Modeling Your Stock Portfolio Performance with Python

Evaluate Your Trade Performance in 200 lines of Code

Matt Grierson Apr 12, 2020 · 15 min read ★

Photo by Jamie Street on Unsplash

When I first began learning Python, one of my goals was to understand how to better evaluate the financial performance of my stock portfolio. I came across a great article by Kevin Boller where he used python to compare a portfolio against the S&P 500. I highly recommend spending a few minutes reading through it to better understand the underlying financial concepts since I won't go as in-depth here.

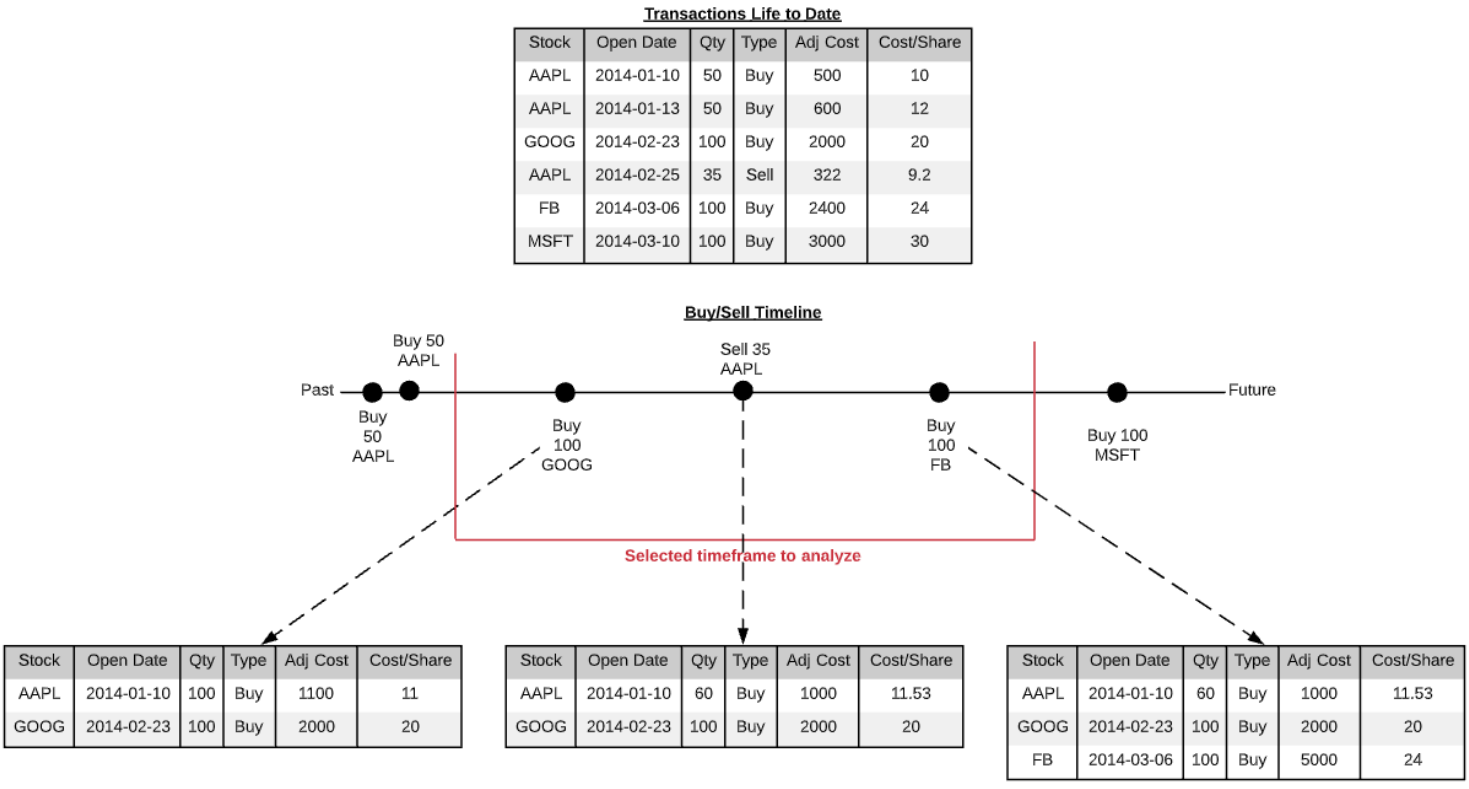
However, one of the things I wanted was a time series display of the data, whereas this was more a summary of a point in time. The original code also assumed stock sales would never happen, and I wanted it to reflect the reality of buying/selling positions over a dynamic time frame.

