormatter)
candlestick_ohlc(ax,tupleKO, width=0.6, colorup='g'
    , colordown='r')
plt.show()
```

Graphical analysis IX



There are many other ways to visualise the returns, among these: **cumulative returns, histograms, Q-Q plots, Box and Whiskers and scatter plots**.

Strategies I

There are two broad categories for "predicting" movements in the market: momentum strategies and mean-reversion strategies. Here we only show an example of:

- Momentum. This type of trading focuses on stocks that are moving in a specific direction on high volume, measuring the rate of change in price changes. We continuously compute price differences at fixed time intervals. Is known to work better in Bull (or rising) markets.

These strategies utilise moving averages (rolling means) of the closing price of stocks. Notice that if moving averages (with larger windows):

- Are bellow the actual stock price, then it means a bull market and the large windows rolling average acts a a floor for the price.

- Are above the actual stock price, represents a resilience to keep the price up.

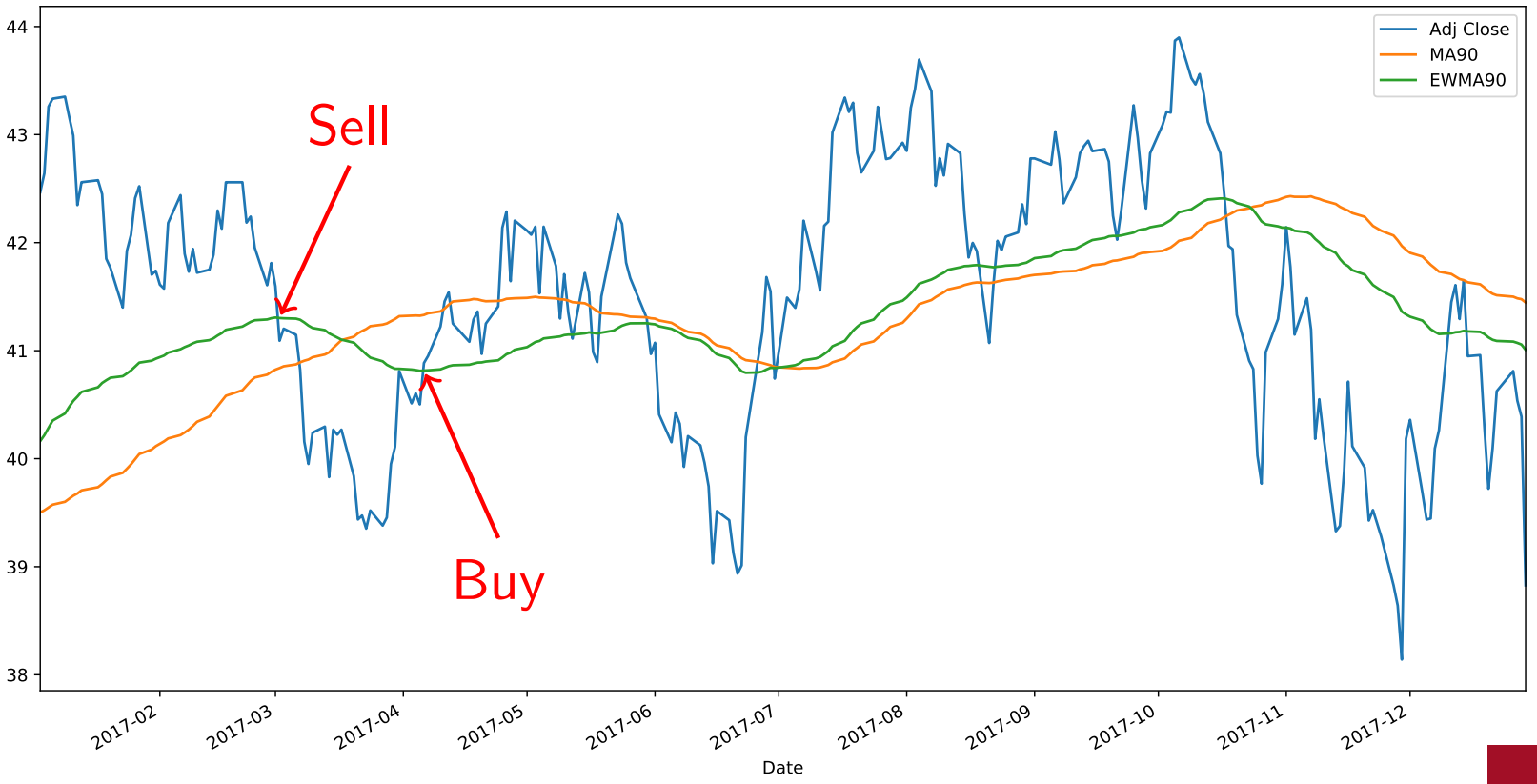
There are some negative point when using the rolling windows to have in mind:

- It only tell you things about the past (If only the past could predict the future).
- With shorter windows we add more the noise into the signal.

