The present challenge at hand is to generate a desired dataframe from a JSON output generated by the get\_options\_chain method using the tda-api. The ideal dataframe that we would like as a result would look like this:

Columns = “expiration date”, “strike price”, “ask price”, … (potential additional columns: vol, iv, Greeks.

The JSON output consists of dozens of columns and a single row. In these columns is the data we want, but it is nested along on columns called “ExpDateMap.xxxxxxx” for the call chain we want. Inside these nested columns are the keys “strike price”, “ask price” and the Greeks. That means that for each of these row, column pairs we want specific key : values to be copied into a dataframe. For example, for key1, value 2, we would like each of them to be added to row n, column n. Key1 would be added to the Key1 column, and Value2 would be added to the Value2 column. Programmatically we need to instruct our program using list comprehension to take each selected key within the “ExpDateMap.xxxxxx” group of columns and appending both the key and value pair to a new existing database.

For k, v in old\_df[“ExpDateMap.\*”]

new\_list = []

new\_df = pd.Dataframe(new\_list)

new\_df.append(k,v)

So far, I have managed to get all the received data into a dataframe with the following format:

“symbol” “callExpDateMap”

F { index [ key: value]}

At first glance the keys and values we want are in rows of nested dictionaries containing sets of tuples. The zip function could be used to put together two lists into a dictionary, but what about creating a new filtered dictionary from an existing dictionary? The challenge persists. Extracting nested key values from each of these dictionaries suggests using iteration, as well as dictionary and list comprehension.

*Update 6.30.21.* I only have to columns left to parse and they are proving to be the hardest to access. I have a dataframe with a symbol column, an expiration date column, and even a volatility column. The last two columns however are the strike price and ask columns, which consist of key value pair nested inside multiple indexed dictionary rows inside a single column.

Upon further examination of the data, it is able to be manipulated such that we have tuples (expiration date, strike price) and a dictionary { with dictionary items for each row in the dictionary list. The key values now being reindexed can hopefully be accessed.

*Update 7.4.21.* Using dir, type, and iter, we now know that each key is the expiration date in our nested dictionary and the values are another nested dictionary with the strike price as the key index. Below is a visual representation of the structure of the nested dictionary we wish to parse.

key-> {'key': [{'key': 'value', 'key': 'value', 'key': 'value', 'key': 'value', 'key': 'value', 'key': 'value', 'key': 'value'}]}

Update: using i and j as key-values for the dictionary it is possible to iterate through it and get the values that we want. Ultimately *j* is the dictionary that we want to unpack further.

*Update 7.5.21* I successfully got the first value from the nested dictionary, the strike price in the requested option chain. I only hope that it is ordered because it does not appear so. I managed to reconstruct the nested dictionary to one with a tuple index, the second element of these being made up of the first key, exposing the values to be a single dictionary. My only challenge left here now is to iterate through the rows in this dataframe and retrieve the value in the dictionary of the values for the tuple dictionary.

*Update 7.6.21* I was now able to get the second value in the nested dictionary, and with the help of list creation and for loops I was able to add these values to my dataframe. From the look of it I now have all the values that I desired in my ideal dataframe for getting options. I ran the code successfully and not I would like to format columns like the expiration date column to display more visually friendly values. In addition, the strike price values appear to be out of order in relation to their corresponding expiration date. There are now smaller bugs left to work out now that I spent almost 3 weeks learning and getting this initial dataframe construction hurdle to work. Some small bugs:

-Connecting the options dataframe retrieval code and the stocks list to iterate through each of the stocks in a defined list.

-Learning the tradebuilder methods that come built in with the broker API that I am using. The placing of buy and sell orders given each final dataframe for both stocks and options while a small bug it is still the largest chunk visibly left to craft.

-Fine tuning the get movers function to only add selected stocks to a list instead of adding all.

From the three bugs above there are a few interesting comparisons to make. All essentially deal with taking a dataframe and then running it through a search function much like how our stocks are searched through by taking in a csv file. Ultimately it appears that if we export our option data frames, trade execution function, and even our get movers code to a csv file then a function to execute should be depending on filtered data frames that yield a target output.

