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 [4]:
```
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 [6]:
```
# Base URL for raw GitHub content
base_url = "https://raw.githubusercontent.com/udacity/CD13649-Project/main/P

# File paths
gdp_url = base_url + "GDP.csv"
inflation_url = base_url + "inflation_monthly.csv"
apple_url = base_url + "apple_historical_data.csv"
microsoft_url = base_url + "microsoft_historical_data.csv"

# Load data directly from GitHub
gdp = pd.read_csv(gdp_url)
inflation = pd.read_csv(inflation_url)
apple = pd.read_csv(apple_url)
microsoft = pd.read_csv(microsoft_url)

# Preview and inspect structure for each
print("GDP:")
display(gdp.head())
gdp.info()

print("\nInflation:")
display(inflation.head())
inflation.info()

print("\nApple:")
display(apple.head())
apple.info()

print("\nMicrosoft:")
display(microsoft.head())
microsoft.info()
```

GDP:

	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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 309 entries, 0 to 308
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   DATE    309 non-null    object 
 1   GDP     309 non-null    float64
dtypes: float64(1), object(1)
memory usage: 5.0+ KB

Inflation:
```

	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

```
<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

Apple:
```

	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

```
<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

Microsoft:
```

	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

```
<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
```

In [7]:

```
# Use methods like .info() and .describe() to explore the data

gdp.info()
gdp.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 309 entries, 0 to 308
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   DATE    309 non-null    object 
 1   GDP     309 non-null    float64
dtypes: float64(1), object(1)
memory usage: 5.0+ KB
```

Out[7]:

	GDP
count	309.000000
mean	7227.754935
std	7478.297734
min	243.164000
25%	804.981000
50%	4386.773000
75%	12527.214000
max	28284.498000

In [8]:

```
# continued

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

In [9]:

```
# continued

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[9]:

	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 [10]:

```
# continued

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[10]:

	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

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 [12]:
```
# Check for nulls

apple.isnull().sum() # -> there are 3 NULL values in Close/Last
```

Out[12]:
```
Date          0
Close/Last    3
Volume        0
Open          0
High          0
Low           0
dtype: int64
```

In [13]:
```
# Forward fill any missing data

apple.ffill(inplace=True)
```

In [14]:
```
# Check again for nulls after using forward fill

apple.isna().sum() # -> Success
```

Out[14]:
```
Date          0
Close/Last    0
Volume        0
Open          0
High          0
Low           0
dtype: int64
```

In [15]:
```
# Repeat same procedure for remaining datasets:
```

```
microsoft.isna().sum() # -> no missing data for msft
inflation.isna().sum() # -> no missing data for inflation
gdp.isna().sum() # -> no missing data for gdp
```

Out[15]:
```
DATE    0
GDP     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 [17]:
```
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 col in numeric_columns:
        df[col] = df[col].replace(r'[\$,]', '', regex=True).astype(float)

    return df
```

In [18]:
```
# Use convert_dollar_columns_to_numeric() to remove the dollar sign from the

price_columns = ['Open', 'High', 'Low', 'Close/Last']
apple = convert_dollar_columns_to_numeric(apple, price_columns)
microsoft = convert_dollar_columns_to_numeric(microsoft, price_columns)

display(apple.head()) # -> Success
display(microsoft.head()) # -> Success
```

	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

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

In [19]:
```
# Fixing bug in code - inflation should use 'DATE' field as index:

inflation.set_index('DATE', inplace=True)
```

In [20]:
```
# 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'])
gdp['DATE'] = pd.to_datetime(gdp['DATE'])
inflation.index = pd.to_datetime(inflation.index)
```

In [21]:
```
# Use .info() and check the type of each column to ensure that the above ste

apple.info() # -> success
microsoft.info() # -> success
gdp.info() # -> success
inflation.info() # -> success
```

```
<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
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 309 entries, 0 to 308
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   DATE    309 non-null    datetime64[ns]
 1   GDP     309 non-null    float64
dtypes: datetime64[ns](1), float64(1)
memory usage: 5.0 KB
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 675 entries, 1968-01-01 to 2024-03-01
Data columns (total 1 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   CORESTICKM159SFRBATL  675 non-null    float64
dtypes: float64(1)
memory usage: 10.5 KB
```