The rest of this article will explore the process of how I structured the code, challenges to accurately calculating performance on a time-weighted basis, and how to display the end result. The files and code explained below can be found and forked here.

calculate the rate of return relative to an index across any specified timeframe. “That doesn’t sound terrible!” — said the arrogant voice in my head prior to starting coding. Turns out that voice is a hopeless optimist and there are actually a lot of hurdles, including:

- For a relative start and end date, how do you calculate what’s “current” holdings at the start?
- How do you factor in multiple buys and sells in the same position over a given timeframe?
- How do you accurately portray cost basis when multiple purchases/sales of the same stock are made at different cost bases?

To better illustrate these challenges, here’s a drawing that attempts to visualize a scenario:



Stock	Open Date	Qty	Type	Adj Cost	Cost/Share
AAPL	2014-01-10	50	Buy	500	10
AAPL	2014-01-13	50	Buy	600	12
GOOG	2014-02-23	100	Buy	2000	20
AAPL	2014-02-25	35	Sell	322	9.2
FB	2014-03-06	100	Buy	2400	24
MSFT	2014-03-10	100	Buy	3000	30

Stock	Open Date	Qty	Type	Adj Cost	Cost/Share
AAPL	2014-01-10	100	Buy	1100	11
GOOG	2014-02-23	100	Buy	2000	20

Stock	Open Date	Qty	Type	Adj Cost	Cost/Share
AAPL	2014-01-10	60	Buy	1000	11.53
GOOG	2014-02-23	100	Buy	2000	20

Stock	Open Date	Qty	Type	Adj Cost	Cost/Share
AAPL	2014-01-10	60	Buy	1000	11.53
GOOG	2014-02-23	100	Buy	2000	20
FB	2014-03-06	100	Buy	5000	24

An example showing correctly calculated cost/share after buys and sales

various points across the selected timeframe. As you can see, trades **prior** to the start date become extremely important in calculating the active balance at the start of the timeframe. Just because I bought 100 shares of Apple before my start date does not mean it should be excluded.

You will also notice on the bottom-middle table cost per share becomes $11.53 when selling AAPL, why is that? Because the AAPL shares were bought two days apart at different prices per share, we need to aggregate and average the cost together. This means the order of what you sell becomes important! For this article, we'll assume everything adheres to FIFO standards, which means the oldest shares bought are the first shares sold. Finally, we see that the buy for MSFT stock will be excluded since it exists outside of the timeframe, so we'll need to account for that as well in our program.

Solution Proposal

After considering the challenges mentioned above, I decided creating a fresh 'daily' calculation of holdings and stock prices would be necessary to generate an accurate analysis. As an added benefit, everything mentioned in Kevin's article could also be feasible since most all calculations were based on taking a snapshot and comparing it to a later snapshot. Keeping this all in mind, we're going to take the following approach to build this:

- Read the portfolio file with dates of all buy/sell transactions, then pull by-day data for all tickers before the specified end date (remember we need to worry about buys/sells before the start date)
- Calculate the 'Active' Portfolio holdings and adjusted per holding costs based on the start date provided
- Create a daily list of 'active holdings' in my portfolio, along with daily close prices of the stocks
- Create calculations based on each day's merged data

Now that we have a loose structure of what we want to accomplish, let's get to typing!

Doggo doing a snake boi program

Step 1 — Grabbing the Data

The first thing we need to do is grab daily data for both the stocks we have in our portfolio, as well as the benchmark we're evaluating our portfolio against. Stock data is generally not available in an open source or free form, and although there are plenty of awesome services like Quandl or IEXFinance for hardcore analysis, they're a Ferrari to my Prius needs. Luckily, there's a brilliant library called yfinance that scrapes Yahoo Finance stock data and returns it in a structured form. So let's start by importing the libraries we'll need and the first few functions for getting the data:

```
import pandas as pd
import numpy as np
import datetime
import plotly.express as px
import yfinance as yf
import pandas_market_calendars as mcal
from plotly.offline import init_notebook_mode, plot
init_notebook_mode(connected=True)


def create_market_cal(start, end):
```

```
    market_cal = market_cal.tz_localize(None)
    market_cal = [i.replace(hour=0) for i in market_cal]
    return market_cal


def get_data(stocks, start, end):
    def data(ticker):
        df = yf.download(ticker, start=start, end=(end + datetime.timedelta(days=1)))
        df['symbol'] = ticker
        df.index = pd.to_datetime(df.index)
        return df
    datas = map(data, stocks)
    return(pd.concat(datas, keys=stocks, names=['Ticker', 'Date'], sort=True))


def get_benchmark(benchmark, start, end):
    benchmark = get_data(benchmark, start, end)
    benchmark = benchmark.drop(['symbol'], axis=1)
    benchmark.reset_index(inplace=True)
    return benchmark
```

