Course 2 Project: ML Pipeline for Feature Engineering

Instructions

In this project, you'll use data related to microeconomic indicators and historical stock prices to explore the data engineering pipline. You'll get to practice:

- Data ingestion
- Data cleaning
- Data imputation
- Exploratory data analysis (EDA) through charts and graphs

Packages

You'll use `pandas` and `matplotlib`, which were covered in the course material, to import, clean, and plot data. They have been installed in this workspace for you. If you're working locally and you installed Jupyter using Anaconda, these packages will already be installed.

In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Load data

The first step in a data engineering pipeline for machine learning is to ingest the data that will be used. For this project, data is hosted on a public GitHub repo.

Your tasks:

- Import data from the provided GitHub repo using `pandas`
- Verify that the data has been imported correctly into `pandas` dataframes. Use methods like `head()` and `info()`
- You may need to change column names to make them easier to work with
- You may need to cast datetime data to the `datetime` format using `pandas` `to_datetime()` method

Data files to import:

1. GDP

2. Inflation
3. Apple stock prices
4. Microsoft stock prices

In [2]:
```
gdp_url = 'https://raw.githubusercontent.com/udacity/CD13649-Project/refs/he
#Load historical GDP data
gdp = pd.read_csv(gdp_url, on_bad_lines='skip')
inflation_url = 'https://raw.githubusercontent.com/udacity/CD13649-Project/r
inflation = pd.read_csv(inflation_url, on_bad_lines='skip')
inflation.head()
```

Out[2]:

	DATE	CORESTICKM159SFRBATL
0	1968-01-01	3.651861
1	1968-02-01	3.673819
2	1968-03-01	4.142164
3	1968-04-01	4.155828
4	1968-05-01	4.088245

In [3]:
```
# Check the first few rows of data
gdp.head()
```

Out[3]:

	DATE	GDP
0	1947-01-01	243.164
1	1947-04-01	245.968
2	1947-07-01	249.585
3	1947-10-01	259.745
4	1948-01-01	265.742

In [4]:
```
# Load the historical stock price data for Apple and Microsoft
apple_historical_data_url = 'https://raw.githubusercontent.com/udacity/CD136
apple = pd.read_csv(apple_historical_data_url, on_bad_lines='skip')
apple.head()
```

Out[4]:

	Date	Close/Last	Volume	Open	High	Low
0	5/3/2024	$183.38	163224100	$186.65	$187.00	$182.66
1	5/2/2024	$173.03	94214920	$172.51	$173.42	$170.89
2	5/1/2024	$169.30	50383150	$169.58	$172.71	$169.11
3	4/30/2024	$170.33	65934780	$173.33	$174.99	$170.00
4	4/29/2024	$173.50	68169420	$173.37	$176.03	$173.10

In [5]:
```
# Check the first few rows of data
```

```
msft_historical_data_url = 'https://raw.githubusercontent.com/udacity/CD1364
microsoft = pd.read_csv(msft_historical_data_url, on_bad_lines='skip')
microsoft.head()
```

Out[5]:

	Date	Close/Last	Volume	Open	High	Low
0	05/03/2024	$406.66	17446720	$402.28	$407.15	$401.86
1	05/02/2024	$397.84	17709360	$397.66	$399.93	$394.6515
2	05/01/2024	$394.94	23562480	$392.61	$401.7199	$390.31
3	04/30/2024	$389.33	28781370	$401.49	$402.16	$389.17
4	04/29/2024	$402.25	19582090	$405.25	$406.32	$399.19

In [6]:
```
# Use methods like .info() and .describe() to explore the data
apple.info()
apple.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2517 entries, 0 to 2516
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Date        2517 non-null   object
 1   Close/Last  2514 non-null   object
 2   Volume      2517 non-null   int64
 3   Open        2517 non-null   object
 4   High        2517 non-null   object
 5   Low         2517 non-null   object
dtypes: int64(1), object(5)
memory usage: 118.1+ KB
```

Out[6]:

	Volume
count	2.517000e+03
mean	1.277394e+08
std	7.357405e+07
min	2.404834e+07
25%	7.741776e+07
50%	1.077601e+08
75%	1.567789e+08
max	7.576780e+08