```
1 tck_data = odata('SEP',datetime(2016, 1, 1),datetime
    (2018, 1, 1))
2 clspx = tck_data[['Adj Close']]
3 ##### Calculate the moving averages ##################
4 clspx['MA7'] = pd.rolling_mean(clspx['Adj Close'], 7)
5 clspx['MA30'] = pd.rolling_mean(clspx['Adj Close'], 30)
6 clspx['MA90'] = pd.rolling_mean(clspx['Adj Close'], 90)
7 clspx['EWMA90'] = pd.ewma(clspx['Adj Close'], span=90)
```

Crossovers I

A crossover is a simple strategy that changes its position whenever the moving average crosses the price of the asset.
Let us explain with the following example:



Crossovers II

Let p_t be the price at t and e_t be its EWMA and notice that:

- If p_t crosses e_t from below, we close any short position and go long (buy) one unit of the asset.
- If p_t crosses e_t from above, we close any existing long position and go short (sell) one unit of the asset.

This is translated into a strategy in that:

$$w_t = \begin{cases} 1 & \text{If } p_t > e_t \equiv p_t - e_t > 0 \\ -1 & \text{If } p_t < e_t \equiv p_t - e_t < 0 \end{cases}$$

and therefore, we may assign

$$w_t = \text{sgn}(p_t - e_t)$$

That is

Crossovers III

```
close_px = clspx['Adj Close']
ewma_px = clspx['EWMA90']
signal_px = ewma_px - close_px
trade_pos = signal_px.apply(np.sign)
```

Since we calculate the signal for the same day at close (i.e. closing price) the execution of the signal is going to happen the next day and therefore we shift the data one day.

```
trade_pos = trade_pos.shift(1)
```

The next step is to calculate the daily (log-)returns and apply our trading strategy:

$$logr_t = \ln\left(\frac{p_t}{p_{t-1}}\right)$$

```
log_ret = np.log(close_px).diff()
r_s = trade_pos * log_ret
```

Crossovers IV

And calculate the cumulative returns

$$cr_{t_k} = \exp\left\{\sum_{i=1}^{k} logr_{t_i}\right\} - 1$$

