**Python programming for**

**Finance**

**Prahlad Mehandiratta**

**ID : 1275563**

**Class ID : 19**

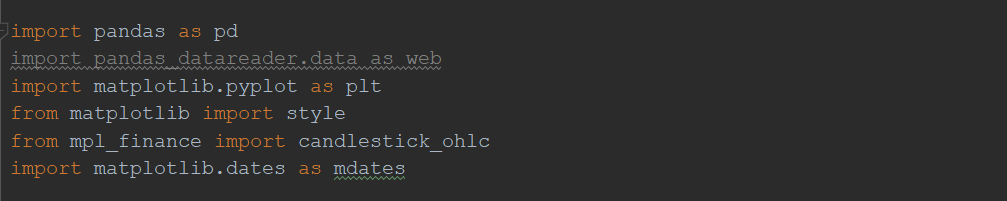
## Abstract

 We will be importing financial (stock) data into Python using the Pandas framework. And do some data manipulation and visualization of that data.

1. Packages to install

* pandas
* matplotlib
* pandas\_datareader
* beautifulsoup4
* Datetime

To begin, we're going to make the following imports:



**Datetime**will easily allow us to work with dates

**matplotlib** to graph things

**pandas** to manipulate data

**pandas\_datareader** is the pandas io library.

style.use('ggplot')

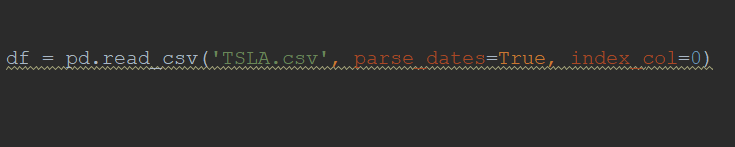
there's a pre-defined style called "ggplot", which emulates the aesthetics of [ggplot](https://ggplot2.tidyverse.org/).

start = dt.datetime(2015, 1, 1)

end = dt.datetime.now()

These are two variables which stores dates of beginning and end.

The line web.DataReader('TSLA', "yahoo", start, end) uses the pandas\_datareader package, looks for the stock ticker TSLA(Tesla), gets the information from yahoo, for the starting date of  start is and ends at the end.



This statement loads the data from the file TSLA.csv to dataframe df.

TSLA.csv is a file downloaded from Yahoo.

print(df.head())

Symbol Date Close High Low Open Volume

TSLA 2015-01-01 222.41 222.41 222.4100 222.41 0

2015-01-02 219.31 223.25 213.2600 222.63 4764443

2015-01-05 210.09 216.50 207.1626 214.50 5368477

2015-01-06 211.28 214.20 204.2100 210.06 6261936

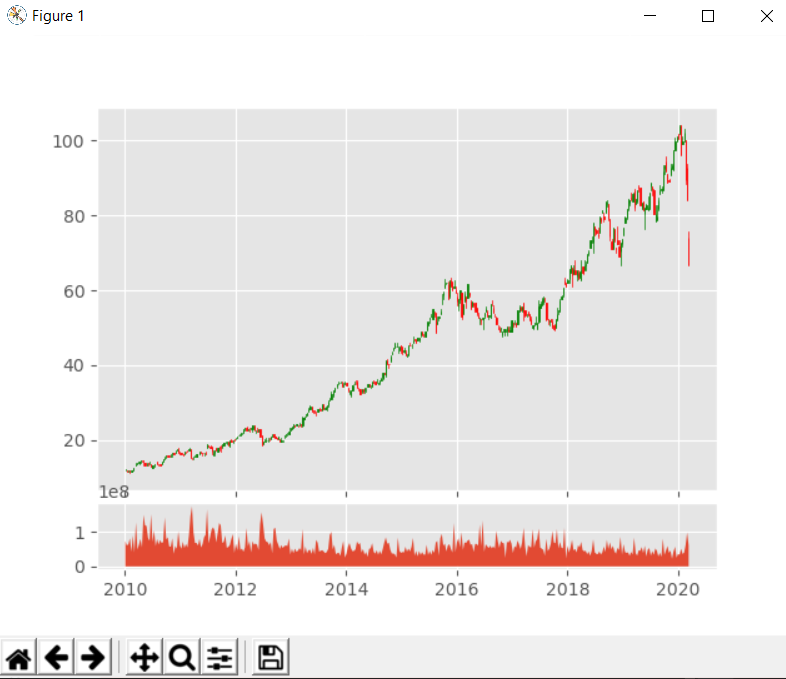
2015-01-07 210.95 214.78 209.7800 213.40 2968390

The .head() is something you can do with Pandas DataFrames, and it will output the first n rows, where n is the optional parameter you pass. If you don't pass a parameter, 5 is the default value. We mosly will use .head() to just get a quick glimpse of our data to make sure we're on the right track. Looks great to me!