-In the case of option dataframe exporting and iterating through the results, the search function is invoked on a csv file. It could very well be invoked on a created list.

-Placing trades also hinges upon reading values from lists or files, and then executing them if there are values present in them.

-the get movers function has already been improved to filter out by parameters as of this writing. The present challenge is to run it for every item on our stock list/file.

The last two bugs above seem to be the most vital, as the first one seems to already have answered itself. Placing trades and getting the movers might very well be different beasts, one is simply invoked by set variables and the other- the movers function, is a complex iteration of a series of steps for dictionary to dataframe creation. This complexity is only relative to the initial start of the movers code, which simply passes values to a file or list. As such:

Movers, to csv

Stocks, to csv

Options, to csv

Execution, to csv

This iteration can be instantiated by a function, meaning our initial code will have to be formatted within a function that can call it.

*Dictionary Creation Using Iteration*

In order to apply our dictionary creation steps for each stock in the list, we need to understand a few key concepts first. Among this is the bridge between generating option dataframes for our given stock lists and executing both sell and buy orders for the target dataframe. For some reason upon completion of the code the author considered searching through the targeted selling variables and searching for their option chains, which is an extension of the initial intention of just selling based on their stock technical. But this additional step of obtaining option dataframes for variables to sell may be crucial in selling options, so it will be added. Perhaps one of the easiest approaches to put this whole program together is to focus on the selling function, which as we have explained contain a step-by-step process to get option dataframes as well as stock information for stocks in the current portfolio to weigh against their selling. One this part of the program is completed the second part which will be the buying function should be quite easier to finish up.

*Option Iteration*

When collecting stock prices yahoo finance has API friendly methods utilize. Using for loops it is possible to iterate through a stock list and searching each one, appending the results to a new list. The same principle should hold for option search iteration, only instead of a for loop just calling the stock search function it would call the option search function.

*Update 7.6.21* After having broken my code I was able to rebuild it from scratch with a better understanding of each of its components. The option retrieval portion is about 40 lines of code. With the get movers, get stocks, and trade execution portions these should ideally total no more than 200 lines of code. Now I just need to get organized and get the seemingly daunting task of cleaning up everything and putting together a clean build. I have a few templates to practice on and once I do this I should be able to export my project to somewhere more portable and where it can be easily implemented.

As the code is put together, I am realizing the importance of articulating properly the relationship between the option dataframe retrieval process and the stock dataframe retrieval process. Both end up with results as dataframes, but each begins differently. The stock search process is invoked from the get movers function, while the option search process is more flexible in how it is invoked. It can be made up of a list of stocks, but this list can come from the get movers list or other lists and files. Then again so can the stock search function, which really places both the stock search and option search process horizontally side by side and with not one being dependent on the other to run.

The challenge still appears to be option iteration now that the dataframes have been successfully created with minimal malfunction. That is, for a list of stocks can we design a function flow that allows out option dataframe creator function to iterate through each? The structure our program is such that iteration occurs in the middle once a list is pulled. Some issues run into include labeling errors and key errors so careful attention to detail Is required.

*Consolidation*

An option search function, a stock search function, and functions for getting lists to these are some of the important parts of this code. If we can wrap out heads around these, the structure of our program can perhaps become clearer. However, upon further consideration the option search function might need to be tabled for development to complete the program in time for deployment. Additionally, the Google Cloud environment is having some trouble recognizing critical modules.

For now, focusing on a patchwork workaround is the main goal. For example, our selling function pulls the stocks in our portfolio and creates a file with a list that can be accessed by a search function, the result being a dataframe that sets a variable should candidates for selling arise. From this an order is placed. In this sense we can sell options and theoretically even buy them based on this stock search function’s result, rather than the iterating get options method which we tabled. The reason for this is because rather than duplicate our work now by creating an options iterating function, we can trade options based on the stock search function’s results.

After migrating this project to a different IDE, it is now possible to use it in Google Cloud, Visual Studio, Spyder, and a Jupyter Notebook. This is promising but the main hurdles of option iteration and crafting a selling function persist now.

This document has discussed option dataframe creation and walked through the process of successfully creating a dataframe, but it now concludes, and further discussions will be documented in other documents that are more relevant, like program architecture, program notes, etc.

-