FINANCIAL DATA ANALYSIS

PROJECT REPORT

Name:

Student ID:

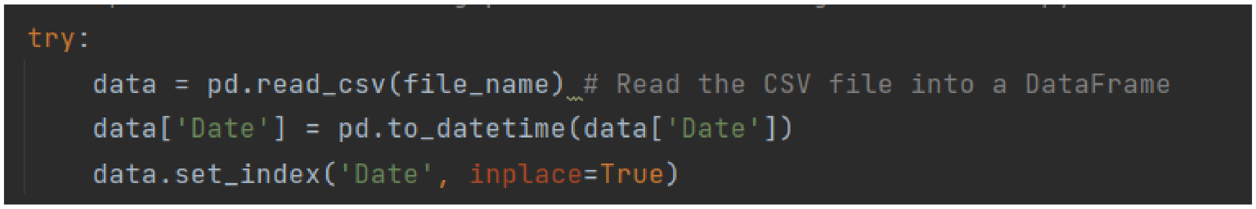
Introduction

Our Python application visualizes stock price analysis for a given date range and stock market. We use the yfinance (Yahoo Finance) library to import stock data and download it as a CSV file. To read the downloaded CSV file and manage the data, we use the Pandas library. Additionally, we use the NumPy library for various operations. After processing the data, we utilize the Matplotlib library to visualize the results.

Data Collection and Preprocessing

As previously mentioned, we used yfinance to fetch and download stock data to a csv file. After downloading the csv file,

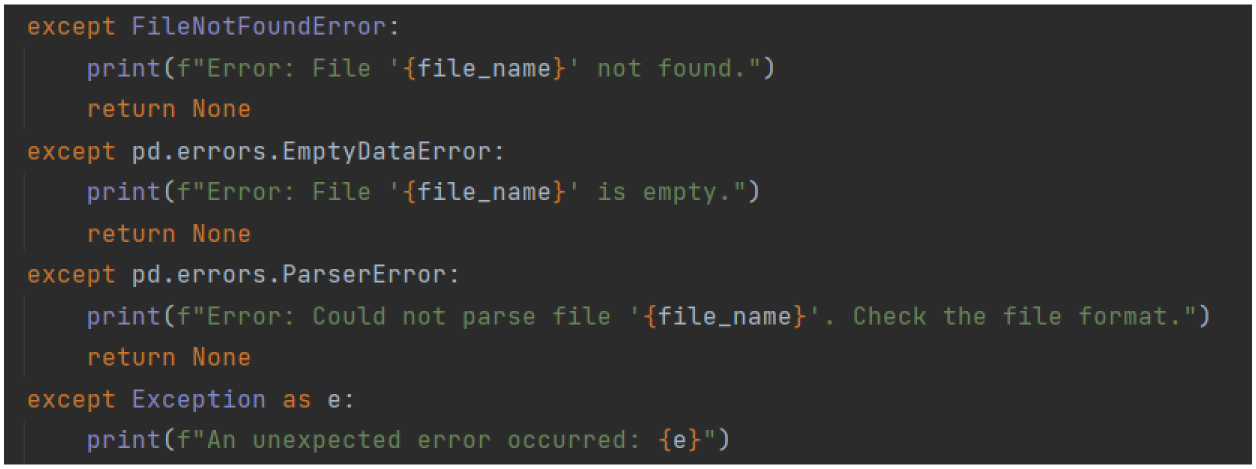
Read the csv file into a Pandas Dataframe.



```
try:
    data = pd.read_csv(file_name) # Read the CSV file into a DataFrame
    data['Date'] = pd.to_datetime(data['Date'])
    data.set_index('Date', inplace=True)
```

- Reads the CSV file into a Pandas DataFrame called data.
- Converts the 'Date' column to a datetime format to ensure proper date handling.
- Sets the 'Date' column as the DataFrame index for easy time-series analysis.