stocks_get_data.py hosted with ♡ by GitHub view raw

The first function we are writing is called `create_market_cal` and uses the pandas_market_calendars library to find all relevant trading days within a specified timeframe. This library automatically filters out non-trading days based on the market, so I don't need to worry about trying to join data to invalid dates by using something like pandas.date_range. Since my stocks are all US-based, I'll select `NYSE` as my calendar, and then standardize the timestamps to make them easy to join on later.

The `get_data` function takes an array of stock tickers along with a start and end date, and then grabs the data using the yfinance library listed above. You'll notice the end date parameter includes a `timedelta` shift, this is because yfinance is exclusive of the end date you provide. Since we don't want to remember this caveat when setting our parameters, we'll shift the date+1 here using timedelta.

everything to variables:

```
portfolio_df = pd.read_csv('stock_transactions.csv')
portfolio_df['Open date'] = pd.to_datetime(portfolio_df['Open date'])

symbols = portfolio_df.Symbol.unique()
stocks_start = datetime.datetime(2017, 9, 1)
stocks_end = datetime.datetime(2020, 4, 7)

daily_adj_close = get_data(symbols, stocks_start, stocks_end)
daily_adj_close = daily_adj_close[['Close']].reset_index()
daily_benchmark = get_benchmark(['SPY'], stocks_start, stocks_end)
daily_benchmark = daily_benchmark[['Date', 'Close']]
market_cal = create_market_cal(stocks_start, stocks_end)
```

As a point of reference, my CSV file contains the following columns and you'll want to make sure your CSV contains the same columns if you're trying to replicate:

Index	Symbol	Security	Qty	Type	Open date	Adj cost per share	Adj cost
10	FB	FACEBOOK INC CL A	4	Buy	2014-12-31 00:00:00	78.86	315.44
11	FB	FACEBOOK INC CL A	13	Sell.FIFO	2018-06-22 00:00:00	202.451	2631.86

Transaction CSV Format

We now have four crucial datasets in order to proceed:

1. `portfolio_df` with our buy/sell transaction history
2. `daily_adj_close` with daily closes for all tickers in our inventory before the end date specified
3. `daily_benchmark` with daily closes for our benchmark comparison
4. `market_cal` contains dates that the market was open in our timeframe

Using this we can move forward to the next step, onward to glory!

Step 2 — Finding our Initial Active Portfolio

functions, `portfolio_start_balance` and `position_adjust`.

```
def position_adjust(daily_positions, sale):
    stocks_with_sales = pd.DataFrame()
    buys_before_start = daily_positions[daily_positions['Type'] == 'Buy'].sort_values(by='Open d
    for position in buys_before_start[buys_before_start['Symbol'] == sale[1]['Symbol']].iterrows
        if position[1]['Qty'] <= sale[1]['Qty']:
            sale[1]['Qty'] -= position[1]['Qty']
            position[1]['Qty'] = 0
        else:
            position[1]['Qty'] -= sale[1]['Qty']
            sale[1]['Qty'] -= sale[1]['Qty']
        stocks_with_sales = stocks_with_sales.append(position[1])
    return stocks_with_sales


def portfolio_start_balance(portfolio, start_date):
    positions_before_start = portfolio[portfolio['Open date'] <= start_date]
    future_sales = portfolio[(portfolio['Open date'] >= start_date) & (portfolio['Type'] == 'Sel
    sales = positions_before_start[positions_before_start['Type'] =='Sell.FIFO'].groupby(['Symbo
    sales = sales.reset_index()
    positions_no_change = positions_before_start[~positions_before_start['Symbol'].isin(sales['S
    adj_positions_df = pd.DataFrame()
    for sale in sales.iterrows():
        adj_positions = position_adjust(positions_before_start, sale)
        adj_positions_df = adj_positions_df.append(adj_positions)
    adj_positions_df = adj_positions_df.append(positions_no_change)
    adj_positions_df = adj_positions_df.append(future_sales)
    adj_positions_df = adj_positions_df[adj_positions_df['Qty'] > 0]
    return adj_positions_df
```

active_positions.py hosted with ♡ by GitHub view raw

Assigning the output to a variable should give you the active positions within your portfolio:

```
active_portfolio = portfolio_start_balance(portfolio_df,
stocks_start)
```

