RAPSTAR

"I've been making like two thousand a minute…"
-*Rapstar* by Polo G

Zion Choi

I. *Purpose*.
Though the application is far from complete, there are some useful functionalities that I believe can help both inexperienced and experienced individuals make better informed decisions when trading stocks. The purpose of this project was to create a comprehensible, all-in-one stop for stock data and information.

As a relatively new individual tackling the stock market, it was important to me to be informed and up-to-date on potential stock opportunities. A large part of this process was examining stock charts (usually candlestick charts) and utilizing online screeners (such as Finviz); the project helps replicate this process by providing users with charts or stocks and a functional screener service. Hopefully, users new to the stock market, such as myself, can find utility and convenience in this program to research new stocks.

II. *Research*.
Most of my research was oriented towards learning more about the stock market. A book that helped guide the direction of my program was *A Beginner's Guide to the Stock Market* by Matthew R. Kratter. Kratter's book goes through the basics of stock chart analysis and stock screeners. When developing the program, I planned to use a greater amount of stock indicators (i.e P/E or P/B), but I ended up focusing solely on price-related data.

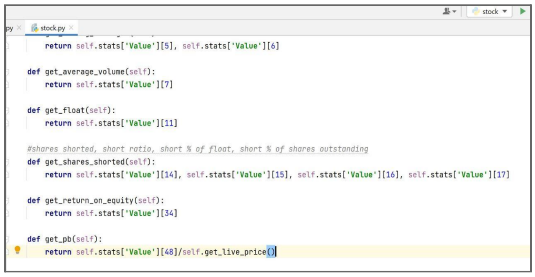
Another pair of books that were a huge help in the development process were *How to Swing Trade* and *How to Day Trade for a Living* by Brian Pezim and Andrew Azziz respectively. In these books, there were more nuanced techniques for trading stocks. The one technique I chose to focus on utilized both price-related data and screeners: reversal indicators (Fig. 1). This was a technique I relied heavily on when starting to trade stocks to good effect. I think starting small with relatively simple techniques can help newer users, as well as provide a strong foundation for further features.

Figure 1: an example of a graph generated by a screener, specifically Finviz

In terms of coding practices, I have tackled similar projects in the past with limited success. However, it was useful to reference previous code, especially regarding the complexity and nuance of advanced data structures such as pandas dataframes or series. Though I didn't reference similar code or projects, I did rely on online resources for graphing and data analysis help.

III. *Development*.
Frankly, development was slow in the beginning. It was difficult to get a footing on the necessary data I would need to execute the project, as well as creating functional, yet runtime efficient, classes. The first aspect of the project I developed was the stock class. Though it was easy to develop, it was apparent that the runtime was just not efficient enough. Creating a new stock class, or even retrieving a single data point, would

take a few seconds. Though this was not a major issue on its own, when working on the screener class — where hundreds of stock objects need to be created and compared — runtime became a huge issue (Fig. 2).



```
        return self.stats['Value'][5], self.stats['Value'][6]

    def get_average_volume(self):
        return self.stats['Value'][7]

    def get_float(self):
        return self.stats['Value'][11]

    #shares shorted, short ratio, short % of float, short % of shares outstanding
    def get_shares_shorted(self):
        return self.stats['Value'][14], self.stats['Value'][15], self.stats['Value'][16], self.stats['Value'][17]

    def get_return_on_equity(self):
        return self.stats['Value'][34]

    def get_pb(self):
        return self.stats['Value'][48]/self.get_live_price()
```

Figure 2: Initial version of the stock class, which utilized many more indicators. An important note, regarding runtime, was the variable self.data. By storing these indicators in one variable, and importing data every time an object was initialized, the program was largely inefficient.

In order to fix this issue, I relied on importing and exporting large amounts of historical data into csv files (Fig. 3). Whereas the data was previously stored in data frame variables, importing/exporting to external files meant initialization of a stock object would not affect runtime. In order to implement this change, I needed to create more functionality for the stock class then previously expected. I needed to know when to build a new data file, update the file, or simply check if the file is up to date. I relied on the datetime module to make these changes and see if a certain stock was up to date on a local level.

