Equal-Weight S&P 500 Index Fund

Introduction & Library Imports

The S&P 500 is the world's most popular stock market index. The largest fund that is benchmarked to this index is the SPDR® S&P 500® ETF Trust. It has more than US$250 billion of assets under management.

The goal of this section of the course is to create a Python script that will accept the value of your portfolio and tell you how many shares of each S&P 500 constituent you should purchase to get an equal-weight version of the index fund.

Library Imports

The first thing we need to do is import the open-source software libraries that we'll be using in this tutorial.

In [1]:
```
import numpy as np #The Numpy numerical computing library
import pandas as pd #The Pandas data science library
import requests #The requests library for HTTP requests in Python
import xlsxwriter #The XlsxWriter libarary for
import math #The Python math module
```

Importing Our List of Stocks

The next thing we need to do is import the constituents of the S&P 500.

These constituents change over time, so in an ideal world you would connect directly to the index provider (Standard & Poor's) and pull their real-time constituents on a regular basis.

Paying for access to the index provider's API is outside of the scope of this course.

There's a static version of the S&P 500 constituents available here. Click this link to download them now. Move this file into the `starter-files` folder so it can be accessed by other files in that directory.

Now it's time to import these stocks to our Jupyter Notebook file.

In [2]:
```
stocks = pd.read_csv('sp_500_stocks.csv')
```

Acquiring an API Token

Now it's time to import our IEX Cloud API token. This is the data provider that we will be using throughout this course.

API tokens (and other sensitive information) should be stored in a `secrets.py` file that doesn't get pushed to your local Git repository. We'll be using a sandbox API token in this course, which means that the data we'll use is randomly-generated and (more importantly) has no cost associated with it.

Click here to download your `secrets.py` file. Move the file into the same directory as this Jupyter Notebook before proceeding.

In [3]:
```
from secrets import IEX_CLOUD_API_TOKEN
```

Making Our First API Call

Now it's time to structure our API calls to IEX cloud.

We need the following information from the API:

- Market capitalization for each stock
- Price of each stock

In [4]:
```
symbol='AAPL'
api_url = f'https://sandbox.iexapis.com/stable/stock/{symbol}/quote?token={IEX_CLOUD_API_T
data = requests.get(api_url).json() # retrieve stock data and pass into .json format (info
data # can type print(data) for json file to list all different variable in this python li
```

Out[4]:

```
{'avgTotalVolume': 99442791,
 'calculationPrice': 'iexlasttrade',
 'change': 2.497,
 'changePercent': 0.01615,
 'close': 0,
 'closeSource': 'oliacffi',
 'closeTime': None,
 'companyName': 'Apple Inc',
 'currency': 'USD',
 'delayedPrice': None,
 'delayedPriceTime': None,
 'extendedChange': None,
 'extendedChangePercent': None,
 'extendedPrice': None,
 'extendedPriceTime': None,
 'high': 0,
 'highSource': None,
 'highTime': None,
 'iexAskPrice': 0,
 'iexAskSize': 0,
 'iexBidPrice': 0,
 'iexBidSize': 0,
 'iexClose': 157.019,
 'iexCloseTime': 1652613261691,
 'iexLastUpdated': 1678977170284,
 'iexMarketPercent': 0.023790457251048606,
 'iexOpen': 158.7,
 'iexOpenTime': 1670913181936,
 'iexRealtimePrice': 161.014,
 'iexRealtimeSize': 8,
 'iexVolume': 2667037,
 'lastTradeTime': 1668019300826,
 'latestPrice': 155.341,
 'latestSource': 'IEX Last Trade',
 'latestTime': 'May 10, 2022',
 'latestUpdate': 1683643085286,
 'latestVolume': None,
 'low': 0,
 'lowSource': None,
 'lowTime': None,
 'marketCap': 2503849151664,
 'oddLotDelayedPrice': None,
 'oddLotDelayedPriceTime': None,
 'open': 0,
 'openTime': None,
 'openSource': 'ifcfliao',
```

```
'peRatio': 25.38,
'previousClose': 159.63,
'previousVolume': 137719595,
'primaryExchange': 'SNDAAQ',
'symbol': 'AAPL',
'volume': None,
'week52High': 184.92,
'week52Low': 125.24,
'ytdChange': -0.13125233605030498,
'isUSMarketOpen': True}
```

Parsing Our API Call

The API call that we executed in the last code block contains all of the information required to build our equal-weight S&P 500 strategy.

With that said, the data isn't in a proper format yet. We need to parse it first.

In [5]:
```
data['latestPrice'] # 'latestPrice' from the above returned python library returns the sto
data['marketCap']
```

Out[5]:
```
2503849151664
```

Adding Our Stocks Data to a Pandas DataFrame

The next thing we need to do is add our stock's price and market capitalization to a pandas DataFrame. Think of a DataFrame like the Python version of a spreadsheet. It stores tabular data.