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 [23]:
```
# Align inflation data so that the date is the month end (e.g. Jan 31, Feb 2

inflation.index = inflation.index + pd.offsets.MonthEnd(0)
```

In [24]:
```
display(inflation.head())
display(inflation.index)
```

	CORESTICKM159SFRBATL
DATE	
1968-01-31	3.651861
1968-02-29	3.673819
1968-03-31	4.142164
1968-04-30	4.155828
1968-05-31	4.088245

```
DatetimeIndex(['1968-01-31', '1968-02-29', '1968-03-31', '1968-04-30',
               '1968-05-31', '1968-06-30', '1968-07-31', '1968-08-31',
               '1968-09-30', '1968-10-31',
               ...
               '2023-06-30', '2023-07-31', '2023-08-31', '2023-09-30',
               '2023-10-31', '2023-11-30', '2023-12-31', '2024-01-31',
               '2024-02-29', '2024-03-31'],
              dtype='datetime64[ns]', name='DATE', length=675, freq=None)
```

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 quarterly frequency.

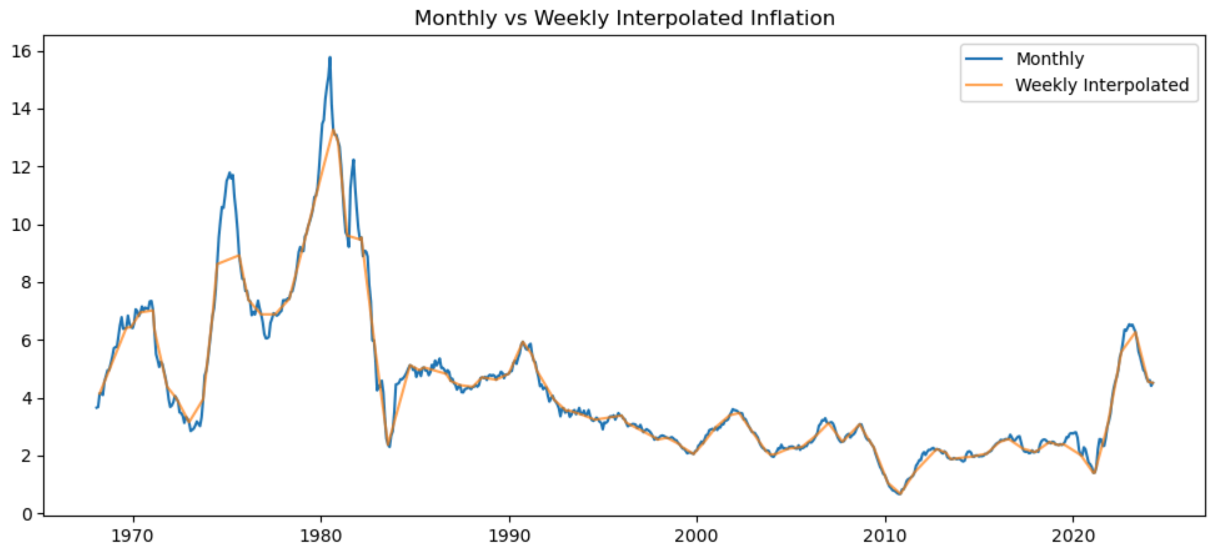
In this step:

- 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 [26]:
```
# Resample to weekly frequency

inflation_weekly = inflation.resample('W').interpolate(method='linear')
```

