TECHNOLOGY NEWS

EXPLORING PYTHON'S PANDAS LIBRARY

Winfred Lam, CFA

Share this article

Python's emergence as the programming language of choice for everything from web development to scientific computing, data analytics, and artificial intelligence is largely due to its array of available open-source libraries. A library is a collection of modules (essentially, a Python file with more code) that allows programmers to build on code already written by others, which serves as a feedback loop to invite even more developers to expand the ecosystem. Unsurprisingly, one of the areas with extensive libraries is data analysis, which will be the focus of this article.

Pandas

We will focus on Pandas, a popular library that greatly enhances Python's ability to work with structured data and spreadsheets. It has obvious similarities to Microsoft Excel and is a good starting point for finance professionals. We provide a walk-through of basic operations in Python and finish with a demo that uses Pandas to manipulate real financial data.

As a third-party library, Pandas is not included in the standard Python package and must be installed separately. Fortunately, this is very easily done via pip install or through Anaconda distribution. Once installed, it should be imported at the beginning of every script.

Figure 1

```
[ ] !pip install pandas
```

Creating a DataFrame

With Pandas, programmers work with two-dimensional arrays called DataFrames, which essentially function like tables. Loading Excel files into a Pandas DataFrame is easy (we will cover this later), but for now, we go through the basics of creating and manipulating DataFrames.

There are several ways to create a DataFrame. Below demonstrates a method using a dictionary where keys represent column labels and values are lists of equal length containing data. The dictionary is then used to create a new DataFrame object called new_df.

Tip: Using an.ipynb notebook is generally preferred when working with DataFrames. It renders DataFrames as HTML-like tables, allowing programmers to visualize and interactively modify them line by line.

Figure 2

```
[ ] import pandas as pd
    our_dataset = {"A": [1,2,3],
                   "B": [4,5,6],
                   "C": [7,8,9]}

    new_df = pd.DataFrame(data=our_dataset)
    new_df
```

	A	B	C
0	1	4	7
1	2	5	8
2	3	6	9

By default, when creating a new DataFrame, Pandas assign row indices unless otherwise specified, while the columns are labeled with the keys "A", "B", "C" that we provided. These labels are not considered part of the dataset, although they appear in the top row. Labels can be any hashable data type (for more on hashable data types, refer to "Hashing in Python").

Selecting individual values in a DataFrame

We can reference values in the DataFrame using the df.loc[] or df.iloc[] methods. This is like selecting a cell in an Excel spreadsheet (e.g., cell "A1").

With df.loc[], values are accessed based on the index (for the row) and the label (for the column).

Figure 3

```
[ ] num1 = new_df.loc[1, "B"] #take index 1 in column "B"
    num2 = new_df.loc[2, "C"] #take index 2 in column "C"

    print(num1, num2)
```

```
5 9
```

With df.iloc[], both rows and columns are referenced based on indices. Here, the column "A" represents index 0, "B" represents index 1, and so on.

Figure 4

```
[ ] num1 = new_df.iloc[1, 1] #take index 1 in column "B"
    num2 = new_df.iloc[2, 2] #take index 2 in column "C"

    print(num1, num2)

    5 9
```

Using df.loc[] or df.iloc[], we can overwrite values in the DataFrame. Note that the data types within a column or row do not need to be homogeneous; we can store a string just as easily as an integer. However, this flexibility can potentially lead to issues, as we will see later.

Figure 5

```
[ ] new_df.loc[1, "B"] = 20 #write 20 into row 1, column "B"
    new_df.iloc[2, 2] = "thirty" #write the string "thirty" into row 2, column 2
    new_df
```

	A	B	C
0	1	4	7
1	2	20	8
2	3	6	thirty

loc and iloc are also great for slicing, enabling us to select portions of a DataFrame. Note that slicing does not modify the original itself but allows programmers to reference subsets of the data. As a general guide, modifications, additions, and overwrites require using the assignment operator "=".

Figure 6

```
[ ] new_df.loc[1:, "B":"C"] #reference row 1 and down, from B to C
```

	B	C
1	20	8
2	6	thirty

```
[ ] new_df #remains unchanged
```

	A	B	C
0	1	4	7
1	2	20	8
2	3	6	thirty

Adding a new column or row

Inserting a new column is simple.