In [7]:
```
microsoft.info()
microsoft.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2517 entries, 0 to 2516
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype 
---  ------      --------------  ----- 
 0   Date        2517 non-null   object
 1   Close/Last  2517 non-null   object
 2   Volume      2517 non-null   int64 
 3   Open        2517 non-null   object
 4   High        2517 non-null   object
 5   Low         2517 non-null   object
dtypes: int64(1), object(5)
memory usage: 118.1+ KB
```

Out[7]:

	Volume
count	2.517000e+03
mean	2.953106e+07
std	1.370138e+07
min	7.425603e+06
25%	2.131892e+07
50%	2.639470e+07
75%	3.360003e+07
max	2.025141e+08

In [8]:
```
inflation.info()
inflation.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 675 entries, 0 to 674
Data columns (total 2 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   DATE                  675 non-null    object 
 1   CORESTICKM159SFRBATL  675 non-null    float64
dtypes: float64(1), object(1)
memory usage: 10.7+ KB
```

Out[8]:

	CORESTICKM159SFRBATL
count	675.000000
mean	4.331276
std	2.694022
min	0.663868
25%	2.453373
50%	3.354398
75%	5.202000
max	15.774167

Data preprocessing: Check for missing data and forward fill

Check the Apple historical prices for missing data. Check for missing data in all columns. If there's data missing, use a forward fill to fill in those missing prices.

In [9]:
```
# Check for nulls
md_a = apple.isnull().sum()
md_m = microsoft.isnull().sum()
md_g = gdp.isnull().sum()
md_i = inflation.isnull().sum()
print("Missing data per column:\n")
print(md_a)
print(md_m)
print(md_g)
print(md_i)
```

```
Missing data per column:

Date          0
Close/Last    3
Volume        0
Open          0
High          0
Low           0
dtype: int64
Date          0
Close/Last    0
Volume        0
Open          0
High          0
Low           0
dtype: int64
DATE    0
GDP     0
dtype: int64
DATE                    0
CORESTICKM159SFRBATL    0
dtype: int64
```

In [10]:
```
# Forward fill any missing data
apple = apple.ffill()
microsoft = microsoft.ffill()
gdp = gdp.ffill()
inflation = inflation.ffill()
```

In [11]:
```
# Check again for nulls after using forward fill
print("Missing data after forward fill:\n")
print(apple.isnull().sum())
print(microsoft.isnull().sum())
print(gdp.isnull().sum())
print(inflation.isnull().sum())
```

```
Missing data after forward fill:

Date          0
Close/Last    0
Volume        0
Open          0
High          0
Low           0
dtype: int64
Date          0
Close/Last    0
Volume        0
Open          0
High          0
Low           0
dtype: int64
DATE    0
GDP     0
dtype: int64
DATE                    0
CORESTICKM159SFRBATL    0
dtype: int64
```

Data preprocessing: Remove special characters and convert to numeric/datetime

The next step in the data engineering process is to standardize and clean up data. In this step, you'll check for odd formatting and special characters that will make it difficult to work with data as numeric or datetime.

In this step:

- Create a function that takes in a dataframe and a list of columns and removes dollar signs ('$') from those columns
- Convert any columns with date/time data into a `pandas datetime` format

In [12]:
```
def convert_dollar_columns_to_numeric(df, numeric_columns):
    '''
        Removes dollar signs ('$') from a list of columns in a given datafra
        Updates dataframe IN PLACE.

        Inputs:
            df: dataframe to be operated on
            numeric_columns: columns that should have numeric data but have

        Returns:
            None - changes to the dataframe can be made in place
    '''
    for column in numeric_columns:
        df[column] = df[column].replace({'\$': '', ',': ''}, regex=True)
        df[column] = pd.to_numeric(df[column], errors='coerce')
    return None
```

```
<>:14: SyntaxWarning: invalid escape sequence '\$'
<>:14: SyntaxWarning: invalid escape sequence '\$'
/var/folders/gx/x4w_yhkx4xddw8kdmqf3m5zw0000gn/T/ipykernel_27687/357849168
3.py:14: SyntaxWarning: invalid escape sequence '\$'
  df[column] = df[column].replace({'\$': '', ',': ''}, regex=True)
```

In [13]:
```
# Use convert_dollar_columns_to_numeric() to remove the dollar sign from the
convert_dollar_columns_to_numeric(apple, ['Close/Last', 'Open', 'High', 'Low
convert_dollar_columns_to_numeric(microsoft, ['Close/Last', 'Open', 'High',
```