Exception Handling: File and Parsing Errors



```
except FileNotFoundError:
    print(f"Error: File '{file_name}' not found.")
    return None
except pd.errors.EmptyDataError:
    print(f"Error: File '{file_name}' is empty.")
    return None
except pd.errors.ParserError:
    print(f"Error: Could not parse file '{file_name}'. Check the file format.")
    return None
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

- If the file is not found, a message is printed, and the function returns **None**
- If the file is empty, a message is printed, and the function returns **None**
- If the file cannot be parsed (e.g., incorrect format), a message is printed, and the function returns **None**.
- Catches any other errors, prints the error message, and returns **None**

Filling Missing Values

```
# Fill missing values
data.ffill(inplace=True) # Forward fill
data.bfill(inplace=True) # Backward fill
```

- Fills missing values by propagating the last valid value forward (forward fill).
- Fills remaining missing values by propagating the next valid value backward (backward fill).

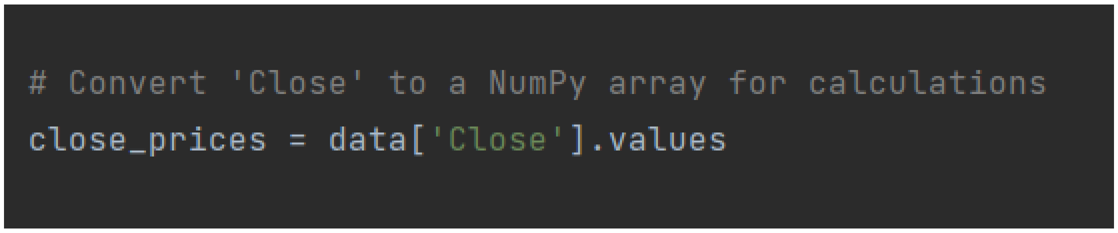
Ensuring 'Close' Column is Numeric

```
# Ensure 'Close' is numeric
data['Close'] = pd.to_numeric(data['Close'], errors='coerce')

# Drop rows with NaN values after conversion
data.dropna(subset=['Close'], inplace=True)
```

- Converts the 'Close' column to numeric values. If conversion fails, invalid entries are replaced with “NaN”.
- Drops rows where the 'Close' value is “NaN” after conversion.

Extracting Close Prices



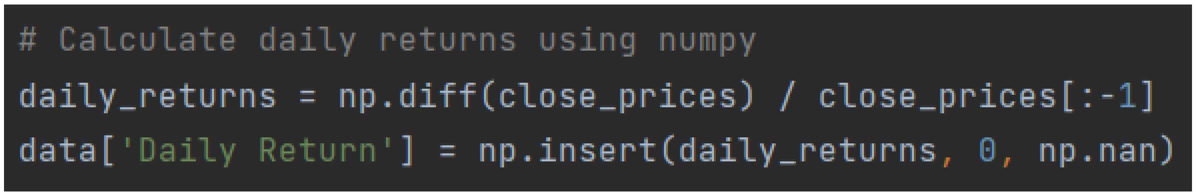
```
# Convert 'Close' to a NumPy array for calculations
close_prices = data['Close'].values
```

- Converts the 'Close' column into a NumPy array for numerical calculations.

Analysis and Findings

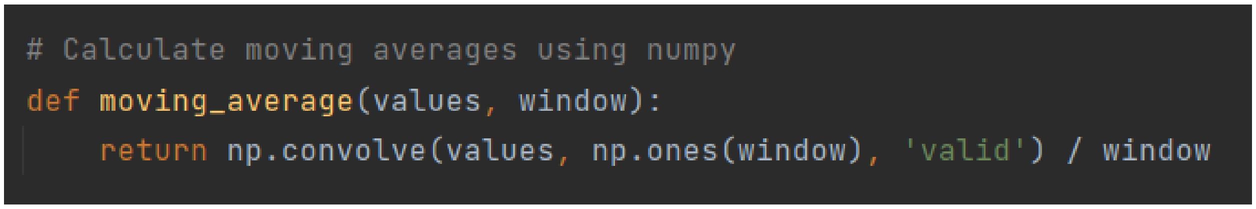
After the data collection and preprocessing, application calculates Daily Returns, Moving Averages and Rolling Volatility and visualize those data in a graph using Matplotlib.

Calculating Daily Returns, Moving Averages and Rolling Volatility



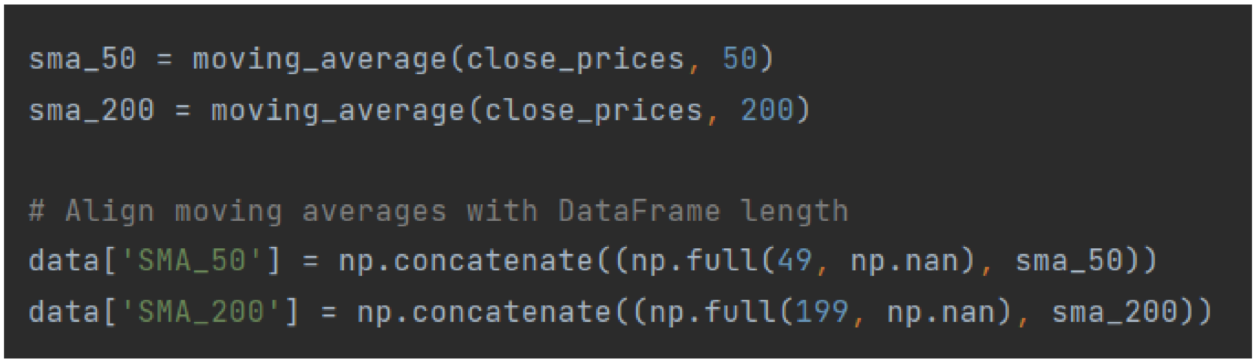
```python
# Calculate daily returns using numpy
daily_returns = np.diff(close_prices) / close_prices[:-1]
data['Daily Return'] = np.insert(daily_returns, 0, np.nan)
```

- Computes the difference between consecutive elements in the “close_prices” array and calculates the percentage change (daily returns).
- Inserts “NaN” at the beginning to align the daily returns with the original DataFrame's length.