Figure 7

```
[ ] new_df["D"] = [10, 11, 12]
    new_df
```

	A	B	C	D
0	1	4	7	10
1	2	20	8	11
2	3	6	thirty	12

To add a new row, we can use the df.loc method. However, the DataFrame's index must be continuous. If there are gaps in the index (like missing numbers), adding a row using df.loc might unintentionally overwrite existing data.

Figure 8

```
[ ] new_df.loc[len(new_df)] = [-1, -4, -7, - 10]
    new_df
```

	A	B	C	D
0	1	4	7	10
1	2	20	8	11
2	3	6	thirty	12
3	-1	-4	-7	-10

Filtering

Filtering allows us to select specific rows or columns based on certain conditions.

Figure 9

```
[ ] new_df[new_df["A"] > 0] # filtering for column "A" larger than 0.
    #What if we filter with new_df[new_df["C"] > 0], which has the string "thirty"? Hint: from prior section, we mentioned the importance of being careful with data types
```

	A	B	C	D
0	1	4	7	10
1	2	20	8	11
2	3	6	thirty	12

```
[ ] new_df[(new_df["A"] > 0) & (new_df["D"] != 11)] #filtering on multiple conditions
```

	A	B	C	D
0	1	4	7	10
2	3	6	thirty	12

Performing computations

One effective method of performing computations across a DataFrame is the apply() function, which has the below syntax.

Figure 10

```
[ ] #DataFrame.apply(func, axis=0, raw=False, result_type=None, args=(), **kwds)
```

For straightforward operations, a lambda function can be used with apply(). Below, we demonstrate adding a new column "E" (specified with axis=1) to the DataFrame. Each value in "E" is computed as the sum of the squared values from columns "A" and "B", multiplied by the corresponding value in column "D". (In this case, we can also make use of vectorized operations, which is even faster. This will be demonstrated at the end of this article.)

Figure 11

```
new_df["E"] = new_df.apply(lambda row: (row["A"] + row["B"])**2*row["D"], axis=1)
new_df
```

	A	B	C	D	E
0	1	4	7	10	250
1	2	20	8	11	5324
2	3	6	thirty	12	972
3	-1	-4	-7	-10	-250

Not all operations can be performed using lambda functions. Fortunately, df.apply() can support more advanced, user-defined functions. This is where the advantage of being part of the Python environment becomes evident, as we can draw on other libraries.

As an example, suppose we have a spreadsheet/DataFrame of customer support messages with the associated customer contact info, and want to extract the phone numbers or emails into a separate column. This can be achieved relatively easily with the use of the regular expression (regex) library (for more on regex, refer to "Regular Expression in Python").

Figure 12

```
import re
pd.set_option('display.max_colwidth', None)

# Create sample_df with messages_data with sentences containing phone numbers and email addresses
sample_data = {
    'Raw_Text': [
        "Contact us at support@hotmail.com or call 123-456-7890/ 000.213-2198.",
        "Send your feedback to feedback@gmail.com or call 123.456.7890.",
        "I am out of office. Please email alice@XYZ.com or bob@XYZ.com.",
        "Hello World",
        "Reach out at sales@CFA.com or 123 456 7890 for sales inquiries.",
        "Phone: 1234567890, Email: contact@cfa-toronto.com"
    ]
}

df = pd.DataFrame(sample_data)

# Custom function to parse contact info
def parse_phone_number_or_email(text):

    # Regex patterns to find 1) email addresses and 2)phone numbers
    email_pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
    phone_pattern = r'\b\d{3}[-.\s]?\d{3}[-.\s]?\d{4}\b'

    # Feed patterns into re.findall() from re module to match patterns for email addresses and phone numbers
    emails = re.findall(email_pattern, text)
    phones = re.findall(phone_pattern, text)

    # Join found emails and phones, and combine to form one final string "contact_info"
    email_str = ', '.join(emails) if emails else None
    phone_str = ', '.join(phones) if phones else None
    contact_info = ', '.join(filter(None, [email_str, phone_str])) if email_str or phone_str else None

    # Return a pandas series
    return pd.Series({'Contact_Info': contact_info})

# Apply the function to the 'Raw Text' column
df[['Contact_Info']] = df['Raw_Text'].apply(parse_phone_number_or_email)
df
```