`portfolio_start_balance`

First, we supply our CSV data and start date to the `portfolio_start_balance` function and create a dataframe of all trades that happened before our start date. We'll then check to see if there are future sales **after** the start_date since we will reconstruct a snapshot of this dataframe in the end:

```
positions_before_start = portfolio[portfolio['Open date'] <=
start_date]
future_sales = portfolio[(portfolio['Open date'] >= start_date) &
(portfolio['Type'] == 'Sell.FIFO')]
```

We'll then create a dataframe of sales that occurred **before** the start_date. We need to make sure that these are all factored out of our active portfolio on the specified start_date:

```
sales = positions_before_start[positions_before_start['Type'] ==
'Sell.FIFO'].groupby(['Symbol'])['Qty'].sum()
sales = sales.reset_index()
```

Next, we'll make a final dataframe of positions that did not have any sales occur over the specified time period:

```
positions_no_change =
positions_before_start[~positions_before_start['Symbol'].isin(sales['
Symbol'].unique())]
```

Now we'll loop through every sale in our `sales` dataframe, call our `position_adjust` function, and then append the output of that into our empty `adj_postitions_df`:

```
adj_positions = position_adjust(positions_before_start, sale)
adj_positions_df = adj_positions_df.append(adj_positions)
```

Let's now look at how the `position_adjust` function works so we can fully understand what's going on here.

`position_adjust`

First, we'll create an empty dataframe called `stocks_with_sales` where we'll later add adjusted positions, and another dataframe holding all of the transactions labeled as 'buys'.

Remember that we already filtered out 'buys in the future' in the `portfolio_start_balance` function, so no need to do it again here. You'll also notice that we're sorting by 'Open Date', and that will be important given we want to subtract positions using the FIFO method. By sorting the dates, we know we can move iteratively through a list of old-to-new positions:

```
stocks_with_sales = pd.DataFrame()
buys_before_start = daily_positions[daily_positions['Type'] ==
'Buy'].sort_values(by='Open date')
```

Now that we have all buys in a single dataframe, we're going to filter for all buys where the stock symbol matches the stock symbol of the sold position:

```
for position in buys_before_start[buys_before_start['Symbol'] ==
sale[1]['Symbol']].iterrows():
```

You'll notice that we're using indexing to access the 'Symbol' column in our data. That's because using `iterrows()` creates a tuple from the index [0] and the series of data [1]. This is the same reason we'll use indexing when we loop through `buys_before_start`:

```
        if position[1]['Qty'] <= sale[1]['Qty']:
            sale[1]['Qty'] -= position[1]['Qty']
            position[1]['Qty'] = 0
        else:
            position[1]['Qty'] -= sale[1]['Qty']
            sale[1]['Qty'] -= sale[1]['Qty']
        stocks_with_sales = stocks_with_sales.append(position[1])
```

So what's happening in the loop here is that for every buy in `buys_before_start`:

- If the quantity of the oldest buy amount is ≤ the sold quantity (aka you sold more than your initial purchase amount), subtract the amount of the buy position from the sell, then set the buy quantity to 0
- Else (the amount you bought on a certain day > the quantity sold), subtract the sales quantity from the buy position, then subtract that same amount from the sales position
- Append that adjusted position to our empty `stock_with_sales` dataframe

Once that loops through every sales position your code will now execute the final lines of `portfolio_start_balance`:

```
adj_positions_df = adj_positions_df.append(positions_no_change)
adj_positions_df = adj_positions_df.append(future_sales)
adj_positions_df = adj_positions_df[adj_positions_df['Qty'] > 0]
```

So we're taking our adjusted positions in `adj_positions_df`, adding back positions that never had sales, adding back sales that occur in the future, and finally filtering out any rows that `position_adjust` zeroed out. You should now have an accurate record of your active holdings as of the start date!