```
cum_ret = r_s.cumsum()
nost_cum_ret = log_ret.cumsum()
# And relative returns
cum_rel_ret = np.exp(cum_ret) - 1
nost_cum_rel_ret = np.exp(nost_cum_ret) - 1
```

Finally we plot the returns and check

Crossovers V

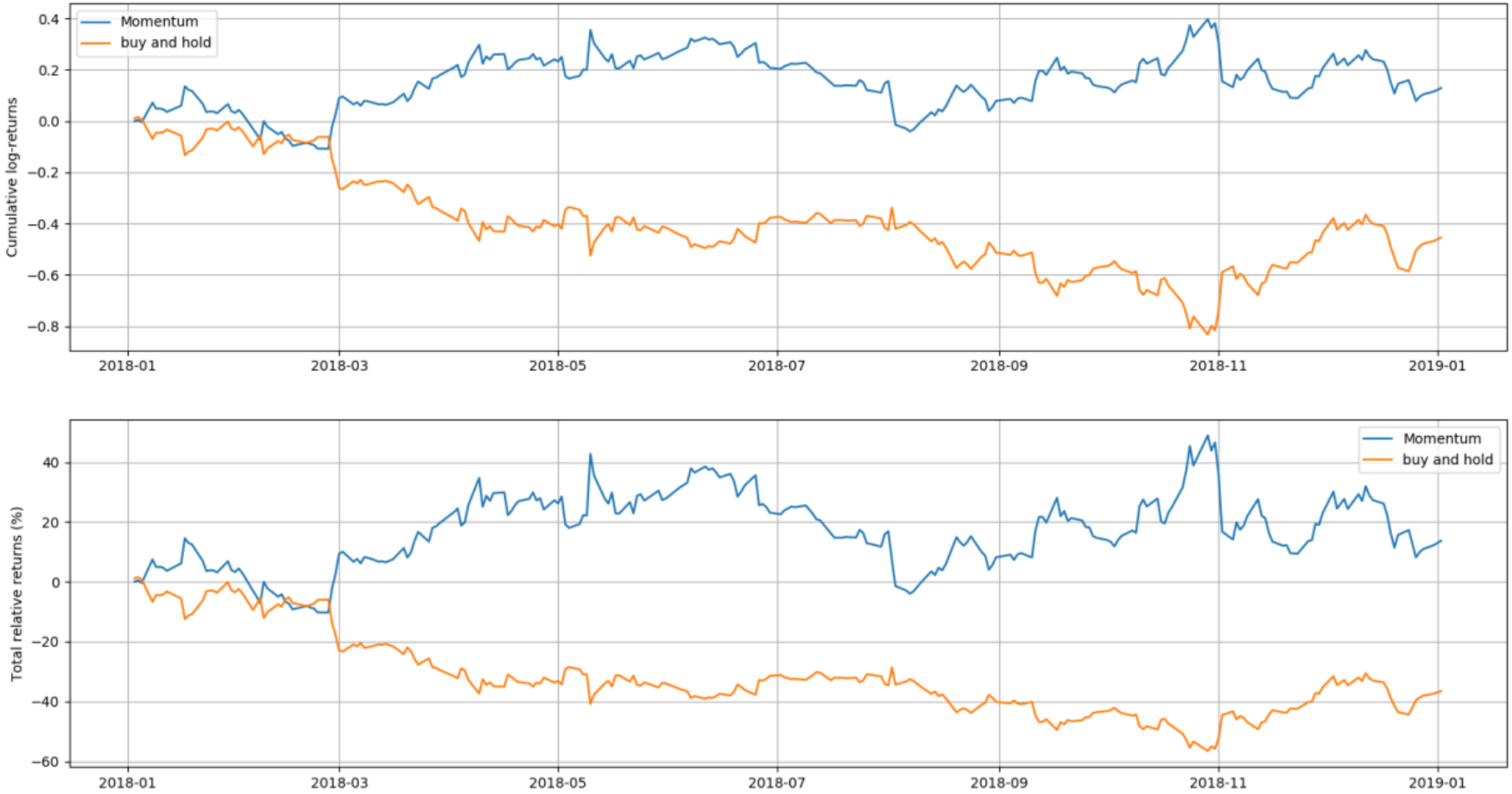
```
fig = plt.figure()

ax = fig.add_subplot(2,1,1)
ax.plot(cum_ret.index, cum_ret, label='Momentum')
ax.plot(nost_cum_ret.index, nost_cum_ret, label='buy and hold')
ax.set_ylabel('Cumulative log-returns')
ax.legend(loc='best')
ax.grid()

ax = fig.add_subplot(2,1,2)
ax.plot(cum_rel_ret.index, 100*cum_rel_ret, label='Momentum')
ax.plot(nost_cum_rel_ret.index, 100*nost_cum_rel_ret, label='buy and hold')
ax.set_ylabel('Total relative returns (%)')
ax.legend(loc='best')
ax.grid()
plt.show()
```

Crossovers VI

Crossovers VII



Zipline I

Previously we simulated trading based on a historical stock data, but these examples were naive in that they omit many features of real trading such as **transaction fees, commissions, slippage**(Different transaction price than quoted price) and many others.
There is a library called `Zipline`, which provides many of this features into our model and is used in the web based algotrading platform `quantopian`. This library provides the **backtesting** feature useful to determine the effectiveness of the strategy on the actual market data.
Here we present the basics for backtesting. Remember that classic installation of python complements is made by:

```
> pip install zipline
```

or

Zipline II

```
> conda install -c Quantopian zipline
```

More information may be found at:
`http://www.zipline.io/install.html`

To use the `zipline` library we usually create a `Class` that inherits of the `TradingAlgorithm` subclass of `zipline`. We have three ways of running programs having inherited the `TradingAlgorithm` class:

- Directly by command line. If we want to run our program from command line first we need to ingest the data to back-test our algorithm, and then run using the `zipline` command, that is