* Open - The starting price of the share
* High - over the course of the trading day, what was the highest value for that day?
* Low - over the course of the trading day, what was the lowest value for that day?
* Close - When the trading day was over, what was the final price?
* Volume - For that day, how many shares were traded?
* Adj Close - This one is slightly more complicated, but, over time, companies may decide to do something called a stock split. For example, Apple did one once their stock price exceeded $1000. Since in most cases, people cannot buy fractions of shares, a stock price of $1,000 is fairly limiting to investors. Companies can do a stock split where they say every share is now 2 shares, and the price is half. Anyone who had 1 share of Apple for $1,000, after a split where Apple doubled the shares, they would have 2 shares of Apple (AAPL), each worth $500. Adj Close is helpful, since it accounts for future stock splits, and gives the relative price to splits.

df['Adj Close'].plot()

plt.show()



The [Pandas](https://pythonprogramming.net/data-analysis-python-pandas-tutorial-introduction/) module comes equipped with a bunch of built-in functionality that you can leverage, along with ways to [create custom Pandas functions](https://pythonprogramming.net/rolling-apply-mapping-functions-data-analysis-python-pandas-tutorial/). We'll cover some custom functions later, but, for now, let's do a very common operation to this data: Moving Averages.

The idea of a simple moving average is to take a window of time, and calculate the average price in that window. Then we shift that window over one period, and do it again. In our case, we'll do a 100 day rolling moving average. So this will take the current price, and the prices from the past 99 days, add them up, divide by 100, and there's your current 100-day moving average. Then we move the window over 1 day, and do the same thing again. Doing this in Pandas is as simple as:

df['100ma'] = df['Adj Close'].rolling(window=100).mean()

Doing df['100ma'] allows us to either re-define what comprises an existing column if we had one called '100ma,' or create a new one, which is what we're doing here. We're saying that the df['100ma'] column is equal to being the df['Adj Close'] column with a rolling method applied to it, with a window of 100, and this window is going to be a mean() (average) operation.

Now, we could do:

print(df.head())

Date Open High Low Close Volume \

Date

2010-06-29 2010-06-29 19.000000 25.00 17.540001 23.889999 18766300

2010-06-30 2010-06-30 25.790001 30.42 23.299999 23.830000 17187100

2010-07-01 2010-07-01 25.000000 25.92 20.270000 21.959999 8218800

2010-07-02 2010-07-02 23.000000 23.10 18.709999 19.200001 5139800

2010-07-06 2010-07-06 20.000000 20.00 15.830000 16.110001 6866900

Adj Close 100ma

Date

2010-06-29 23.889999 NaN

2010-06-30 23.830000 NaN

2010-07-01 21.959999 NaN

2010-07-02 19.200001 NaN

2010-07-06 16.110001 NaN

Under the 100ma column we just see NaN. We chose a 100 moving average, which theoretically requires 100 prior datapoints to compute, so we wont have any data here for the first 100 rows. NaN means "Not a Number." With Pandas, you can decide to do lots of things with missing data, but, for now, let's actually just change the minimum periods parameter:

ax1 = plt.subplot2grid((6,1), (0,0), rowspan=5, colspan=1)

ax2 = plt.subplot2grid((6,1), (5,0),rowspan=1,colspan=1,sharex=ax1

Basically, we're saying we want to create two subplots, and both subplots are going to act like they're on a 6x1 grid, where we have 6 rows and 1 column. The first subplot starts at (0,0) on that grid, spans 5 rows, and spans 1 column. The next axis is also on a 6x1 grid, but it starts at (5,0), spans 1 row, and 1 column. The 2nd axis also has the sharex=ax1, which means that ax2 will always align its x axis with whatever ax1's is, and visa-versa. Now we just make our plots:

ax1.plot(df.index, df['Adj Close'])

ax1.plot(df.index, df['100ma'])

ax2.bar(df.index, df['Volume'])

plt.show()

Above, we've graphed the close and the 100ma on the first axis, and the volume on the 2nd axis. Our result:

High Low Open Close Volume Adj Close \

Date

2010-06-29 25.00 17.540001 19.000000 23.889999 18766300 23.889999

2010-06-30 30.42 23.299999 25.790001 23.830000 17187100 23.830000

2010-07-01 25.92 20.270000 25.000000 21.959999 8218800 21.959999

2010-07-02 23.10 18.709999 23.000000 19.200001 5139800 19.200001

2010-07-06 20.00 15.830000 20.000000 16.110001 6866900 16.110001

100ma

Date

2010-06-29 23.889999

2010-06-30 23.860000

2010-07-01 23.226666

2010-07-02 22.220000

2010-07-06 20.998000

Unfortunately, making candlestick graphs right from Pandas isn't built in, even though creating OHLC data is.The alternative to that is.

from mpl\_finance import candlestick\_ohlc

import matplotlib.dates as mdates

First, we need proper OHLC data. Our current data does have OHLC values, and, unless we am mistaken, Tesla has never had a split, but you wont always be this lucky. Thus, we're going to create our own OHLC data, which will also allow us to show another data transformation that comes from Pandas:

df\_ohlc = df['Adj Close'].resample('10D').ohlc()

What we've done here is created a new dataframe, based on the df['Adj Close']column, resamped with a 10 day window, and the resampling is an ohlc (open high low close). We could also do things like .mean() or .sum() for 10 day averages, or 10 day sums. Keep in mind, this 10 day average would be a 10 day average, not a rolling average. Since our data is daily data, resampling it to 10day data effectively shrinks the size of our data significantly. This is how you can normalize multiple datasets. Sometimes, you might have data that tracks once a month on the 1st of the month, other data that logs at the end of each month, and finally some data that logs weekly. You can resample this dataframe to the end of the month, every month, and effectively normalize it all! That's a more advanced Pandas feature that you can learn more about from the [Pandas](https://pythonprogramming.net/data-analysis-python-pandas-tutorial-introduction/) series if you like.

We'd like to graph both the candlestick data, as well as the volume data. We don't HAVE to resample the volume data, but we should, since it would be too granular compared to our 10D pricing data.

df\_volume = df['Volume'].resample('10D').sum()

We're using sum here, since we really want to know the total volume traded over those 10 days, but you could also use mean instead. Now if we do:

print(df\_ohlc.head())

We get:

open high low close

Date

2010-06-29 23.889999 23.889999 15.800000 17.459999

2010-07-09 17.400000 20.639999 17.049999 20.639999

2010-07-19 21.910000 21.910000 20.219999 20.719999

2010-07-29 20.350000 21.950001 19.590000 19.590000

2010-08-08 19.600000 19.600000 17.600000 19.150000

That's expected, but, we want to now move this information to matplotlib, as well as convert the dates to the mdates version. Since we're just going to graph the columns in Matplotlib, we actually don't want the date to be an index anymore, so we can do:

df\_ohlc = df\_ohlc.reset\_index()

Now dates is just a regular column. Next, we want to convert it:

df\_ohlc['Date'] = df\_ohlc['Date'].map(mdates.date2num)

Now we're going to setup the figure:

fig = plt.figure()

ax1 = plt.subplot2grid((6,1), (0,0), rowspan=5, colspan=1)

ax2 = plt.subplot2grid((6,1), (5,0), rowspan=1, colspan=1,sharex=ax1)

ax1.xaxis\_date()

Everything here you've already seen, except ax1.xaxis\_date(). What this does for us is converts the axis from the raw mdate numbers to dates.

Now we can graph the candlestick graph:

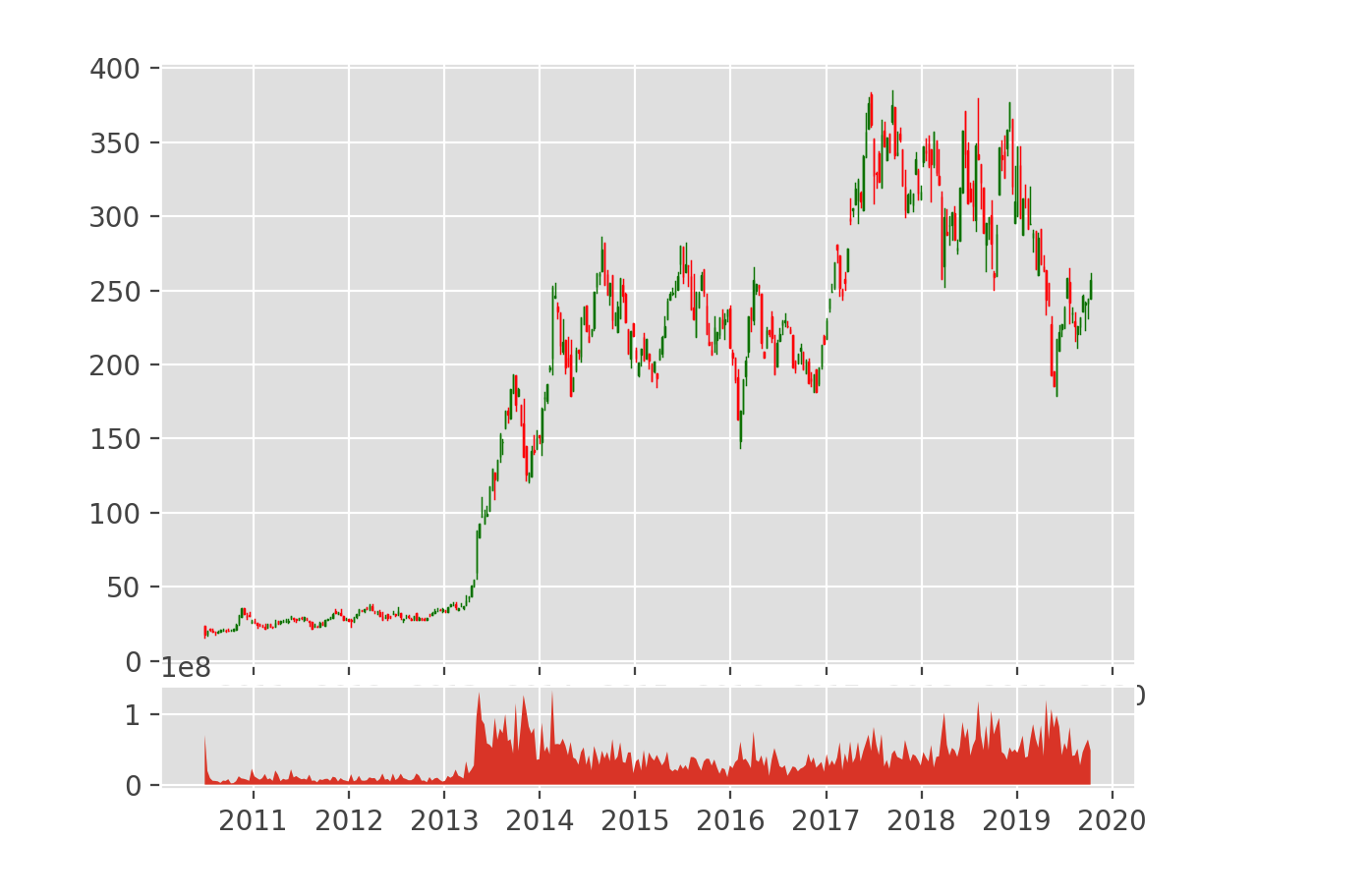
candlestick\_ohlc(ax1, df\_ohlc.values, width=2, colorup='g')

Then do volume:

ax2.fill\_between(df\_volume.index.map(mdates.date2num),df\_volume.values,0)

The fill\_between function will graph x, y, then what to fill to/between. In our case, we're choosing 0.

plt.show()



We want a Python list of the S&P 500 companies. we're going to grab the list from

Wikipedia:http://en.wikipedia.org/wiki/List\_of\_S%26P\_500\_companies. The tickers/symbols

in Wikipedia are organized on a table. To handle for this, we used the HTML parsing

library, Beautiful Soup. bs4 is for Beautiful Soup, pickle is so we can easily just save this list

of companies, rather than hitting Wikipedia every time we run (though remember, in time, you will want to update this list!), and we'll be using requests to grab the source code from

Wikipedia's page. First, we visit the Wikipedia page, and are given the response, which