Step 3 — Creating Daily Performance Snapshots

So now that we have an accurate statement of positions held at the start date, let's create daily performance data! Our strategy is similar to what we did in step 2, in fact, we'll re-

`fifo`, and I'll explain what each does in more detail:

```
def fifo(daily_positions, sales, date):
    sales = sales[sales['Open date'] == date]
    daily_positions = daily_positions[daily_positions['Open date'] <= date]
    positions_no_change = daily_positions[~daily_positions['Symbol'].isin(sales['Symbol'].unique
    adj_positions = pd.DataFrame()
    for sale in sales.iterrows():
        adj_positions = adj_positions.append(position_adjust(daily_positions, sale))
    adj_positions = adj_positions.append(positions_no_change)
    adj_positions = adj_positions[adj_positions['Qty'] > 0]
    return adj_positions


def time_fill(portfolio, market_cal):
    sales = portfolio[portfolio['Type'] == 'Sell.FIFO'].groupby(['Symbol','Open date'])['Qty'].s
    sales = sales.reset_index()
    per_day_balance = []
    for date in market_cal:
        if (sales['Open date'] == date).any():
            portfolio = fifo(portfolio, sales, date)
        daily_positions = portfolio[portfolio['Open date'] <= date]
        daily_positions = daily_positions[daily_positions['Type'] == 'Buy']
        daily_positions['Date Snapshot'] = date
        per_day_balance.append(daily_positions)
    return per_day_balance
```

time_fill_daily.py hosted with ♡ by GitHub view raw

time_fill

Similar to `portfolio_start_balance`, our goal is to provide our dataframe of active positions, find the sales, and zero-out sales against buy positions. The main difference here is that we are going to loop through using our `market_cal` list with valid trading days:

```
sales = portfolio[portfolio['Type'] ==
'Sell.FIFO'].groupby(['Symbol','Open date'])['Qty'].sum()
```

```
for date in market_cal:
        if (sales['Open date'] == date).any():
            portfolio = fifo(portfolio, sales, date)
```

This way we can go day-by-day and see if any sales occurred, adjust positions correctly, and then return a correct snapshot of the daily data. In addition, we'll also filter to positions that have occurred before or at the current `date` and make sure there are only buys. We'll then add a `Date Snapshot` column with the current `date` in the market_cal loop, then append it to our `per_day_balance` list:

```
daily_positions = portfolio[portfolio['Open date'] <= date]
daily_positions = daily_positions[daily_positions['Type'] == 'Buy']
daily_positions['Date Snapshot'] = date
per_day_balance.append(daily_positions)
```

fifo

Our `fifo` function takes your active portfolio positions, the sales dataframe created in `time_fill`, and the current `date` in the `market_cal` list. It then filters `sales` to find any that have occurred on the current `date`, and create a dataframe of positions not affected by `sales`:

```
sales = sales[sales['Open date'] == date]
daily_positions = daily_positions[daily_positions['Open date'] <=
date]
positions_no_change = daily_positions[~daily_positions['Symbol'].
isin(sales['Symbol'].unique())]
```

We'll then use our trusty `position_adjust` function to zero-out any positions with active sales. If there were no sales for the specific date, our function will simply append the `positions_no_change` onto the empty `adj_positions` dataframe, leaving you with an accurate daily snapshot of positions:

```
daily_positions, sale))
adj_positions = adj_positions.append(positions_no_change)
adj_positions = adj_positions[adj_positions['Qty'] > 0]
```

Running this line of code should return back a list of all trading days within the time range specified, along with an accurate count of positions per-day:

```
positions_per_day = time_fill(active_portfolio, market_cal)
```