```python
# Calculate moving averages using numpy
def moving_average(values, window):
    return np.convolve(values, np.ones(window), 'valid') / window
```

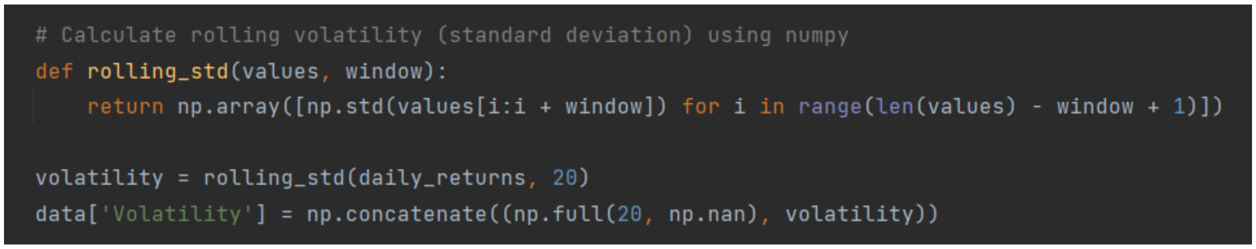
- In function “moving _averages”, applies a convolution operation to compute the rolling sum for the specified window size.
- After that it divides by the window size to calculate the average.



```python
sma_50 = moving_average(close_prices, 50)
sma_200 = moving_average(close_prices, 200)

# Align moving averages with DataFrame length
data['SMA_50'] = np.concatenate((np.full(49, np.nan), sma_50))
data['SMA_200'] = np.concatenate((np.full(199, np.nan), sma_200))
```

- "sma_50" computes the 50-day simple moving average.
- **"**sma_200**"** computes the 200-day simple moving average.
- Creates an array of "NaN" values for the initial 49 rows (because moving averages require a full window).
- Combines "NaN" values with the computed moving average to align with the DataFrame length.



```python
# Calculate rolling volatility (standard deviation) using numpy
def rolling_std(values, window):
    return np.array([np.std(values[i:i + window]) for i in range(len(values) - window + 1)])

volatility = rolling_std(daily_returns, 20)
data['Volatility'] = np.concatenate((np.full(20, np.nan), volatility))
```

- In "rolling_std" function, iterates through values in sliding windows of size window.
- **"**np.std" computes the standard deviation for each window.
- Volatility computes the 20-day rolling standard deviation of daily returns.
- **"**np.full(20, np.nan)" adds "NaN" values for the initial rows to align with the DataFrame length.

Creating the Figure

```python
# Data Visualization
plt.figure(figsize=(14, 10))
```

- Creates a new figure for plotting with a specified size of 14 inches wide by 10 inches tall.

Plotting Stock Price and Moving Averages

```python
# Plot stock price and moving averages
plt.subplot(2, 1, 1)
plt.plot(data.index, data['Close'], label='Close Price', color='blue', linewidth=1.5)
plt.plot(data.index, data['SMA_50'], label='50-Day MA', color='red', linestyle='--')
plt.plot(data.index, data['SMA_200'], label='200-Day MA', color='green', linestyle='--')
plt.title('Stock Price and Moving Averages')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
```

- Sets up a subplot layout with 2 rows and 1 column, activating the first subplot for plotting.
- Represents the x-axis values (dates).
- Plots the stock's closing prices on the y-axis in **blue** with a solid line and line width of 1.5.
- Plots the 50-day simple moving average in **red** with a dashed line
- Plots the 200-day simple moving average in **green** with a dashed line.
- Sets the title of the first plot as "Stock Price and Moving Averages."
- Labels the x-axis as "Date."

- Labels the y-axis as "Price."
- Displays a legend to identify the different lines in the plot.

Plotting Volatility