```
> zipline ingest -b quantopian-quandl
> zipline run -f my_zipline_algorithm.py ...
--start 2017-1-1 --end 2017-10-10 -o myrecords.pickle
```

This command tells `zipline` to run the file my_zipline_algorithm.py from $2017-1-1$ to $2017-10-10$ and record the output in myrecords.pickle. Off course, you will need a pickle reader to check this data.
other option to run the algorithm may be consulted by using

```
> zipline run --help
```

- In `IPython` (via `Jupyter` or else) just by adding the command

```
%%zipline --start 2017-1-1 --end 2018-1-1
```

Which tells `IPython` this is a `zipline` algorithm, and that it should run it from $2017-1-1$ to $2018-1-1$.

- With python normally, we must explicitly create an object and call the run function. Although since there have been changes in the API we must tweak somewhat the traditional ingesting of data.

TradingAlgorithm I

Every `zipline` algorithm consists of at least two functions:

1. `initialize(context)`. This method is called to start the trading algorithm
2. `handle_data(context, data)`. This method is called at each simulated trading period (i.e. day-to-day). Here we must analyse and decide how to trade.

where

- `context` is a `namespace` used to store variables needed from iteration to iteration (i.e. day-to-day trading).
- `data` contains the current trading bar with ohlc and volume in our setting.

Note. This functions are mandatory to implement since these are inherited from `TradingAlgorithm`.
The basic structure of the .py file is

TradingAlgorithm II

```
1 from zipline.api import order, record, symbol
2
3 def initialize(context):
4 #    Here you write your initialising code
5     pass
6
7 def handle_data(context, data):
8 #    Here you write your handle code
9   pass
```

There are some important functions to have in mind:

1. `order(asset, amount,...)`. Place an order of `amount` of shares of the asset `asset`. If `amount` is positive means buy and negative sell. (Similar to `order_value`)
2. `order_target(asset, target)`. Place an order to adjust a position to a target number of shares, so the final amount of shares after the order is executed is `target`. (Similar `order_target_value`)

3. `symbol(symbol_str)`. Lookup an Equity by its ticker symbol.
4. `record(self, *args, **kwargs)`. Track and record values each day.

More functions and more detailed explanation may be found in the API documentation: `http://www.zipline.io/`

Zipline simple Example I

As a simplest example we create an algorithm to buy 1 asset of Apple constantly during the period of trading.
In this example we first create our `Class` and to do so, we must first import some libraries

```
from zipline.api import order, record, symbol
from zipline import TradingAlgorithm
```

To create the `Class` we use the `class` command and call it `BuyApple`, Next we add our two functions and tell them to print the activity.

```
class BuyApple(TradingAlgorithm):

    def initialize(context):
        print("---> initialize")
        context.asset = 'AAPL'
        print(context)
        print("<--- initialize")

```

Zipline simple Example II

```
9       def handle_data(self, context):
10          print("---> handle_data")
11          print(context)
12          self.order(symbol(self.asset), 1)
13          print("<-- handle_data")
```

Next we must create the a program that ingest some data to feed the trading algorithm, and then creates an object of type `BuyApple` to run it.
The first part is to ingest the data, this was usually done with

```
1 import zipline.data.loader as zpf
2 data = zpf.load_from_yahoo(stocks=['AAPL'], indexes={},
      start=datetime(2010, 1, 1), end=datetime(2014, 1,
    1), adjusted=False)
3 clspx = data[['Close']]
4 clspx.columns = [['AAPL']]
```

This may be unstable in Windows.
Next, we instanciate an object and run the trading algorithm

Zipline simple Example III

```
BA_result = BuyApple().run(clspx['2013-01-03':'
    2013-01-15'])
```

Note that we use just a smaller period of time of that digested previously.
For this strategy you may have the `starting_cash`, `ending_cash` and `ending_value`

```
BA_result[['starting_cash', 'ending_cash', '
    ending_value']]
```

Finally we could check our results for a larger (and/or more recent) period of time, the next example runs for all 2011

```
result_for_2011 = BuyApple().run(clspx['2011'])
```

Projects

- Neural Networks to give trading signals. (`sklearn`).
- Google trends trading. (Using sentiment from google trends)
- Twiter sentiment trading. (`tweepy`).
- Crypto currencies trading. using quandl.
- Trading in Quantopian.
- Balancing an optimal portfolio.