APIs Used:

Alpaca API:

OpenAI API:

Finnhub API:

Data Models:

RiskItem:

- *symbol:* A string that represents the ticker of a stock
- *risk:* Literal representation of risk associated with the analyzed stock based on ChatGPT's analysis
- *trade_block*: Boolean value that determines if the stock trade should be allowed to happen, if true, the trade is blocked. Boolean value is determined by ChatGPT's analysis.
- *rationale:* A short text explanation returned by ChatGPT that summarizes why it gave the stock that specific risk assessment.

RiskReport:

- *asof:* Timestamp of report
- *items:* List of *RiskItem*, one risk item per stock.

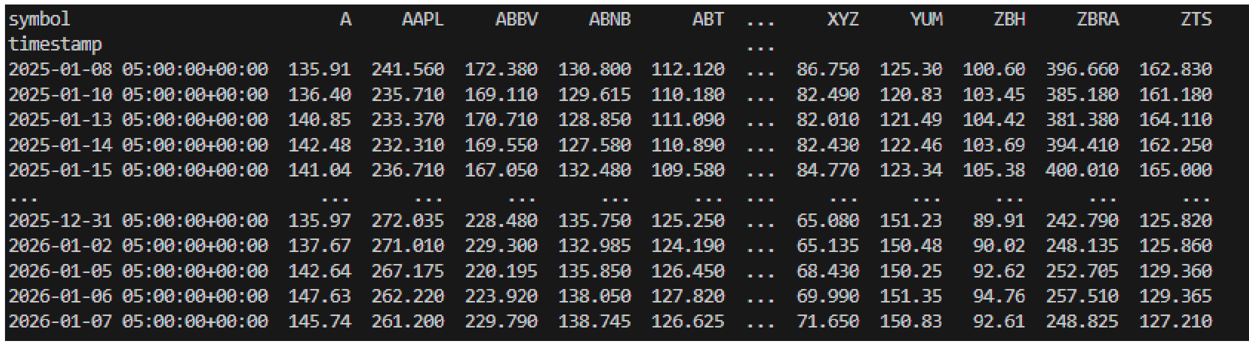
Functions:

get_SP500_companies:

- *Input(s): None*
- *Output(s): pd.DateFrame*
- *Operation:* Using the given Wikipedia page that lists, in the first table, all current companies in the S&P 500, treating the bot request as a user request, the function gets all the information from that webpage. It then parses all HTML tables from the webpage into a list of DataFrames. Since the first table is all we need, we copy, make a copy of the first table and edit its columns for our ease of access and then return that newly edited S&P 500 table.
- *Notes:*
    - "request.get(..., headers=headers, timeout=30) tells the Wikipedia server, "Hey, I'm a normal web browser, not some python bot requesting info." The reason I do

this is because initially the servers wouldn't give me the info, so I believed it blocked bot activity, so treating my GET request as a non-bot action wouldn't block the request.
-

fetch_close_matrix:

- *Input(s):* 'data_client' (*StockHistoricalDataClient)*, 'symbols' *(List[str])*
- *Output(s): pd.DateFrame*
- *Operation:* Given a list of stock symbols, this function fetches about one year of daily price bars from Alpaca (using the IEX feed and split/dividend-adjusted prices), extracts the closing prices, and reshapes the data into a matrix where rows are timestamps, columns are symbols, and values are closing prices. It then sorts the rows chronologically (earliest at the top, latest at the bottom) and returns the result as a pandas DataFrame.
- *Notes:*
    - The reason why the stock prices used in the program are adjusted for stock splits and dividends is because unadjusted prices can make a stock like it suddenly become volatile because they show the raw trading price without accounting for events like stock splits and dividends. For example, a 2-for-1 split will make the price appear to drop about 50% overnight even though nothing bad happened and your investment value didn't actually halve itself. Adjusting for these splits rescale the earlier history so the chart of the stock price stays smooth. Dividends can also cause a visible drop on the ex-dividend date in unadjusted data, but that drop is offset by the cash dividend you receive. In short, while the stock prices may dip, you cover that loss by the dividend payout.
    - 

```
symbol                          A     AAPL     ABBV     ABNB      ABT  ...     XYZ     YUM     ZBH     ZBRA      ZTS
timestamp                                                              ...
2025-01-08 05:00:00+00:00  135.91  241.560  172.380  130.800  112.120  ...  86.750  125.30  100.60  396.660  162.830
2025-01-10 05:00:00+00:00  136.40  235.710  169.110  129.615  110.180  ...  82.490  120.83  103.45  385.180  161.180
2025-01-13 05:00:00+00:00  140.85  233.370  170.710  128.850  111.090  ...  82.010  121.49  104.42  381.380  164.110
2025-01-14 05:00:00+00:00  142.48  232.310  169.550  127.580  110.890  ...  82.430  122.46  103.69  394.410  162.250
2025-01-15 05:00:00+00:00  141.04  236.710  167.050  132.480  109.580  ...  84.770  123.34  105.38  400.010  165.000
...                           ...      ...      ...      ...      ...  ...     ...     ...     ...      ...      ...
2025-12-31 05:00:00+00:00  135.97  272.035  228.480  135.750  125.250  ...  65.080  151.23   89.91  242.790  125.820
2026-01-02 05:00:00+00:00  137.67  271.010  229.300  132.985  124.190  ...  65.135  150.48   90.02  248.135  125.860
2026-01-05 05:00:00+00:00  142.64  267.175  220.195  135.850  126.450  ...  68.430  150.25   92.62  252.705  129.360
2026-01-06 05:00:00+00:00  147.63  262.220  223.920  138.050  127.820  ...  69.990  151.35   94.76  257.510  129.365
2026-01-07 05:00:00+00:00  145.74  261.200  229.790  138.745  126.625  ...  71.650  150.83   92.61  248.825  127.210
```

compute_momentum _scores:

- *Input(s):* 'close' *(pd.DataFrame)*
- *Output(s): pd.Series*
- *Operation:* This function takes a close-price matrix (from the DataFrame produced by *fetch_close_matrix*) and computes each stock's percent return over 126 (*LOOKBACK_TRADING_DAYS*) trading days, then lags the signal by 5 (*SKIP_TRADING_DAYS*) days to ignore the most recent days and reduce short-term reversal noise. It extracts the most recent row of momentum values, removes invalid

entries (NaN and +/- inf), and returns a pandas Series of the latest momentum scores sorted from highest to lowest.

pick_top_with_sector_limits:

- *Input(s):* 'scores' *(pd.Series)*, 'sector_map' *(Dict[str, str])*
- *Output(s): List[str]*
- *Operation:* Given a momentum-ranked *scores* Series and a *sector_map* of ["symbol" : "sector"], this function iterates through the symbols in score order and builds a list of picks until it reaches a list length of 20 (*TARGET_HOLDING*). It enforces diversification by skipping any symbol whose sector already has 5 (*MAX_PER_SECTOR*) picks in the list. It returns the selected symbols as a list.

fetches_headlines:

- *Input(s):* 'news_client' , 'symbols' *(List[str])*, 'days_back' *(int = 7)*
- *Output(s): Dict[str, List[str]]*
- *Operation:* This function pulls recent news and returns a dictionary mapping each stock ticker to up to three headline strings.
- *Notes:*
    - Originally used Alpaca's new API, but the lack of publisher variety and difficulties getting the headline caused me to pivot to Finnhub's API which has a much better system for getting news on a specific company

openai_risk_filter:

- *Input(s):* 'openai_risk_filter' *(OpenAI)*, 'candidates' *(List[str])*, 'sector_map' *(Dict[str, str])*, 'headlines' *(Dict[str, List[str]])*
- *Output(s): Dict[str, RiskItem]*
- *Operation:* Given the top 20 momentum list, this functions grabs the headlines obtained from *fetch_headlines* for each of the stocks, and based on predefined guidelines and prompts, queries ChatGPT (GPT 4.1-mini) to rank the risk of these stocks based on the recent headlines, block the trade based off of the information, and the answer why it chose the answer it did. Returns analysis of each stock as a Dictionary of ticker symbols associated with their risk assessment, *RiskItem*.

rebalance_equal_weight:

- *Input(s):* 'trading_client' *(TradingClient)*, 'data_client' (StockHistoricalDataClient), 'target_symbols' *(List[str])*
- *Output(s): None*
- *Operation:* This function first sells any positions you currently hold that are not in *target_symbols*, then it buys the symbols in *target_symbols* using available buying power. It starts by fetching all open positions, then builds a dictionary that maps each ticker symbol to its Position object so membership checks against *target_symbols* are easy. It loops through that dictionary and, for each symbol not in *target_symbols*, it

reads the share quantity (pos.qty), skips invalid quantities, fetches the latest trade price to estimate the position value, submits a market sell for the full quantity, and tracks that it sold something so it can wait briefly before starting the buy step. If anything was sold, the function only uses the money from said sold assets (asset_sold) to buy other stocks. If nothing was sold, then it uses the cash (account.cash) from the user's account. The function that calculates the length of *target_symbols*, and uses said length to evenly split the buy power capital amongst the stocks in *target_symbols*, via the notional (per_stock_notional). It also multiplies the notional for each stock by 0.99 (buffer) to account for price change between the time assets_sold was calculated and the actual confirmation that said assets were sold. During this time, small price changes are still possible. To avoid spending more money than one has, we use less than our total price. From there the function invests the calculated notional into each stock on *target_symbols*, displaying the total amount of cash used in this buy round, and returning nothing.
- *Notes:*
    - For tallying up the "total sell price" (assets_sold), the reason I don't use the already present 'pos.avg_entry_price' is because it is an average entry price. I need close to exact current pricing on a stock as I use that "total sell price" to dictate my total buying power, thus, I use 'StockLatestTradeRequest'.
    - The way I programmed this function unintentionally makes it so it self-weights my portfolio. If a stock continuously tops the charts in 6-month momentum, it will continuously be on *target_symbols*, meaning every week I will continuously invest into that same company. Each week it will take more and more of my portfolio percentage.

run_once:

- *Input(s): None*
- *Output(s): None*
- *Operation:* This function coordinates the full weekly trading run. It initializes the Alpaca, Finnhub and OpenAI clients, pulls the S&P 500 universe and sector map, fetches ~1 year of daily price bars and builds a close-price matrix, verifies there is enough data to compute momentum, compute 6-month momentum and selects top momentum candidates while enforcing per-sector limits. It then fetches news headlines for the candidates, runs an OpenAI-based risk filter to block unsafe tickers, and aborts if fewer than 5 symbols remain approved. Finally, it only proceeds on Mondays, waits for the market to open, and rebalances the portfolio into the approved tickers using equal weights.

main_loop:

- *Input(s): None*
- *Output(s): None*
- *Operation:* This function runs an infinite loop and checks every 10 seconds whether the current EST is within a 1-minute catch window around the scheduled run time. If it's in

that window and the bot hasn’t already run today (based on *last_daily_run* in the saved JSON state), it executes *run_once()* inside a try/except block so errors are caught and printed instead of crashing the bot. On success it records today’s date as *last_daily_run*, and if today is Monday it also updates *last_rebalance_week*. It then saves the updated state back to the JSON file.