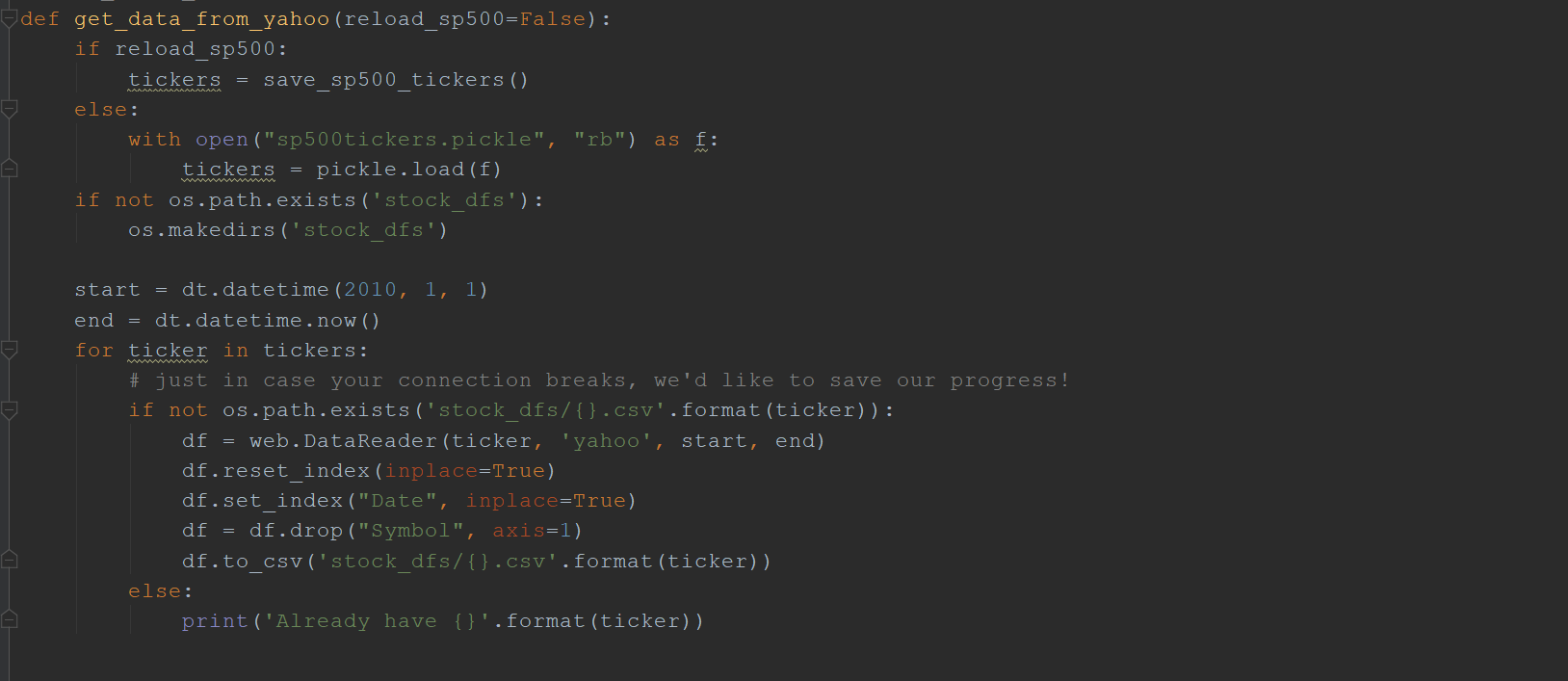
contains our source code. To treat the source code how we want, we want to access

the .text attribute, which we turn to soup using BeautifulSoup. For each row, after the

header row (this is why we're going through with [1:]), we're saying the ticker is the "table

data" (td), we grab the .text of it, and we append this ticker to our list. Now, it'd be nice if