In [27]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 5))
plt.plot(inflation.index, inflation['CORESTICKM159SFRBATL'], label='Monthly'
plt.plot(inflation_weekly.index, inflation_weekly['CORESTICKM159SFRBATL'], l
plt.legend()
plt.title('Monthly vs Weekly Interpolated Inflation')
plt.show()
```



In [28]:
```
# Checking some stuff

print(inflation.columns)
display(inflation.head())
display(inflation.index)
```

```
Index(['CORESTICKM159SFRBATL'], dtype='object')
```

	CORESTICKM159SFRBATL
DATE	
1968-01-31	3.651861
1968-02-29	3.673819
1968-03-31	4.142164
1968-04-30	4.155828
1968-05-31	4.088245

```
DatetimeIndex(['1968-01-31', '1968-02-29', '1968-03-31', '1968-04-30',
               '1968-05-31', '1968-06-30', '1968-07-31', '1968-08-31',
               '1968-09-30', '1968-10-31',
               ...
               '2023-06-30', '2023-07-31', '2023-08-31', '2023-09-30',
               '2023-10-31', '2023-11-30', '2023-12-31', '2024-01-31',
               '2024-02-29', '2024-03-31'],
              dtype='datetime64[ns]', name='DATE', length=675, freq=None)
```

In [29]:

```
# Downsample from monthly to quarterly

inflation_quarterly = inflation.resample('QE').mean()
display(inflation_quarterly.head())
```

	CORESTICKM159SFRBATL
DATE	
1968-03-31	3.822615
1968-06-30	4.263214
1968-09-30	4.882643
1968-12-31	5.429443
1969-03-31	5.873770

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 [31]:

```
# Standardize the GDP measure

gdp['GDP_standardized'] = (gdp['GDP'].mean()) / gdp['GDP'].std()
```

In [32]:

```
# Check the dataframe to make sure the calculation worked as expected

display(gdp.head())
```

	DATE	GDP	GDP_standardized
0	1947-01-01	243.164	0.966497
1	1947-04-01	245.968	0.966497
2	1947-07-01	249.585	0.966497
3	1947-10-01	259.745	0.966497
4	1948-01-01	265.742	0.966497

In [33]:
```
# Now let's try to sklearn way

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
gdp['GDP_standardized'] = scaler.fit_transform(gdp[['GDP']])

display(gdp.head())
```

	DATE	GDP	GDP_standardized
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

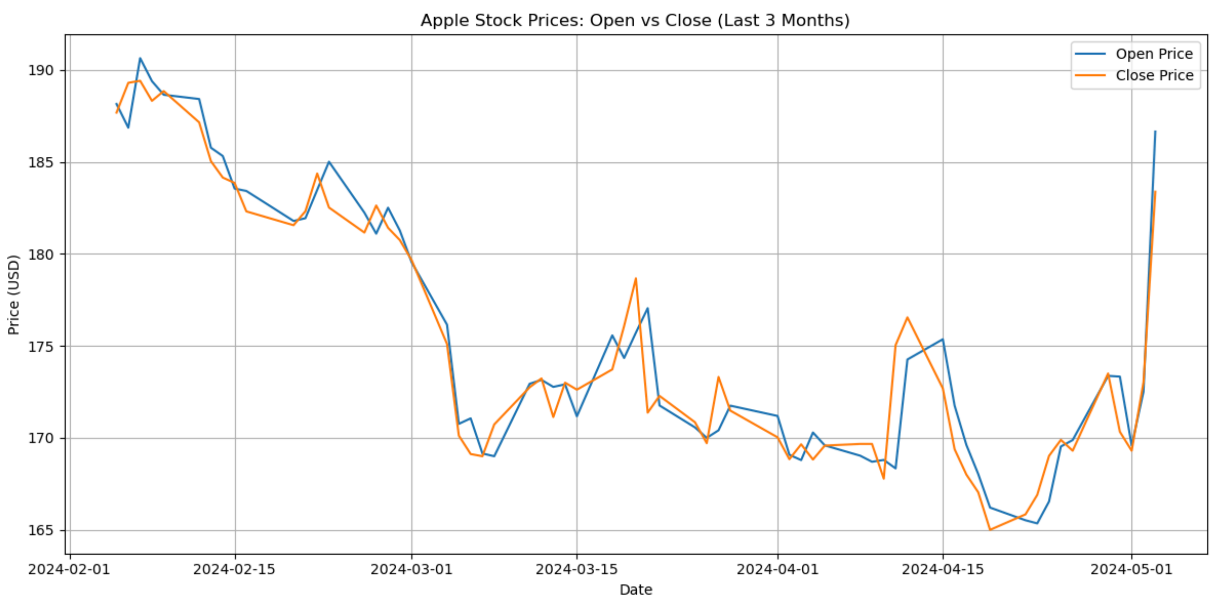
NOTE: This is a large dataset. If you try to plot the entire series, your graph will be hard to interpret and may take a long time to plot. Be sure to use only the most recent three months of data.

In [35]:
```
# Get max date in timeseries