Step 4 — Making Portfolio Calculations

If you're still following along we're in the home stretch! Now that we have an accurate by-day ledger of our active holdings, we can go ahead and create the final calculations needed to generate graphs! We'll be adding an additional six functions to our code to accomplish this:

```
def modified_cost_per_share(portfolio, adj_close, start_date):
    df = pd.merge(portfolio, adj_close, left_on=['Date Snapshot', 'Symbol'],
                  right_on=['Date', 'Ticker'], how='left')
    df.rename(columns={'Close': 'Symbol Adj Close'}, inplace=True)
    df['Adj cost daily'] = df['Symbol Adj Close'] * df['Qty']
    df = df.drop(['Ticker', 'Date'], axis=1)
    return df


def benchmark_portfolio_calcs(portfolio, benchmark):
    portfolio = pd.merge(portfolio, benchmark, left_on=['Date Snapshot'],
                         right_on=['Date'], how='left')
    portfolio = portfolio.drop(['Date'], axis=1)
    portfolio.rename(columns={'Close': 'Benchmark Close'}, inplace=True)
    benchmark_max = benchmark[benchmark['Date'] == benchmark['Date'].max()]
    portfolio['Benchmark End Date Close'] = portfolio.apply(lambda x: benchmark_max['Close'], ax
    benchmark_min = benchmark[benchmark['Date'] == benchmark['Date'].min()]
    portfolio['Benchmark Start Date Close'] = portfolio.apply(lambda x: benchmark_min['Close'],
    return portfolio


def portfolio_end_of_year_stats(portfolio, adj_close_end):
```

```
                                    right_on='Ticker')
    portfolio_end_data.rename(columns={'Close': 'Ticker End Date Close'}, inplace=True)
    portfolio_end_data = portfolio_end_data.drop(['Ticker', 'Date'], axis=1)
    return portfolio_end_data


def portfolio_start_of_year_stats(portfolio, adj_close_start):
    adj_close_start = adj_close_start[adj_close_start['Date'] == adj_close_start['Date'].min()]
    portfolio_start = pd.merge(portfolio, adj_close_start[['Ticker', 'Close', 'Date']],
                               left_on='Symbol', right_on='Ticker')
    portfolio_start.rename(columns={'Close': 'Ticker Start Date Close'}, inplace=True)
    portfolio_start['Adj cost per share'] = np.where(portfolio_start['Open date'] <= portfolio_s
                                                     portfolio_start['Ticker Start Date Clo
                                                     portfolio_start['Adj cost per share'])
    portfolio_start['Adj cost'] = portfolio_start['Adj cost per share'] * portfolio_start['Qty']
    portfolio_start = portfolio_start.drop(['Ticker', 'Date'], axis=1)
    portfolio_start['Equiv Benchmark Shares'] = portfolio_start['Adj cost'] / portfolio_start['B
    portfolio_start['Benchmark Start Date Cost'] = portfolio_start['Equiv Benchmark Shares'] * p
    return portfolio_start


def calc_returns(portfolio):
    portfolio['Benchmark Return'] = portfolio['Benchmark Close'] / portfolio['Benchmark Start Da
    portfolio['Ticker Return'] = portfolio['Symbol Adj Close'] / portfolio['Adj cost per share']
    portfolio['Ticker Share Value'] = portfolio['Qty'] * portfolio['Symbol Adj Close']
    portfolio['Benchmark Share Value'] = portfolio['Equiv Benchmark Shares'] * portfolio['Benchm
    portfolio['Abs Value Compare'] = portfolio['Ticker Share Value'] - portfolio['Benchmark Star
    portfolio['Abs Value Return'] = portfolio['Abs Value Compare']/portfolio['Benchmark Start Da
    portfolio['Stock Gain / (Loss)'] = portfolio['Ticker Share Value'] - portfolio['Adj cost']
    portfolio['Benchmark Gain / (Loss)'] = portfolio['Benchmark Share Value'] - portfolio['Adj c
    portfolio['Abs. Return Compare'] = portfolio['Ticker Return'] - portfolio['Benchmark Return'
    return portfolio


def per_day_portfolio_calcs(per_day_holdings, daily_benchmark, daily_adj_close, stocks_start):
    df = pd.concat(per_day_holdings, sort=True)
    mcps = modified_cost_per_share(df, daily_adj_close, stocks_start)
    bpc = benchmark_portfolio_calcs(mcps, daily_benchmark)
    pes = portfolio_end_of_year_stats(bpc, daily_adj_close)
    pss = portfolio_start_of_year_stats(pes, daily_adj_close)
    returns = calc_returns(pss)
    return returns
```

Let's start with the last function `per_day_portfolio_calcs` since it will use all the other functions.

per_day_portfolio_calcs

Now that we have our `positions_per_day` from step 3, our goal is to pass that along with `daily_benchmark`, `daily_adj_close`, and `stocks_start` to this new function:

```
combined_df = per_day_portfolio_calcs(positions_per_day,
daily_benchmark, daily_adj_close, stocks_start)
```

We'll then concatenate our list of dataframes into a single list using `pd.concat`:

```
df = pd.concat(per_day_holdings, sort=True)
```