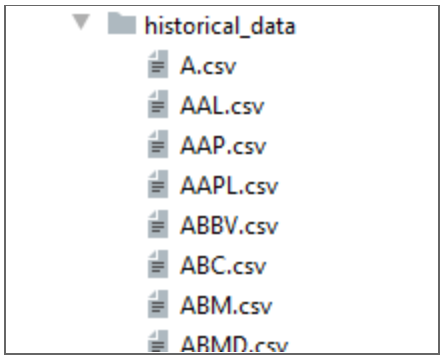


Figure 3: An example of the new storage system — utilizes a new folder called historical_data where YTD data (open, low, high, close, adj. close) is stored per ticker

This change to the stock class necessitated change in the screener class as well. Before, I decided to go simpler and store settings in a dictionary. However, drawing from the stock class changes, I imported and exported new settings to a local folder. This was more efficient in terms of runtime, however, it was much more difficult to import and export data. I decided to store the settings for a specified screener within a pandas series. Exporting the data to a csv file was easy enough, however I ran into trouble when importing the data. When opening a csv file with pandas, it reads the information as a data frame rather than a series. It took some experimentation with importing, and eventually it worked (Figs. 4 & 5).

```
#imports screener from screeener_name.csv, reads in as pd series
def import_screener(self, screener_name):
    self.settings = pd.read_csv(screener_name, header=None,squeeze=True)
```

Figure 4. Import screener method. Reads in from a csv file, where squeeze=True and header=None transforms data from Dataframe to Series

```
def save_screener(self, screener_name):
    self.settings.to_csv(screener_name+'.csv', header=None, index=False)
```

Figure 5. Save Screener method. Sends settings to csv files.

The next big step was creating a user interface for the program. Though I hadn't

planned to do this initially, it seemed reasonable enough to execute and would add a lot to the project in terms of visualization — visualizing data is the entire purpose of the project (Fig. 6)! I used Python's built-in tkinter module as well as matplotlib to create the user interface. Though this was mostly new to me, it went pretty seamlessly and I didn't run into any major issues during this process.

Figure 6. The user interface when loaded up.

One of the unusual difficulties of the project was Github. When working with a virtual environment (Fig. 7), I didn't know to not import the venv folder. As a result, setup was unusually difficult when testing the program across multiple computers. However, by using a .gitignore file and adding a requirements.txt document (Fig. 8), the problem was resolved.

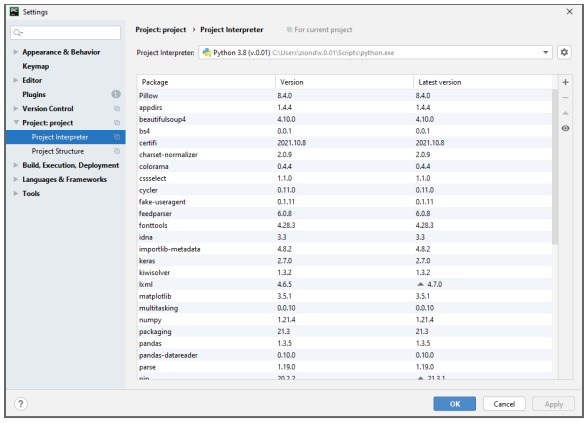


Figure 7: The virtual environment in PyCharm.

```
This file is a part of Rapstar, a Zion Choi prc
See README.md for full details.

This project, as licensed under Zion Choi, requ

The required dependencies for the virtual envir

Package                 Version

pandas                  1.3.5
matplotlib              3.5.1
pandas-datareader       0.10.0
yahoo-fin               0.8.9.1
yfinance                0.1.67
```

Figure 8: The new requirements.txt document to help resolve the venv folder problem. Also helps users with easier setup when running the program.

The current program, after resolving these issues, has three main functionalities. The first is the charting feature. A user is able to input a ticker, and a chart will be outputted in the user interface (Fig. 9).