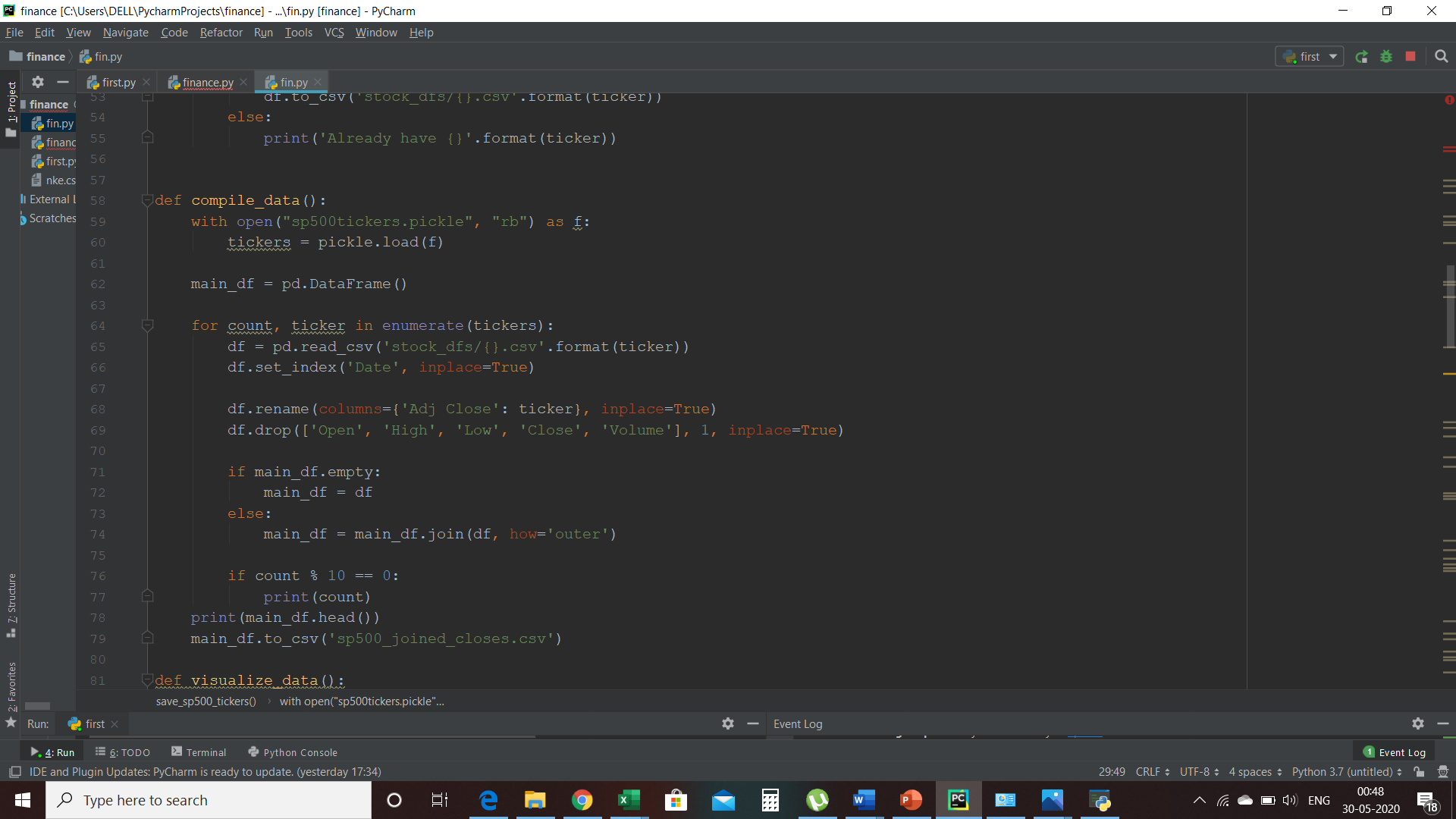
we could just save this list. We'll use the pickle module for this, which serializes Python objects for us.



This is the code to pull stock pricing data on all s&p 500 companies. use datetime to specify

dates for the Pandas datareader, os is to check for, and create, directories. The goal is to

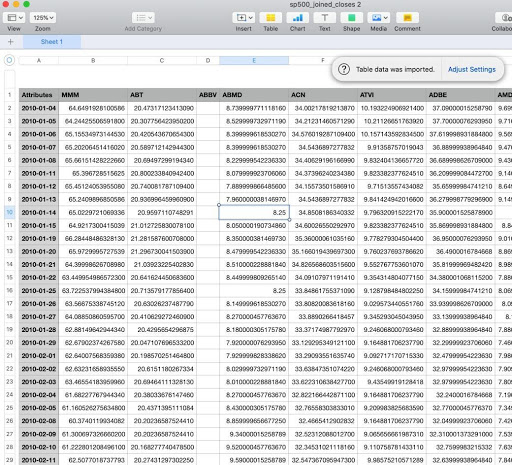
pull everything we can from Yahoo, returns to us for every stock and just save it.



In this step we brought this companies names and their prices together,we just joined them

together. Each of the stock files at the moment come with: Open, High, Low, Close, Volume, and Adj Close. we pulled them previously-made list of tickers, and begin with an empty data

frame called main\_df. we just iterated the tickers list so they can join.



pd  
  
style.use('ggplot')  
  
  
def visualize\_data():  
 df = pd.read\_csv("sp500\_joined\_closes-2.csv", parse\_dates=True, index\_col=0)  
  
 print(df.head())  
 df\_corr = df.corr(method='pearson')  
 print(df\_corr.head())  
 df\_corr.to\_csv('sp500corr.csv')  
 df\_corr = pd.read\_csv("sp500corr.csv", parse\_dates=True, index\_col=0)  
 data1 = df\_corr.values  
 fig1 = plt.figure()  
 ax1 = fig1.add\_subplot(111)  
 heatmap1 = ax1.pcolor(data1, cmap=plt.cm.RdYlGn)  
 fig1.colorbar(heatmap1)  
  
 ax1.set\_xticks(np.arange(data1.shape[1]) + 0.5, minor=False)  
 ax1.set\_yticks(np.arange(data1.shape[0]) + 0.5, minor=False)  
 ax1.invert\_yaxis()  
 ax1.xaxis.tick\_top()  
 column\_labels = df\_corr.columns  
 row\_labels = df\_corr.index  
 ax1.set\_xticklabels(column\_labels)  
 ax1.set\_yticklabels(row\_labels)  
 plt.xticks(rotation=90)  
 heatmap1.set\_clim(-1, 1)  
 plt.tight\_layout()  
 plt.show();  
  
  
visualize\_data()