Now that we have a single large dataframe we'll pass it to the remaining functions in `per_day_portfolio_calcs`.

modified_cost_per_share

If we want to track daily performance we'll need to know the theoretical value of our holdings per day. This requires taking the amount of securities currently owned and then multiplying it by the daily close for each security owned.

```
mcps = modified_cost_per_share(df, daily_adj_close, stocks_start)
```

To do this, we provide our new single df along with the per-day data we pulled using yfinance, as well as our start date. We'll then merge our portfolio to the daily close data by joining the date of the portfolio snapshot to the date of the daily data, as well as joining on the ticker. For people more familiar with SQL this is essentially a left join:

Once we have our merged `df` we'll rename the daily close to 'Symbol Adj Close', and then multiply the daily close by the quantity of shares owned. Dropping extra columns will return the dataframe we need to proceed:

```
df.rename(columns={'Close': 'Symbol Adj Close'}, inplace=True)
df['Adj cost daily'] = df['Symbol Adj Close'] * df['Qty']
df = df.drop(['Ticker', 'Date'], axis=1)
```

benchmark_portfolio_calcs

Now that we have an accurate daily cost of our securities, we'll want to add in our benchmark to the dataset in order to make comparisons against our portfolio:

```
bpc = benchmark_portfolio_calcs(mcps, daily_benchmark)
```

We start by merging our daily benchmark data to the correct snapshots by using a merge similar to the one in `modified_cost_per_share`:

```
portfolio = pd.merge(portfolio, benchmark, left_on=['Date Snapshot'],
right_on=['Date'], how='left')
portfolio = portfolio.drop(['Date'], axis=1)
portfolio.rename(columns={'Close': 'Benchmark Close'}, inplace=True)
```

Now that we have daily closes for our benchmark merged to our portfolio dataset, we'll filter our `daily_benchmark` data based on its max and min dates. It's important to use max and min vs. your start and end date because the max/min will take into account days where the market was open:

```
benchmark_max = benchmark[benchmark['Date'] ==
benchmark['Date'].max()]
```

```
benchmark_min = benchmark[benchmark['Date'] ==
benchmark['Date'].min()]
portfolio['Benchmark Start Date Close'] = portfolio.apply(lambda x:
benchmark_min['Close'], axis=1)
```

Great! So now we have absolute start and end closes for our benchmark in the portfolio dataset as well, which will be important when calculating returns on a daily basis.

portfolio_end_of_year_stats

So now that our benchmark data is added, let's move onto the next step:

```
pes = portfolio_end_of_year_stats(bpc, daily_adj_close)
```

Our goal here is to take the output of `benchmark_portfolio_calcs`, find the last day of close for all the stocks in the portfolio, and then add a `Ticker End Date Close` column to our portfolio dataset. We'll do this by once again merging to the daily stock data, filtering for the max date, and then joining based on the ticker symbol:

```
adj_close_end = adj_close_end[adj_close_end['Date'] ==
adj_close_end['Date'].max()]

portfolio_end_data = pd.merge(portfolio, adj_close_end,
left_on='Symbol', right_on='Ticker')

portfolio_end_data.rename(columns={'Close': 'Ticker End Date Close'},
inplace=True)

portfolio_end_data = portfolio_end_data.drop(['Ticker', 'Date'],
axis=1)
```

Now just one more step until we generate our calculations!

portfolio_start_of_year_stats

This final step takes the updated portfolio dataframe, the daily stock data from yfinance, and assigns start of year equivalent positions for the benchmark:

We'll first filter the daily close data to its beginning date, then merge our portfolio data to it using the ticker symbol. We'll then call this close `Ticker Start Date Close` for convenience:

```
adj_close_start = adj_close_start[adj_close_start['Date'] ==
adj_close_start['Date'].min()]

portfolio_start = pd.merge(portfolio, adj_close_start[['Ticker',
'Close', 'Date']], left_on='Symbol', right_on='Ticker')

portfolio_start.rename(columns={'Close': 'Ticker Start Date Close'},
inplace=True)
```

Then we need to 'true up' our adjusted cost per share costs, but why? Imagine you bought Google a long time ago at $500/share, but now you want to calculate YTD returns on your position in 2020. If you use $500 as your cost basis for the beginning of 2020, you're not going to have an accurate comparison since the cost basis is from years ago. To fix this, we're going to use Numpy's `where` function:

```
portfolio_start['Adj cost per share'] =
np.where(portfolio_start['Open date'] <= portfolio_start['Date'],
         portfolio_start['Ticker Start Date Close'],
         portfolio_start['Adj cost per share'])
```

Simply put, this is saying 'if the open date is ≤ the date of the start date, then `Adj cost per share` is equal to `Ticker Start Date Close`' (closing price of the stock from the min date on the yfinance data). If not, then use the existing `Adj cost per share`.