	Raw_Text	Contact_Info
0	Contact us at support@hotmail.com or call 123-456-7890/ 000.213-2198.	support@hotmail.com, 123-456-7890, 000.213-2198
1	Send your feedback to feedback@gmail.com or call 123.456.7890.	feedback@gmail.com, 123.456.7890
2	I am out of office. Please email alice@XYZ.com or bob@XYZ.com.	alice@XYZ.com, bob@XYZ.com
3	Hello World	None
4	Reach out at sales@CFA.com or 123 456 7890 for sales inquiries.	sales@CFA.com, 123 456 7890
5	Phone: 1234567890, Email: contact@cfa-toronto.com	contact@cfa-toronto.com, 1234567890

Reading and writing data

We have walked through the basics of creating and manipulating DateFrames, but professionals often work with existing datasets in formats like .csv, .xlsx, .json, and SQL databases.

Pandas has extensive input/output (I/O) support, allowing users to read various types of files into DataFrames. Here are the functions for some of the popular file types, as well as required arguments:

- pd.read_excel("full_file_path")
- pd.read_csv("full_file_path")
- pd.read_json("full_file_path")
- pd.read_sql("SQL QUERY", connection) connection supported: ADBC Connection, SQLAlchemy connectable, str, or sqlite3

Below is a simple load example, using a data set provided by Google Colab. We use the method df.head(5) to view only the first five rows.

Figure 13

```
california_housing_df = pd.read_csv("sample_data/california_housing_train.csv")
california_housing_df.head(5)
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
0	-114.31	34.19	15.0	5612.0	1283.0	1015.0	472.0	1.4936	66900.0
1	-114.47	34.40	19.0	7650.0	1901.0	1129.0	463.0	1.8200	80100.0
2	-114.56	33.69	17.0	720.0	174.0	333.0	117.0	1.6509	85700.0
3	-114.57	33.64	14.0	1501.0	337.0	515.0	226.0	3.1917	73400.0
4	-114.57	33.57	20.0	1454.0	326.0	624.0	262.0	1.9250	65500.0

Naturally, output is also widely supported by Pandas. Below is a list of output functions for the aforementioned types.

- pd.to_excel("full_file_path")
- pd.to_csv("full_file_path")
- pd.to_json("full_file_path")
- pd.to_sql("Table_name", connection)

Here is an example of writing the DataFrame to Excel.

Figure 14

```
california_housing_df.to_excel("my_file_path/california_housing_data.xlsx")
```

Comparisons to Excel: Speed and scalability of Pandas

Like other Python data types, the DataFrame is an object that resides in computer memory (RAM), enabling faster in-memory computations than Excel, which relies on more I/O when handling large datasets or complex formulas that reference other sheets. (For more on computer memory, refer to "Copy and Reference in Python.")

Excel's graphical user interface (GUI), which makes it easy for beginners to pick up, also contributes to slower performance due to the overhead of real-time updates and handling user interactions. Meanwhile, Pandas computations are highly optimized, benefiting from its efficient implementation in C, as well as leveraging NumPy (another Python library) for vectorized operations, which are significantly faster for numerical computations compared to traditional looping (i.e., in VBA macros). Scalability is another factor, as Excel has a cap of just over a million rows of data, while Pandas DataFrames are theoretically bounded by the amount of computer RAM, which for a standard 16 GB RAM computer can go up to several hundred million rows.

In addition, being part of the Python ecosystem means that operations are easier to automate and can be easily version-controlled using systems like Git, which can track line-by-line changes. This can be more challenging for VBA macros embedded in Excel files.

Although it comes at the cost of a steeper initial learning curve, Pandas's speed, scalability, and integration with the rest of the Python ecosystem make it a must-have for large-scale, complex data analysis. However, Excel remains an excellent choice for small datasets and quick visualizations thanks to its GUI.

Putting it all together using real-world financial data

There are many other DataFrame operations, such as merging and summarizing, that we do not have enough scope to cover. Instead, we will finish with a demo to showcase how beginners can leverage Python's extensive programming environment to automate basic extract, transform, load (ETL). We highly recommend readers check out the Pandas documentation for a deeper dive into Pandas.

An important advantage of Python is the availability of many application programming interfaces (APIs) to retrieve data. An API is a set of protocols that enable a software application to interact with another—in this case to retrieve data from a source or server.

Below, we showcase the use of the Yahoo Finance Python API, which allows us to easily pull historical data into a Pandas DataFrame. From there, we can perform various automated data computations, then export and/or share. Best of all, this is all free.

We will break the process down cell by cell before presenting it all in one final script.

Pip install Yahoo Finance

Figure 15