In this we have correlate the data and visualize it. The .corr() automatically will look at

the entire DataFrame, and determine the correlation of every column to every column. This

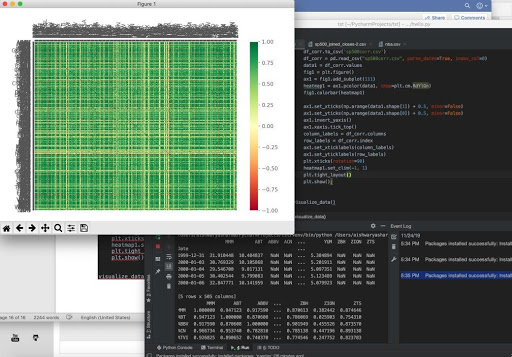
heatmap is made using a range of colors, which can be a range of anything to anything, and

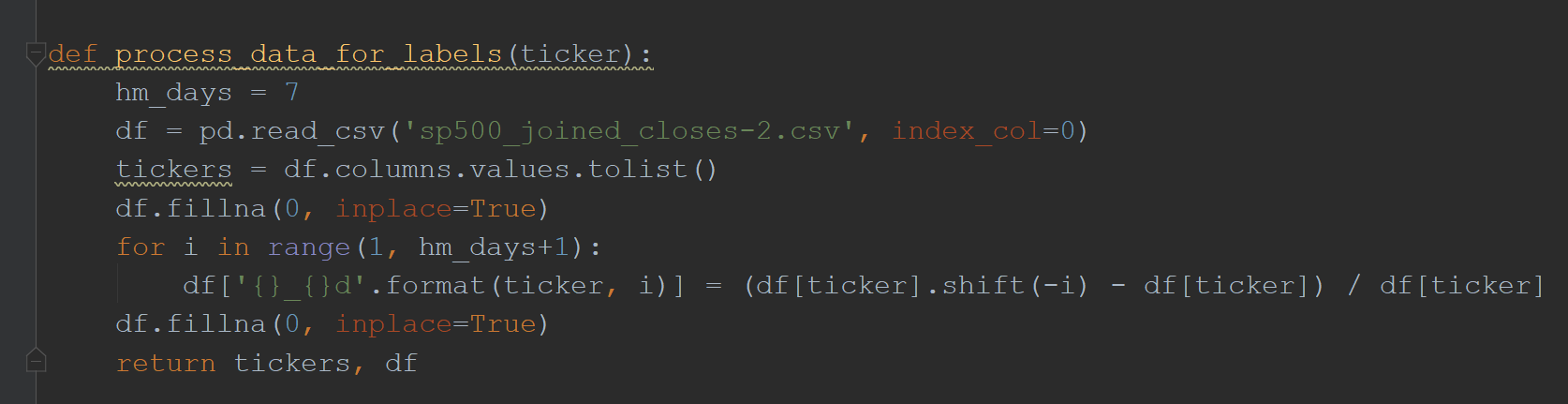
the color scale is generated from the cmap that we use. You can find all of the options

for color maps here. I’m going to use RdYlGn, which is a colormap that goes from red on the

low side, yellow for the middle, and green for the higher part of the scale, which will give us

red for negative correlations, green for positive correlations, and yellow for no-correlations.





This function will take one parameter: the ticker in question. Each model will be

trained on a single company. Next, we want to know how many days into the future we

need prices for. We're choosing 7 here. Now, we'll read in the data for the close prices for

all companies that we've saved in the past, grab a list of the existing tickers, and we'll fill any

missing values with 0. Now, we want to grab the % change values for the next 7 days, creates

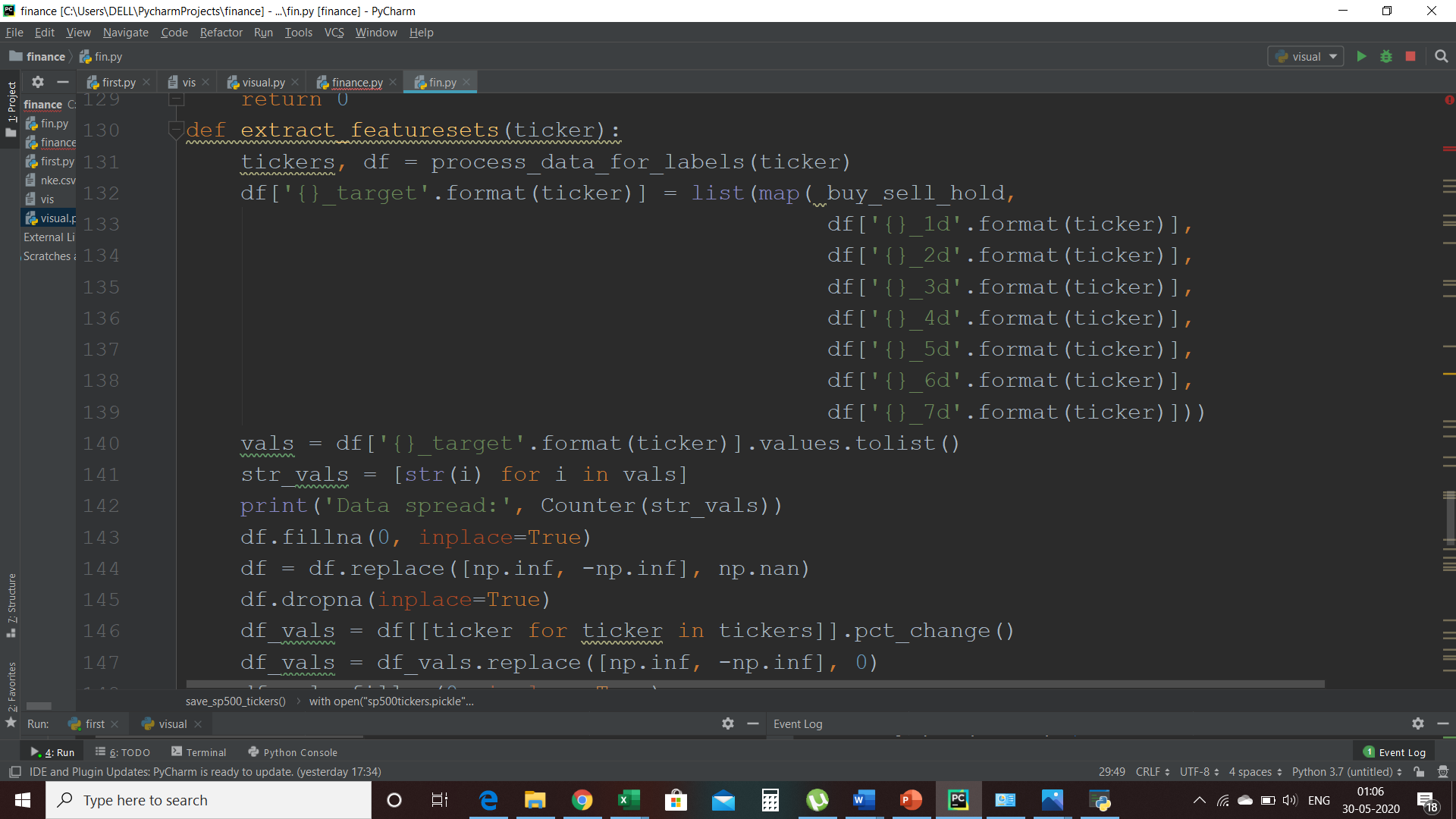
new dataframe columns for our specific ticker in question, using string formatting to create

the custom names. The way we're getting future values is with .shift, which basically will

shift a column up or down. In this case, we shift a negative amount, which will take that

column and, if you could see it visually, it would shift that column UP by we rows. This gives us

the future values 1days in advanced, which we can calculate percent change against.



In this we have created the function that creates our label that dictates buy, sell, or hold.if

the price rises more than 2% in the next 7 days, we're going to say that's a buy. If it drops

more than 2% in the next 7 days, that's a sell. If it doesn't do either of those, then it's not

moving enough, and we have to just hold whatever our position is. If we have shares in that

company, we do nothing, we keep our position. If we don't have shares in that company, it

does nothing. And this “extract\_featuresets(ticker):” This function will take any ticker, create the needed

dataset, and create our "target" column, which is our label. The target column will have

either a -1, 0, or 1 for each row, based on our function and the columns and then if we have

any empty spaces or NAN then we normalize them to zero by using dropna() function. The

idea here being that some companies will change in price before others, and we can profit

maybe on the laggards. We'll convert the stock prices to % changes. The capital X contains

our feature sets (daily % changes for every company in the S&P 500). The lowercase y is our “target” or our “label”. Basically, we are trying to map our feature sets.