In [14]:
```
# Use pandas's to_datetime() to convert any columns that are in a datetime f
apple['Date'] = pd.to_datetime(apple['Date'])
microsoft['Date'] = pd.to_datetime(microsoft['Date'])
inflation['DATE'] = pd.to_datetime(inflation['DATE'])
```

In [15]:
```
# Use .info() and check the type of each column to ensure that the above ste
apple.info()
microsoft.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2517 entries, 0 to 2516
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Date        2517 non-null   datetime64[ns]
 1   Close/Last  2517 non-null   float64
 2   Volume      2517 non-null   int64
 3   Open        2517 non-null   float64
 4   High        2517 non-null   float64
 5   Low         2517 non-null   float64
dtypes: datetime64[ns](1), float64(4), int64(1)
memory usage: 118.1 KB
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2517 entries, 0 to 2516
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Date        2517 non-null   datetime64[ns]
 1   Close/Last  2517 non-null   float64
 2   Volume      2517 non-null   int64
 3   Open        2517 non-null   float64
 4   High        2517 non-null   float64
 5   Low         2517 non-null   float64
dtypes: datetime64[ns](1), float64(4), int64(1)
memory usage: 118.1 KB
```

In [16]:
```
apple.head()
```

Out[16]:

	Date	Close/Last	Volume	Open	High	Low
0	2024-05-03	183.38	163224100	186.65	187.00	182.66
1	2024-05-02	173.03	94214920	172.51	173.42	170.89
2	2024-05-01	169.30	50383150	169.58	172.71	169.11
3	2024-04-30	170.33	65934780	173.33	174.99	170.00
4	2024-04-29	173.50	68169420	173.37	176.03	173.10

In [17]:
```
microsoft.head()
```

Out[17]:

	Date	Close/Last	Volume	Open	High	Low
0	2024-05-03	406.66	17446720	402.28	407.1500	401.8600
1	2024-05-02	397.84	17709360	397.66	399.9300	394.6515
2	2024-05-01	394.94	23562480	392.61	401.7199	390.3100
3	2024-04-30	389.33	28781370	401.49	402.1600	389.1700
4	2024-04-29	402.25	19582090	405.25	406.3200	399.1900

Data preprocessing: Align datetime data

Data engineering includes changing data with a datetime component if needed so that different time series can be more easily compared or plotted against each other.

In this step:

- Align the inflation date so that it falls on the last day of the month instead of the first

Helpful hints:

- Use the `pandas offsets` method using `MonthEnd(0)` to set the 'Date' column to month-end

In [18]:
```
# Align inflation data so that the date is the month end (e.g. Jan 31, Feb 2
from pandas.tseries.offsets import MonthEnd
inflation['DATE'] = inflation['DATE'] + MonthEnd(0)
```

Data preprocessing: Upsample, downsample and interpolate data

Inflation data is presented monthly in this dataset. However, for some models, you may need it at a quarterly frequency, and for some models you may need it at a weekly frequency.

In this step:

- Create a new quarterly inflation dataframe by downsampling the monthly inflation data to quarterly using the mean (e.g. for quarter 1 in a given year, use the average values from January, February, and March)
- Create a new weekly inflation dataframe by upsampling the monthly inflation data. For this, you'll need to use `resample` and then you'll need to `interpolate` to fill in the missing data at the weekly frequency

Note that you may need to change the index for some of these operations!

In [19]:
```
inflation.set_index('DATE', inplace=True)
apple.set_index('Date', inplace=True)
microsoft.set_index('Date', inplace=True)
```

In [20]:
```
# Upsample and interpolate from monthly to weekly
weekly_inflation = inflation.resample('W').interpolate()
weekly_inflation.head(20)
```

Out[20]:

DATE	CORESTICKM159SFRBATL
1968-02-04	NaN
1968-02-11	NaN
1968-02-18	NaN
1968-02-25	NaN
1968-03-03	NaN
1968-03-10	NaN
1968-03-17	NaN
1968-03-24	NaN
1968-03-31	4.142164
1968-04-07	4.173195
1968-04-14	4.204226
1968-04-21	4.235258
1968-04-28	4.266289
1968-05-05	4.297320
1968-05-12	4.328351
1968-05-19	4.359382
1968-05-26	4.390413
1968-06-02	4.421445
1968-06-09	4.452476
1968-06-16	4.483507

In [21]:
```
# Downsample from monthly to quarterly
quarterly_inflation = inflation.resample('QE').mean()
```

Data preprocessing: Normalize/standardize a feature

Economic time series data often involve variables measured on different scales (e.g., GDP in trillions of dollars, inflation in percentage points). Standardizing these variables (typically by subtracting the mean and dividing by the standard deviation) puts them on a common scale, allowing for meaningful comparisons and analyses.