# Can do in one swoop by calling max():
def get_last_3_months(df):
```

```
    last_3_months = df[df['Date'] >= apple['Date'].max() - pd.DateOffset(mor
    return last_3_months

apple_last_3_months = get_last_3_months(apple)
```

In [36]:
```
# Use the max date calculated above to get the last three months of data in

# Done above
```

In [37]:
```
# Plot time series of open v. close stock price for Apple using the last 3 m

plt.figure(figsize=(12, 6))
plt.plot(apple_last_3_months['Date'], apple_last_3_months['Open'], label='Op
plt.plot(apple_last_3_months['Date'], apple_last_3_months['Close/Last'], lab
plt.title('Apple Stock Prices: Open vs Close (Last 3 Months)')
plt.xlabel('Date')
plt.ylabel('Price (USD)')
plt.legend()
plt.grid(True)
plt.tight_layout()
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 [39]:
```
# Plot the histogram of Apple's closing price over the last 3 months
```

```
plt.figure(figsize=(10, 6))
plt.hist(apple_last_3_months['Close/Last'], bins=30, color='skyblue', edgecc
plt.title('Histogram of Apple Closing Prices (Last 3 Months)')
plt.xlabel('Closing Price (USD)')
plt.ylabel('Frequency')
plt.grid(True)
plt.tight_layout()
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 [42]:
```
# Calculate daily returns for Apple and Microsoft and the percent change in

# Sort data
apple.sort_values('Date', inplace=True)
microsoft.sort_values('Date', inplace=True)
inflation.sort_index(inplace=True)  # inflation['DATE'] was set as index ear

# Calculate returns and percent changes
apple['Daily Return'] = apple['Close/Last'].pct_change()
microsoft['Daily Return'] = microsoft['Close/Last'].pct_change()
inflation['Inflation Change'] = inflation['CORESTICKM159SFRBATL'].pct_change

# Show display
display(apple.head())
display(microsoft.head())
display(inflation.head())
```

	Date	Close/Last	Volume	Open	High	Low	Daily Return
2516	2014-05-06	21.23	373872650	21.49	21.59	21.23	NaN
2515	2014-05-07	21.15	282128727	21.26	21.33	20.99	-0.003768
2514	2014-05-08	21.00	228973884	21.01	21.23	20.94	-0.007092
2513	2014-05-09	20.91	291068564	20.88	20.94	20.73	-0.004286
2512	2014-05-12	21.17	212736019	20.98	21.20	20.98	0.012434

	Date	Close/Last	Volume	Open	High	Low	Daily Return
2516	2014-05-06	39.060	27105700	39.29	39.35	38.95	NaN
2515	2014-05-07	39.425	41731030	39.22	39.51	38.51	0.009345
2514	2014-05-08	39.640	32089010	39.34	39.90	38.97	0.005453
2513	2014-05-09	39.540	29646100	39.54	39.85	39.37	-0.002523
2512	2014-05-12	39.970	22761620	39.74	40.02	39.65	0.010875

	CORESTICKM159SFRBATL	Inflation Change
DATE		
1968-01-31	3.651861	NaN
1968-02-29	3.673819	0.006013
1968-03-31	4.142164	0.127482
1968-04-30	4.155828	0.003299
1968-05-31	4.088245	-0.016262

2. Interpolate stock returns from daily to monthly

In [44]:
```
display(apple.head())
```

	Date	Close/Last	Volume	Open	High	Low	Daily Return
2516	2014-05-06	21.23	373872650	21.49	21.59	21.23	NaN
2515	2014-05-07	21.15	282128727	21.26	21.33	20.99	-0.003768
2514	2014-05-08	21.00	228973884	21.01	21.23	20.94	-0.007092
2513	2014-05-09	20.91	291068564	20.88	20.94	20.73	-0.004286
2512	2014-05-12	21.17	212736019	20.98	21.20	20.98	0.012434

In [45]:
```
# Resample to monthly

# Ensure 'Date' is datetime
microsoft['Date'] = pd.to_datetime(microsoft['Date'])
apple['Date'] = pd.to_datetime(apple['Date'])

# Set as index
microsoft.set_index('Date', inplace=True)
apple.set_index('Date', inplace=True)
```

In [46]:
```
# Resample
microsoft_monthly_return = microsoft['Daily Return'].resample('M').apply(lam
apple_monthly_return = apple['Daily Return'].resample('M').apply(lambda r: (
inflation_monthly_change = inflation['Inflation Change'].resample('M').apply
```

```
/var/folders/v4/sm00xvsd7l3cn1whnf5h88ww0000gn/T/ipykernel_22237/722813924.p
y:2: FutureWarning: 'M' is deprecated and will be removed in a future versio
n, please use 'ME' instead.
  microsoft_monthly_return = microsoft['Daily Return'].resample('M').apply(l
ambda r: (1 + r).prod() - 1)
/var/folders/v4/sm00xvsd7l3cn1whnf5h88ww0000gn/T/ipykernel_22237/722813924.p
y:3: FutureWarning: 'M' is deprecated and will be removed in a future versio
n, please use 'ME' instead.
  apple_monthly_return = apple['Daily Return'].resample('M').apply(lambda r:
(1 + r).prod() - 1)
/var/folders/v4/sm00xvsd7l3cn1whnf5h88ww0000gn/T/ipykernel_22237/722813924.p
y:4: FutureWarning: 'M' is deprecated and will be removed in a future versio
n, please use 'ME' instead.
  inflation_monthly_change = inflation['Inflation Change'].resample('M').app
ly(lambda r: (1 + r).prod() - 1)
```

3. Merge the dataframes and calculate / plot the correlation

In [48]:

```python
# Combine all into a single DataFrame

combined = pd.concat([
    apple_monthly_return.rename('Apple Monthly Return'),
    microsoft_monthly_return.rename('Microsoft Monthly Return'),
    inflation_monthly_change.rename('Monthly Inflation Change')
], axis=1)

# Drop rows with any missing values
combined.dropna(inplace=True)
```

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

In [50]:

```python
# Calculate correlation matrix

correlation = combined.corr()
print(correlation)
```

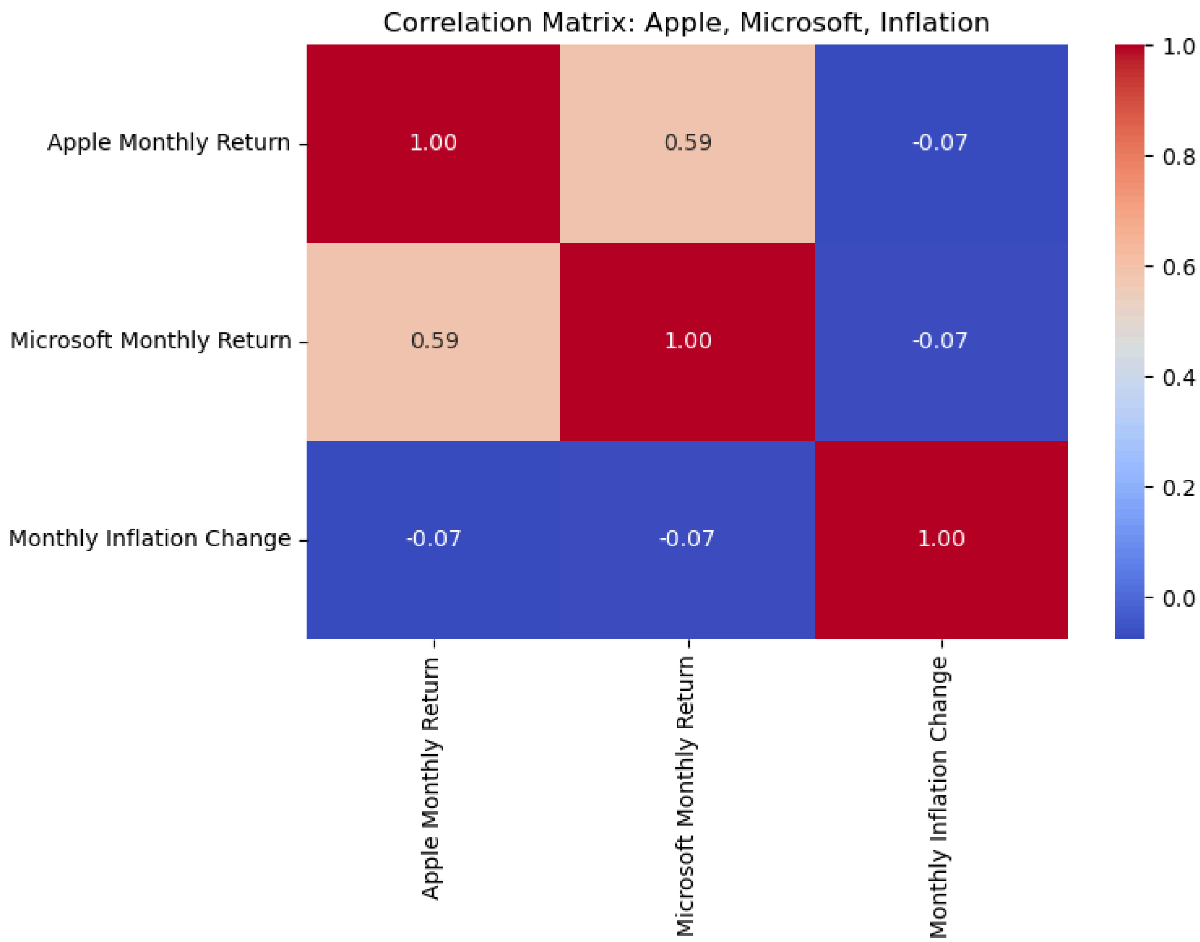
```
                          Apple Monthly Return  Microsoft Monthly Return  \
Apple Monthly Return                  1.000000                  0.588237   
Microsoft Monthly Return              0.588237                  1.000000   
Monthly Inflation Change             -0.074699                 -0.070176   

                          Monthly Inflation Change  
Apple Monthly Return                     -0.074699  
Microsoft Monthly Return                 -0.070176  
Monthly Inflation Change                  1.000000  
```

5. Plot the correlation matrix as a heatmap

In [52]:

```python
# Plot heatmap

import seaborn as sns
import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(8, 6))
sns.heatmap(correlation, annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Correlation Matrix: Apple, Microsoft, Inflation")
plt.tight_layout()
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.