In [6]:
```
my_columns = ['Ticker', 'Price','Market Capitalization', 'Number Of Shares to Buy'] # crea
final_dataframe = pd.DataFrame(columns = my_columns) # create a pandas data frame using pi
final_dataframe
```

Out[6]:

Ticker	Price	Market Capitalization	Number Of Shares to Buy

In [7]:
```
final_dataframe = final_dataframe.append(
                                        pd.Series(['AAPL', # a panda series is how a table
                                                   data['latestPrice'],
                                                   data['marketCap'],
                                                   'N/A'],
                                                  index = my_columns), # adds symbol, pric
                                        ignore_index = True)
final_dataframe
```

Out[7]:

	Ticker	Price	Market Capitalization	Number Of Shares to Buy
0	AAPL	155.341	2503849151664	N/A

Looping Through The Tickers in Our List of Stocks

Using the same logic that we outlined above, we can pull data for all S&P 500 stocks and store their data in the DataFrame using a `for` loop.

In [8]:
```
final_dataframe = pd.DataFrame(columns = my_columns)
for symbol in stocks['Ticker']: # for loop to obtain stock info in json through API then a
```

```
    api_url = f'https://sandbox.iexapis.com/stable/stock/{symbol}/quote?token={IEX_CLOUD_A
    data = requests.get(api_url).json()
    final_dataframe = final_dataframe.append(
                                        pd.Series([symbol,
                                                   data['latestPrice'],
                                                   data['marketCap'],
                                                   'N/A'],
                                                  index = my_columns),
                                        ignore_index = True)
```

In [9]:
```
final_dataframe
```

Out[9]:

	Ticker	Price	Market Capitalization	Number Of Shares to Buy
0	A	118.850	36752196069	N/A
1	AAL	17.100	11021688347	N/A
2	AAP	212.220	12951255189	N/A
3	AAPL	157.042	2598483482832	N/A
4	ABBV	158.000	269811748275	N/A
...	...	...	...	...
130	DG	234.730	52785293540	N/A
131	DGX	136.270	16390295883	N/A
132	DHI	70.870	25007126424	N/A
133	DHR	240.720	179529123141	N/A
134	DIS	108.360	199356950803	N/A

135 rows × 4 columns

Using Batch API Calls to Improve Performance

Batch API calls are one of the easiest ways to improve the performance of your code.

This is because HTTP requests are typically one of the slowest components of a script.

Also, API providers will often give you discounted rates for using batch API calls since they are easier for the API provider to respond to.

IEX Cloud limits their batch API calls to 100 tickers per request. Still, this reduces the number of API calls we'll make in this section from 500 to 5 - huge improvement! In this section, we'll split our list of stocks into groups of 100 and then make a batch API call for each group.

In [10]:
```
# Function sourced from
# https://stackoverflow.com/questions/312443/how-do-you-split-a-list-into-evenly-sized-chu
def chunks(lst, n):
    """Yield successive n-sized chunks from lst."""
    for i in range(0, len(lst), n):
        yield lst[i:i + n]
```

In [11]:
```
symbol_groups = list(chunks(stocks['Ticker'], 100))
symbol_strings = []
```

```
for i in range(0, len(symbol_groups)):
    symbol_strings.append(','.join(symbol_groups[i]))
#     print(symbol_strings[i])

final_dataframe = pd.DataFrame(columns = my_columns)

for symbol_string in symbol_strings: # perform batch calls to obtain desired information
#     print(symbol_strings)
    batch_api_call_url = f'https://sandbox.iexapis.com/stable/stock/market/batch/?types=qu
    data = requests.get(batch_api_call_url).json()
    for symbol in symbol_string.split(','):
        final_dataframe = final_dataframe.append(
                                        pd.Series([symbol,
                                                   data[symbol]['quote']['latestPrice'],
                                                   data[symbol]['quote']['marketCap'],
                                                   'N/A'],
                                                  index = my_columns),
                                        ignore_index = True)


final_dataframe
```

Out[11]:

	Ticker	Price	Market Capitalization	Number Of Shares to Buy
0	A	122.280	35187220524	N/A
1	AAL	16.590	10869058979	N/A
2	AAP	211.260	12607343901	N/A
3	AAPL	154.866	2570477187734	N/A
4	ABBV	159.000	269726032011	N/A
...	...	...	...	...
130	DG	238.340	52746181931	N/A
131	DGX	140.610	15959862331	N/A
132	DHI	70.500	24683308842	N/A
133	DHR	243.360	183233117455	N/A
134	DIS	109.850	201997285333	N/A

135 rows × 4 columns

Calculating the Number of Shares to Buy

As you can see in the DataFrame above, we stil haven't calculated the number of shares of each stock to buy.