Your task:

- Standardize the GDP data. You may do this manually by subtracting the mean and dividing by the standard deviation, or you may use a built-in method from a library

like `sklearn's StandardScaler`

In [22]:
```
# Standardize the GDP measure
! pip install scikit-learn
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# Fit and transform the 'GDP' data
gdp['GDP_standardize'] = scaler.fit_transform(gdp[['GDP']])
```

```
Requirement already satisfied: scikit-learn in /Users/ishanklal/miniconda3/e
nvs/aitsnd/lib/python3.12/site-packages (1.6.1)
Requirement already satisfied: numpy>=1.19.5 in /Users/ishanklal/miniconda3/
envs/aitsnd/lib/python3.12/site-packages (from scikit-learn) (1.26.4)
Requirement already satisfied: scipy>=1.6.0 in /Users/ishanklal/miniconda3/e
nvs/aitsnd/lib/python3.12/site-packages (from scikit-learn) (1.15.2)
Requirement already satisfied: joblib>=1.2.0 in /Users/ishanklal/miniconda3/
envs/aitsnd/lib/python3.12/site-packages (from scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /Users/ishanklal/mini
conda3/envs/aitsnd/lib/python3.12/site-packages (from scikit-learn) (3.6.0)
```

In [23]:
```
# Check the dataframe to make sure the calculation worked as expected
gdp.head()
```

Out[23]:

	DATE	GDP	GDP_standardize
0	1947-01-01	243.164	-0.935496
1	1947-04-01	245.968	-0.935121
2	1947-07-01	249.585	-0.934636
3	1947-10-01	259.745	-0.933276
4	1948-01-01	265.742	-0.932472

EDA: Plotting a time series of adjusted open vs close price

As part of your EDA, you'll frequently want to plot two time series on the same graph and using the same axis to compare their movements.

Your task:

- Plot the Apple open and close price time series on the same chart **for the last three months only**. Be sure to use a legend to label each line

NOTE: This is a large dataset. If you try to plot the entire series, your graph will be hard to interpret and may take a long time to plot. Be sure to use only the most recent three months of data.

In [24]:
```
# Get max date in timeseries
max_date = apple.index.max()
```

```
three_months_ago = max_date - pd.DateOffset(months=3)
three_months_ago
```

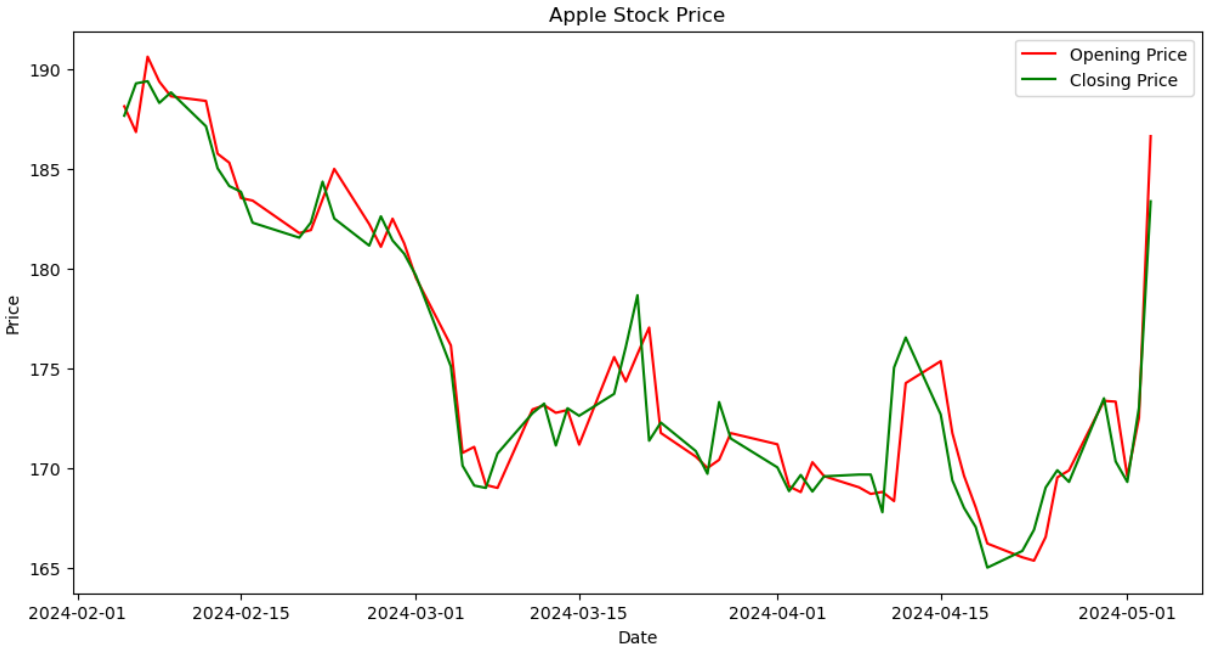
Out[24]: Timestamp('2024-02-03 00:00:00')