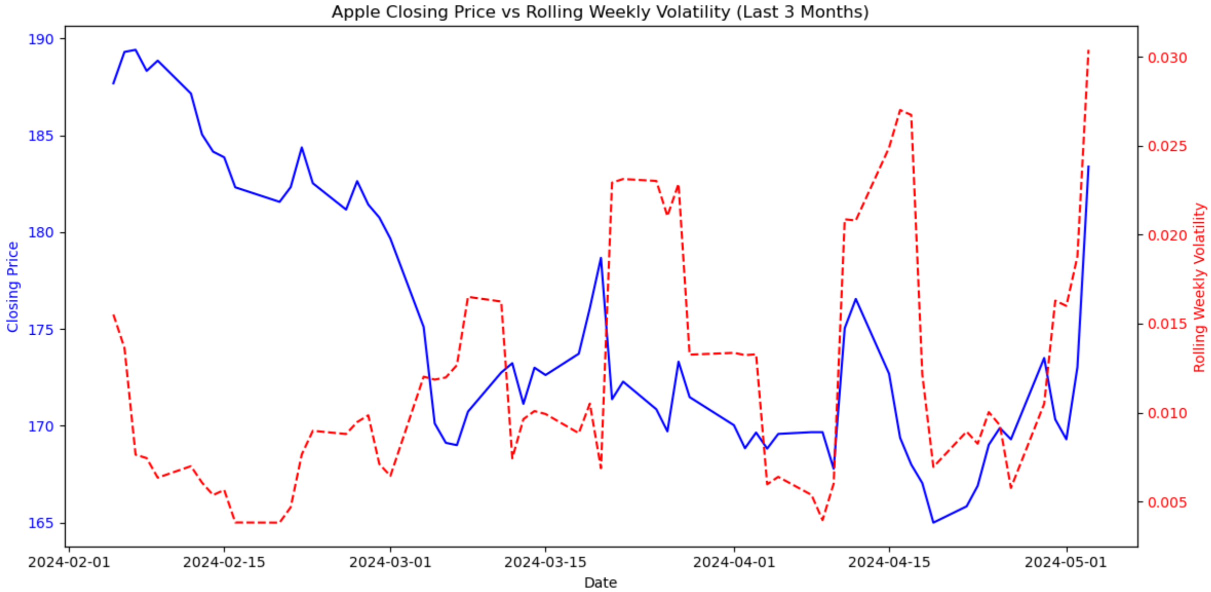
Your task:

- 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 [54]:
```
# Calculate rolling one-week volatility once on full dataset
apple['Rolling Weekly Volatility'] = apple['Daily Return'].rolling(window=5)

# Filter to last 3 months using Date index
last_3_months = apple[apple.index >= apple.index.max() - pd.DateOffset(month

# Plot
import matplotlib.pyplot as plt

fig, ax1 = plt.subplots(figsize=(12, 6))

# Plot closing price
ax1.plot(last_3_months.index, last_3_months['Close/Last'], color='blue', lab
ax1.set_xlabel("Date")
ax1.set_ylabel("Closing Price", color='blue')
ax1.tick_params(axis='y', labelcolor='blue')

# Plot rolling volatility from precomputed column
ax2 = ax1.twinx()
ax2.plot(last_3_months.index, last_3_months['Rolling Weekly Volatility'],
         color='red', linestyle='--', label='Rolling Weekly Volatility')
ax2.set_ylabel("Rolling Weekly Volatility", color='red')
ax2.tick_params(axis='y', labelcolor='red')

plt.title("Apple Closing Price vs Rolling Weekly Volatility (Last 3 Months)"
plt.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 [56]:
```
# Save Apple data with rolling volatility and daily return
apple.to_csv('apple_processed.csv', index=True)

# Save Microsoft data with daily return
microsoft.to_csv('microsoft_processed.csv', index=True)

# Save inflation data with both monthly and weekly versions
inflation.to_csv('inflation_monthly_processed.csv', index=False)
inflation_weekly.to_csv('inflation_weekly_processed.csv', index=True)

# Save standardized GDP (including DATE column)
gdp.to_csv('gdp_standardized.csv', index=False)

# Convert DATE to datetime & resample to quarterly before saving
gdp['DATE'] = pd.to_datetime(gdp['DATE'], errors='coerce')  # ensure datetim
gdp_quarterly = gdp.set_index('DATE').resample('QE').last()
gdp_quarterly.to_csv('gdp_quarterly_processed.csv', index=True)

# Save merged dataframe used for correlation analysis
combined.to_csv('merged_data.csv', index=True)
```