```
[ ] !pip install yfinance
```

Define stock tickers to pull

In this case we will pull the daily closing prices of the popular Magnificent Seven stocks and the S&P 500 Index and put them all in one DataFrame. The more detailed documentation is available on PyPI.

We will only use the yfinance.download() call, which returns a Pandas DataFrame of historical data for a given ticker. This is what the DataFrame looks like for S&P 500 Index.

Figure 16

```
[ ] import yfinance as yf
    tickers = ["^SPX", "AMZN", "AAPL", "GOOGL", "NVDA", "MSFT", "META", "TSLA"]

    #the API pulls the earliest/latest available data, so don't worry about the end_date being in the future
    start_date = "1928-01-01"
    end_date = "2024-12-31"

    yf.download(tickers[0], start_date, end_date) #yf.download returns a pandas dataframe
```

```
[*********************100%***********************]  1 of 1 completed
```

Date	Open	High	Low	Close	Adj Close	Volume
1928-01-03	17.760000	17.760000	17.760000	17.760000	17.760000	0
1928-01-04	17.719999	17.719999	17.719999	17.719999	17.719999	0
1928-01-05	17.549999	17.549999	17.549999	17.549999	17.549999	0
1928-01-06	17.660000	17.660000	17.660000	17.660000	17.660000	0
1928-01-09	17.500000	17.500000	17.500000	17.500000	17.500000	0
...	...	...	...	...	...	...
2024-06-14	5424.080078	5432.390137	5403.750000	5431.600098	5431.600098	3438650000
2024-06-17	5431.109863	5488.500000	5420.399902	5473.229980	5473.229980	3447840000
2024-06-18	5476.149902	5490.379883	5471.319824	5487.029785	5487.029785	3544330000
2024-06-20	5499.990234	5505.529785	5455.560059	5473.169922	5473.169922	3847060000
2024-06-21	5466.770020	5478.310059	5452.029785	5464.620117	5464.620117	6773800000

24233 rows × 6 columns

We do for this for all eight tickers, extract only the "Adj Close" column, and stack them into a DataFrame.

Figure 17

```
[ ] price_hist_mag_seven_df = pd.DataFrame()

    for ticker in tickers:
      adj_close_hist = yf.download(ticker, start_date, end_date)["Adj Close"]
      price_hist_mag_seven_df[ticker] = adj_close_hist

    price_hist_mag_seven_df
```

Date	^SPX	AMZN	AAPL	GOOGL	NVDA	MSFT	META	TSLA
1928-01-03	17.760000	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1928-01-04	17.719999	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1928-01-05	17.549999	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1928-01-06	17.660000	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1928-01-09	17.500000	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...
2024-06-14	5431.600098	183.660004	212.490005	176.789993	131.880005	442.570007	504.160004	178.009995
2024-06-17	5473.229980	184.059998	216.669998	177.240005	130.979996	448.369995	506.630005	187.440002
2024-06-18	5487.029785	182.809998	214.289993	175.089996	135.580002	446.339996	499.489990	184.860001
2024-06-20	5473.169922	186.100006	209.679993	176.300003	130.779999	445.700012	501.700012	181.570007
2024-06-21	5464.620117	189.080002	207.490005	179.630005	126.570000	449.779999	494.779999	183.009995

24233 rows × 8 columns

Pulling all the tickers and combining them into one DataFrame, we note that all the Magnificent Seven stocks have a NaN, which makes sense as none of them had IPOed back in the 1920s. Pandas allows us to easily deal with null values by dropping rows with NaN via df.dropna(). As such, we are left with data as recent as 2012, the day of Meta's IPO.

Figure 18

```
[ ] price_hist_mag_seven_df = price_hist_mag_seven_df.dropna()
    price_hist_mag_seven_df
```

Date	^SPX	AMZN	AAPL	GOOGL	NVDA	MSFT	META	TSLA
2012-05-18	1295.219971	10.692500	16.014681	15.007801	0.277002	23.486095	38.151604	1.837333
2012-05-21	1315.989990	10.905500	16.947702	15.350501	0.281818	23.871246	33.960213	1.918000
2012-05-22	1316.630005	10.766500	16.817568	15.017799	0.278378	23.879274	30.936428	2.053333
2012-05-23	1318.859985	10.864000	17.227915	15.234267	0.285258	23.357718	31.934378	2.068000
2012-05-24	1320.680054	10.762000	17.069689	15.089290	0.277690	23.325607	32.962265	2.018667
...	...	...	...	...	...	...	...	...
2024-06-14	5431.600098	183.660004	212.490005	176.789993	131.880005	442.570007	504.160004	178.009995
2024-06-17	5473.229980	184.059998	216.669998	177.240005	130.979996	448.369995	506.630005	187.440002
2024-06-18	5487.029785	182.809998	214.289993	175.089996	135.580002	446.339996	499.489990	184.860001
2024-06-20	5473.169922	186.100006	209.679993	176.300003	130.779999	445.700012	501.700012	181.570007
2024-06-21	5464.620117	189.080002	207.490005	179.630005	126.570000	449.779999	494.779999	183.009995

3042 rows × 8 columns

Suppose we are interested in comparing cumulative returns over time. Leveraging several built-in Pandas vectorized operations like df.pct_change() and df.cumprod(), we can easily convert the daily prices to cumulative returns, as below.

(Note that this computation method will fail to capture the true total return, as the data is not dividend-adjusted. Fortunately for us, the period captured is relatively short, while several of these stocks have not paid dividends over that time.)

Figure 19

```
[ ] dod_return_mag_seven_df = price_hist_mag_seven_df.pct_change().dropna() #find the percent change and drop NaN (from the first row)
    cumulative_returns_df = (1 + dod_return_mag_seven_df).cumprod()-1
    cumulative_returns_pct_df = cumulative_returns_df*100
    cumulative_returns_pct_df
```

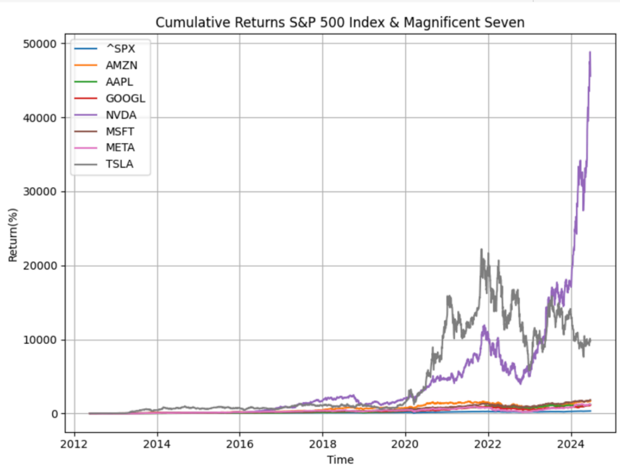
Date	^SPX	AMZN	AAPL	GOOGL	NVDA	MSFT	META	TSLA
2012-05-21	1.603590	1.992053	5.826039	2.283479	1.738396	1.639910	-10.986146	4.390441
2012-05-22	1.653004	0.692077	5.013443	0.066621	0.496683	1.674092	-18.911854	11.756175
2012-05-23	1.825174	1.603930	7.575761	1.508990	2.980130	-0.546612	-16.296107	12.554454
2012-05-24	1.965696	0.649988	6.587755	0.542975	0.248325	-0.683333	-13.601889	9.869415
2012-05-25	1.744875	-0.448916	6.016468	-1.477351	2.649026	-0.717466	-16.531523	8.164013
...	...	...	...	...	...	...	...	...
2024-06-14	319.357346	1617.652576	1226.845082	1077.987319	47509.683193	1784.391591	1221.464774	9588.499472
2024-06-17	322.571463	1621.393459	1252.946088	1080.985841	47184.773054	1809.087002	1227.938948	10101.743838
2024-06-18	323.636904	1609.703022	1238.084693	1066.659897	48845.410188	1800.443595	1209.224100	9961.322811
2024-06-20	322.566826	1640.472332	1209.298602	1074.722415	47112.572637	1797.718646	1215.016837	9782.259279
2024-06-21	321.906722	1668.342294	1195.623730	1096.910888	45592.730999	1815.090570	1196.878639	9860.633053

3041 rows × 8 columns

With the cumulative returns ready, we can use Python's Matplotlib library to display them all on the same chart. We will define a function called plot_returns() which takes the DataFrame as an argument, in case we want to make more plots later.

Figure 20

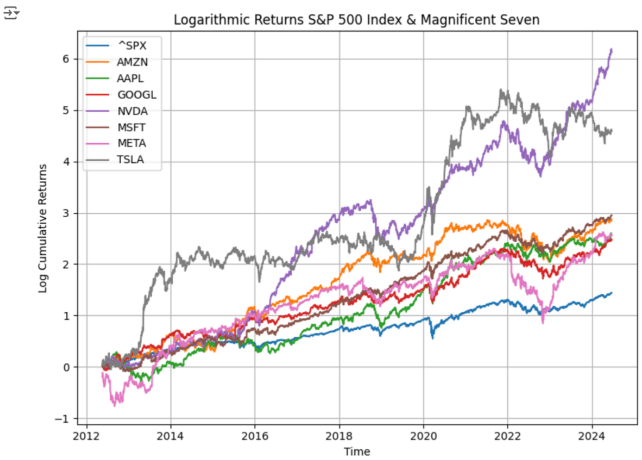
```
[ ] import matplotlib
    from matplotlib import pyplot as plt

    def plot_returns(return_df, title, x_axis_label, y_axis_label):
      plt.figure(figsize=(8, 6))  # Set the figure size (width, height)

      # Plot each column as a separate series
      for col in return_df.columns:
          plt.plot(return_df.index, return_df[col], label=col)

      # Customize the plot
      plt.title(title)
      plt.xlabel(x_axis_label)
      plt.ylabel(y_axis_label)
      plt.legend()  # Show legend based on 'label' in plt.plot()
      plt.grid()  # Show grid
      plt.tight_layout()  # Adjust layout to fit the figure area

      # Display the plot
      plt.show()

    plot_returns(cumulative_returns_pct_df,'Cumulative Returns S&P 500 Index & Magnificent Seven', 'Time', 'Return(%)')
```



Due to the standout performance of Nvidia Corp stock, which beat the S&P 500 Index by a factor of over 100 during the period, the scale of the chart is thrown off and does not adequately capture the relative movement of the stocks in periods of market turmoil (like in 2022). With Python, we can easily apply a logarithmic transformation by leveraging the NumPy library to better visualize the relative performance.

Figure 21

```
[ ] import numpy as np

    log_cumul_returns_df = np.log(1+ cumulative_returns_df)
    plot_returns(log_cumul_returns_df, "Logarithmic Returns S&P 500 Index & Magnificent Seven", "Time", "Log Cumulative Returns")
```



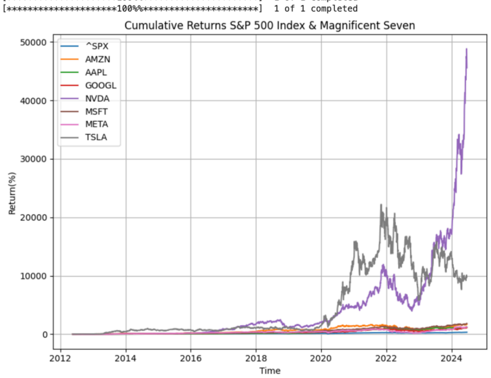
Below presents the entire program in a concise Python script, with minimal comments. We have added an export_results() function to illustrate how one might export the data to Excel, though this can be easily substituted for other types of storage, such as an SQL database. Then, the process can be scheduled to run at regular intervals via Task Scheduler (Windows) or Cron Job (Mac/Linux).

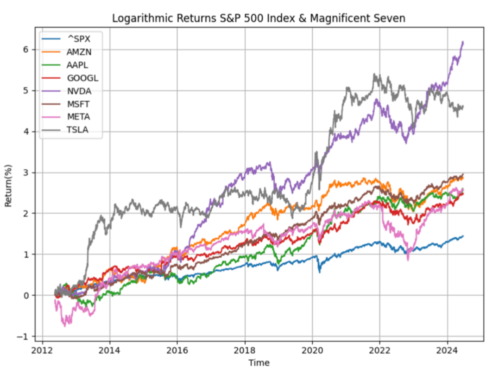
Thanks to Python's simple syntax and rich libraries, we just performed an entire ETL, plus basic visualization, in about 50 lines. Not bad!

Figure 22

```
[ ] import yfinance as yf
    import pandas as pd
    import numpy as np
    import matplotlib
    from matplotlib import pyplot as plt

    def get_closing_prices_df(tickers):

      price_hist_mag_seven_df = pd.DataFrame()

      for ticker in tickers:
        adj_close_hist = yf.download(ticker, "1928-01-01" , "2024-12-31")["Adj Close"] #pull historical data, taking only "Adj Close"
        price_hist_mag_seven_df[ticker] = adj_close_hist #add new column for each ticker to the dataframe
      return price_hist_mag_seven_df.dropna()

    def build_cumul_pct_log_cumul_dfs(close_price_df):
      dod_return_mag_seven_df = close_price_df.pct_change().dropna() #find the percent change and drop NaN (from the first row)
      cumulative_returns_df = (1 + dod_return_mag_seven_df).cumprod()-1 #Use vectorized operations, df.cumprod(), df + 1/-1 to quickly find cumulative return

      cumulative_returns_pct_df = cumulative_returns_df*100
      log_cumul_returns_df = np.log(1+ cumulative_returns_df) #Leverage Numpy library to apply log transformation

      return cumulative_returns_pct_df, log_cumul_returns_df

    def plot_returns(return_df, title, x_axis_label, y_axis_label): #plot returns
      plt.figure(figsize=(8, 6))

      for col in return_df.columns:
          plt.plot(return_df.index, return_df[col], label=col)

      plt.title(title)
      plt.xlabel(x_axis_label)
      plt.ylabel(y_axis_label)
      plt.legend()
      plt.grid()
      plt.tight_layout()
      plt.savefig("my_file_path/" + title +".pdf") #Changed to output to .pdf, though matplotlib accepts many types like .png, .jpg

    def export_results(raw_price_df, cumul_ret_df, log_cumul_df):

      with pd.ExcelWriter("my_file_path/mag_seven_spx.xlsx") as writer: #outputs the dataframes to an excel file called "mag_seven_spx.xlsx"
        raw_price_df.to_excel(writer, sheet_name="Raw_Prices")
        cumul_ret_df.to_excel(writer, sheet_name="Cumul_Ret_Pct")
        log_cumul_df.to_excel(writer, sheet_name="Log_Cumul_Ret")

      #plot the returns and save as .pdf
      plot_returns(cumul_ret_df, 'Cumulative Returns S&P 500 Index & Magnificent Seven', 'Time', 'Return(%)')
      plot_returns(log_cumul_df, 'Logarithmic Returns S&P 500 Index & Magnificent Seven', 'Time', 'Return(%)')

    def main():
      tickers = ["^SPX", "AMZN", "AAPL", "GOOGL", "NVDA", "MSFT", "META", "TSLA"]
      raw_price_df = get_closing_prices_df(tickers)
      cumul_ret_df, log_cumul_df = build_cumul_pct_log_cumul_dfs(raw_price_df)
      export_results(raw_price_df, cumul_ret_df, log_cumul_df)

    if __name__ == "__main__": #ensure main() only runs when the script is executed directly, not when it is imported
      main()
```

```
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
```





Winfred Lam, CFA, is a Manager at BMO Corporate Treasury responsible for Balance Sheet Management. He holds a Master's degree in Computer Science from the University of Pennsylvania and is a volunteer member of CFA Society Toronto's Member Communication Committee.