In [25]:
```
# Use the max date calculated above to get the last three months of data in
last_3_months_apple = apple[ apple.index >= three_months_ago ]
last_3_months_apple.head()
```

Out[25]:

Date	Close/Last	Volume	Open	High	Low
2024-05-03	183.38	163224100	186.65	187.00	182.66
2024-05-02	173.03	94214920	172.51	173.42	170.89
2024-05-01	169.30	50383150	169.58	172.71	169.11
2024-04-30	170.33	65934780	173.33	174.99	170.00
2024-04-29	173.50	68169420	173.37	176.03	173.10

In [26]:
```
# Plot time series of open v. close stock price for Apple using the last 3 m
plt.figure(figsize=(12,6))
plt.plot(last_3_months_apple['Open'], label='Opening Price', color='red')
plt.plot(last_3_months_apple['Close/Last'], label='Closing Price', color='gr
plt.xlabel('Date')
plt.ylabel('Price')
plt.title('Apple Stock Price')
plt.legend()
plt.show()
```



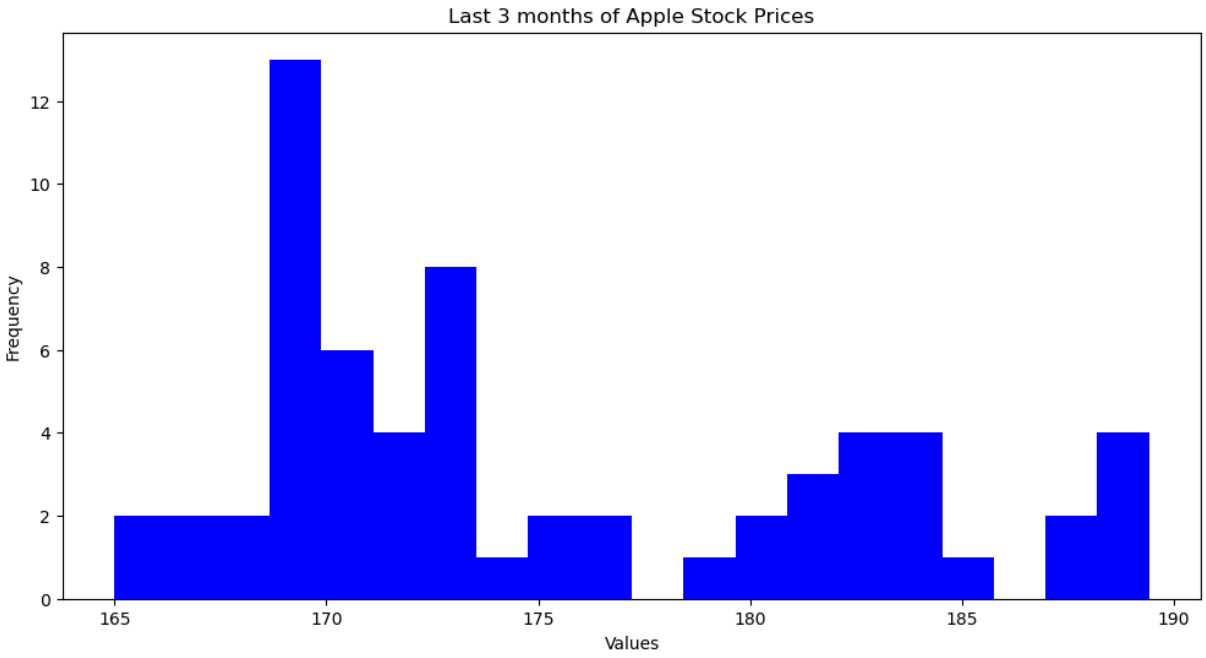
EDA: Plotting a histogram of a stock's closing price in the last three months

One way to see how much a stock's price generally moves is to plot the frequency of closing prices over a set time period.