We'll do that next.

In [17]:
```
portfolio_size = input("Enter the value of your portfolio:") # get invested money input fi

try:
    val = float(portfolio_size)
except ValueError:
    print("That's not a number! \n Try again:")
    portfolio_size = input("Enter the value of your portfolio:")
```

In [18]:
```
position_size = float(portfolio_size) / len(final_dataframe.index) # Divide portfolio size
```

```
for i in range(0, len(final_dataframe['Ticker'])-1):
    final_dataframe.loc[i, 'Number Of Shares to Buy'] = math.floor(position_size / final_d
final_dataframe
```

Out[18]:

	Ticker	Price	Market Capitalization	Number Of Shares to Buy
0	A	122.280	35187220524	60577424
1	AAL	16.590	10869058979	446498336
2	AAP	211.260	12607343901	35062990
3	AAPL	154.866	2570477187734	47831075
4	ABBV	159.000	269726032011	46587467
...	...	...	...	...
130	DG	238.340	52746181931	31079161
131	DGX	140.610	15959862331	52680516
132	DHI	70.500	24683308842	105069608
133	DHR	243.360	183233117455	30438064
134	DIS	109.850	201997285333	N/A

135 rows × 4 columns

Formatting Our Excel Output

We will be using the XlsxWriter library for Python to create nicely-formatted Excel files.

XlsxWriter is an excellent package and offers tons of customization. However, the tradeoff for this is that the library can seem very complicated to new users. Accordingly, this section will be fairly long because I want to do a good job of explaining how XlsxWriter works.

Initializing our XlsxWriter Object

In [19]:

```
writer = pd.ExcelWriter('recommended_trades.xlsx', engine='xlsxwriter') # output the recor
final_dataframe.to_excel(writer, sheet_name='Recommended Trades', index = False)
```

Creating the Formats We'll Need For Our `.xlsx` File

Formats include colors, fonts, and also symbols like `%` and `$`. We'll need four main formats for our Excel document:

- String format for tickers
- $XX.XX format for stock prices
- $XX,XXX format for market capitalization
- Integer format for the number of shares to purchase

In [20]:

```
background_color = '#0a0a23' # format the xlsx file accordingly
font_color = '#ffffff'

string_format = writer.book.add_format(
        {
```

```
            'font_color': font_color,
            'bg_color': background_color,
            'border': 1
        }
    )

dollar_format = writer.book.add_format(
        {
            'num_format':'$0.00',
            'font_color': font_color,
            'bg_color': background_color,
            'border': 1
        }
    )

integer_format = writer.book.add_format(
        {
            'num_format':'0',
            'font_color': font_color,
            'bg_color': background_color,
            'border': 1
        }
    )
```

Applying the Formats to the Columns of Our `.xlsx` File

We can use the `set_column` method applied to the `writer.sheets['Recommended Trades']` object to apply formats to specific columns of our spreadsheets.

Here's an example:

```
writer.sheets['Recommended Trades'].set_column('B:B', #This tells the method to apply
the format to column B
                    18, #This tells the method to apply a column width of 18 pixels
                    string_format #This applies the format 'string_format' to the
column
                   )
```

In [21]:

```
# writer.sheets['Recommended Trades'].write('A1', 'Ticker', string_format)
# writer.sheets['Recommended Trades'].write('B1', 'Price', string_format)
# writer.sheets['Recommended Trades'].write('C1', 'Market Capitalization', string_format)
# writer.sheets['Recommended Trades'].write('D1', 'Number Of Shares to Buy', string_format
# writer.sheets['Recommended Trades'].set_column('A:A', 20, string_format)
# writer.sheets['Recommended Trades'].set_column('B:B', 20, dollar_format)
# writer.sheets['Recommended Trades'].set_column('C:C', 20, dollar_format)
# writer.sheets['Recommended Trades'].set_column('D:D', 20, integer_format)
```

This code works, but it violates the software principle of "Don't Repeat Yourself".

Let's simplify this by putting it in 2 loops:

In [22]:

```
column_formats = {
                    'A': ['Ticker', string_format],
                    'B': ['Price', dollar_format],
                    'C': ['Market Capitalization', dollar_format],
                    'D': ['Number of Shares to Buy', integer_format]
                    }

for column in column_formats.keys():
```

```
    writer.sheets['Recommended Trades'].set_column(f'{column}:{column}', 20, column_format
    writer.sheets['Recommended Trades'].write(f'{column}1', column_formats[column][0], str
```

Saving Our Excel Output

Saving our Excel file is very easy:

In [23]:

```
writer.save()
```