The remaining part modifies the adjusted cost based on the modified cost per share, drops unneeded columns from the merge, and then calculates the equivalent amount of benchmarks shares you would have owned based on your newly calculated adjusted cost:

```
portfolio_start = portfolio_start.drop(['Ticker', 'Date'], axis=1)

portfolio_start['Equiv Benchmark Shares'] = portfolio_start['Adj
cost'] / portfolio_start['Benchmark Start Date Close']

portfolio_start['Benchmark Start Date Cost'] = portfolio_start['Equiv
Benchmark Shares'] * portfolio_start['Benchmark Start Date Close']
```

Congratulations, we now have all the necessary data to calculate returns properly! Let's knock out this last section and then dive into visualizing this!

calc_returns

The final step here simply takes the aggregated dataframe from all the other functions, applies a bunch of calculations against the data we've been modifying, and returns a final dataframe:

```
returns = calc_returns(pss)
```

The first set, `Benchmark Return` and `Ticker Return`, both use a current close price divided by their beginning cost basis to calculate a return:

```
portfolio['Benchmark Return'] = portfolio['Benchmark Close'] /
portfolio['Benchmark Start Date Close'] - 1

portfolio['Ticker Return'] = portfolio['Symbol Adj Close'] /
portfolio['Adj cost per share'] - 1
```

Share value for each is calculated the same way, using the modified per-day quantities and equivalent benchmark shares we calculated earlier:

```
portfolio['Ticker Share Value'] = portfolio['Qty'] *
portfolio['Symbol Adj Close']
```

We'll do the same thing again to calculate monetary gain/loss, subtracting the share value columns from the modified adjusted cost we calculated in the `portfolio_start_of_year_stats` function:

```
portfolio['Stock Gain / (Loss)'] = portfolio['Ticker Share Value'] -
portfolio['Adj cost']

portfolio['Benchmark Gain / (Loss)'] = portfolio['Benchmark Share
Value'] - portfolio['Adj cost']
```

Finally, we'll calculate absolute return values using the benchmark metrics we calculated earlier:

```
portfolio['Abs Value Compare'] = portfolio['Ticker Share Value'] -
portfolio['Benchmark Start Date Cost']

portfolio['Abs Value Return'] = portfolio['Abs Value
Compare']/portfolio['Benchmark Start Date Cost']

portfolio['Abs. Return Compare'] = portfolio['Ticker Return'] -
portfolio['Benchmark Return']
```

Boom! Now let's figure out how to graph our new data and finish this up.

Step 4 — Visualize the Data

So now that we went through all of that to get our daily performance data, how should we best display it? The biggest benefit of this daily data is to see how your positions perform *over time*, so let's try looking at our data on an aggregated basis first.

I've been using Plotly a lot for recent side projects, so for doing this I'm going to opt for simple and go with the Plotly Express library. Since we'll need to aggregate each day's stocks into a single metric per-day, I'm going to write this as a function that takes your completed dataframe and two metrics you want to plot against each other:

return something similar to this!

Aggregated by-day gain/loss vs. benchmark

You can also aggregate using different metrics like `Abs Value Compare` to see this as a single line:

Displaying using Absolute Value Compare metric

benchmark against each ticker's performance:

We'll also use the `facet_col_wrap` parameter in order to limit the amount of graphs per row. Running this code should generate something similar to the output below!

Example data of benchmark return comparison per ticker

Conclusion

We covered a lot of ground here, and hopefully, this has been helpful for learning more about populating and analyzing financial data! There's a lot more that can be explored in the future, including:

- Factoring in dividends and splits — yfinance has this data available, but I wanted to publish this as a first step before adding more features
- Adding in more exotic metrics, such as volatility comparisons and rolling averages. This could be interesting to compare on a portfolio-level vs. a benchmark
- Signals for better actionability. For example, you could take your by-day data and look at when a short-term moving average crosses a long-term moving average as a

Hope this was helpful to anyone out there, and feel free to reach out or comment below with questions/comments. Thanks for reading!