Your task:

- Use the **last three months** of Apple stock data and plot a histogram of closing price

In [27]:
```
# Plot the histogram of Apple's closing price over the last 3 months
plt.figure(figsize=(12,6))
plt.hist(last_3_months_apple['Close/Last'], bins=20, color='blue')
plt.title('Last 3 months of Apple Stock Prices')
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.show()
```



Calculating correlation between a stock price and a macroeconomic variable

Inflation affects the purchasing power of money and can influence corporate profits, interest rates, and consumer behavior. By analyzing the correlation between stock prices and inflation, one can gauge how inflationary trends impact stock market performance. For instance, high inflation might erode profit margins and reduce stock prices, while moderate inflation might indicate a growing economy, benefiting stocks.

Your task:

- Plot a heatmap that shows the correlation between Microsoft and Apple returns and inflation

This will require several steps:

1. Calculate the returns for Apple and Microsoft and the change in monthly inflation (use the `pct_change` method for each)
2. Interpolate the daily stock returns data to monthly so it can be compared to the monthly inflation data
3. Merge the stock returns (Apple and Microsoft) and inflation data series into a single dataframe
4. Calculate the correlation matrix between the Apple returns, Microsoft returns, and inflation change
5. Plot the correlation matrix as a heatmap

1. Calculate returns for Microsoft / Apple and the monthly change in inflation

In [28]:
```
# Calculate daily returns for Apple and Microsoft and the percent change in
#fix data order before pct_change()
apple = apple.sort_index(ascending=True)
microsoft = microsoft.sort_index(ascending=True)
apple['daily_returns'] = apple['Close/Last'].pct_change()
microsoft['daily_returns'] = microsoft['Close/Last'].pct_change()
inflation['monthly_change'] = inflation['CORESTICKM159SFRBATL'].pct_change()
```

2. Interpolate stock returns from daily to monthly

In [29]:
```
apple_monthly_returns = (1 + apple['daily_returns']).resample('ME').prod() -
microsoft_monthly_returns = (1 + microsoft['daily_returns']).resample('ME').
apple_monthly_returns.name = 'apple_monthly_returns'
microsoft_monthly_returns.name = 'microsoft_monthly_returns'
```

3. Merge the dataframes and calculate / plot the correlation

In [30]:
```
merged_stock = pd.merge(apple_monthly_returns, microsoft_monthly_returns, le
merged_df = pd.merge(merged_stock, inflation['monthly_change'], left_index=T
merged_df.rename(columns={'monthly_change': 'Inflation_Change'}, inplace=Tru
merged_df.dropna(inplace=True)
merged_df
```

Out[30]:

Date	apple_monthly_returns	microsoft_monthly_returns	Inflation_Change
2014-05-31	0.065002	0.048131	0.050251
2014-06-30	0.027421	0.018564	0.007147
2014-07-31	0.028842	0.035012	-0.020724
2014-08-31	0.072385	0.052595	-0.066387
2014-09-30	-0.017167	0.020471	-0.004516
...	...	...	...
2023-11-30	0.112315	0.120671	-0.039860
2023-12-31	0.013583	-0.007574	-0.028560
2024-01-31	-0.042227	0.057281	0.010874
2024-02-29	-0.019794	0.040394	-0.043619
2024-03-31	-0.051286	0.017116	0.024099