```
# Plot Volatility
plt.subplot(2, 1, 2)
plt.plot(data.index, data['Volatility'], label='Volatility', color='purple', linewidth=1.5)
plt.title('Stock Price Volatility')
plt.xlabel('Date')
plt.ylabel('Volatility')
plt.legend()

plt.tight_layout()
plt.show()
```

- Activates the second subplot in the 2-row layout.
- Represents the x-axis values (dates).
- Plots the stock's volatility on the y-axis in **purple** with a line width of 1.5.
- Sets the title of the second plot as "Stock Price Volatility."
- Labels the x-axis as "Date."
- Labels the y-axis as "Volatility."
- Displays a legend to identify the volatility line.

Adjusting Layout and Displaying the Plot

```
plt.tight_layout()
plt.show()
```

- Automatically adjusts subplot parameters to minimize overlap between elements.
- Displays the final combined figure with the two subplots

Conclusion

After all the process is done, the application provides the information below.

- Stock Price and Moving Averages
- Stock Price Volatility

Users can enter the start date, end date and ticker symbol as necessary.

Sample output

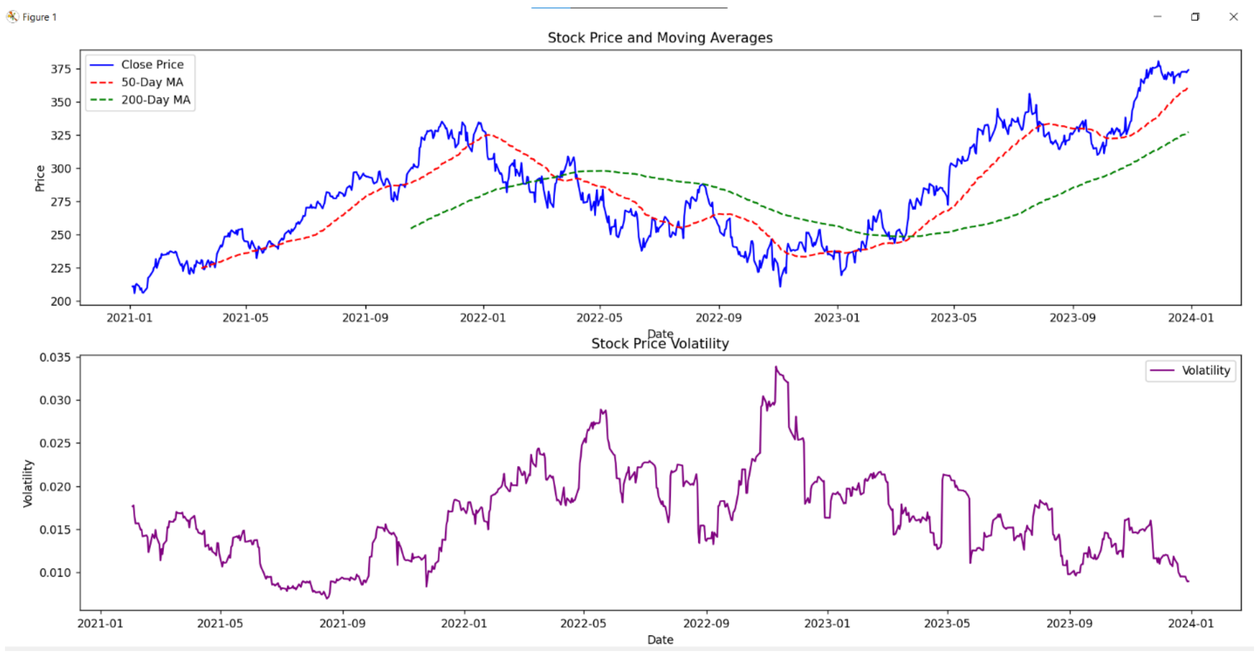


Figure 1
Stock Price and Moving Averages
Close Price
50-Day MA
200-Day MA
Price
Date
Stock Price Volatility
Volatility
Volatility
Date