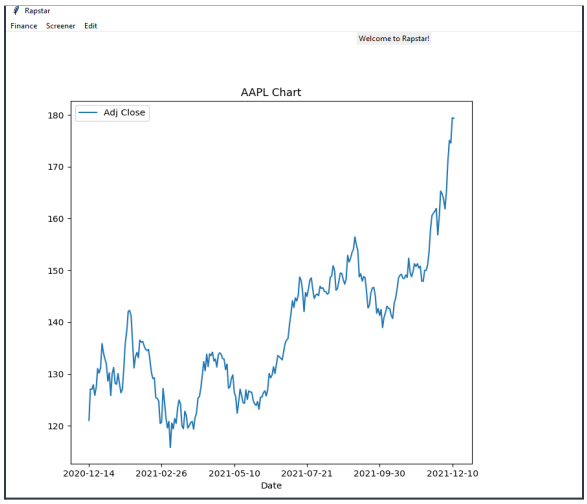


Figure 9: An example of the user interface, with the stock AAPL.

The second main feature is the screener functionality. Users can set their own settings, save their screeners, and import them (Fig. 10).

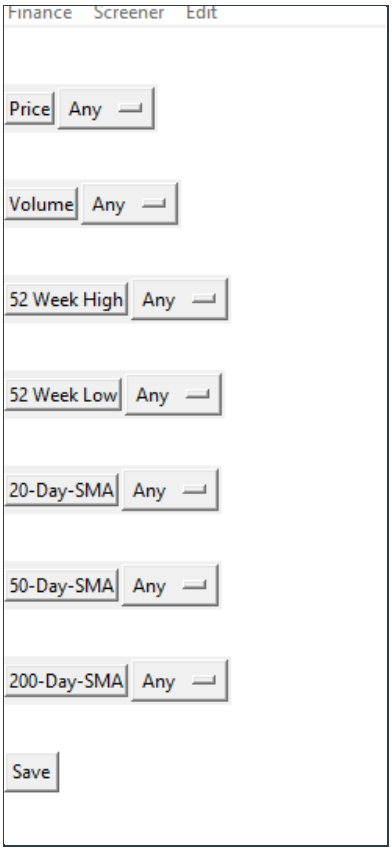


Figure 10: An example of the user interface's screener settings. They are able to use dropdown menus to set their custom screener.

The third main feature is an advanced stock screener. Users can now add SMAs and other price indicators (specifically, opens, highs, lows, closes, and adj. closes). A graph is produced by matplotlib (Fig. 11).

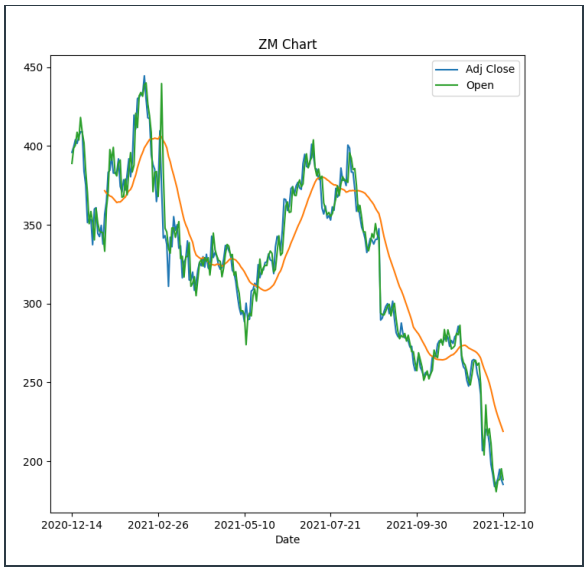


Figure 11: An example of the advanced charting tool. The above is an example of ZM.

There are no current bugs I am aware of. There are some features, however, that may be complicated to users (basically, not very user friendly). Regarding the advanced charting tool, how to use it may be unclear.

IV. *Further Development.*

Regarding further development, I would like to add more features to the program — specifically the indicators previously mentioned and left out. Additionally, I would like to add candlestick graphs to the finance tab. There is of course much more to implement.

Some different features that should be considered are a neural network for price prediction and one for twitter analysis. Using keras and theanos, it shouldn't be too difficult to train a model, especially given the framework in the stock class. Additionally, StockTwits has been a useful tool in my own trading experience. To create an all-in-one stop for trading, it would be nice to implement this feature.

For future collaborators, it's important to note the project structure. Further notes are in the README.md file.