119 rows × 3 columns

4. Calculate the correlation matrix between the Apple returns, Microsoft returns, and inflation change

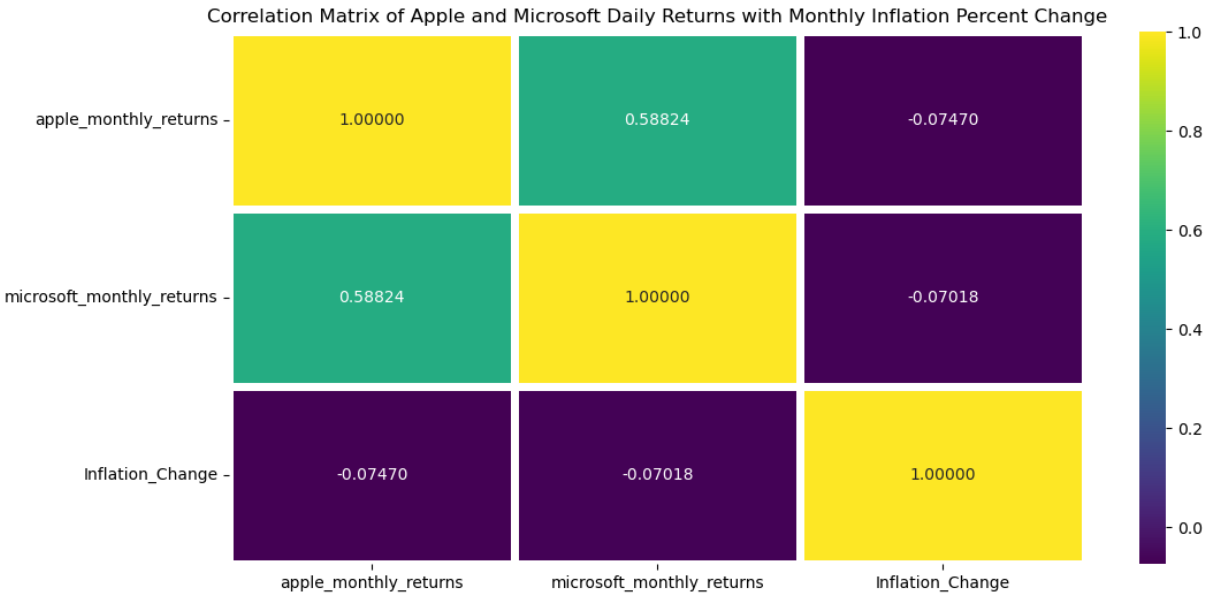
In [31]:
```
# Calculate correlation matrix
correlation_matrix = merged_df[['apple_monthly_returns', 'microsoft_monthly_
print(correlation_matrix)
```

```
                          apple_monthly_returns  microsoft_monthly_returns
\
apple_monthly_returns                  1.000000                   0.588237
microsoft_monthly_returns              0.588237                   1.000000
Inflation_Change                      -0.074699                  -0.070176

                          Inflation_Change
apple_monthly_returns            -0.074699
microsoft_monthly_returns        -0.070176
Inflation_Change                  1.000000
```

5. Plot the correlation matrix as a heatmap

In [32]:
```
# Plot heatmap
plt.figure(figsize=(12, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='viridis', fmt='.5f', linew
plt.title('Correlation Matrix of Apple and Microsoft Daily Returns with Mont
plt.show()
```



Calculating rolling volatility (standard deviation) of a stock's price for last 3 months

Volatility is a measure of the dispersion of returns for a given security. By calculating rolling volatility, investors can assess the risk associated with a stock over time: Higher volatility indicates higher risk, as the stock's price is more likely to experience significant fluctuations. In portfolio optimization, understanding the volatility of individual stocks and how it changes over time is crucial for diversification and optimization. By analyzing rolling volatility, investors can adjust their portfolios to maintain a desired risk level, potentially improving the risk-return profile.

One possible way to calculate volatility is by using the standard deviation of returns for a stock over time.

Your task:

- Calculate the weekly rolling standard deviation for Apple's closing price
- Plot the calculated rolling weekly volatility of Apple's closing price against Apple's closing price. Plot these **on the same chart, but using different y-axes**

Helpful hints:

- You'll need to use the `pandas rolling()` method with a given `window_size` parameter to make it a *weekly* rolling calculation
- Use **only the last three months of data**; data much older than this may not be as useful for portfolio optimization
- You'll need to create two axes on the matplotlib figure to be able to use two different y-axes (one for the closing price and one for the rolling volatility calculated here)

In [33]:
```
# Define the window size for the rolling calculation (e.g., one week)
window_size = 5
```

In [34]:
```
# Calculate rolling one-week volatility
apple['rolling_volatility'] = apple['daily_returns'].rolling(window=window_s
```

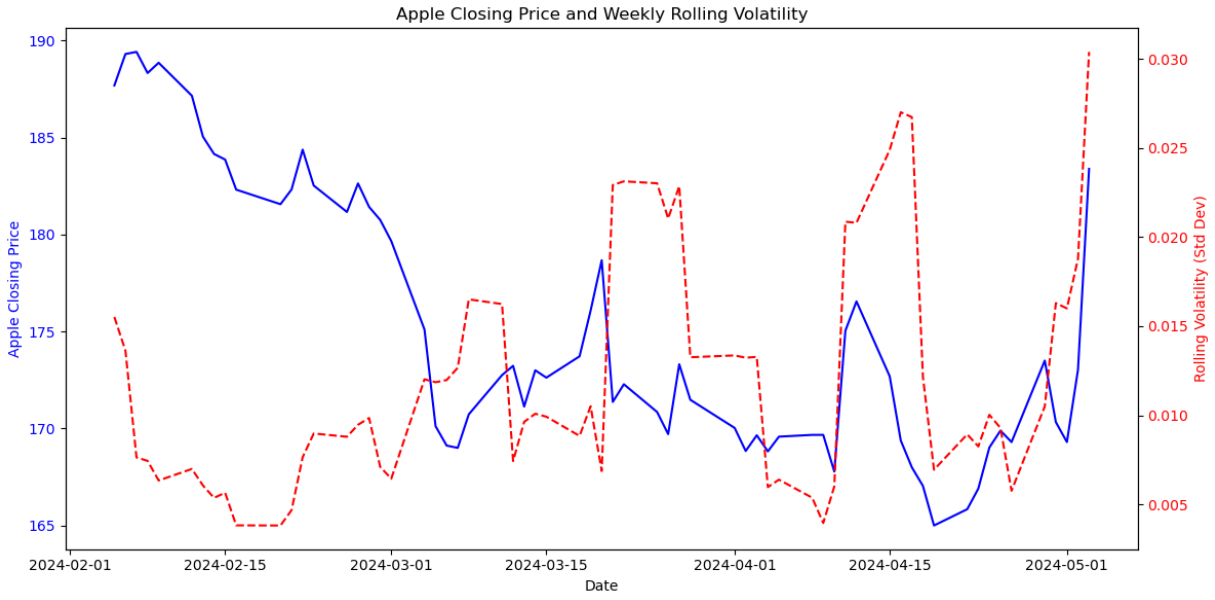
In [35]:
```
max_date = apple.index.max()
three_months_ago = max_date - pd.DateOffset(months=3)
last_3_months_rol_vol_apple = apple[ apple.index >= three_months_ago ]
```

In [36]:
```
# Plot the calculated rolling weekly volatility of Apple's closing price aga
# Plot these on the same chart, but using different y-axes
fig, ax1 = plt.subplots(figsize=(12, 6))
ax1.plot(last_3_months_rol_vol_apple.index, last_3_months_rol_vol_apple['Clo
ax1.set_xlabel('Date')
ax1.set_ylabel('Apple Closing Price', color='blue')
ax1.tick_params(axis='y', labelcolor='blue')

ax2 = ax1.twinx()
ax2.plot(last_3_months_rol_vol_apple.index, last_3_months_rol_vol_apple['rol
ax2.set_ylabel('Rolling Volatility (Std Dev)', color='red')
ax2.tick_params(axis='y', labelcolor='red')

plt.title('Apple Closing Price and Weekly Rolling Volatility')
fig.tight_layout()

plt.show()
```



Export data

Now that you have preprocessed your data, you should save it in new csv files so that it can be used in downstream tasks without having to redo all the preprocessing steps.

Your task:

- Use `pandas` to export all modified datasets back to new CSV files

In [37]:
```
gdp.to_csv('modified_gdp.csv', index=False)
inflation.to_csv('modified_inflation.csv', index=False)
apple.to_csv('modified_apple.csv', index=False)
microsoft.to_csv('modified_microsoft.csv', index=False)
weekly_inflation.to_csv('weekly_inflation_dataset.csv', index=False)
quarterly_inflation.to_csv('quarterly_inflation_dataset.csv', index=False)
last_3_months_apple.to_csv('last_3_months_apple_dataset.csv', index=False)
merged_df.to_csv('merged_stock_inflation_dataset.csv', index=False)
last_3_months_rol_vol_apple.to_csv('last_3_months_rol_vol_apple_dataset.csv'
apple_monthly_returns.to_csv('apple_monthly_returns_dataset.csv', index=Fals
microsoft_monthly_returns.to_csv('microsoft_monthly_returns_dataset.csv', in
```

In [ ]:

In [ ]:

In [ ]: