Python Code Documentation

How to put the input values

The above library setup enables users to input stock symbols from different exchanges, such as the National Stock Exchange (NSE) in India, the New York Stock Exchange (NYSE), and others, to compare financial metrics like dividend yield. Stock symbols need to be formatted appropriately for each exchange as per the conventions supported by Yahoo Finance (used here). For example, for NSE stocks, you append .NS to the stock symbol (e.g., RELIANCE.NS for Reliance Industries on the NSE). For NYSE, you use the standard stock ticker without any suffix (e.g., AAPL for Apple Inc.). This flexibility allows users to analyze and compare stocks globally by simply entering their respective symbols in the input fields, enabling insights into different markets seamlessly.

Yes, the library covers other exchanges supported by Yahoo Finance, allowing users to input stock symbols from a wide range of global markets. Each exchange has its own suffix or format for the stock symbols. Here are a few examples:

- **NSE (India)**: Append .NS to the stock symbol (e.g., TCS.NS for Tata Consultancy Services).
- **BSE (India)**: Append .BO to the stock symbol (e.g., 500325.BO for Reliance Industries).
- **NYSE (USA)**: Use the stock ticker as-is (e.g., AAPL for Apple Inc.).
- **NASDAQ (USA)**: Similarly, use the ticker without a suffix (e.g., MSFT for Microsoft).
- **LSE (London Stock Exchange)**: Append .L (e.g., BP.L for BP).
- **TSE (Tokyo Stock Exchange)**: Append .T (e.g., 7203.T for Toyota Motor).
- **ASX (Australian Securities Exchange)**: Append .AX (e.g., BHP.AX for BHP Group).

This approach provides flexibility to analyze stocks across various global exchanges. Users can input symbols formatted for the respective exchanges, ensuring accurate data retrieval and analysis. For specific cases, users can refer to Yahoo Finance's symbol guidelines for additional exchanges and suffixes.

get_pe_ratio()

This function, get_pe_ratio, is designed to calculate the Price-to-Earnings (P/E) ratio for a specified stock. The P/E ratio is a financial metric widely used by investors and traders to evaluate the relative value of a company's stock. It represents the relationship between a company's current stock price and its earnings per share (EPS).

Definition of get_pe_ratio

- Function Input: A stock ticker symbol (e.g., "AAPL" for Apple).
- Process: It retrieves data from the Yahoo Finance API via the yfinance library, fetching the current price (current_price) and trailing earnings per share (trailingEps) of the stock.
- Calculation: The P/E ratio is calculated by dividing the current_price by eps.
- Output: The function returns the P/E ratio if available, or None if the required data is missing.

Significance for Traders

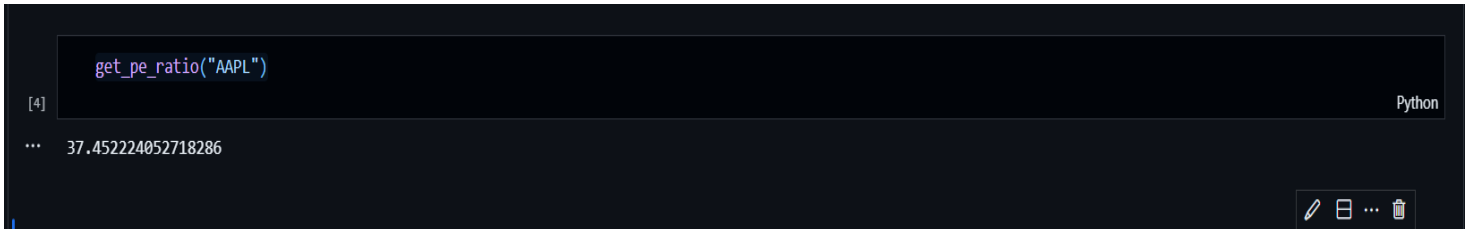
For traders, the P/E ratio is essential for comparing the valuation of companies. A high P/E ratio could suggest that the stock is overvalued or that investors expect high growth rates in the future, while a low P/E might indicate undervaluation or slower growth expectations. By analyzing the P/E ratio, traders can make more informed decisions about whether to buy, hold, or sell a stock.

What the Function Covers

- Fetches Current Price: Retrieves the latest market price of the stock.
- Fetches EPS: Gathers earnings data, which is essential for calculating the P/E ratio.
- Handles Data Availability: Includes error handling in case the data for current price or EPS is unavailable for the given ticker, which is common for certain stocks or newly listed companies.

Overall, this function is a practical tool for traders aiming to quickly assess a stock's valuation relative to its earnings, aiding in fundamental analysis and comparison across stocks.

Use Case



Source Code for the get_pe_ratio() function

```
#Creating the function which takes the ticket as input and returns the price
to earnings ratio

def get_pe_ratio(ticker):
    # Fetch the stock data using yfinance
    stock = yf.Ticker(ticker)

    try:
        # Get the current price
        current_price = stock.info['currentPrice']

        # Get the earnings per share (EPS)
        eps = stock.info['trailingEps']

        # Calculate the P/E ratio
        pe_ratio = current_price / eps

        return pe_ratio
    except KeyError:
        print(f"Unable to calculate P/E ratio for {ticker}. Required data not
available.")
        return None
```

get_pb_ratio(ticker)

Significance of the Financial Metric: Price-to-Book (P/B) Ratio

The Price-to-Book (P/B) ratio is a financial metric that compares a company's market value to its book value. It is calculated as:

P/B=Market Price per ShareBook Value per ShareP/B = \frac{\text{Market Price per Share}}{\text{Book Value per Share}}P/B=Book Value per ShareMarket Price per Share

- Market Price per Share: The current trading price of a single share of the company.
- Book Value per Share: The value of the company's net assets (total assets minus liabilities) divided by the total number of outstanding shares.

Why is the P/B Ratio Important?

1. Valuation Indicator:
    - A low P/B ratio (< 1) may indicate that the stock is undervalued relative to its book value.
    - A high P/B ratio may suggest overvaluation or that investors expect high growth.
2. Industry-Specific Insight: The P/B ratio is more relevant for asset-heavy industries like banking, real estate, or manufacturing.
3. Risk Assessment: It provides insights into the company's financial health and investor sentiment.

Source Code

```
def get_pb_ratio(ticker):
    # Fetch the stock data using yfinance
    stock = yf.Ticker(ticker)

    try:
        # Get the current price
        current_price = stock.info['currentPrice']

        # Get the book value per share
        book_value_per_share = stock.info['bookValue']

        # Calculate the P/B ratio
        pb_ratio = current_price / book_value_per_share

        return pb_ratio
    except KeyError:
```

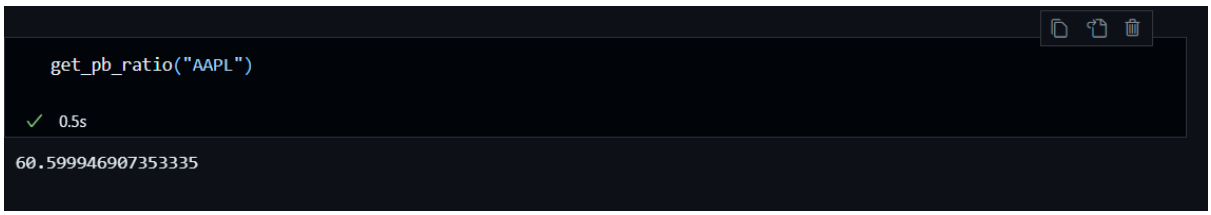
```
        print(f"Unable to calculate P/B ratio for {ticker}. Required data not
available.")
        return None

# Testing the created function
#print(get_pb_ratio("AAPL"))
```

Explanation of the Function

The function get_pb_ratio retrieves the P/B ratio for a specified stock ticker using the yfinance library. Here's a breakdown:

1. Input:
    - ticker: The stock's ticker symbol (e.g., "AAPL" for Apple Inc.).
2. Fetching Stock Data:
    - The yfinance library's Ticker object fetches detailed stock information for the specified ticker.
3. Data Retrieval:
    - The currentPrice and bookValue are extracted from the info dictionary provided by yfinance.
4. P/B Ratio Calculation:
    - If the required data (currentPrice and bookValue) is available, the P/B ratio is calculated using: P/B Ratio=currentPricebookValue\text{P/B Ratio} = \frac{\text{currentPrice}}{\text{bookValue}}P/B Ratio=bookValuecurrentPrice
5. Error Handling:
    - If any of the required fields (currentPrice or bookValue) are missing, a KeyError exception is handled gracefully. A message is printed indicating that the data is unavailable, and the function returns None.
6. Output:
    - Returns the calculated P/B ratio if successful, or None if the data is not available.

Use Case



get_ps_ratio(ticker)

Significance of the Financial Metric: Price-to-Sales (P/S) Ratio

The Price-to-Sales (P/S) ratio is a financial metric that measures the value investors place on a company's sales relative to its market value. It is calculated as:

P/S=Market Price per ShareRevenue per ShareP/S = \frac{\text{Market Price per Share}}{\text{Revenue per Share}}P/S=Revenue per ShareMarket Price per Share

- Market Price per Share: The current trading price of a single share of the company.
- Revenue per Share: The total revenue of the company divided by the number of outstanding shares.

Why is the P/S Ratio Important?

1. **Valuation Benchmark:**
    - A low P/S ratio may indicate that a company is undervalued compared to its sales.
    - A high P/S ratio may suggest that investors expect significant future growth.
2. **Comparative Analysis:** Useful for comparing companies in the same industry, especially for firms with no profits (e.g., startups or growth companies).
3. **Industry Neutrality:** While more relevant for revenue-driven industries, the P/S ratio can be applied across a broad range of sectors.

Explanation of the Function

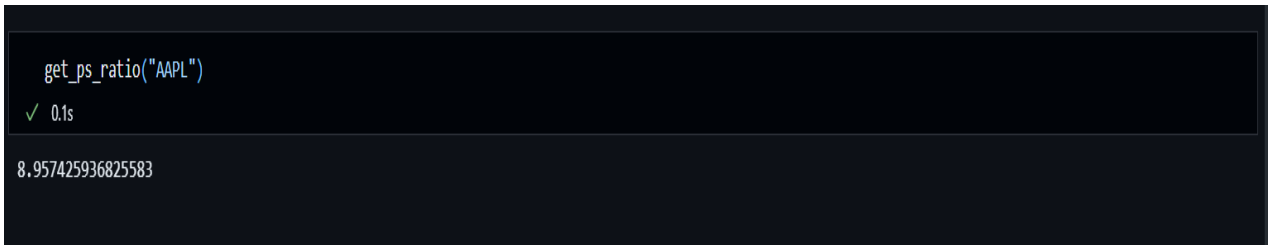
The function get_ps_ratio retrieves the P/S ratio for a specified stock ticker using the yfinance library. Here's how it works:

1. **Input:**
    - ticker: The stock's ticker symbol (e.g., "AAPL" for Apple Inc.).
2. **Fetching Stock Data:**
    - The yfinance library's Ticker object fetches comprehensive stock information for the specified ticker.
3. **Data Retrieval:**
    - The currentPrice and revenuePerShare fields are extracted from the info dictionary provided by yfinance.
4. **P/S Ratio Calculation:**
    - If the required data (currentPrice and revenuePerShare) is available, the P/S ratio is calculated using: P/S Ratio=currentPricerevenuePerShare\text{P/S Ratio} =

\frac{\text{currentPrice}}{\text{revenuePerShare}}P/S Ratio=revenuePerSharecurrent Price

5. **Error Handling:**
    - If either currentPrice or revenuePerShare is unavailable, the function handles the KeyError gracefully. It prints a message and returns None.
6. **Output:**
    - Returns the calculated P/S ratio if successful, or None if the data is unavailable.

Use Case



Source Code

```
def get_ps_ratio(ticker):
    # Fetch the stock data using yfinance
    stock = yf.Ticker(ticker)

    try:
        # Get the current price
        current_price = stock.info['currentPrice']

        # Get the revenue per share
        revenue_per_share = stock.info['revenuePerShare']

        # Calculate the P/S ratio
        ps_ratio = current_price / revenue_per_share

        return ps_ratio
    except KeyError:
        print(f"Unable to calculate P/S ratio for {ticker}. Required data not
available.")
        return None

# Testing the created function
#print(get_ps_ratio("AAPL"))
```

get_ps_ratio(ticker)

Significance of the Financial Metric: Enterprise Value (EV)

Enterprise Value (EV) is a key financial metric that represents the total value of a company. It is a comprehensive measure that includes equity value, debt, and cash, providing a clear picture of the firm's financial health and valuation. The formula is:

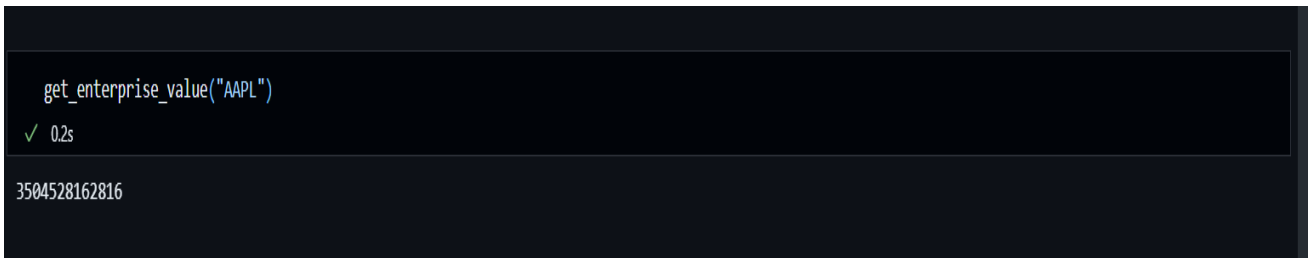
EV=Market Capitalization+Total Debt−Cash and Cash EquivalentsEV = \text{Market Capitalization} + \text{Total Debt} - \text{Cash and Cash Equivalents}EV=Market Capitalization+Total Debt−Cash and Cash Equivalents

However, some financial data platforms like yfinance directly provide the pre-calculated **Enterprise Value** to streamline analysis.

Why is EV Important?

1. **Complete Valuation**:
    - Unlike market capitalization, EV incorporates debt and cash to give a fuller view of a company's worth.
2. **Capital Structure-Neutral**:
    - Useful for comparing companies with different financing structures (e.g., debt-heavy vs. equity-heavy firms).
3. **M&A Valuation**:
    - Frequently used in mergers and acquisitions to assess the true cost of acquiring a business.

Use Case



Source Code

```
def get_enterprise_value(ticker):
    # Fetch the stock data using yfinance
    stock = yf.Ticker(ticker)

    try:
        # Get the enterprise value
        enterprise_value = stock.info['enterpriseValue']

        return enterprise_value
    except KeyError:
        print(f"Unable to retrieve Enterprise Value for {ticker}. Required
data not available.")
        return None

# Testing the created function
#print(get_enterprise_value("AAPL"))
```

Explanation of the Function

1. **Input**:
    - ticker: The stock's ticker symbol (e.g., "AAPL" for Apple Inc.).
2. **Fetching Stock Data**:
    - The yfinance library's Ticker object retrieves comprehensive stock information for the specified ticker.
3. **Data Retrieval**:
    - The function accesses the enterpriseValue key from the info dictionary provided by yfinance. This value represents the pre-calculated Enterprise Value.
4. **Output**:
    - Returns the **Enterprise Value** if the data is successfully retrieved.
5. **Error Handling**:
    - If the enterpriseValue key is unavailable in the stock data, a KeyError is caught. The function prints an error message and returns None.

get_ev_ebitda(ticker)

The **Enterprise Value to EBITDA (EV/EBITDA)** ratio is a widely used valuation metric that compares a company's total value (EV) to its Earnings Before Interest, Taxes, Depreciation, and Amortization (EBITDA). It provides a measure of how much an investor is willing to pay for a company's operational performance. The formula is:

EV/EBITDA=Enterprise Value (EV)EBITDA\text{EV/EBITDA} = \frac{\text{Enterprise Value (EV)}}{\text{EBITDA}}EV/EBITDA=EBITDAEnterprise Value (EV)

- **Enterprise Value (EV)**: The total valuation of a company, including equity, debt, and cash adjustments.
- **EBITDA**: A proxy for cash flow from operations, excluding non-operating expenses.

Why is EV/EBITDA Important?

1. **Valuation Benchmark**:
    - A low EV/EBITDA ratio may indicate that a company is undervalued.
    - A high EV/EBITDA ratio suggests higher valuation expectations, possibly due to strong growth potential.
2. **Cross-Industry Comparison**:
    - Suitable for comparing companies across industries with different capital structures.
3. **Operational Focus**:
    - Unlike other ratios, it focuses on operational performance without being affected by financing or accounting policies.

Use Case

Source Code

```
def get_ev_ebitda(ticker):
    # Fetch the stock data using yfinance
    stock = yf.Ticker(ticker)

    try:
        # Get the Enterprise Value
        enterprise_value = stock.info['enterpriseValue']

        # Get the EBITDA
        ebitda = stock.info['ebitda']

        # Calculate the EV/EBITDA ratio
        ev_ebitda_ratio = enterprise_value / ebitda

        return ev_ebitda_ratio
    except KeyError:
        print(f"Unable to calculate EV/EBITDA ratio for {ticker}. Required
data not available.")
        return None
```

get_gross_margin()

Significance of the Financial Metric: Gross Margin

Gross Margin is a profitability ratio that measures the percentage of revenue remaining after deducting the Cost of Goods Sold (COGS). It represents how efficiently a company produces and sells its products. The formula is:

Gross Margin (%)=(Gross ProfitRevenue)×100\text{Gross Margin (\%)} = \left(\frac{\text{Gross Profit}}{\text{Revenue}} \right) \times 100Gross Margin (%)=(RevenueGross Profit)×100

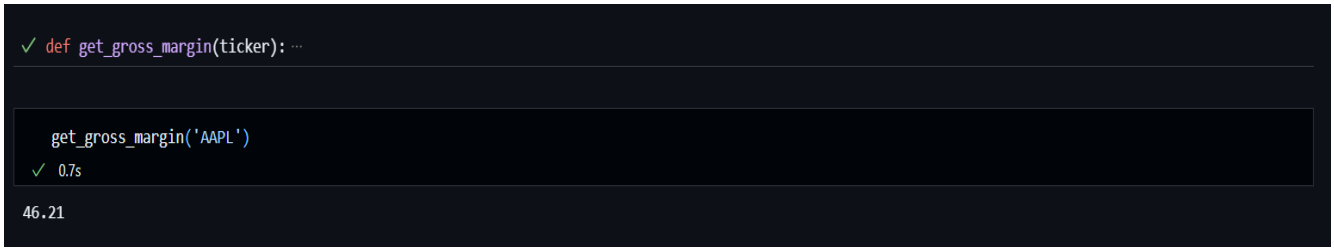
Where:

- Gross Profit = Revenue - Cost of Goods Sold (COGS)

Why is Gross Margin Important?

1. **Operational Efficiency:**
    - Indicates how well a company controls its production costs relative to its sales.
2. **Profitability Assessment:**
    - A higher gross margin implies a more profitable company, able to cover operating and non-operating expenses.
3. **Industry Comparison:**
    - Enables benchmarking against competitors to evaluate cost management strategies.
4. **Financial Health:**
    - A declining gross margin may indicate rising costs or inefficiencies, warranting further investigation.

Use Case



Source Code

```python
def get_gross_margin(ticker):
    try:
        # Fetch the stock data
        stock = yf.Ticker(ticker)

        # Get the financial data
        financials = stock.financials

        # Calculate Gross Margin
        revenue = financials.loc['Total Revenue'].iloc[0]
        cogs = financials.loc['Cost Of Revenue'].iloc[0]
        gross_profit = revenue - cogs
        gross_margin = (gross_profit / revenue) * 100

        return round(gross_margin, 2)
    except Exception as e:
        print(f"Error calculating Gross Margin for {ticker}: {str(e)}")
        return None
```

How the Function Works

1. **Input:**
    - **ticker: The stock's ticker symbol (e.g., "AAPL" for Apple Inc.).**
2. **Fetching Stock Data:**
    - **The yfinance library's Ticker object retrieves detailed stock data.**
    - **The financials property provides the company's income statement.**
3. **Data Retrieval:**
    - **Total Revenue: Total income generated from sales.**
    - **Cost Of Revenue: Expenses directly associated with producing and delivering goods or services (COGS).**
4. **Gross Margin Calculation:**
    - **Gross Profit = Total Revenue - Cost Of Revenue.**
    - **Gross Margin (%) = (Gross Profit/Total Revenue)×100(\text{Gross Profit} / \text{Total Revenue}) \times 100(Gross Profit/Total Revenue)×100.**
    - **The result is rounded to two decimal places for better readability.**

5. **Error Handling:**
    - **If any data is missing or an error occurs (e.g., incorrect data labels, empty financials), an exception is caught, and a descriptive message is printed.**
6. **Output:**
    - **Returns the Gross Margin (%) if successful, or None if an error occurs.**

get_operating_margin()

Significance of the Financial Metric: Operating Margin

Operating Margin is a profitability ratio that measures the percentage of revenue remaining after covering the operating expenses of a company. It shows how efficiently a company manages its core operations to generate profit. The formula is:

Operating Margin (%)=(Operating IncomeRevenue)×100\text{Operating Margin (\%)} = \left(\frac{\text{Operating Income}}{\text{Revenue}} \right) \times 100Operating Margin (%)=(RevenueOperating Income)×100

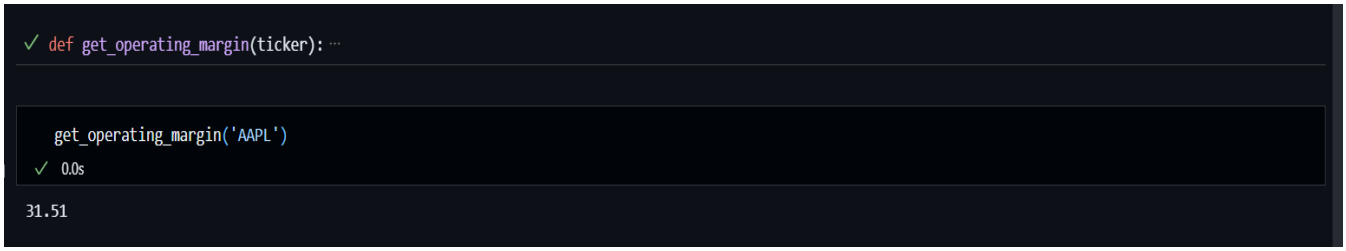
Where:

- Operating Income: Revenue minus operating expenses, including costs such as wages, depreciation, and administrative expenses, but excluding taxes and interest.

Why is Operating Margin Important?

1. **Efficiency Indicator:**
    - A higher operating margin indicates that the company generates more profit from its operations for every dollar of revenue.
2. **Cost Management Insight:**
    - Reflects how well the company controls its operating costs.
3. **Industry Comparisons:**
    - Allows for benchmarking against competitors within the same industry.
4. **Financial Health:**
    - A declining operating margin might signal rising costs or decreasing efficiency, requiring further investigation.

Use Case



Source Code

```
def get_operating_margin(ticker):
    try:
        # Fetch the stock data
        stock = yf.Ticker(ticker)

        # Get the financial data
        financials = stock.financials

        # Calculate Operating Margin
        revenue = financials.loc['Total Revenue'].iloc[0]
        operating_income = financials.loc['Operating Income'].iloc[0]
        operating_margin = (operating_income / revenue) * 100

        return round(operating_margin, 2)
    except Exception as e:
        print(f"Error calculating Operating Margin for {ticker}: {str(e)}")
        return None
```

How the Function Works

1. **Input**:
    - ticker: The stock's ticker symbol (e.g., "AAPL" for Apple Inc.).
2. **Fetching Stock Data**:
    - The yfinance library's Ticker object retrieves detailed financial data.
3. **Data Retrieval**:
    - **Total Revenue**: Total income generated by the company from sales or services.
    - **Operating Income**: Earnings from core business operations after deducting operating expenses.
4. **Operating Margin Calculation**:
    - Formula:
      Operating Margin (%)=(Operating IncomeTotal Revenue)×100\text{Operating Margin (\%)} = \left(\frac{\text{Operating Income}}{\text{Total Revenue}} \right) \times 100Operating Margin (%)=(Total RevenueOperating Income)×100
    - Rounds the result to two decimal places for clarity.
5. **Error Handling**:
    - If any required financial data is missing or an error occurs (e.g., incorrect labels), the exception is caught, and a descriptive error message is printed.
6. **Output**:

- Returns the **Operating Margin (%)** if successful, or None if an error occurs.

get_net_profit_margin()

Significance of the Financial Metric: Net Profit Margin

Net Profit Margin is a key profitability metric that measures the percentage of revenue left as profit after all expenses, taxes, and costs have been deducted. It reflects the company's ability to convert sales into net income. The formula is:

Net Profit Margin (%)=(Net IncomeTotal Revenue)×100\text{Net Profit Margin (\%)} = \left(\frac{\text{Net Income}}{\text{Total Revenue}} \right) \times 100Net Profit Margin (%)=(Total RevenueNet Income)×100

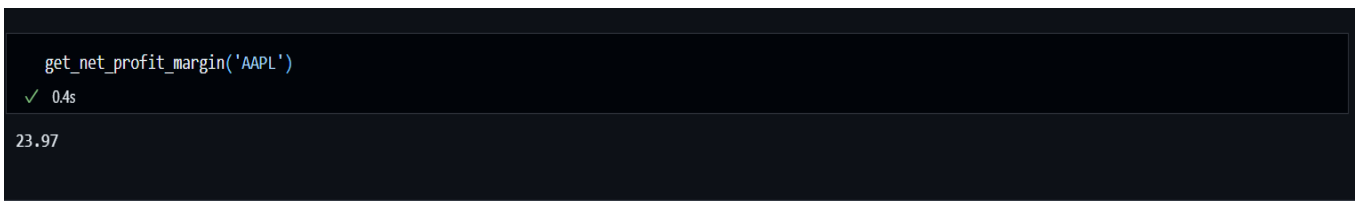
Where:

- **Net Income:** The total profit of a company after deducting all costs, taxes, interest, and other expenses.
- **Total Revenue:** The total income generated from the sale of goods or services.

Why is Net Profit Margin Important?

1. **Profitability Indicator:**
    - Demonstrates how efficiently a company converts revenue into actual profit.
2. **Comparative Analysis:**
    - Allows comparison of profitability between companies or industries, regardless of their size.
3. **Financial Health:**
    - A low or declining net profit margin may indicate inefficiencies or rising costs.
4. **Investment Decisions:**
    - Higher margins attract investors, as they signal strong cost control and profitability.

Use Case



Source Code

```
def get_net_profit_margin(ticker):
    try:
        # Fetch the stock data
        stock = yf.Ticker(ticker)

        # Get the financial data
        financials = stock.financials

        # Calculate Net Profit Margin
        revenue = financials.loc['Total Revenue'].iloc[0]
        net_income = financials.loc['Net Income'].iloc[0]
        net_profit_margin = (net_income / revenue) * 100

        return round(net_profit_margin, 2)
    except Exception as e:
        print(f"Error calculating Net Profit Margin for {ticker}: {str(e)}")
        return None
```

How the Function Works

1. **Input**:
    - ticker: The stock's ticker symbol (e.g., "AAPL" for Apple Inc.).
2. **Fetching Stock Data**:
    - Uses the yfinance library to retrieve financial data associated with the stock.
3. **Data Retrieval**:
    - **Total Revenue**: Total income from sales or services.
    - **Net Income**: Total profit after all expenses and taxes.
4. **Net Profit Margin Calculation**:
    - Formula: Net Profit Margin (%)=(Net IncomeTotal Revenue)×100\text{Net Profit Margin (\%)} = \left(\frac{\text{Net Income}}{\text{Total Revenue}} \right) \times 100Net Profit Margin (%)=(Total RevenueNet Income)×100
    - The result is rounded to two decimal places for clarity.
5. **Error Handling**:
    - If required data is missing or an error occurs, an exception is caught, and a descriptive error message is printed.
6. **Output**:
    - Returns the **Net Profit Margin (%)** if successful or None if an error occurs.

get_return_on_equity()

Significance of the Financial Metric: Return on Equity (ROE)

Return on Equity (ROE) is a profitability metric that measures a company's ability to generate profit from its shareholders' equity. It indicates how effectively the company is using the funds invested by shareholders to generate earnings. The formula is:

ROE (%)=(Net IncomeShareholders' Equity)×100\text{ROE (\%)} = \left(\frac{\text{Net Income}}{\text{Shareholders' Equity}} \right) \times 100ROE (%)=(Shareholders' EquityNet Income)×100

Where:

- **Net Income**: The total profit after all expenses, taxes, and costs are deducted.
- **Shareholders' Equity**: The residual interest in the assets of the company after deducting liabilities.

Why is Return on Equity Important?

1. **Profitability Indicator**:
    - Shows how efficiently the company is generating profits from shareholders' investments.
2. **Management Effectiveness**:
    - Reflects management's ability to deploy equity capital effectively.
3. **Investment Decisions**:
    - A higher ROE often attracts investors as it signifies better returns on their investment.
4. **Comparative Analysis**:
    - Useful for comparing companies in the same sector or industry.

Use Case

Source Code

```
def get_return_on_equity(ticker):
    try:
        # Fetch the stock data
        stock = yf.Ticker(ticker)

        # Get the financial data and balance sheet
        financials = stock.financials
        balance_sheet = stock.balance_sheet

        # Calculate Return on Equity (ROE)
        net_income = financials.loc['Net Income'].iloc[0]
        stockholders_equity = balance_sheet.loc['Stockholders Equity'].iloc[0]
        roe = (net_income / stockholders_equity) * 100

        return round(roe, 2)
    except Exception as e:
        print(f"Error calculating Return on Equity (ROE) for {ticker}:
{str(e)}")
        return None
```

How the Function Works

1. **Input**:
    - ticker: The stock's ticker symbol (e.g., "AAPL" for Apple Inc.).
2. **Fetching Stock Data**:
    - Uses the yfinance library to retrieve financial and balance sheet data for the specified stock.
3. **Data Retrieval**:
    - **Net Income**: Obtained from the financial data (stock.financials).
    - **Shareholders' Equity**: Obtained from the balance sheet (stock.balance_sheet).
4. **Return on Equity Calculation**:
    - Formula: ROE (%)=(Net IncomeShareholders' Equity)×100\text{ROE (\%)} = \left(\frac{\text{Net Income}}{\text{Shareholders' Equity}} \right) \times 100ROE (%)=(Shareholders' EquityNet Income)×100
    - The result is rounded to two decimal places for readability.
5. **Error Handling**:
    - If required data is missing or an error occurs during execution, an exception is caught, and a descriptive error message is displayed.

6. **Output**:
    - Returns the **ROE (%)** if the calculation is successful or None if an error occurs.

profitability_metrics_finance()

Significance of Profitability Metrics

Profitability metrics are key indicators used to assess a company's ability to generate earnings relative to its revenue, assets, and shareholders' equity. These metrics provide a detailed view of financial health and operational efficiency, enabling investors and analysts to evaluate performance across different dimensions.

Purpose of the Function:

The profitability_metrics_finance function automates the calculation of multiple profitability metrics for a given stock, using financial data retrieved via the yfinance library. It consolidates several key metrics into a structured output for easy interpretation.

Applications of the Function

1. **Comprehensive Profitability Analysis**:
    - Combines multiple metrics into a single output for a holistic view of a company's financial health.
2. **Investment Decision-Making**:
    - Provides key insights into margins, returns, and profitability, aiding in evaluating investment opportunities.
3. **Comparison Across Companies**:
    - Enables benchmarking of financial performance within the same industry.
4. **Operational Efficiency Assessment**:
    - Highlights how well a company utilizes its revenue, equity, and assets to generate profit.

Advantages of the Function

- **Multi-Metric Integration**:
    - Calculates and returns several critical financial metrics in a structured format.
- **Dynamic Data Retrieval**:
    - Fetches real-time financial data for accurate analysis.
- **Robust Error Handling**:
    - Ensures stability by handling missing or invalid data gracefully.

Source Code

```
def profitability_metrics_finance(ticker):
    stock = yf.Ticker(ticker)

    # Get the stock financials and balance sheet data
    financials = stock.financials
    balance_sheet = stock.balance_sheet

    # Calculate metrics
    try:
        # Retrieve necessary values
        gross_profit = financials.loc['Gross Profit'].iloc[0]
        total_revenue = financials.loc['Total Revenue'].iloc[0]
        operating_income = financials.loc['Operating Income'].iloc[0]
        net_income = financials.loc['Net Income'].iloc[0]
        total_assets = balance_sheet.loc['Total Assets'].iloc[0]
        stockholders_equity = balance_sheet.loc['Stockholders Equity'].iloc[0]

        # Calculate margins and ratios
        gross_margin = (gross_profit / total_revenue) * 100 if total_revenue
!= 0 else 0
        operating_margin = (operating_income / total_revenue) * 100 if
total_revenue != 0 else 0
        net_profit_margin = (net_income / total_revenue) * 100 if
total_revenue != 0 else 0
        roe = (net_income / stockholders_equity) * 100 if stockholders_equity
!= 0 else 0
        roa = (net_income / total_assets) * 100 if total_assets != 0 else 0

        # Prepare the data to return
        data = [
            {"Metric": "Gross Margin", "Value": f"{gross_margin:.2f}%"},
            {"Metric": "Operating Margin", "Value":
f"{operating_margin:.2f}%"},
            {"Metric": "Net Profit Margin", "Value":
f"{net_profit_margin:.2f}%"},
            {"Metric": "Return on Equity (ROE)", "Value": f"{roe:.2f}%"},
            {"Metric": "Return on Assets (ROA)", "Value": f"{roa:.2f}%"}
        ]

        return data

    except Exception as e:
        return f"Error calculating profitability metrics for {ticker}:
{str(e)}"
```

How the Function Works

1. **Input**:
    - ticker: The stock ticker symbol (e.g., "AAPL" for Apple Inc.).
2. **Fetching Data**:
    - The function retrieves financial and balance sheet data for the specified stock using the yfinance library.
3. **Calculating Metrics**:
    - **Gross Margin**: Gross Margin (%)=(Gross ProfitTotal Revenue)×100\text{Gross Margin (\%)} = \left(\frac{\text{Gross Profit}}{\text{Total Revenue}} \right) \times 100Gross Margin (%)=(Total RevenueGross Profit)×100
    - **Operating Margin**:
      Operating Margin (%)=(Operating IncomeTotal Revenue)×100\text{Operating Margin (\%)} = \left(\frac{\text{Operating Income}}{\text{Total Revenue}} \right) \times 100Operating Margin (%)=(Total RevenueOperating Income)×100
    - **Net Profit Margin**: Net Profit Margin (%)=(Net IncomeTotal Revenue)×100\text{Net Profit Margin (\%)} = \left(\frac{\text{Net Income}}{\text{Total Revenue}} \right) \times 100Net Profit Margin (%)=(Total RevenueNet Income)×100
    - **Return on Equity (ROE)**: ROE (%)=(Net IncomeStockholders' Equity)×100\text{ROE (\%)} = \left(\frac{\text{Net Income}}{\text{Stockholders' Equity}} \right) \times 100ROE (%)=(Stockholders' EquityNet Income)×100
    - **Return on Assets (ROA)**: ROA (%)=(Net IncomeTotal Assets)×100\text{ROA (\%)} = \left(\frac{\text{Net Income}}{\text{Total Assets}} \right) \times 100ROA (%)=(Total AssetsNet Income)×100
4. **Data Preparation**:
    - The calculated metrics are formatted into a list of dictionaries, where each dictionary contains the metric name and its value.
5. **Error Handling**:
    - If an error occurs (e.g., missing data), the function catches the exception and returns a descriptive error message.
6. **Output**:
    - A list of key profitability metrics, each labeled with its name and rounded to two decimal places.

Use Case

```
profitability_metrics_finance('AAPL')
```

✓ 1.1s

```
[{'Metric': 'Gross Margin', 'Value': '46.21%'},
 {'Metric': 'Operating Margin', 'Value': '31.51%'},
 {'Metric': 'Net Profit Margin', 'Value': '23.97%'},
 {'Metric': 'Return on Equity (ROE)', 'Value': '164.59%'},
 {'Metric': 'Return on Assets (ROA)', 'Value': '25.68%'}]
```

get_eps_growth('AAPL')

Explanation of the get_eps_growth Function

This function calculates the Earnings Per Share (EPS) Growth for a specified stock ticker using historical financial data. EPS Growth is a key indicator used by investors to assess a company's profitability growth on a per-share basis over time.

Purpose of the Function

The function provides a way to:

- Analyze the growth in a company's profitability on a per-share basis over two years.
- Compare the performance of different companies in terms of earnings growth.
- Assist in evaluating potential investments by considering historical profitability trends.

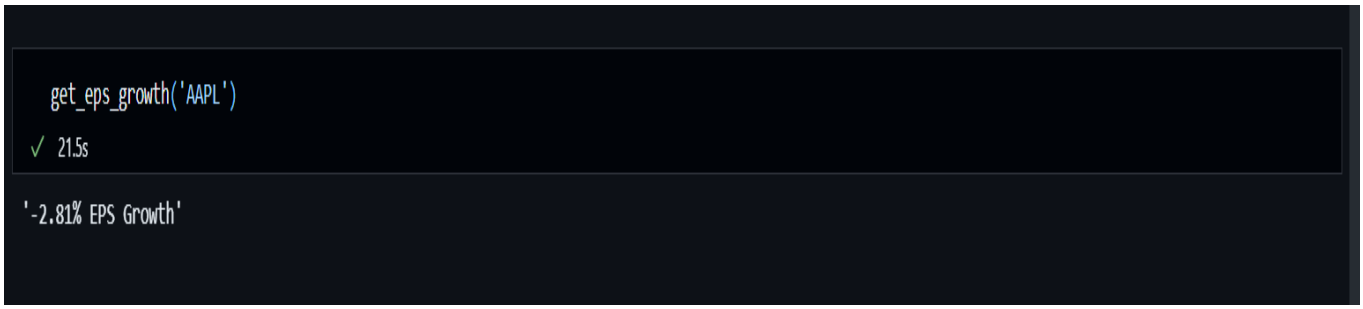
Source Code

```
def get_eps_growth(ticker):
    # Download stock data
    stock = yf.Ticker(ticker)
    try:
        # Fetch net income and shares outstanding data
        financials = stock.financials  # Annual financial statements
        net_income = financials.loc['Net Income'] if 'Net Income' in
financials.index else None
        shares_outstanding = stock.info.get("sharesOutstanding")

        # Ensure both values are available for EPS calculation
        if net_income is not None and shares_outstanding:
            # Calculate EPS for the last two years if available
            eps_last_year = net_income.iloc[1] / shares_outstanding
            eps_two_years_ago = net_income.iloc[2] / shares_outstanding
            # Calculate EPS growth
            eps_growth = ((eps_last_year - eps_two_years_ago) /
eps_two_years_ago) * 100
            return f"{eps_growth:.2f}% EPS Growth"
        else:
            return "Not enough data to calculate EPS growth."

    except Exception as e:
        return f"Error: Could not retrieve EPS growth data for {ticker}.
Details: {e}"
```

How the Function Works

1. **Input**:
    - ticker: The stock ticker symbol (e.g., "AAPL" for Apple Inc.).
2. **Fetching Data**:
    - **Financials**: Retrieves the company's annual financial statements using yfinance.
    - **Net Income**: Extracts the "Net Income" data from the financial statements.
    - **Shares Outstanding**: Retrieves the total number of shares outstanding from the stock's metadata.
3. **EPS Calculation**:
    - **Earnings Per Share (EPS)**: EPS=Net IncomeShares Outstanding\text{EPS} = \frac{\text{Net Income}}{\text{Shares Outstanding}}EPS=Shares OutstandingNet Income
    - The function calculates EPS for:
        - Last year (eps_last_year).
        - Two years ago (eps_two_years_ago).
4. **EPS Growth Calculation**:
    - The growth in EPS over the two-year period is calculated as:
      EPS Growth (%)=(EPS Last Year−EPS Two Years AgoEPS Two Years Ago)×100\text{EPS Growth (\%)} = \left(\frac{\text{EPS Last Year} - \text{EPS Two Years Ago}}{\text{EPS Two Years Ago}} \right) \times 100EPS Growth (%)=(EPS Two Years AgoEPS Last Year−EPS Two Years Ago)×100
5. **Data Validation**:
    - Checks if both **Net Income** and **Shares Outstanding** data are available. If not, returns a message indicating insufficient data.
6. **Error Handling**:
    - If an exception occurs (e.g., missing financial data), it returns a descriptive error message.
7. **Output**:
    - If successful, returns the EPS Growth as a percentage, rounded to two decimal places.
    - If data is missing, returns an appropriate error or message.

Use Case



get_revenue_growth_rate(ticker)

Function: get_revenue_growth_rate

The get_revenue_growth_rate function calculates the Revenue Growth Rate for a given stock ticker. Revenue Growth Rate measures the percentage increase (or decrease) in a company's revenue over the past two years, offering insights into its sales performance.

Similar Function: get_profit_growth_rate

The following function is similar but calculates the Profit Growth Rate instead, which is the percentage change in a company's net income (profit) over the last two years.

Source Code

```
def get_revenue_growth_rate(ticker):
    # Download stock data
    stock = yf.Ticker(ticker)
    try:
        # Fetch revenue data from the financials statement
        financials = stock.financials  # Annual financial statements
        revenue = financials.loc['Total Revenue'] if 'Total Revenue' in
financials.index else None

        # Ensure revenue data is available and has at least two years of data
        if revenue is not None and len(revenue) >= 2:
            # Calculate revenue growth rate based on the latest two years
            revenue_growth = ((revenue.iloc[0] - revenue.iloc[1]) /
revenue.iloc[1]) * 100
            return f"{revenue_growth:.2f}% Revenue Growth Rate"
        else:
            return "Not enough revenue data available to calculate growth
rate."

    except Exception as e:
        return f"Error: Could not retrieve revenue growth data for {ticker}.
Details: {e}"
```

Explanation of the get_profit_growth_rate Function

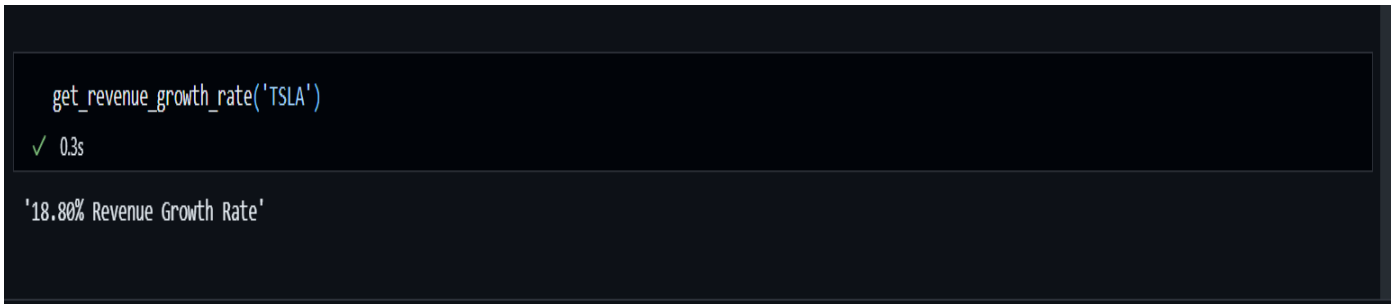
Purpose

This function calculates the **percentage change in net income** for a company over the past two years, offering insights into its profitability growth trajectory.

Steps in the Code

1. **Input**:
    - ticker: A stock symbol representing the company (e.g., "AAPL").
2. **Fetching Data**:
    - Uses yfinance to fetch the company's **financial statements**.
    - Retrieves **Net Income** data from the annual financials.
3. **Validation**:
    - Checks whether the **Net Income** data exists and has at least two years of entries.
4. **Calculation**:
    - Computes the growth rate as:
      Profit Growth Rate (%)=(Net Income (Current Year)−Net Income (Previous Year)Net I ncome (Previous Year))×100\text{Profit Growth Rate (\%)} = \left(\frac{\text{Net Income (Current Year)} - \text{Net Income (Previous Year)}}{\text{Net Income (Previous Year)}} \right) \times 100Profit Growth Rate (%)=(Net Income (Previous Year)Net Income (Current Year)−N et Income (Previous Year))×100
5. **Output**:
    - Returns the calculated **Profit Growth Rate** as a formatted string (e.g., "12.34% Profit Growth Rate").
    - If data is unavailable or an error occurs, it returns an appropriate message.
6. **Error Handling**:
    - Handles exceptions, such as missing financial data, and provides a descriptive error message.

Example Usage



get_eps_growth('AAPL')

Function: get_debt_to_equity_ratio

The get_debt_to_equity_ratio function calculates the Debt-to-Equity Ratio for a given stock ticker. This ratio measures a company's financial leverage by comparing its total liabilities to shareholders' equity, offering insights into the company's capital structure.

Explanation of the get_debt_to_equity_ratio Function

Purpose

This function calculates the Debt-to-Equity Ratio, which shows how much debt a company is using to finance its operations relative to its equity. A higher ratio may indicate greater financial risk.

Source Code

```
def get_debt_to_equity_ratio(ticker):
    # Fetch the stock's balance sheet data
    stock = yf.Ticker(ticker)
    balance_sheet = stock.balance_sheet

    # Print available row labels to identify the exact label names
    print("Available labels in balance sheet:", balance_sheet.index)

    # Set the labels based on available data
    total_liabilities_label = 'Total Liabilities Net Minority Interest'
    total_equity_label = 'Stockholders Equity'

    # Extract Total Liabilities and Shareholders' Equity
    try:
        total_liabilities = balance_sheet.loc[total_liabilities_label][0]
        total_shareholder_equity = balance_sheet.loc[total_equity_label][0]

        # Calculate Debt-to-Equity Ratio
        debt_to_equity_ratio = total_liabilities / total_shareholder_equity
        return debt_to_equity_ratio

    except KeyError as e:
        return f"Key error: {e}. Check if the label names have changed."
    except Exception as e:
        return f"An error occurred: {e}"
```

Code Explaination

1. **Input**:
    - The function takes a stock ticker as input (e.g., "AAPL") to identify the company.
2. **Fetching Data**:
    - Uses yfinance to fetch the **balance sheet** data of the specified stock.
3. **Debugging Support**:
    - Prints all available labels in the balance sheet to help ensure the correct fields are used for calculations.
    - Example output:

css

Copy code

Available labels in balance sheet: Index(['Total Liabilities Net Minority Interest', 'Stockholders Equity', ...])

4. **Setting Labels**:
    - Sets the labels for **Total Liabilities** and **Stockholders' Equity**.
    - These labels correspond to rows in the balance sheet and may vary depending on the data source.
5. **Extracting Financial Data**:
    - Retrieves:
        - Total Liabilities using the label "Total Liabilities Net Minority Interest".
        - Stockholders' Equity using the label "Stockholders Equity".
6. **Calculation**:
    - Computes the Debt-to-Equity Ratio as: Debt-to-Equity Ratio=Total LiabilitiesStockholders' Equity\text{Debt-to-Equity Ratio} = \frac{\text{Total Liabilities}}{\text{Stockholders' Equity}}Debt-to-Equity Ratio=Stockholders' EquityTotal Liabilities
7. **Output**:
    - Returns the calculated Debt-to-Equity Ratio as a numeric value.
8. **Error Handling**:
    - Handles KeyError if the specified label does not exist in the balance sheet, suggesting the user check for updated label names.
    - Catches any other exceptions and returns a descriptive error message.

```
get_debt_to_equity_ratio('AAPL')
```

✓ 0.1s

```
Available labels in balance sheet: Index(['Treasury Shares Number', 'Ordinary Shares Number', 'Share Issued',
       'Net Debt', 'Total Debt', 'Tangible Book Value', 'Invested Capital',
```

```
5.408779631255487
```

get_interest_coverage_ratio()

Function: get_interest_coverage_ratio

The get_interest_coverage_ratio function calculates the Interest Coverage Ratio for a given stock ticker. This ratio indicates how well a company can cover its interest obligations with its operating income, providing insight into the company's financial health.

Explanation of the get_interest_coverage_ratio Function

Purpose

The function calculates the Interest Coverage Ratio, which measures a company's ability to pay its interest expenses using its earnings before interest and taxes (EBIT). A higher ratio signifies stronger financial stability, while a lower ratio may indicate potential difficulty in meeting debt obligations.

Applications of the Function

1. **Financial Stability Analysis**:
    - Helps determine whether a company generates enough income to comfortably cover its interest expenses.
    - A higher ratio indicates less risk of default on debt payments.
2. **Investment Decisions**:
    - Investors use this metric to assess the financial health of a company before making investment decisions.
3. **Creditworthiness Evaluation**:
    - The ratio is commonly used by lenders and credit analysts to evaluate a company's ability to manage debt.

Source Code

```
def get_interest_coverage_ratio(ticker):
    # Fetch the stock's financial data
    stock = yf.Ticker(ticker)

    # Get the income statement data
    income_statement = stock.financials
```

```
    # Print available row labels in the income statement
    print("Available labels in income statement:", income_statement.index)

    try:
        # Extract EBIT (Operating Income) and Interest Expense
        ebit = income_statement.loc['Operating Income'][0]  # This is often
used as EBIT
        interest_expense = income_statement.loc['Interest Expense'][0]

        # Calculate Interest Coverage Ratio
        interest_coverage_ratio = ebit / interest_expense
        return interest_coverage_ratio

    except KeyError as e:
        return f"Key error: {e}. Check if the label names have changed."
    except Exception as e:
        return f"An error occurred: {e}"
```

Code Explanation

🔹 Input:

- **The function takes a stock ticker as input (e.g., "AAPL") to identify the company.**

🔹 Fetching Data:

- **Uses yfinance to fetch the income statement of the specified stock.**

🔹 Debugging Support:

- **Prints all available row labels in the income statement to help ensure the correct fields are used for calculations.**
- **Example output:**

css

Copy code

Available labels in income statement: Index(['Operating Income', 'Interest Expense', ...])

🔹 Extracting Financial Data:

- **Retrieves:**
    - **EBIT (Earnings Before Interest and Taxes), typically represented by the "Operating Income" row in the income statement.**
    - **Interest Expense, representing the cost incurred by the company for its borrowed funds.**

Calculation:

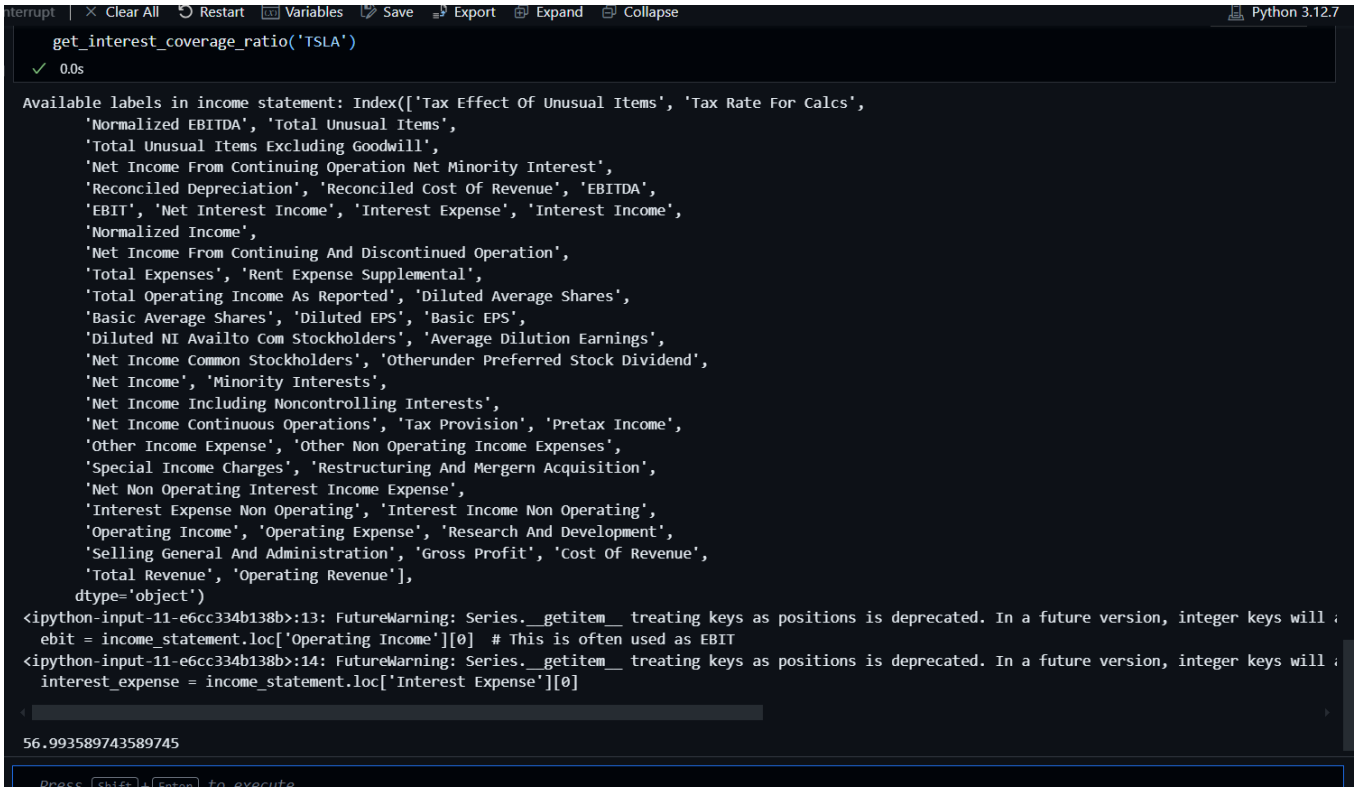
- **Computes the Interest Coverage Ratio as:**
  Interest Coverage Ratio=EBITInterest Expense\text{Interest Coverage Ratio} = \frac{\text{EBIT}}{\text{Interest Expense}}Interest Coverage Ratio=Interest ExpenseEBIT

Output:

- **Returns the calculated Interest Coverage Ratio as a numeric value.**

Error Handling:

- **Handles KeyError if the specified labels (e.g., "Operating Income" or "Interest Expense") are missing from the income statement, suggesting the user verify the label names.**
- **Catches any other exceptions and returns a descriptive error message.**



get_total_debt_to_total_assets_ratio

The get_total_debt_to_total_assets_ratio function calculates the **Total Debt to Total Assets Ratio** for a given stock ticker. This ratio is a measure of financial leverage, showing the proportion of a company's assets financed through debt.

Explanation of the get_total_debt_to_total_assets_ratio Function

Purpose

The function computes the **Total Debt to Total Assets Ratio**, which indicates the percentage of a company's total assets funded by its total debt. A higher ratio may imply greater financial risk, while a lower ratio suggests more reliance on equity financing.

Applications of the Function

1. **Risk Assessment**:
    - A high ratio may indicate a higher level of financial risk, as the company relies more on debt financing.
2. **Financial Analysis**:
    - The metric helps analysts and investors assess the company's capital structure and long-term financial health.
3. **Comparative Benchmarking**:
    - Useful for comparing financial leverage across companies in the same industry.
4. **Creditworthiness Evaluation**:
    - Lenders and credit analysts often examine this ratio to gauge the company's ability to manage its liabilities relative to its assets.

Key Points

- **Dynamic Labels**:
    - The function prints the balance sheet labels to accommodate potential differences in data naming conventions.
- **Error Handling**:
    - Provides user-friendly error messages to handle missing or mismatched data labels effectively.
- **Significance of the Metric**:
    - A **Total Debt to Total Assets Ratio** close to 1 indicates that most of the company's assets are financed by debt.

- A lower ratio indicates a healthier balance between debt and equity financing.

Summary

The get_total_debt_to_total_assets_ratio function calculates the ratio of a company's total debt to its total assets, fetched using yfinance. It helps users understand the financial leverage of the company, providing insight into its risk level and capital structure. The function is robust, accommodating potential data discrepancies and returning descriptive error messages when necessary.

Source Code

```
def get_total_debt_to_total_assets_ratio(ticker):
    # Fetch the stock's balance sheet data
    stock = yf.Ticker(ticker)
    balance_sheet = stock.balance_sheet

    # Print available row labels to identify the exact label names
    print("Available labels in balance sheet:", balance_sheet.index)

    try:
        # Extract Total Debt and Total Assets
        total_debt = balance_sheet.loc['Total Debt'][0]  # This often
represents the total liabilities
        total_assets = balance_sheet.loc['Total Assets'][0]

        # Calculate Total Debt to Total Assets Ratio
        total_debt_to_total_assets_ratio = total_debt / total_assets
        return total_debt_to_total_assets_ratio

    except KeyError as e:
        return f"Key error: {e}. Check if the label names have changed."
    except Exception as e:
        return f"An error occurred: {e}"
```

How the function works

Steps in the Code

1. **Input:**
    - **Accepts a stock ticker symbol (e.g., "AAPL") to identify the target company.**

2. **Fetching Data:**
    - **Utilizes yfinance to retrieve the balance sheet data for the specified stock.**
3. **Debugging Support:**
    - **Prints all available row labels in the balance sheet for reference.**
    - **Example output:**

css

Copy code

Available labels in balance sheet: Index(['Total Debt', 'Total Assets', ...])

4. **Extracting Financial Data:**
    - **Retrieves:**
        - **Total Debt, which includes all liabilities (short-term and long-term debt).**
        - **Total Assets, representing the total value of the company's owned resources.**
5. **Calculation:**
    - **Computes the Total Debt to Total Assets Ratio using the formula:**
      Total Debt to Total Assets Ratio=Total DebtTotal Assets\text{Total Debt to Total Assets Ratio} = \frac{\text{Total Debt}}{\text{Total Assets}}Total Debt to Total Assets Ratio=Total AssetsTotal Debt
6. **Output:**
    - **Returns the calculated ratio as a numeric value.**
7. **Error Handling:**
    - **Catches KeyError if the specified labels (e.g., "Total Debt" or "Total Assets") are not found in the balance sheet.**
    - **Handles other unexpected errors, providing a descriptive message.**

Example Usage

asset_turnover_ratio

The asset_turnover_ratio function calculates the **Asset Turnover Ratio** for a given stock ticker. This financial metric measures how efficiently a company uses its assets to generate revenue.

Explanation of the asset_turnover_ratio Function

Purpose

The function computes the **Asset Turnover Ratio**, which indicates how effectively a company utilizes its total assets to produce sales. A higher ratio suggests better efficiency, while a lower ratio might indicate underutilization of assets.

Applications of the Function

1. **Operational Efficiency Analysis**:
    - Helps investors and analysts assess how effectively a company generates revenue from its asset base.
2. **Comparative Benchmarking**:
    - Useful for comparing companies in the same industry to identify leaders in operational efficiency.
3. **Trend Analysis**:
    - Observing changes in the ratio over time can provide insights into how asset utilization improves or declines.
4. **Management Performance**:
    - Reflects the management's ability to maximize asset productivity.

Key Points

- **Significance of the Metric**:
    - A **higher ratio** indicates better efficiency in asset utilization.
    - A **lower ratio** may indicate excess or underperforming assets.
- **Dynamic Data Handling**:
    - Prints balance sheet labels dynamically to accommodate potential label differences.
    - Example of available labels:

Copy code

Available labels in balance sheet: Index(['Total Assets', ...])

- **Fallbacks for Missing Data**:
    - If only one year's total assets are available, the function adapts by using the most recent year's value as the average.

Limitations

1. **Data Availability**:
    - Relies on the presence of 'Total Revenue' and 'Total Assets' in the financial statements. Missing labels result in None.
2. **Industry Differences**:
    - The ratio may vary significantly by industry (e.g., capital-intensive vs. service industries), so direct comparisons may not always be valid.

Summary

The asset_turnover_ratio function calculates a company's ability to generate revenue relative to its asset base. It uses financial data fetched via yfinance, dynamically handles missing or incomplete data, and returns a numeric ratio or None for unavailable data. This metric is widely used in financial analysis to evaluate operational efficiency and asset management.

Source Code

```
def asset_turnover_ratio(ticker: str) -> float:
    # Fetch financial data
    stock = yf.Ticker(ticker)

    # Get the total revenue (net sales) for the last fiscal year
    financials = stock.financials
    revenue = financials.loc['Total Revenue'][0] if 'Total Revenue' in
financials.index else None

    # Get the total assets for the last two fiscal years to compute average
assets
    balance_sheet = stock.balance_sheet
    if 'Total Assets' in balance_sheet.index:
        total_assets_end = balance_sheet.loc['Total Assets'][0]
        total_assets_start = balance_sheet.loc['Total Assets'][1] if
balance_sheet.shape[1] > 1 else None
    else:
        return None  # Return None if 'Total Assets' not available

    # Calculate the average total assets
```

```
    if total_assets_start is not None:
        avg_total_assets = (total_assets_end + total_assets_start) / 2
    else:
        avg_total_assets = total_assets_end

    # Calculate Asset Turnover Ratio
    if revenue and avg_total_assets:
        asset_turnover_ratio = revenue / avg_total_assets
        return asset_turnover_ratio
    else:
        return None  # Return None if data is missing
```

Code Explanation

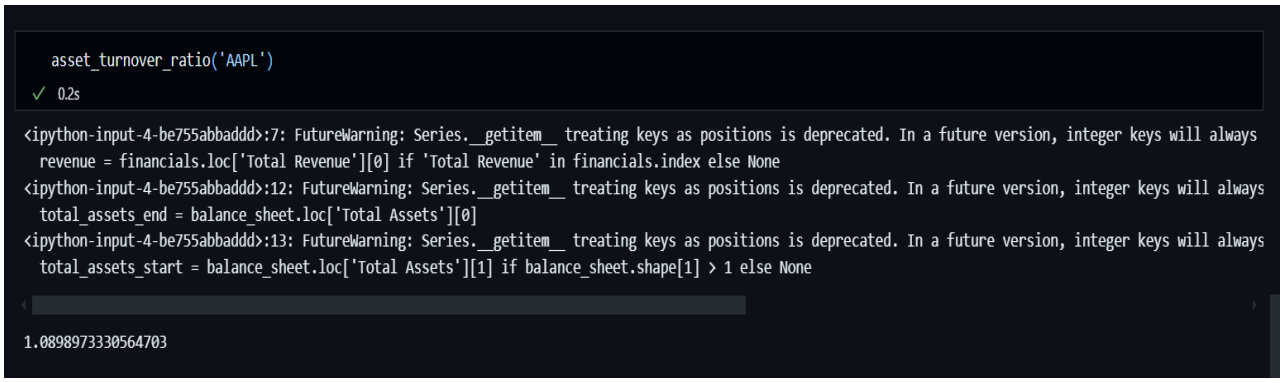
Steps in the Code

1. **Input**:
    - Accepts a stock ticker symbol (e.g., "AAPL") as the input.
2. **Fetching Data**:
    - Utilizes the yfinance library to fetch the company's financial data:
        - **Income Statement** (financials) for the Total Revenue (Net Sales).
        - **Balance Sheet** (balance_sheet) for the Total Assets.
3. **Data Extraction**:
    - Retrieves:
        - **Total Revenue** from the income statement ('Total Revenue' row).
        - **Total Assets** for the last two fiscal years from the balance sheet ('Total Assets' row).
4. **Average Total Assets Calculation**:
    - Computes the **average total assets** over the last two fiscal years using:
      Average Total Assets=Total Assets (End of Year)+Total Assets (Start of Year)2\text{Average Total Assets} = \frac{\text{Total Assets (End of Year)} + \text{Total Assets (Start of Year)}}{2}Average Total Assets=2Total Assets (End of Year)+Total Assets (Start of Year)
    - If only one year's data is available, it uses the most recent year's total assets.
5. **Asset Turnover Ratio Calculation**:
    - Calculates the Asset Turnover Ratio using:
      Asset Turnover Ratio=Total RevenueAverage Total Assets\text{Asset Turnover Ratio}

= \frac{\text{Total Revenue}}{\text{Average Total Assets}}Asset Turnover Ratio=Average Total AssetsTotal Revenue

6. **Output**:
    - Returns the calculated **Asset Turnover Ratio** as a numeric value.
    - Returns None if data required for the calculation is unavailable.
7. **Error Handling**:
    - Handles missing or incomplete data by returning None.

Use Case



Function: inventory_turnover_ratio

The inventory_turnover_ratio function calculates the **Inventory Turnover Ratio** for a given stock ticker. This financial metric measures how efficiently a company manages its inventory by analyzing how many times inventory is sold and replaced over a specific period.

Explanation of the inventory_turnover_ratio Function

Purpose

The function computes the **Inventory Turnover Ratio**, which reflects how effectively a company utilizes its inventory to generate revenue. A higher ratio indicates better inventory management, while a lower ratio suggests inefficiency or overstocking.

Applications of the Function

1. **Inventory Management Efficiency**:
    - Helps assess how quickly a company can convert inventory into sales, reducing holding costs and improving cash flow.
2. **Operational Benchmarking**:
    - Useful for comparing companies within the same industry to evaluate efficiency in managing inventory.
3. **Trend Analysis**:
    - Observing changes in the ratio over time can highlight improvements or declines in inventory management practices.
4. **Working Capital Analysis**:
    - Indicates how inventory contributes to a company's liquidity and operational cycle.

Key Points

- **Significance of the Metric**:
    - A **higher ratio** indicates efficient inventory turnover and strong sales performance.
    - A **lower ratio** may signal overstocking, weak sales, or inventory obsolescence.
- **Dynamic Data Handling**:
    - Prints labels dynamically to identify possible mismatches in financial statement terms.
    - Example:

css

Copy code

Available labels in balance sheet: Index(['Inventory', ...])

- **Fallback for Missing Data**:
    - Adapts to situations where only one year's inventory data is available by using the most recent value as the average.

Limitations

1. **Data Dependency**:
    - Relies on the presence of 'Cost Of Revenue' and 'Inventory' in the financial statements. Missing or misnamed labels result in None.
2. **Industry Variations**:
    - Ratios vary significantly across industries (e.g., retail vs. manufacturing). Industry context is essential for meaningful interpretation.
3. **COGS Representation**:
    - Assumes 'Cost Of Revenue' accurately represents COGS, which may vary by company reporting standards.

Summary

The inventory_turnover_ratio function calculates a company's efficiency in managing its inventory. It uses financial data fetched via yfinance, dynamically handles missing or incomplete data, and provides a numeric ratio or None if required data is unavailable. This metric is widely used for evaluating operational efficiency, inventory management, and financial health.

Source Code

```
def inventory_turnover_ratio(ticker: str) -> float:
    # Fetch financial data
    stock = yf.Ticker(ticker)

    # Get the Cost of Goods Sold (COGS) for the last fiscal year
    financials = stock.financials
    cogs = financials.loc['Cost Of Revenue'][0] if 'Cost Of Revenue' in
financials.index else None

    # Get the inventory values for the last two fiscal years to compute
average inventory
    balance_sheet = stock.balance_sheet
    if 'Inventory' in balance_sheet.index:
        inventory_end = balance_sheet.loc['Inventory'][0]
```

```
        inventory_start = balance_sheet.loc['Inventory'][1] if
balance_sheet.shape[1] > 1 else None
    else:
        return None  # Return None if 'Inventory' is not available

    # Calculate the average inventory
    if inventory_start is not None:
        avg_inventory = (inventory_end + inventory_start) / 2
    else:
        avg_inventory = inventory_end

    # Calculate Inventory Turnover Ratio
    if cogs and avg_inventory:
        inventory_turnover_ratio = cogs / avg_inventory
        return inventory_turnover_ratio
    else:
        return None  # Return None if data is missing
```

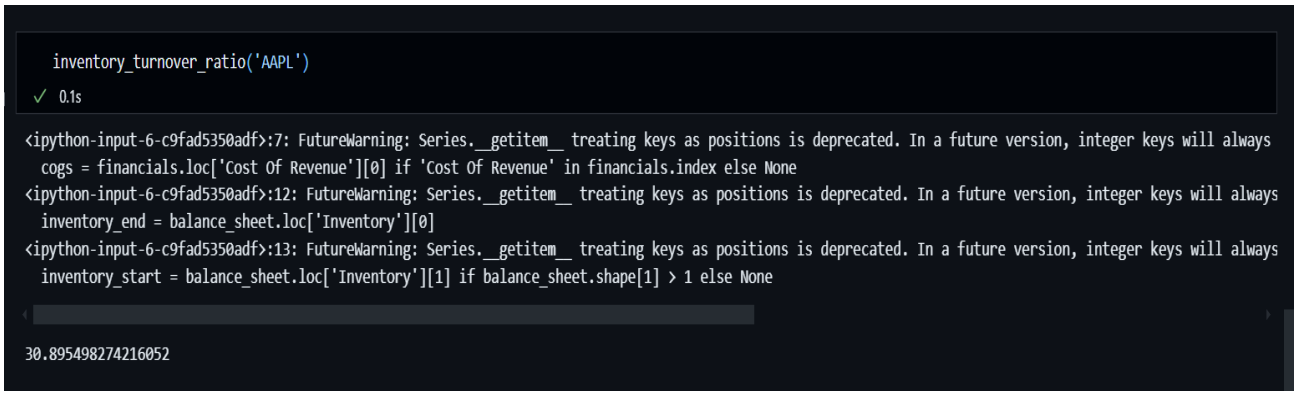
Code Explanation

Steps in the Code

1. **Input:**
    - **Accepts a stock ticker symbol (e.g., "AAPL") as the input.**
2. **Fetching Data:**
    - **Utilizes the yfinance library to fetch financial data:**
        - **Income Statement (financials) for the Cost of Goods Sold (COGS).**
        - **Balance Sheet (balance_sheet) for the Inventory values.**
3. **Data Extraction:**
    - **Retrieves:**
        - **COGS from the income statement ('Cost Of Revenue' row).**
        - **Inventory values for the last two fiscal years from the balance sheet ('Inventory' row).**
4. **Average Inventory Calculation:**
    - **Computes the average inventory using:**
      Average Inventory=Inventory (End of Year)+Inventory (Start of Year)2\text{Average Inventory} = \frac{\text{Inventory (End of Year)} + \text{Inventory (Start of Year)}}{2}Average Inventory=2Inventory (End of Year)+Inventory (Start of Year)
    - **If only one year's inventory data is available, the function uses the most recent year's inventory value as the average.**

5. **Inventory Turnover Ratio Calculation:**
    - **Calculates the ratio using:**
      Inventory Turnover Ratio=COGSAverage Inventory\text{Inventory Turnover Ratio} = \frac{\text{COGS}}{\text{Average Inventory}}Inventory Turnover Ratio=Average InventoryCOGS
6. **Output:**
    - **Returns the calculated Inventory Turnover Ratio as a numeric value.**
    - **Returns None if required data is unavailable.**
7. **Error Handling:**
    - **Handles missing or incomplete data gracefully by returning None.**

Example Usage



days_sales_outstanding

The days_sales_outstanding (DSO) function calculates the **average number of days** it takes a company to collect payment after making a sale. This metric is useful for analyzing a company's efficiency in managing accounts receivable and cash flow.

Explanation of the days_sales_outstanding Function

Purpose

The function determines how efficiently a company collects payments from customers by calculating the **Days Sales Outstanding (DSO)**. A lower DSO indicates prompt payment collection, while a higher DSO may suggest inefficiencies or potential collection issues.

Applications of the Function

1. **Efficiency Analysis**:
    - Helps evaluate how quickly a company converts credit sales into cash.
2. **Working Capital Management**:
    - Identifies potential issues in receivables collection that may impact liquidity.
3. **Industry Benchmarking**:
    - Provides insight into a company’s performance compared to peers in the same sector.
4. **Trend Monitoring**:
    - Tracks changes in DSO over time to detect improvements or declines in payment collection efficiency.

Key Points

- **Dynamic Label Matching**:
    - Searches for multiple possible labels for revenue and accounts receivable, accounting for variations in financial statement naming conventions.
    - Example debug output:

css

Copy code

Financials Data:

Index(['Total Revenue', 'Cost Of Revenue'], dtype='object')

Balance Sheet Data:

Index(['Net Receivables', 'Inventory', 'Total Assets'], dtype='object')

- **Adjustable Timeframe**:
    - The days parameter allows customization of the analysis period (e.g., 30 days for a monthly analysis).
- **Fallback for Missing Data**:
    - Returns None and provides debug output when either revenue or accounts receivable data is unavailable.

Limitations

1. **Data Dependency**:
    - Relies on the availability of 'Total Revenue' and 'Net Receivables' (or equivalent labels). Missing or alternative naming can affect results.
2. **Static Period Assumption**:
    - Assumes constant sales and receivables over the period, which may not reflect seasonal or cyclical variations.
3. **Industry Variations**:
    - DSOs vary widely across industries (e.g., technology vs. utilities). Proper context is needed for interpretation.
4. **Receivables Detail**:
    - Does not account for the aging of receivables, which can provide deeper insights into collection efficiency.

Summary

The days_sales_outstanding function calculates a company's efficiency in collecting payments by analyzing accounts receivable and revenue data. It is a valuable metric for evaluating cash flow, financial health, and operational efficiency. The function handles variations in financial statement labels dynamically and provides debugging outputs to assist with data availability issues.

Source Code

```
def days_sales_outstanding(ticker: str, days: int = 365) -> float:
    # Fetch financial data
    stock = yf.Ticker(ticker)

    # Get the total revenue (net sales) for the last fiscal year
    financials = stock.financials
    print("Financials Data:\n", financials)  # Print financials data to debug

    # Check for alternative names for total revenue
    revenue = None
    if 'Total Revenue' in financials.index:
        revenue = financials.loc['Total Revenue'].iloc[0]
    elif 'Revenues' in financials.index:
        revenue = financials.loc['Revenues'].iloc[0]
    elif 'Net Sales' in financials.index:
        revenue = financials.loc['Net Sales'].iloc[0]
    else:
        print(f"Total Revenue or equivalent not found for {ticker}")

    # Get the accounts receivable for the last fiscal year from the balance
sheet
    balance_sheet = stock.balance_sheet
    print("Balance Sheet Data:\n", balance_sheet)  # Print balance sheet data
to debug

    # Check for alternative names for accounts receivable
    accounts_receivable = None
    if 'Net Receivables' in balance_sheet.index:
        accounts_receivable = balance_sheet.loc['Net Receivables'].iloc[0]
    elif 'Receivables' in balance_sheet.index:
        accounts_receivable = balance_sheet.loc['Receivables'].iloc[0]
    elif 'Trade Receivables' in balance_sheet.index:
        accounts_receivable = balance_sheet.loc['Trade Receivables'].iloc[0]
    else:
        print(f"Net Receivables or equivalent not found for {ticker}")

    # Calculate DSO if data is available
    if revenue and accounts_receivable:
        dso = (accounts_receivable / revenue) * days
        return dso
    else:
        return None  # Return None if data is missing
```

Code Explanation

□ Input:

- **Ticker: The stock symbol of the company (e.g., "AAPL") to fetch financial data.**
- **Days: The number of days in a period (default: 365 days, representing a year).**

□ Fetching Data:

- **Uses the yfinance library to retrieve:**
  - **Income Statement (financials) for Total Revenue (or equivalent).**
  - **Balance Sheet for Accounts Receivable (or equivalent).**

□ Revenue Extraction:

- **Searches for Total Revenue under different potential labels:**
  - **'Total Revenue', 'Revenues', 'Net Sales'.**
- **Retrieves the value if available or prints a message if not found.**

□ Accounts Receivable Extraction:

- **Searches for Accounts Receivable under different potential labels:**
  - **'Net Receivables', 'Receivables', 'Trade Receivables'.**
- **Retrieves the value if available or prints a message if not found.**

□ DSO Calculation:

- **If both revenue and accounts receivable are available, calculates the DSO using: DSO=(Accounts ReceivableRevenue)×Days\text{DSO} = \left(\frac{\text{Accounts Receivable}}{\text{Revenue}}\right) \times \text{Days}DSO=(RevenueAccounts Receivable)×Days**
- **Returns None if any required data is missing.**

□ Debugging Output:

- **Prints financial statement data to assist in identifying potential mismatches in label names.**

□ Error Handling:

- **Handles missing data gracefully by returning None and printing relevant messages.**

Use Case

```
days_sales_outstanding('AAPL', days=365)
```

✓ 0.5s

```
Financials Data:
                                                  2024-09-30  \
Tax Effect Of Unusual Items                              0.0
Tax Rate For Calcs                                  0.240912
Normalized EBITDA                             134661000000.0
Net Income From Continuing Operation Net Minori...  93736000000.0
Reconciled Depreciation                       11445000000.0
Reconciled Cost Of Revenue                    210352000000.0
EBITDA                                        134661000000.0
EBIT                                          123216000000.0
Net Interest Income                                      NaN
Interest Expense                                         NaN
Interest Income                                          NaN
Normalized Income                              93736000000.0
Net Income From Continuing And Discontinued Ope...  93736000000.0
Total Expenses                                267819000000.0
Total Operating Income As Reported            123216000000.0
Diluted Average Shares                                   NaN
Basic Average Shares                                     NaN
Diluted EPS                                              NaN
Basic EPS                                                NaN
Diluted NI Availto Com Stockholders            93736000000.0
Net Income Common Stockholders                 93736000000.0
Net Income                                     93736000000.0
Net Income Including Noncontrolling Interests  93736000000.0
...
Cash Equivalents                              17635000000.0        NaN
Cash Financial                                17305000000.0        NaN
```

```
61.83255974529135
```

get_detailed_financials()

The get_detailed_financials function retrieves the detailed financial statements for a specified company using Yahoo Finance data. This provides a comprehensive overview of the company's financial performance, including key income statement metrics such as revenue, expenses, and profits.

Explanation of the get_detailed_financials Function

Purpose

The function fetches detailed financial data for a given company using the yfinance library, making it easy to access essential financial information such as Total Revenue, Operating Income, Net Income, and other key metrics.

Applications of the Function

1. **Financial Analysis**:
    - Provides a comprehensive view of a company's revenue, expenses, and profit trends over time.
2. **Investor Insights**:
    - Helps investors evaluate a company's financial health and profitability before making investment decisions.
3. **Corporate Benchmarking**:
    - Enables comparison of financial metrics across different companies in the same industry.
4. **Data Preparation**:
    - Supplies structured financial data for advanced analysis, such as ratio calculations or time-series modeling.

Key Features

1. **Data Retrieval**:
    - Retrieves data directly from Yahoo Finance, ensuring up-to-date and reliable financial metrics.
2. **Ease of Use**:
    - Simple to call and immediately returns financials in a structured DataFrame format.
3. **Versatility**:

- Works for any publicly traded company as long as the stock ticker is valid.

Limitations

1. **Data Availability**:
    - Relies on Yahoo Finance's data. If financials are unavailable for the specified ticker, the function may return an empty DataFrame.
2. **Static Data**:
    - Retrieves pre-compiled financial statements and does not offer real-time updates or custom calculations.
3. **Financial Statement Focus**:
    - Only fetches the **income statement** data. For balance sheet or cash flow data, additional attributes like stock.balance_sheet or stock.cashflow must be used.

Source Code

```
def get_detailed_financials(ticker: str):
    # Fetch the stock data from Yahoo Finance
    stock = yf.Ticker(ticker)

    # Get the financials data
    financials = stock.financials

    # Return the financials data
    return financials

# Example usage:
#get_detailed_financials('AAPL')
```

Code Explanation

1. **Input:**
    - **Ticker: A string representing the stock symbol of the company (e.g., 'AAPL' for Apple).**
2. **Fetching Data:**
    - **The function creates an instance of the yf.Ticker class for the specified stock ticker.**
    - **Calls the financials attribute of the yf.Ticker object to fetch the company's income statement data.**

3. **Return Value:**
    - **Returns the retrieved financial data in a pandas DataFrame format for further analysis or display.**

Use Case

```
get_detailed_financials('AAPL')
```
✓ 0.4s

	2024-09-30	2023-09-30	2022-09-30	2021-09-30	2020-09-30
Tax Effect Of Unusual Items	0.0	0.0	0.0	0.0	NaN
Tax Rate For Calcs	0.240912	0.147192	0.162	0.133	NaN
Normalized EBITDA	134661000000.0	125820000000.0	130541000000.0	123136000000.0	NaN
Net Income From Continuing Operation Net Minority Interest	93736000000.0	96995000000.0	99803000000.0	94680000000.0	NaN
Reconciled Depreciation	11445000000.0	11519000000.0	11104000000.0	11284000000.0	NaN
Reconciled Cost Of Revenue	210352000000.0	214137000000.0	223546000000.0	212981000000.0	NaN
EBITDA	134661000000.0	125820000000.0	130541000000.0	123136000000.0	NaN
EBIT	123216000000.0	114301000000.0	119437000000.0	111852000000.0	NaN
Net Interest Income	NaN	-183000000.0	-106000000.0	198000000.0	890000000.0
Interest Expense	NaN	3933000000.0	2931000000.0	2645000000.0	2873000000.0
Interest Income	NaN	3750000000.0	2825000000.0	2843000000.0	3763000000.0
Normalized Income	93736000000.0	96995000000.0	99803000000.0	94680000000.0	NaN
Net Income From Continuing And Discontinued Operation	93736000000.0	96995000000.0	99803000000.0	94680000000.0	NaN
Total Expenses	267819000000.0	268984000000.0	274891000000.0	256868000000.0	NaN
Total Operating Income As Reported	123216000000.0	114301000000.0	119437000000.0	108949000000.0	NaN
Diluted Average Shares	NaN	15812547000.0	16325819000.0	16864919000.0	17528214000.0
Basic Average Shares	NaN	15744231000.0	16215963000.0	16701272000.0	17352119000.0
Diluted EPS	NaN	6.13	6.11	5.61	3.28
Basic EPS	NaN	6.16	6.15	5.67	3.31
Diluted NI Availto Com Stockholders	93736000000.0	96995000000.0	99803000000.0	94680000000.0	NaN

Interrupt | Clear All | Restart | Variables | Save | Export | Expand | Collapse | Python 3.12.7

Basic Average Shares	NaN	15744231000.0	16215963000.0	16701272000.0	17352119000.0
Diluted EPS	NaN	6.13	6.11	5.61	3.28
Basic EPS	NaN	6.16	6.15	5.67	3.31
Diluted NI Availto Com Stockholders	93736000000.0	96995000000.0	99803000000.0	94680000000.0	NaN
Net Income Common Stockholders	93736000000.0	96995000000.0	99803000000.0	94680000000.0	NaN
Net Income	93736000000.0	96995000000.0	99803000000.0	94680000000.0	NaN
Net Income Including Noncontrolling Interests	93736000000.0	96995000000.0	99803000000.0	94680000000.0	NaN
Net Income Continuous Operations	93736000000.0	96995000000.0	99803000000.0	94680000000.0	NaN
Tax Provision	29749000000.0	16741000000.0	19300000000.0	14527000000.0	NaN
Pretax Income	123485000000.0	113736000000.0	119103000000.0	109207000000.0	NaN
Other Income Expense	269000000.0	-565000000.0	-334000000.0	60000000.0	NaN
Other Non Operating Income Expenses	269000000.0	-565000000.0	-334000000.0	60000000.0	NaN
Net Non Operating Interest Income Expense	NaN	-183000000.0	-106000000.0	198000000.0	890000000.0
Interest Expense Non Operating	NaN	3933000000.0	2931000000.0	2645000000.0	2873000000.0
Interest Income Non Operating	NaN	3750000000.0	2825000000.0	2843000000.0	3763000000.0
Operating Income	123216000000.0	114301000000.0	119437000000.0	108949000000.0	NaN
Operating Expense	57467000000.0	54847000000.0	51345000000.0	43887000000.0	NaN
Research And Development	31370000000.0	29915000000.0	26251000000.0	21914000000.0	NaN
Selling General And Administration	26097000000.0	24932000000.0	25094000000.0	21973000000.0	NaN
Gross Profit	180683000000.0	169148000000.0	170782000000.0	152836000000.0	NaN
Cost Of Revenue	210352000000.0	214137000000.0	223546000000.0	212981000000.0	NaN
Total Revenue	391035000000.0	383285000000.0	394328000000.0	365817000000.0	NaN
Operating Revenue	391035000000.0	383285000000.0	394328000000.0	365817000000.0	NaN

get_dividend_yield()

The get_dividend_yield function retrieves the dividend yield of a specified stock using Yahoo Finance. The dividend yield is expressed as a percentage, representing the annual dividend payment relative to the stock's current price.

Explanation of the get_dividend_yield Function

Purpose

This function calculates the dividend yield for a given company's stock, providing insight into the stock's income-generating potential. It is a key metric for income-focused investors.

Applications of the Function

1. **Income Investing**:
    - Helps investors focus on stocks with higher dividend yields, which may provide regular income.
2. **Portfolio Analysis**:
    - Enables evaluation of income potential across multiple stocks in a portfolio.
3. **Screening Dividend Stocks**:
    - Assists in filtering stocks with desirable yields for further analysis.

Key Features

1. **Simplicity**:
    - Easy to use with just the stock's ticker symbol as input.
2. **Real-Time Data**:
    - Fetches up-to-date dividend yield information directly from Yahoo Finance.
3. **Error Handling**:
    - Gracefully handles cases where data is unavailable or errors occur during data fetching.
4. **Versatility**:
    - Can be integrated into larger financial analysis workflows.

Limitations

1. **Data Availability**:

- The dividendYield field may not be available for all stocks, especially non-dividend-paying companies.

2. **Static Metric**:
   - Provides only the current dividend yield, not historical trends or projections.
3. **Dependent on Yahoo Finance**:
   - Relies on Yahoo Finance's API, which may occasionally have data discrepancies or outages.

Source Code

```
def get_dividend_yield(ticker_symbol):
    """
    Fetches the dividend yield of a stock using the yfinance library.

    Parameters:
        ticker_symbol (str): The stock ticker symbol (e.g., 'AAPL', 'MSFT').

    Returns:
        float: The dividend yield as a percentage, or None if no dividend
yield is available.
    """
    try:
        stock = yf.Ticker(ticker_symbol)
        dividend_yield = stock.info.get('dividendYield')

        if dividend_yield is not None:
            return round(dividend_yield * 100, 2)  # Convert to percentage
        else:
            print(f"No dividend yield data available for {ticker_symbol}.")
            return None
    except Exception as e:
        print(f"Error fetching data for {ticker_symbol}: {e}")
        return None
```

Code Explanation

Steps in the Code

1. Input:

- Ticker Symbol: A string representing the stock's ticker symbol (e.g., 'AAPL' for Apple).

2. Data Fetching:
    - The function uses the yf.Ticker class from the yfinance library to fetch information about the specified stock.
    - It retrieves the dividendYield field from the stock's info dictionary.
3. Dividend Yield Processing:
    - If the dividendYield data is available, it is converted from a decimal to a percentage (e.g., 0.015 becomes 1.5%) and rounded to two decimal places.
    - If no data is available, the function prints a message indicating the absence of dividend yield information.
4. Error Handling:
    - Any issues during the data-fetching process are caught by an Exception block, and an error message is displayed.
5. Return Value:
    - The dividend yield as a percentage (e.g., 1.5 for 1.5%), or None if the data is unavailable or an error occurs.

Example Use Case

```
get_dividend_yield('AAPL')
✓ 0.7s
0.41
```

get_dividend_payout_ratio

The get_dividend_payout_ratio function retrieves the dividend payout ratio of a specified stock using Yahoo Finance. The dividend payout ratio indicates the proportion of earnings a company distributes as dividends to shareholders, expressed as a percentage.

Explanation of the get_dividend_payout_ratio Function

Purpose

This function calculates the dividend payout ratio, a crucial metric for evaluating how much of a company's earnings are being paid out as dividends. It helps investors understand a company's dividend policy and sustainability.

Applications of the Function

1. **Dividend Policy Analysis**:
    - Understand how much of a company's earnings are allocated to dividends and retained for growth.
2. **Sustainability Check**:
    - Assess whether a company's dividend payouts are sustainable, especially during economic downturns.
3. **Investment Decisions**:
    - Compare payout ratios across companies to identify those with stable and consistent dividend policies.

Key Features

1. **Simple Input**:
    - Requires only the stock's ticker symbol to retrieve data.
2. **Real-Time Data**:
    - Provides the latest dividend payout ratio from Yahoo Finance.
3. **Error Handling**:
    - Handles cases where data is unavailable or issues arise during data fetching.
4. **Versatile Integration**:
    - Can be used alongside other financial metrics for comprehensive stock analysis.

Limitations

1. **Data Availability**:
    - The payoutRatio field may not be available for all companies, particularly those that do not pay dividends.
2. **Static Metric**:
    - Reflects only the most recent dividend payout ratio, without historical or trend data.
3. **Dependent on Yahoo Finance**:
    - Relies on Yahoo Finance's API, which may occasionally have data discrepancies or outages.

Source Code

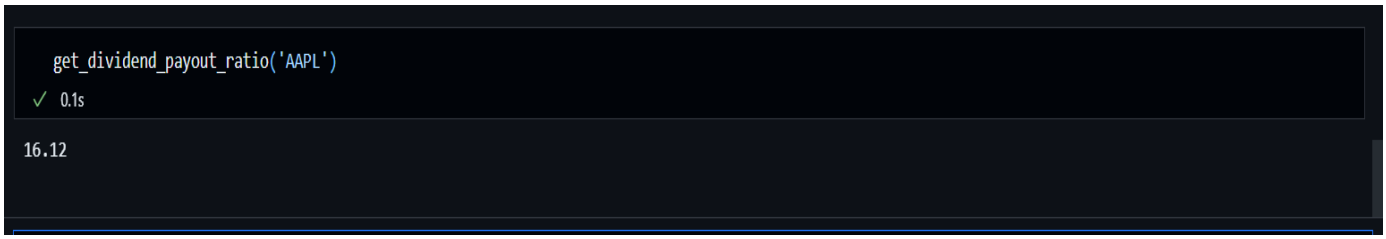
```
def get_dividend_payout_ratio(ticker_symbol):
    """
    Fetches the dividend payout ratio of a stock using the yfinance library.

    Parameters:
        ticker_symbol (str): The stock ticker symbol (e.g., 'AAPL', 'MSFT').

    Returns:
        float: The dividend payout ratio as a percentage, or None if no data
is available.
    """
    try:
        stock = yf.Ticker(ticker_symbol)
        payout_ratio = stock.info.get('payoutRatio')

        if payout_ratio is not None:
            return round(payout_ratio * 100, 2)  # Convert to percentage
        else:
            print(f"No dividend payout ratio data available for
{ticker_symbol}.")
            return None
    except Exception as e:
        print(f"Error fetching data for {ticker_symbol}: {e}")
        return None
```

Code Explanation

Steps in the Code

1. Input:
    - Ticker Symbol: A string representing the stock's ticker symbol (e.g., 'AAPL' for Apple).
2. Data Fetching:
    - The function uses the yf.Ticker class from the yfinance library to retrieve the stock's information.
    - It extracts the payoutRatio field from the stock's info dictionary.
3. Dividend Payout Ratio Processing:
    - If the payoutRatio data is available, it is converted from a decimal to a percentage (e.g., 0.25 becomes 25%) and rounded to two decimal places.
    - If no data is available, the function prints a message indicating the absence of dividend payout ratio information.
4. Error Handling:
    - Any issues during data fetching are caught by an Exception block, and an error message is displayed.
5. Return Value:
    - The dividend payout ratio as a percentage (e.g., 25.0 for 25%), or None if the data is unavailable or an error occurs.

Example Usage



get_dividend_growth_rate()

The get_dividend_growth_rate function calculates the compound annual growth rate (CAGR) of a stock's dividend over a specified number of years. This metric helps investors evaluate how consistently and rapidly a company has increased its dividends.

Explanation of the get_dividend_growth_rate Function

Purpose

This function determines the historical growth rate of a stock's dividends over a given time frame, offering insight into the company's financial health and commitment to rewarding shareholders.

Applications of the Function

1. **Long-Term Dividend Trends**:
    - Assess how consistently a company increases dividends over time.
2. **Comparison Across Companies**:
    - Compare the dividend growth rates of multiple companies to identify those with strong dividend policies.
3. **Income Investing**:
    - Use growth rate data to select stocks for dividend income portfolios.

Key Features

1. **Custom Time Frame**:
    - Allows users to specify the number of years for the calculation.
2. **Real-Time Data**:
    - Fetches the latest dividend data directly from Yahoo Finance.
3. **Accurate CAGR Calculation**:
    - Adjusts for the actual time span of the data to ensure accuracy.
4. **Error Handling**:
    - Handles cases where data is unavailable, incomplete, or invalid.

Limitations

1. **Data Availability**:

- The function relies on Yahoo Finance's historical dividend data, which may be missing or incomplete for some companies.

2. **Assumption of Regular Dividends**:
   - Assumes consistent dividend payouts; irregular or skipped dividends can distort the growth rate.
3. **Dependent on yfinance**:
   - Requires the yfinance library and an internet connection to fetch data.

Source Code

```
def get_dividend_growth_rate(ticker_symbol, years=5):
    """
    Calculates the dividend growth rate of a stock over a given number of
years using yfinance.

    Parameters:
        ticker_symbol (str): The stock ticker symbol (e.g., 'AAPL', 'MSFT').
        years (int): The number of years over which to calculate the growth
rate.

    Returns:
        float: The dividend growth rate as a percentage, or None if data is
unavailable.
    """
    try:
        stock = yf.Ticker(ticker_symbol)
        # Fetch historical dividend data
        dividends = stock.dividends

        if dividends.empty:
            print(f"No dividend data available for {ticker_symbol}.")
            return None

        # Filter dividends for the last 'years' years
        recent_dividends = dividends.last(f'{years}Y')

        if len(recent_dividends) < 2:
            print(f"Not enough dividend data for {ticker_symbol} to calculate
growth rate.")
            return None
```

```
        # Calculate CAGR for dividends
        first_dividend = recent_dividends[0]
        last_dividend = recent_dividends[-1]
        num_years = (recent_dividends.index[-1] -
recent_dividends.index[0]).days / 365.25

        if first_dividend > 0 and last_dividend > 0:
            cagr = ((last_dividend / first_dividend) ** (1 / num_years) - 1) *
100
            return round(cagr, 2)
        else:
            print(f"Dividend values are invalid for {ticker_symbol}.")
            return None
    except Exception as e:
        print(f"Error fetching data for {ticker_symbol}: {e}")
        return None
```

Code Explanation

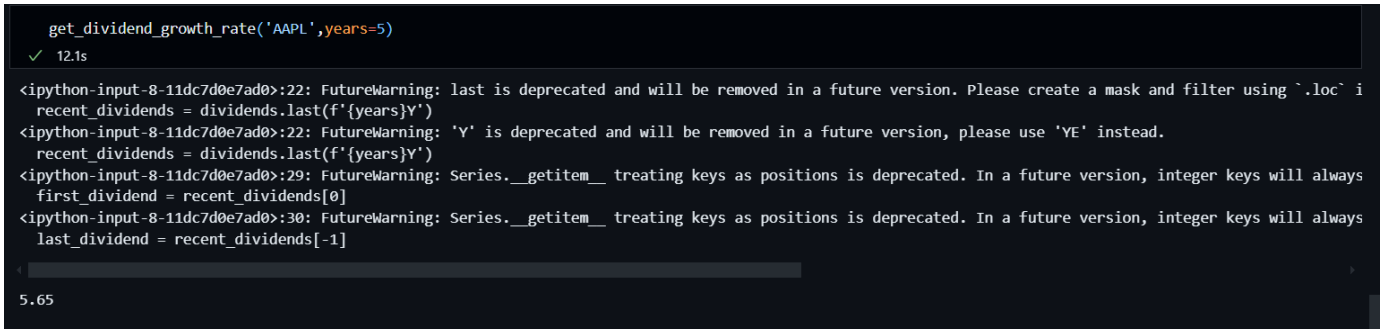
Steps in the Code

1. Input:
    - ticker_symbol: A string representing the stock's ticker symbol (e.g., 'AAPL' for Apple).
    - years: The number of years over which to calculate the dividend growth rate (default is 5).
2. Data Fetching:
    - The yf.Ticker class from the yfinance library retrieves the stock's historical dividend payments using the dividends attribute.
3. Data Validation:
    - If no dividend data is found, or if there are fewer than two data points in the specified period, the function returns None with an appropriate message.
4. Filter Recent Data:
    - The function narrows down the dividend data to the last years years using the last method.
5. Calculate CAGR:
    - Using the formula for CAGR:
      CAGR=(Last DividendFirst Dividend)1Number of Years−1\text{CAGR} = \left(\frac{\text{Last Dividend}}{\text{First Dividend}}\right)^{\frac{1}{\text{Number of Years}}} - 1CAGR=(First DividendLast Dividend)Number of Years1−1

- Converts the CAGR into a percentage and rounds it to two decimal places.

6. Error Handling:
    - If any issues arise during data fetching or processing, an error message is displayed.
7. Return Value:
    - The dividend growth rate as a percentage (e.g., 5.67 for 5.67%), or None if insufficient data is available or an error occurs.

Example Usage



get_market_cap()

The get_market_cap function retrieves the market capitalization of a stock from Yahoo Finance and formats it into a human-readable string. Market capitalization is a measure of a company's total value in the stock market and is calculated as the total number of outstanding shares multiplied by the share price.

Explanation of the get_market_cap Function

Purpose

This function provides a quick, user-friendly way to determine the market cap of a stock and understand the company's size category (e.g., small-cap, mid-cap, large-cap, etc.).

Applications of the Function

1. **Company Valuation**:
    - Helps investors quickly gauge the total value of a company.
2. **Comparison Across Stocks**:
    - Compare the market caps of multiple companies to identify large-cap, mid-cap, or small-cap stocks.
3. **Portfolio Diversification**:
    - Use market cap data to diversify a portfolio by including stocks of various sizes.
4. **Stock Screening**:
    - Identify companies with specific market cap thresholds (e.g., only large-cap stocks).

Key Features

1. **Human-Readable Output**:
    - Formats large numbers into Trillions, Billions, or Millions for easier interpretation.
2. **Real-Time Data**:
    - Fetches the most recent market cap directly from Yahoo Finance.
3. **Error Handling**:
    - Provides feedback in cases of missing data or errors during execution.

Limitations

1. **Data Availability**:

- The function depends on Yahoo Finance's marketCap data, which may not always be available for certain stocks.

2. **Assumption of Currency**:
   - Assumes that the market cap is denominated in USD. For international stocks, currency variations might need adjustment.
3. **Dependent on yfinance**:
   - Requires the yfinance library and an internet connection to retrieve stock data.

Source Code

```
def get_market_cap(ticker_symbol):
    """
    Fetches the market capitalization of a stock using the yfinance library.

    Parameters:
        ticker_symbol (str): The stock ticker symbol (e.g., 'AAPL', 'MSFT').

    Returns:
        str: The market capitalization formatted as a human-readable string,
             or None if no data is available.
    """
    try:
        stock = yf.Ticker(ticker_symbol)
        market_cap = stock.info.get('marketCap')

        if market_cap is not None:
            # Format the market cap into billions/trillions for readability
            if market_cap >= 1e12:
                formatted_cap = f"${market_cap / 1e12:.2f} Trillion"
            elif market_cap >= 1e9:
                formatted_cap = f"${market_cap / 1e9:.2f} Billion"
            elif market_cap >= 1e6:
                formatted_cap = f"${market_cap / 1e6:.2f} Million"
            else:
                formatted_cap = f"${market_cap:.2f}"

            return formatted_cap
        else:
            print(f"No market capitalization data available for
{ticker_symbol}.")
            return None
    except Exception as e:
```

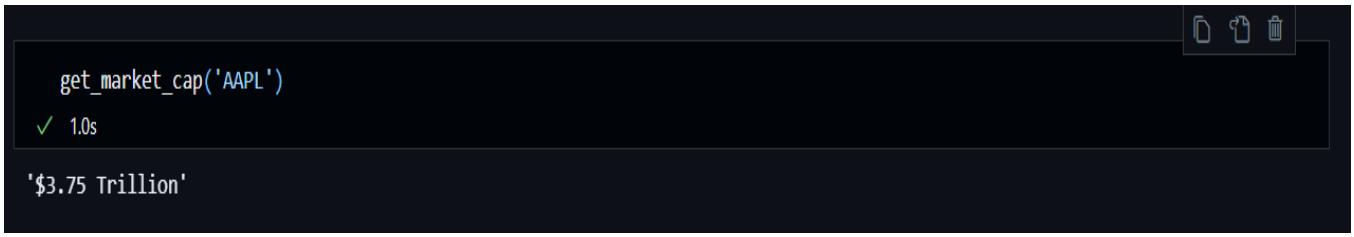
```
        print(f"Error fetching data for {ticker_symbol}: {e}")
        return None
```

Code Explanation

Steps in the Code

1. **Input**:
    - **ticker_symbol**: A string representing the stock's ticker symbol (e.g., 'AAPL' for Apple).
2. **Data Fetching**:
    - Uses the yf.Ticker class from the yfinance library to fetch the stock's information.
    - Retrieves the marketCap value from the stock's data using the info dictionary.
3. **Data Validation**:
    - Checks whether the marketCap data is available. If it is missing, a message is displayed, and the function returns None.
4. **Formatting**:
    - Converts the market cap into a readable format:
        - Values ≥ $1 trillion are formatted as "Trillion."
        - Values ≥ $1 billion are formatted as "Billion."
        - Values ≥ $1 million are formatted as "Million."
        - Values below $1 million are displayed as the exact value.
5. **Error Handling**:
    - If an error occurs while fetching data, the function prints an error message and returns None.
6. **Return Value**:
    - The formatted market capitalization as a string (e.g., $2.57 Trillion).
    - Returns None if the market cap data is unavailable or an error occurs.

Usecase



get_beta()

The get_beta function retrieves the Beta value of a stock from Yahoo Finance using the yfinance library. Beta is a measure of a stock's volatility relative to the overall market and is commonly used to assess investment risk.

Explanation of the get_beta Function

Purpose

This function provides the Beta value of a stock, helping investors understand how the stock's price movements are expected to correlate with market trends.

Applications of the Function

1. **Risk Assessment**:
    - Beta helps determine how much risk a stock carries compared to the overall market.
        - **Beta > 1**: Stock is more volatile than the market.
        - **Beta < 1**: Stock is less volatile than the market.
        - **Beta = 1**: Stock moves in line with the market.
2. **Portfolio Diversification**:
    - By knowing the Beta of individual stocks, investors can balance their portfolios with a mix of high- and low-Beta stocks.
3. **Investment Strategy**:
    - High-Beta stocks might appeal to growth-focused investors, while low-Beta stocks may be suitable for risk-averse investors.
4. **Financial Analysis**:
    - Beta is a critical input in calculating a stock's **cost of equity** and is widely used in valuation models like the Capital Asset Pricing Model (CAPM).

Key Features

1. **Real-Time Data**:
    - Fetches the most recent Beta value from Yahoo Finance.
2. **Error Handling**:
    - Provides descriptive error messages if data fetching fails.
3. **Easy Interpretation**:
    - Returns a simple numerical value that is rounded for clarity.

Limitations

1. **Data Availability**:
    - Beta data may not be available for some stocks, especially those with limited trading history or smaller market capitalization.
2. **Assumes Stability**:
    - The function retrieves a single Beta value, which may not reflect changes in volatility over time.
3. **Dependent on yfinance**:
    - Requires the yfinance library and an active internet connection.

Source Code

```
def get_beta(ticker_symbol):
    """
    Fetches the Beta (volatility measure) of a stock using the yfinance
library.

    Parameters:
        ticker_symbol (str): The stock ticker symbol (e.g., 'AAPL', 'MSFT').

    Returns:
        float: The Beta value, or None if no data is available.
    """
    try:
        stock = yf.Ticker(ticker_symbol)
        beta = stock.info.get('beta')

        if beta is not None:
            return round(beta, 2)
        else:
            print(f"No Beta data available for {ticker_symbol}.")
            return None
    except Exception as e:
        print(f"Error fetching data for {ticker_symbol}: {e}")
        return None
```

Code Explanation

Steps in the Code

1. **Input**:
    - **ticker_symbol**: A string representing the stock's ticker symbol (e.g., 'AAPL' for Apple).
2. **Data Fetching**:
    - Uses the yf.Ticker class from the yfinance library to fetch the stock's information.
    - Retrieves the Beta value from the stock's info dictionary using the key 'beta'.
3. **Data Validation**:
    - Checks if the Beta value is available in the retrieved data.
    - If the Beta value is missing, the function prints a message and returns None.
4. **Data Formatting**:
    - If the Beta value is present, it is rounded to two decimal places for clarity and returned.
5. **Error Handling**:
    - Catches and displays any errors that occur during data fetching, returning None if an issue arises.
6. **Return Value**:
    - The Beta value as a float rounded to two decimal places.
    - Returns None if the Beta value is unavailable or an error occurs.

Use Case

get_52_week_high_low()

The get_52_week_high_low function retrieves the **52-week high** and **52-week low** stock prices from Yahoo Finance using the yfinance library. These values represent the highest and lowest trading prices of the stock over the past year, providing insights into its performance range.

Explanation of the get_52_week_high_low Function

Purpose

This function provides the 52-week high and low prices for a stock, helping investors gauge the stock's recent performance and volatility.

Applications of the Function

1. **Performance Analysis**:
    - The 52-week high and low provide a quick snapshot of how the stock has performed over the past year.
2. **Trading Strategies**:
    - **Support and Resistance Levels**:
        - The 52-week low can act as a support level, while the 52-week high can act as a resistance level for traders.
    - **Breakout Opportunities**:
        - A stock surpassing its 52-week high may indicate bullish momentum, while dropping below its 52-week low might signal bearish trends.
3. **Risk Assessment**:
    - The range between the high and low can indicate the stock's volatility.
4. **Investment Decisions**:
    - Investors may use this information to determine if a stock is trading near its highs (potential overvaluation) or lows (potential undervaluation).

Key Features

1. **Real-Time Data**:
    - Fetches the most recent 52-week high and low prices from Yahoo Finance.
2. **Error Handling**:
    - Provides descriptive error messages if data fetching fails.
3. **Readable Output**:

- Returns rounded prices for better clarity and usability.

4. **Convenient Tuple Format**:
    - Bundles the high and low prices into a tuple for easier access and manipulation.

Limitations

1. **Data Availability**:
    - The function may return None if the 52-week high/low data is unavailable, especially for new or thinly traded stocks.
2. **Dependent on yfinance**:
    - Requires the yfinance library and an active internet connection.
3. **Static Snapshot**:
    - Provides the latest 52-week high/low values but does not show historical changes over time.

Source Code

```
def get_52_week_high_low(ticker_symbol):
    """
    Fetches the 52-week high and low prices of a stock using the yfinance
library.

    Parameters:
        ticker_symbol (str): The stock ticker symbol (e.g., 'AAPL', 'MSFT').

    Returns:
        tuple: A tuple containing the 52-week high and low prices as floats,
or None if data is unavailable.
    """
    try:
        stock = yf.Ticker(ticker_symbol)
        high_52_week = stock.info.get('fiftyTwoWeekHigh')
        low_52_week = stock.info.get('fiftyTwoWeekLow')

        if high_52_week is not None and low_52_week is not None:
            return round(high_52_week, 2), round(low_52_week, 2)
        else:
            print(f"No 52-week high/low data available for {ticker_symbol}.")
            return None
    except Exception as e:
```

```
        print(f"Error fetching data for {ticker_symbol}: {e}")
        return None
```

Code Explanation

Steps in the Code

1. **Input:**
    - **ticker_symbol: A string representing the stock's ticker symbol (e.g., 'AAPL' for Apple).**
2. **Data Fetching:**
    - **Uses the yf.Ticker class from the yfinance library to fetch the stock's information.**
    - **Retrieves the 52-week high and low prices from the info dictionary using the keys 'fiftyTwoWeekHigh' and 'fiftyTwoWeekLow'.**
3. **Data Validation:**
    - **Checks if both the 52-week high and low prices are available in the retrieved data.**
    - **If either value is missing, the function prints a message and returns None.**
4. **Data Formatting:**
    - **If both values are present, they are rounded to two decimal places for readability and returned as a tuple.**
5. **Error Handling:**
    - **Catches and displays any errors that occur during data fetching, returning None if an issue arises.**
6. **Return Value:**
    - **A tuple containing the 52-week high and low prices as floats rounded to two decimal places.**
    - **Returns None if the data is unavailable or an error occurs.**

Usecase

```
get_52_week_high_low('AAPL')
✓ 0.6s

(250.8, 164.08)
```

get_average_volume()

The get_average_volume function retrieves the **average trading volume** of a stock using the yfinance library. Average trading volume is a key indicator of a stock's liquidity and investor interest.

Explanation of the get_average_volume Function

Purpose

This function fetches the average number of shares traded per day for a stock, giving insight into the stock's trading activity and liquidity over time.

Applications of the Function

1. **Liquidity Analysis**:
    - Stocks with higher average trading volumes are generally more liquid, making them easier to buy or sell without impacting the price significantly.
2. **Investor Sentiment**:
    - High trading volumes often indicate strong investor interest or activity in a stock.
3. **Trading Strategy**:
    - Helps traders identify stocks with sufficient liquidity to support large trades.
    - Low-volume stocks may have higher price volatility, which can influence risk.
4. **Comparison Across Stocks**:
    - Compare the average trading volumes of different stocks to identify which ones have higher liquidity or are more actively traded.

Key Features

1. **Simple and Clear Output**:
    - Returns the raw average trading volume value for easy integration into calculations or reports.
2. **Error Handling**:
    - Provides descriptive error messages to handle issues with data fetching.
3. **Insightful Indicator**:
    - Offers a key metric for assessing stock liquidity and market activity.

Limitations

1. **Data Availability**:

- The function relies on Yahoo Finance's data and may return None for new or thinly traded stocks with no recorded average volume.

2. **Static Value**:
   - The function fetches the current average volume but does not show how it has changed over time.
3. **Dependent on yfinance**:
   - Requires the yfinance library and an active internet connection.

Source Code

```
def get_average_volume(ticker_symbol):
    """
    Fetches the average trading volume of a stock using the yfinance library.

    Parameters:
        ticker_symbol (str): The stock ticker symbol (e.g., 'AAPL', 'MSFT').

    Returns:
        int: The average volume, or None if no data is available.
    """
    try:
        stock = yf.Ticker(ticker_symbol)
        avg_volume = stock.info.get('averageVolume')

        if avg_volume is not None:
            return avg_volume
        else:
            print(f"No average volume data available for {ticker_symbol}.")
            return None
    except Exception as e:
        print(f"Error fetching data for {ticker_symbol}: {e}")
        return None
```

Code Explanation

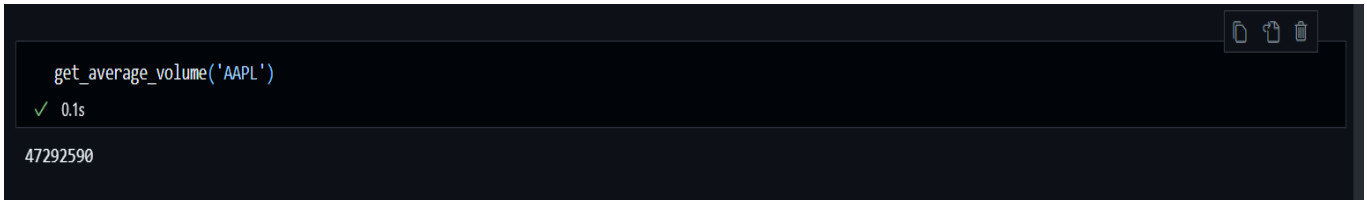
Steps in the Code

1. **Input:**
   - **ticker_symbol: A string representing the stock's ticker symbol (e.g., 'AAPL' for Apple).**
2. **Data Fetching:**

- Uses the yf.Ticker class from the yfinance library to fetch the stock's information.
- Retrieves the average trading volume from the info dictionary using the key 'averageVolume'.

3. Data Validation:
   - Checks if the averageVolume data is available.
   - If the data is unavailable, the function prints a message and returns None.
4. Return Value:
   - If the average volume data is available, it is returned as an integer.
   - If the data is unavailable or an error occurs, the function returns None.
5. Error Handling:
   - Catches and displays any errors encountered during data fetching, ensuring the program does not crash due to unexpected issues.

Usecase



calculate_moving_averages()

The calculate_moving_averages function computes **Simple Moving Averages (SMA)** and **Exponential Moving Averages (EMA)** for a stock using historical price data obtained from the yfinance library. Moving averages are commonly used in technical analysis to identify trends and price direction.

Explanation of the calculate_moving_averages Function

Purpose

This function fetches historical stock data for the past year and calculates **SMA** and **EMA** for various time periods (e.g., 5-day, 10-day). The results provide insights into the stock's recent and long-term price trends.

Applications of the Function

1. **Trend Identification**:
    - Short-term averages (e.g., SMA_5, EMA_10) can help identify recent price movements.
    - Long-term averages (e.g., SMA_200) are useful for understanding broader market trends.
2. **Trading Strategies**:
    - Moving average crossovers (e.g., SMA_50 crossing SMA_200) can signal buying or selling opportunities.
3. **Support and Resistance Levels**:
    - Stocks often respect their moving averages as key support or resistance levels.
4. **Risk Assessment**:
    - Analyzing moving averages can help investors gauge market volatility and risk.

Key Features

1. **Comprehensive Periods**:
    - Computes moving averages for commonly used periods (e.g., 5, 10, 20 days).
2. **Simple and Exponential Averages**:
    - SMA provides a straightforward average of prices.
    - EMA gives more weight to recent prices, making it more responsive to recent changes.
3. **User-Friendly Output**:
    - Returns a dictionary of values, ready to be used for analysis or plotting.

4. **Error Resilience**:
    - Handles data errors gracefully, ensuring robust execution.

Limitations

1. **Historical Data Dependency**:
    - Requires at least 200 days of historical data to compute the SMA and EMA for longer periods.
2. **Static Periods**:
    - Fixed periods may not fit every trader's or investor's needs.
3. **Real-Time Data**:
    - Does not account for intraday price movements; uses historical data only.

Source Code

```
def calculate_moving_averages(ticker):
    """
    Fetches historical stock data for the given ticker and calculates Simple
and Exponential Moving Averages.

    Parameters:
        ticker (str): The stock ticker symbol (e.g., 'AAPL').

    Returns:
        dict: A dictionary containing SMAs and EMAs for different periods.
    """
    try:
        # Download historical data for the past year
        stock_data = yf.download(ticker, period="1y")

        if stock_data.empty:
            raise ValueError("No data found for the ticker.")

        # Calculate moving averages for specified periods
        periods = [5, 10, 20, 50, 100, 200]
        moving_averages = {}

        for period in periods:
            sma_column = f'SMA_{period}'
            ema_column = f'EMA_{period}'
```

```
            # Simple Moving Average (SMA)
            stock_data[sma_column] =
stock_data['Close'].rolling(window=period).mean()

            # Exponential Moving Average (EMA)
            stock_data[ema_column] = stock_data['Close'].ewm(span=period,
adjust=False).mean()

            moving_averages[sma_column] = stock_data[sma_column].iloc[-1]
            moving_averages[ema_column] = stock_data[ema_column].iloc[-1]

        return moving_averages

    except Exception as e:
        print(f"An error occurred: {e}")
        return None
```

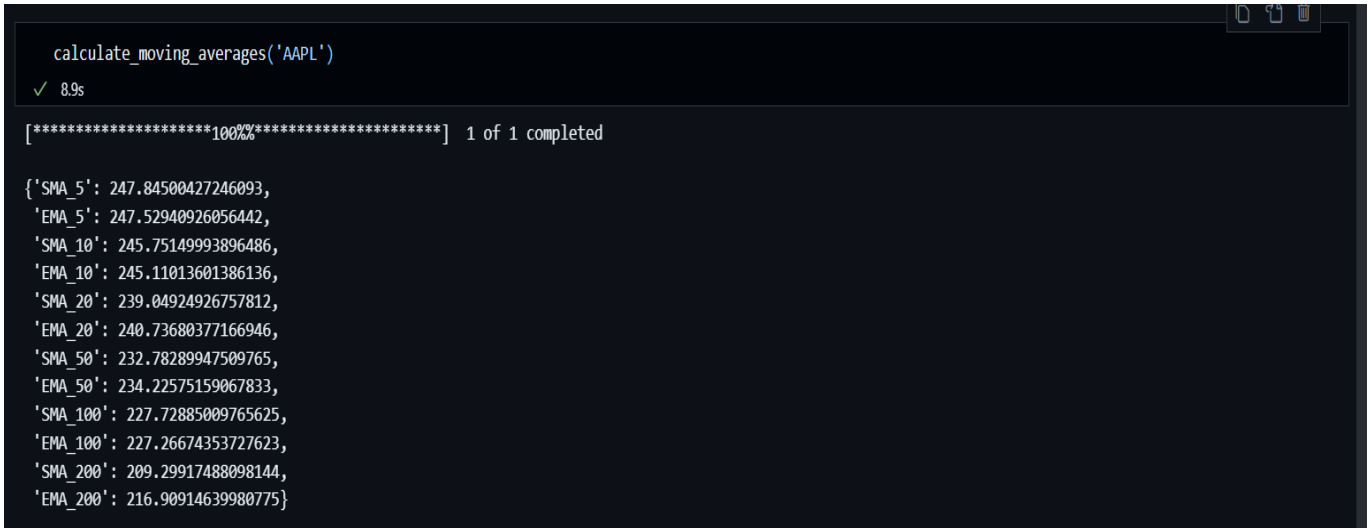
Code Explanation

Steps in the Code

1. **Input:**
    - ticker: A string representing the stock's ticker symbol (e.g., 'AAPL' for Apple).
2. **Fetching Historical Data:**
    - Retrieves 1 year of historical price data using the yfinance.download function.
    - If no data is found (e.g., invalid ticker), raises a ValueError.
3. **Define Periods:**
    - Specifies a list of periods for which moving averages will be calculated: [5, 10, 20, 50, 100, 200] days.
4. **Calculate Moving Averages:**
    - SMA (Simple Moving Average):
        - Uses the rolling().mean() function on the stock's closing prices over the specified period.
    - EMA (Exponential Moving Average):
        - Uses the ewm().mean() function, which gives more weight to recent prices.
5. **Extract and Store Results:**
    - The most recent values for each SMA and EMA are extracted and stored in a dictionary, moving_averages, with keys like 'SMA_5' or 'EMA_50'.

6. **Return Value:**
    - Returns the moving_averages dictionary, containing the computed values for each period.
7. **Error Handling:**
    - If any exception occurs (e.g., invalid ticker, data unavailability), it prints the error and returns None.

Usecase



calculate_rsi()

The calculate_rsi function computes the **Relative Strength Index (RSI)** for a given stock. RSI is a popular momentum oscillator used in technical analysis to measure the speed and change of price movements, helping identify overbought or oversold conditions.

Explanation of the calculate_rsi Function

Purpose

This function fetches historical stock data for the past year and calculates the RSI based on a specified lookback period (default is 14 days). The RSI value is returned, indicating the stock's current momentum.

Applications of RSI

1. **Overbought/Oversold Conditions**:
    - **Overbought**: RSI > 70 (potential reversal or sell signal).
    - **Oversold**: RSI < 30 (potential reversal or buy signal).
2. **Divergence Analysis**:
    - RSI diverging from price trends can indicate a potential reversal.
3. **Trend Confirmation**:
    - RSI levels can help confirm the strength of ongoing trends.
4. **Support/Resistance Levels**:
    - RSI often respects levels during strong trends, e.g., staying above 40 in an uptrend.

Key Features

1. **Customizable Lookback Period**:
    - The period parameter allows users to adjust the sensitivity of the RSI.
2. **Momentum Analysis**:
    - Identifies momentum shifts, helping traders spot potential trend reversals.
3. **User-Friendly Output**:
    - Returns a single, actionable RSI value for the latest date.
4. **Error Resilience**:
    - Handles missing or invalid data gracefully.

Limitations

1. **Historical Data Dependency**:
    - Requires sufficient historical data to calculate RSI, especially for longer periods.
2. **Static Period**:
    - While 14 days is a standard, it may not fit every trading strategy.
3. **Lagging Indicator**:
    - RSI reacts to past data and may lag during rapid price movements.

Source Code

```
def calculate_rsi(ticker, period=14):
    """
    Fetches historical stock data for the given ticker and calculates the
Relative Strength Index (RSI).

    Parameters:
        ticker (str): The stock ticker symbol (e.g., 'AAPL').
        period (int): The lookback period for RSI calculation. Default is 14.

    Returns:
        float: The RSI value.
    """
    try:
        # Download historical data for the past year
        stock_data = yf.download(ticker, period="1y")

        if stock_data.empty:
            raise ValueError("No data found for the ticker.")

        # Calculate price changes
        delta = stock_data['Close'].diff()

        # Separate gains and losses
        gain = delta.where(delta > 0, 0)
        loss = -delta.where(delta < 0, 0)

        # Calculate average gain and loss
        avg_gain = gain.rolling(window=period, min_periods=1).mean()
        avg_loss = loss.rolling(window=period, min_periods=1).mean()

        # Calculate the Relative Strength (RS)
```

```
        rs = avg_gain / avg_loss

        # Calculate the Relative Strength Index (RSI)
        rsi = 100 - (100 / (1 + rs))

        # Return the most recent RSI value
        return rsi.iloc[-1]

    except Exception as e:
        print(f"An error occurred: {e}")
        return None
```

Code Explanation

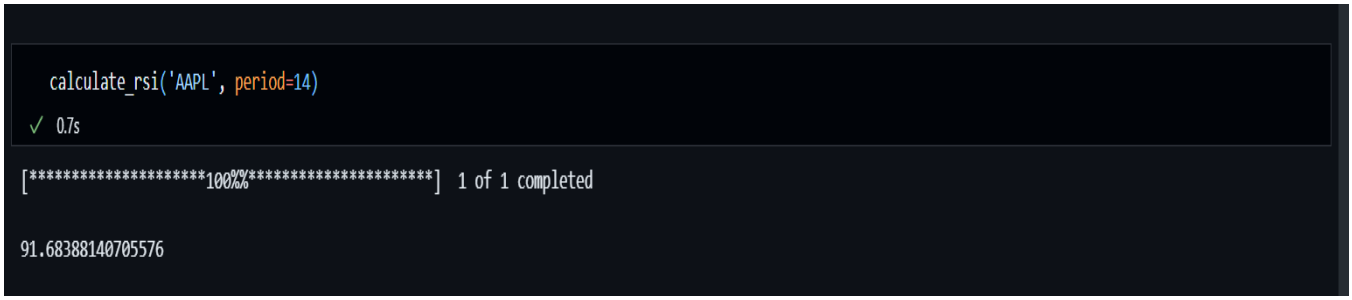
Steps in the Code

1. Input:
    - ticker: The stock ticker symbol as a string (e.g., 'AAPL' for Apple).
    - period: The number of days used for the RSI calculation. The default value is 14 days, a standard in technical analysis.
2. Fetching Historical Data:
    - Retrieves 1 year of historical stock prices using yfinance.download.
    - If no data is found (e.g., invalid ticker), raises a ValueError.
3. Calculate Price Changes:
    - Computes the difference between consecutive closing prices using .diff():
        - Positive differences are gains.
        - Negative differences are losses.
4. Separate Gains and Losses:
    - Extracts gains (positive price changes) and sets all other values to 0.
    - Extracts losses (negative price changes, converted to positive values) and sets all other values to 0.
5. Calculate Average Gain and Loss:
    - Uses a rolling window to compute the average gain and average loss over the specified period (default 14 days).
6. Calculate Relative Strength (RS):
    - The Relative Strength (RS) is calculated as: RS=Average GainAverage LossRS = \frac{\text{Average Gain}}{\text{Average Loss}}RS=Average LossAverage Gain
7. Calculate RSI:

- RSI is computed using the formula: RSI=100−(1001+RS)RSI = 100 - \left(\frac{100}{1 + RS} \right)RSI=100−(1+RS100)
- RSI values range between 0 and 100:
  - Values above 70 suggest the stock is overbought.
  - Values below 30 suggest the stock is oversold.

8. Return Value:
   - Returns the most recent RSI value (rsi.iloc[-1]).
9. Error Handling:
   - Prints an error message and returns None if an exception occurs (e.g., invalid ticker, calculation issues).

Usecase



calculate_macd()

The calculate_macd function computes the **Moving Average Convergence Divergence (MACD)** and its associated **Signal Line** for a given stock. The MACD is a widely used momentum indicator in technical analysis that helps identify potential trend reversals and the strength of price movements.

Explanation of the calculate_macd Function

Purpose

This function retrieves historical stock data and calculates the MACD line and Signal line values based on the stock's closing prices. It provides insights into the stock's momentum and possible buy/sell signals.

Applications of MACD

1. **Buy/Sell Signals:**
    - Bullish Signal: When the MACD line crosses above the Signal line (indicates upward momentum).
    - Bearish Signal: When the MACD line crosses below the Signal line (indicates downward momentum).
2. **Trend Strength:**
    - Larger differences between the MACD and Signal lines indicate stronger trends.
3. **Divergence Analysis:**
    - Divergence between MACD and price action can signal potential reversals.
4. **Identify Momentum:**
    - Helps traders confirm trend direction and gauge the momentum of price movements.

Key Features

1. **Customizable Periods:**
    - The function allows customization of short-term, long-term, and Signal line periods to suit different trading strategies.
2. **Momentum Indicator:**
    - Provides insights into the momentum of the stock, helping traders identify potential entry/exit points.
3. **User-Friendly Output:**
    - Returns the latest values of the MACD and Signal lines for immediate use.

4. **Error Resilience:**
    - Handles missing or invalid data gracefully with informative error messages.

Source Code

```
def calculate_macd(ticker, short_period=12, long_period=26, signal_period=9):
    """
    Fetches historical stock data for the given ticker and calculates the MACD
and Signal line values.

    Parameters:
        ticker (str): The stock ticker symbol (e.g., 'AAPL').
        short_period (int): The short EMA period. Default is 12.
        long_period (int): The long EMA period. Default is 26.
        signal_period (int): The signal line EMA period. Default is 9.

    Returns:
        dict: A dictionary containing the latest MACD line and Signal line
values.
    """
    try:
        # Download historical data for the past year
        stock_data = yf.download(ticker, period="1y")

        if stock_data.empty:
            raise ValueError("No data found for the ticker.")

        # Calculate the short-term EMA (12-day by default)
        stock_data['Short_EMA'] = stock_data['Close'].ewm(span=short_period,
adjust=False).mean()

        # Calculate the long-term EMA (26-day by default)
        stock_data['Long_EMA'] = stock_data['Close'].ewm(span=long_period,
adjust=False).mean()

        # Calculate the MACD line
        stock_data['MACD'] = stock_data['Short_EMA'] - stock_data['Long_EMA']

        # Calculate the Signal line (9-day EMA of MACD by default)
        stock_data['Signal_Line'] = stock_data['MACD'].ewm(span=signal_period,
adjust=False).mean()

        # Get the latest values
        macd_line = stock_data['MACD'].iloc[-1]
        signal_line = stock_data['Signal_Line'].iloc[-1]
```

```
        return {'MACD_Line': macd_line, 'Signal_Line': signal_line}

    except Exception as e:
        print(f"An error occurred: {e}")
        return None
```

Code Explanation

Steps in the Code

1. Input Parameters:
    - ticker: The stock ticker symbol as a string (e.g., 'AAPL' for Apple).
    - short_period: The period for the short-term Exponential Moving Average (EMA), default is 12 days.
    - long_period: The period for the long-term EMA, default is 26 days.
    - signal_period: The period for the Signal line (EMA of the MACD), default is 9 days.
2. Fetching Historical Data:
    - Downloads 1 year of historical stock prices using yfinance.download.
    - If no data is found (e.g., invalid ticker), raises a ValueError.
3. Calculate Short-Term and Long-Term EMAs:
    - Short-Term EMA:
        - Calculated for the closing prices using a span of short_period (default 12 days).
    - Long-Term EMA:
        - Calculated for the closing prices using a span of long_period (default 26 days).
4. Calculate the MACD Line:
    - The MACD line is computed as: MACD=Short_EMA−Long_EMA\text{MACD} = \text{Short_EMA} - \text{Long_EMA}MACD=Short_EMA−Long_EMA
5. Calculate the Signal Line:
    - The Signal line is the EMA of the MACD line over signal_period (default 9 days).
6. Retrieve Latest Values:
    - Extracts the most recent values of:
        - MACD Line: The difference between the short-term and long-term EMAs.
        - Signal Line: The smoothed EMA of the MACD line.

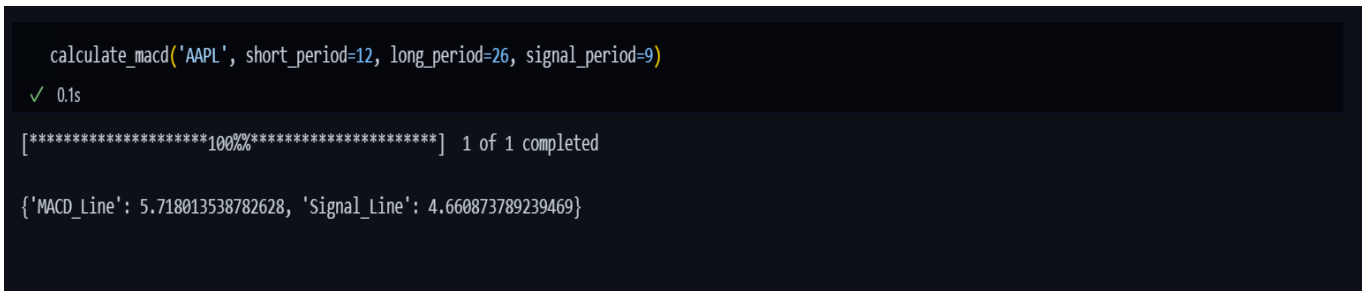
7. Error Handling:
    - Prints an error message and returns None if an exception occurs (e.g., invalid ticker, calculation issues).

Return Value

The function returns a dictionary with:

- MACD_Line: The latest MACD line value.
- Signal_Line: The latest Signal line value.

Use Case



calculate_current_ratio()

The calculate_current_ratio function calculates the **Current Ratio** of a given company based on its balance sheet data. The Current Ratio is a financial metric that assesses a company's ability to meet short-term obligations using its short-term assets.

Explanation of the calculate_current_ratio Function

Purpose

This function retrieves the balance sheet data for a specified stock ticker and calculates the Current Ratio to evaluate the company's liquidity position.

Applications of Current Ratio

1. **Liquidity Analysis**:
    - Measures a company's ability to pay off its short-term debts with short-term assets.
2. **Financial Health Indicator**:
    - A Current Ratio above **1** indicates that the company has more current assets than current liabilities, which is generally favorable.
    - A ratio below **1** suggests potential liquidity issues.
3. **Comparison**:
    - Used to compare companies within the same industry to assess relative financial stability.

Key Features

1. **Dynamic Label Adjustment**:
    - Handles variations in balance sheet labels (e.g., "Current Assets" vs. "Total Current Assets").
2. **Robust Error Handling**:
    - Prevents division by zero and gracefully handles missing or improperly structured data.
3. **User-Friendly Output**:
    - Returns a straightforward numerical result for easy interpretation.

Source Code

```
def calculate_current_ratio(ticker):
    """
    Fetches balance sheet data for the given ticker and calculates the Current
Ratio.

    Current Ratio = Current Assets / Current Liabilities

    Parameters:
        ticker (str): The stock ticker symbol (e.g., 'AAPL').

    Returns:
        float: The Current Ratio, or None if the data is unavailable.
    """
    try:
        # Download financial information
        stock = yf.Ticker(ticker)
        balance_sheet = stock.balance_sheet

        # Debugging: Print the balance sheet data to check the structure
        print(balance_sheet)

        # Try to fetch "Total Current Assets" and "Total Current Liabilities"
        if balance_sheet.empty:
            raise ValueError("Balance sheet data is not available for this
ticker.")

        # Adjust labels based on actual structure
        current_assets_label = 'Current Assets' if 'Current Assets' in
balance_sheet.index else 'Total Current Assets'
        current_liabilities_label = 'Current Liabilities' if 'Current
Liabilities' in balance_sheet.index else 'Total Current Liabilities'

        # Extract values
        current_assets = balance_sheet.loc[current_assets_label].iloc[0]
        current_liabilities =
balance_sheet.loc[current_liabilities_label].iloc[0]

        # Calculate Current Ratio
        if current_liabilities == 0:
            raise ValueError("Current liabilities are zero, division by zero
is not allowed.")

        current_ratio = current_assets / current_liabilities
        return current_ratio
```

```
    except Exception as e:
        print(f"An error occurred: {e}")
        return None
```

Code Explanation

Steps in the Code

1. Input Parameters:
    - ticker: The stock ticker symbol (e.g., 'AAPL' for Apple).
2. Fetch Balance Sheet Data:
    - Retrieves financial data using the yfinance.Ticker object and accesses the balance_sheet attribute.
3. Check for Data Availability:
    - If the balance sheet data is missing or empty, raises a ValueError.
4. Identify Labels for Assets and Liabilities:
    - Searches for the appropriate labels for:
        - Current Assets: Either "Current Assets" or "Total Current Assets".
        - Current Liabilities: Either "Current Liabilities" or "Total Current Liabilities".
5. Extract Values:
    - Extracts the most recent values for Current Assets and Current Liabilities from the balance sheet.
6. Calculate Current Ratio:
    - Formula: Current Ratio=Current AssetsCurrent Liabilities\text{Current Ratio} = \frac{\text{Current Assets}}{\text{Current Liabilities}}Current Ratio=Current LiabilitiesCurrent Assets
    - Checks if Current Liabilities is zero to prevent division by zero.
7. Error Handling:
    - Prints an error message and returns None in case of missing data, invalid structure, or calculation issues.

Use Case

```
calculate_current_ratio('AAPL')
```

✓ 0.5s

	2024-09-30	\
Treasury Shares Number	NaN	
Ordinary Shares Number	15116786000.0	
Share Issued	15116786000.0	
Net Debt	76686000000.0	
Total Debt	106629000000.0	
...	...	
Cash Cash Equivalents And Short Term Investments	65171000000.0	
Other Short Term Investments	35228000000.0	
Cash And Cash Equivalents	29943000000.0	
Cash Equivalents	2744000000.0	
Cash Financial	27199000000.0	

	2023-09-30	\
Treasury Shares Number	0.0	
Ordinary Shares Number	15550061000.0	
Share Issued	15550061000.0	
Net Debt	81123000000.0	
Total Debt	111088000000.0	
...	...	
Cash Cash Equivalents And Short Term Investments	61555000000.0	
Other Short Term Investments	31590000000.0	
Cash And Cash Equivalents	29965000000.0	
Cash Equivalents	1606000000.0	
Cash Financial	28359000000.0	
...		
Cash Equivalents	17635000000.0	NaN
Cash Financial	17305000000.0	NaN

[68 rows x 5 columns]

```
0.8673125765340832
```

calculate_quick_ratio()

The calculate_quick_ratio function computes the **Quick Ratio** of a company, which measures its ability to meet short-term obligations without relying on the sale of inventory. This ratio is also known as the "Acid-Test Ratio."

Explanation of the calculate_quick_ratio Function

Purpose

The function retrieves balance sheet data for a specified stock ticker and calculates the Quick Ratio. It offers a more stringent measure of liquidity than the Current Ratio by excluding inventory from current assets.

Applications of Quick Ratio

1. **Liquidity Analysis**:
    - Indicates how easily a company can meet its short-term liabilities without relying on inventory liquidation.
2. **Financial Stability Assessment**:
    - A Quick Ratio above **1** suggests sufficient liquid assets to cover current liabilities, which is favorable.
3. **Industry Comparisons**:
    - Helps compare companies within the same sector for financial health.

Key Features of the Function

1. **Dynamic Label Adjustment**:
    - Handles variations in balance sheet label names for different companies.
2. **Error Handling**:
    - Provides meaningful error messages for missing or improperly formatted data.
3. **Precise Liquidity Measure**:
    - Excludes inventory, which might not be as easily converted to cash as other current assets.

Challenges and Error Scenarios

1. **Missing or Empty Data**:
    - The function handles missing balance sheet data by raising a ValueError.

2. **Zero Liabilities**:
    - Avoids division by zero errors by raising an exception when Current Liabilities is zero.
3. **Inventory Not Reported**:
    - Assumes a value of 0 if inventory data is unavailable.

Comparison with Current Ratio

The Quick Ratio is a stricter liquidity measure than the Current Ratio:

- **Quick Ratio** excludes inventory, considering only assets that can be quickly converted to cash.
- **Current Ratio** includes all current assets, making it less conservative.

Source Code

```
def calculate_quick_ratio(ticker):
    """
    Fetches balance sheet data for the given ticker and calculates the Quick
Ratio.

    Quick Ratio = (Current Assets - Inventory) / Current Liabilities

    Parameters:
        ticker (str): The stock ticker symbol (e.g., 'AAPL').

    Returns:
        float: The Quick Ratio, or None if the data is unavailable.
    """
    try:
        # Download financial information
        stock = yf.Ticker(ticker)
        balance_sheet = stock.balance_sheet

        # Debugging: Print the balance sheet data to check its structure
        print(balance_sheet)

        # Ensure balance sheet data is available
        if balance_sheet.empty:
            raise ValueError("Balance sheet data is not available for this
ticker.")

        # Adjust labels based on actual structure
```

```
        current_assets_label = 'Current Assets' if 'Current Assets' in
balance_sheet.index else 'Total Current Assets'
        current_liabilities_label = 'Current Liabilities' if 'Current
Liabilities' in balance_sheet.index else 'Total Current Liabilities'
        inventory_label = 'Inventory' if 'Inventory' in balance_sheet.index
else None

        # Extract values
        current_assets = balance_sheet.loc[current_assets_label].iloc[0]
        current_liabilities =
balance_sheet.loc[current_liabilities_label].iloc[0]
        inventory = balance_sheet.loc[inventory_label].iloc[0] if
inventory_label else 0

        # Check for division by zero
        if current_liabilities == 0:
            raise ValueError("Current liabilities are zero, division by zero
is not allowed.")

        # Calculate Quick Ratio
        quick_ratio = (current_assets - inventory) / current_liabilities
        return quick_ratio

    except Exception as e:
        print(f"An error occurred: {e}")
        return None
```

Code Explanation

Steps in the Code

1. Input Parameters:
    - ticker: The stock ticker symbol (e.g., 'AAPL' for Apple).
2. Fetch Balance Sheet Data:
    - Retrieves financial data using the yfinance.Ticker object and accesses the balance_sheet attribute.
3. Check for Data Availability:
    - Ensures that the balance sheet data is available and non-empty. If missing, raises a ValueError.
4. Identify Relevant Labels:
    - Dynamically identifies the labels for:
        - Current Assets: "Current Assets" or "Total Current Assets".

        - Current Liabilities: "Current Liabilities" or "Total Current Liabilities".
        - Inventory: Looks for "Inventory" if present; otherwise assumes a value of 0.
5. Extract Values:
    - Retrieves the most recent values for Current Assets, Current Liabilities, and Inventory.
6. Handle Division by Zero:
    - Checks if Current Liabilities equals zero to prevent division by zero errors.
7. Calculate Quick Ratio:
    - Formula: Quick Ratio=Current Assets−InventoryCurrent Liabilities\text{Quick Ratio} = \frac{\text{Current Assets} - \text{Inventory}}{\text{Current Liabilities}}Quick Ratio=Current LiabilitiesCurrent Assets−Inventory
8. Error Handling:
    - Catches exceptions, prints an error message, and returns None if an issue occurs.

Use Case

```
calculate_quick_ratio('AAPL')
```

✓ 0.1s

```
                                          2024-09-30  \
Treasury Shares Number                           NaN
Ordinary Shares Number                15116786000.0
Share Issued                          15116786000.0
Net Debt                              76686000000.0
Total Debt                           106629000000.0
...                                            ...
Cash Cash Equivalents And Short Term Investments   65171000000.0
Other Short Term Investments          35228000000.0
Cash And Cash Equivalents             29943000000.0
Cash Equivalents                       2744000000.0
Cash Financial                        27199000000.0

                                          2023-09-30  \
Treasury Shares Number                           0.0
Ordinary Shares Number                15550061000.0
Share Issued                          15550061000.0
Net Debt                              81123000000.0
Total Debt                           111088000000.0
...                                            ...
Cash Cash Equivalents And Short Term Investments   61555000000.0
Other Short Term Investments          31590000000.0
Cash And Cash Equivalents             29965000000.0
Cash Equivalents                       1606000000.0
Cash Financial                        28359000000.0
...
Cash Equivalents                      17635000000.0       NaN
Cash Financial                        17305000000.0       NaN

[68 rows x 5 columns]
```

```
0.8260068483831466
```

create_price_chart()

The create_price_chart function builds a web-based stock price dashboard using the Dash framework. The dashboard fetches stock data, allows users to customize the time period and chart type, and displays the data as an interactive chart.

Explanation of the create_price_chart Function

Purpose

This function sets up a Dash app that displays stock price data for a user-specified ticker symbol. It provides an interactive and visually appealing interface where users can:

1. Enter a stock ticker symbol.
2. Choose a time period for the stock price data.
3. Select between candlestick and line charts.

Applications

1. **Stock Price Visualization**:
    - Ideal for analyzing trends and patterns in stock prices.
2. **Financial Dashboards**:
    - Can be integrated into larger dashboards for portfolio management.
3. **Educational Tools**:
    - Useful for teaching stock market concepts and technical analysis.

Key Functions Explained

fetch_stock_data

- Retrieves historical stock data for a given ticker and period using yfinance.

create_stock_chart

- Generates an interactive candlestick or line chart using Plotly.

Callback Function

- Updates the chart dynamically when user inputs change.

Expected Output

1. **App Title**:
    - Displays "Stock Price Dashboard" at the top.

2. **User Inputs**:
    - A text box for stock ticker input.
    - Dropdown menus for selecting time period and chart type.
3. **Chart**:
    - Displays an interactive candlestick or line chart of the stock price.

Error Handling

1. **Invalid Ticker**:
    - If the ticker symbol is invalid, the app may display an empty chart or an error message in the console.
2. **Empty Data**:
    - Ensures valid data before plotting; otherwise, a fallback is needed.
3. **Connection Issues**:
    - Network or API issues could result in data-fetching errors.

Source Code

```
def create_price_chart():
    app = dash.Dash(__name__)

    # Function to fetch stock data using yfinance
    def fetch_stock_data(ticker, period='1y'):
        stock = yf.Ticker(ticker)
        data = stock.history(period=period)
        return data

    # Function to generate stock price chart using plotly
    def create_stock_chart(data, chart_type='candlestick'):
        fig = go.Figure()

        if chart_type == 'candlestick':
            fig.add_trace(go.Candlestick(x=data.index,
                                         open=data['Open'],
                                         high=data['High'],
                                         low=data['Low'],
                                         close=data['Close'],
                                         name='Candlestick'))
        else:  # Line chart
```

```
            fig.add_trace(go.Scatter(x=data.index, y=data['Close'],
mode='lines', name='Line'))

        # Dark theme settings
        fig.update_layout(
            template="plotly_dark",
            title=f"Stock Price Chart ({chart_type.capitalize()})",
            xaxis_title="Date",
            yaxis_title="Price (USD)",
            font=dict(color="white"),
            plot_bgcolor="black",
            paper_bgcolor="black",
            hovermode="x",
        )
        return fig

    # Layout for the Dash app
    app.layout = html.Div(
        style={'backgroundColor': 'black', 'color': 'white', 'padding':
'20px'},
        children=[
            html.H1("Stock Price Dashboard", style={'textAlign': 'center'}),

            # Input for ticker symbol
            html.Label("Enter Stock Ticker:", style={'font-size': '20px'}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': 'orange'}),

            # Dropdown for duration
            html.Label("Select Duration:", style={'font-size': '20px',
'color': 'orange'}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Month', 'value': '1mo'},
                    {'label': '3 Months', 'value': '3mo'},
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': 'orange'}
            ),

            # Dropdown for chart type
```

```
            html.Label("Select Chart Type:", style={'font-size': '20px',
'color': 'orange'}),
            dcc.Dropdown(
                id='chart-type-dropdown',
                options=[
                    {'label': 'Candlestick', 'value': 'candlestick'},
                    {'label': 'Line', 'value': 'line'},
                ],
                value='candlestick',
                style={'font-size': '20px', 'color': 'orange'}
            ),

            # Graph to display stock price chart
            dcc.Graph(id='price-chart', style={'height': '600px'}),
        ]
    )

    # Callback to update the chart based on user inputs
    @app.callback(
        Output('price-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value'),
        Input('chart-type-dropdown', 'value')
    )
    def update_chart(ticker, period, chart_type):
        # Fetch the stock data
        data = fetch_stock_data(ticker, period)

        # Generate the chart
        fig = create_stock_chart(data, chart_type)
        return fig

    # Run the Dash app
    app.run_server(debug=True)
```

Code Explanation

Steps in the Code

1. Initialize the Dash App

- Creates a Dash app instance (app = dash.Dash(__name__)) to serve the web application.

2. Fetch Stock Data

- Defines a helper function fetch_stock_data that retrieves historical stock price data using the yfinance library:
    - Inputs:

        - ticker (e.g., 'AAPL'): The stock ticker.
        - period (default '1y'): The duration for historical data (e.g., '1mo', '1y').
    - Output: A DataFrame containing columns such as Open, High, Low, Close, and Volume.

3. Generate Stock Chart

- Defines another helper function create_stock_chart that generates a chart using the Plotly library:
    - Inputs:
        - data: The historical stock data.
        - chart_type (default 'candlestick'): The chart type ('candlestick' or 'line').
    - Process:
        - Candlestick Chart:
            - Uses the go.Candlestick Plotly trace for open-high-low-close (OHLC) visualization.
        - Line Chart:
            - Uses the go.Scatter Plotly trace for plotting closing prices.
    - Output: A Plotly Figure object with a dark theme, hover interactions, and axis titles.

4. Design the App Layout

- Defines the layout of the dashboard using Dash components:
    - html.H1: Displays the title "Stock Price Dashboard."
    - dcc.Input: Allows the user to input the stock ticker symbol.
    - dcc.Dropdown: Lets the user select:
        - Duration: Time period for stock data.
        - Chart Type: Candlestick or line chart.
    - dcc.Graph: Displays the stock price chart.

5. Update Chart Dynamically

- Uses a Dash callback to update the chart when user inputs change:
    - Inputs:
        - ticker-input: The stock ticker.
        - duration-dropdown: Selected time period.
        - chart-type-dropdown: Selected chart type.
    - Process:

- Fetches new stock data using fetch_stock_data.
- Generates an updated chart using create_stock_chart.

- Output: Updates the figure property of the dcc.Graph component.

6. Run the Dash App

- Launches the Dash app in debug mode (app.run_server(debug=True)).

Example Usage

create_volume_chart()

Purpose

The create_volume_chart() function creates an interactive dashboard for visualizing the trading volume of a stock over a selected period using the Dash framework, yfinance for stock data retrieval, and Plotly for visualization.

Key Features

1. Ticker Input: Users can input a stock ticker symbol (e.g., AAPL) to view its volume data.
2. Duration Dropdown: Users can select a time range (e.g., 1 Month, 1 Year) to filter the data.
3. Dynamic Volume Chart: A bar chart shows the trading volume for the specified stock and duration.

How It Works

1. Users enter a stock ticker (e.g., AAPL) and select a time range (e.g., 1 Month).
2. The app fetches the corresponding stock data from Yahoo Finance.
3. The trading volume data is plotted as a bar chart and displayed in the dashboard.
4. The user can modify inputs to dynamically update the chart.

Applications

- Visualizing the trading activity of stocks over different periods.
- Gaining insights into market trends and volume spikes.
- Providing an interactive tool for investors and analysts.

Source Code

```
def create_volume_chart():
    app = dash.Dash(__name__)

    # Function to fetch stock data using yfinance
    def fetch_stock_data(ticker, period='1y'):
        stock = yf.Ticker(ticker)
        data = stock.history(period=period)
        return data
```

```
    # Function to generate stock volume chart using plotly
    def create_volume_chart(data):
        fig = go.Figure()

        # Adding volume trace
        fig.add_trace(go.Bar(x=data.index, y=data['Volume'], name='Volume'))

        # Dark theme settings
        fig.update_layout(
            template="plotly_dark",
            title="Stock Volume Chart",
            xaxis_title="Date",
            yaxis_title="Volume",
            font=dict(color="white"),
            plot_bgcolor="black",
            paper_bgcolor="black",
            hovermode="x",
        )
        return fig

    # Layout for the Dash app
    app.layout = html.Div(
        style={'backgroundColor': 'black', 'color': 'white', 'padding':
'20px'},
        children=[
            html.H1("Stock Volume Dashboard", style={'textAlign': 'center'}),

            # Input for ticker symbol
            html.Label("Enter Stock Ticker:", style={'font-size': '20px'}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': 'orange'}),

            # Dropdown for duration
            html.Label("Select Duration:", style={'font-size': '20px',
'color': 'orange'}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Month', 'value': '1mo'},
                    {'label': '3 Months', 'value': '3mo'},
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': 'orange'}
```

```
            ),

            # Graph to display stock volume chart
            dcc.Graph(id='volume-chart', style={'height': '600px'}),
        ]
    )

    # Callback to update the chart based on user inputs
    @app.callback(
        Output('volume-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch the stock data
        data = fetch_stock_data(ticker, period)

        # Generate the volume chart
        fig = create_volume_chart(data)
        return fig

    # Run the Dash app
    app.run_server(debug=True)
```

Code Explanation

Code Breakdown

1. Dash App Initialization

python
Copy code
app = dash.Dash(__name__)

- Initializes the Dash app for creating the dashboard.

2. Fetch Stock Data

python
Copy code
def fetch_stock_data(ticker, period='1y'):
  stock = yf.Ticker(ticker)
  data = stock.history(period=period)
  return data

- Retrieves stock data from Yahoo Finance using the yfinance library.
- Inputs:
    - ticker: Stock ticker symbol (e.g., AAPL).
    - period: Time range for data (default: 1 year).

- Returns:
    - A DataFrame with historical stock data, including the Volume column.

3. Generate Volume Chart

```
python
Copy code
def create_volume_chart(data):
    fig = go.Figure()

    # Adding volume as a bar chart
    fig.add_trace(go.Bar(x=data.index, y=data['Volume'], name='Volume'))

    # Setting up the dark theme
    fig.update_layout(
        template="plotly_dark",
        title="Stock Volume Chart",
        xaxis_title="Date",
        yaxis_title="Volume",
        font=dict(color="white"),
        plot_bgcolor="black",
        paper_bgcolor="black",
        hovermode="x",
    )
    return fig
```

- Plots the Volume data using a bar chart with Plotly.
- Customizes the chart with:
    - Dark theme (plotly_dark) for modern aesthetics.
    - Titles for the chart and axes.
    - Hover interaction for detailed data points.

4. App Layout

```
python
Copy code
app.layout = html.Div(
    style={'backgroundColor': 'black', 'color': 'white', 'padding': '20px'},
    children=[
        html.H1("Stock Volume Dashboard", style={'textAlign': 'center'}),

        # Ticker input
        html.Label("Enter Stock Ticker:", style={'font-size': '20px'}),
        dcc.Input(id='ticker-input', type='text', value='AAPL', style={'font-size': '20px', 'color': 'orange'}),

        # Dropdown for duration
        html.Label("Select Duration:", style={'font-size': '20px', 'color': 'orange'}),
        dcc.Dropdown(
            id='duration-dropdown',
            options=[
                {'label': '1 Month', 'value': '1mo'},
```

```
            {'label': '3 Months', 'value': '3mo'},
            {'label': '6 Months', 'value': '6mo'},
            {'label': '1 Year', 'value': '1y'},
            {'label': '2 Years', 'value': '2y'},
            {'label': '5 Years', 'value': '5y'},
            {'label': '10 Years', 'value': '10y'},
        ],
        value='1y',
        style={'font-size': '20px', 'color': 'orange'}
    ),

    # Volume chart display
    dcc.Graph(id='volume-chart', style={'height': '600px'}),
  ]
)
```

- **Ticker Input**: Text box (dcc.Input) for entering the stock ticker (default: AAPL).
- **Duration Dropdown**: Dropdown menu (dcc.Dropdown) for selecting the time period (default: 1 Year).
- **Graph Component**: Displays the bar chart (dcc.Graph).

5. Callback for Interactivity

```
python
Copy code
@app.callback(
  Output('volume-chart', 'figure'),
  Input('ticker-input', 'value'),
  Input('duration-dropdown', 'value')
)
def update_chart(ticker, period):
  # Fetch the stock data
  data = fetch_stock_data(ticker, period)

  # Generate the volume chart
  fig = create_volume_chart(data)
  return fig
```

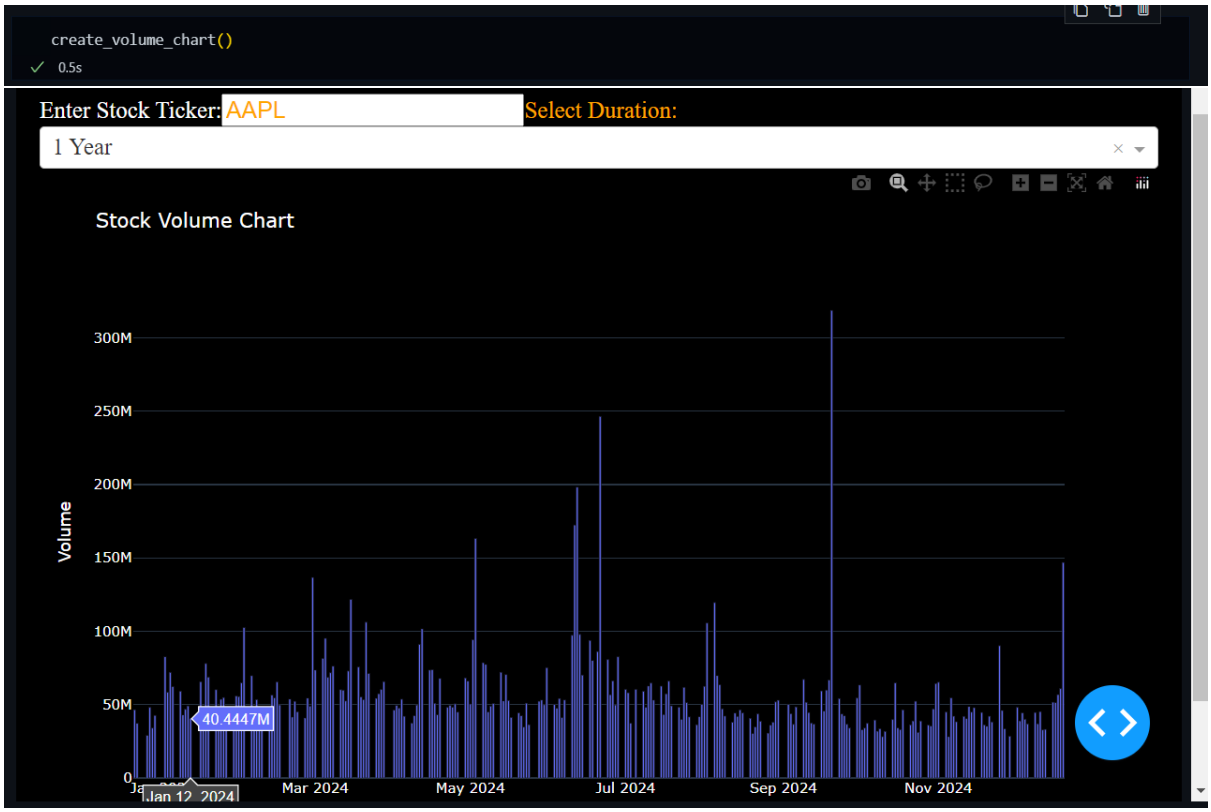
- Triggers updates to the chart when the user modifies:
  - **Stock Ticker**: Retrieves data for the specified stock.
  - **Duration**: Adjusts the time range of data.
- Regenerates the volume chart dynamically.

6. Run the App

```
python
Copy code
app.run_server(debug=True)
```

- Launches the app in debug mode for live updates and error tracking.

Usecase


create_volume_chart()
0.5s
Enter Stock Ticker: AAPL
Select Duration:
1 Year
Stock Volume Chart
300M
250M
200M
150M
100M
50M
0
Volume
40.4447M
Jan 12, 2024
Mar 2024
May 2024
Jul 2024
Sep 2024
Nov 2024

macd_dashboard()

Purpose

The macd_dashboard() function creates an interactive dashboard for visualizing the **MACD (Moving Average Convergence Divergence)** indicator for a stock over a specified time period using the Dash framework, yfinance for stock data retrieval, and Plotly for creating the chart.

Key Features

1. **Ticker Input**: Users can input a stock ticker symbol (e.g., AAPL) to calculate the MACD for that stock.
2. **Duration Dropdown**: Users can select a time range (e.g., 1 Month, 1 Year) to filter the data.
3. **MACD Chart**: A chart displaying the MACD line, Signal line, and MACD histogram is dynamically updated based on user input.

How It Works

1. Users enter a stock ticker (e.g., AAPL) and select a time range (e.g., 1 Month).
2. The app fetches the corresponding stock data from Yahoo Finance.
3. The MACD indicator is calculated based on the Close prices.
4. The MACD line, Signal line, and Histogram are plotted and displayed in the dashboard.
5. Users can modify inputs to dynamically update the chart.

Applications

- Analyzing stock price trends and momentum using the MACD indicator.
- Identifying buy/sell signals based on MACD crossover points.
- Providing an interactive tool for traders and financial analysts.

Source Code

```
def macd_dashboard():
    app = dash.Dash(__name__)

    # Function to fetch stock data using yfinance
    def fetch_stock_data(ticker, period='1y'):
        stock = yf.Ticker(ticker)
```

```
        data = stock.history(period=period)
        return data

    # Function to calculate MACD
    def calculate_macd(data):
        short_ema = data['Close'].ewm(span=12, adjust=False).mean()  # 12-
period EMA
        long_ema = data['Close'].ewm(span=26, adjust=False).mean()   # 26-
period EMA
        macd = short_ema - long_ema  # MACD line
        signal = macd.ewm(span=9, adjust=False).mean()  # Signal line
        return macd, signal

    # Function to generate MACD chart using plotly
    def create_macd_chart(data):
        macd, signal = calculate_macd(data)
        fig = go.Figure()

        # MACD line trace
        fig.add_trace(go.Scatter(x=data.index, y=macd, mode='lines',
name='MACD Line'))

        # Signal line trace
        fig.add_trace(go.Scatter(x=data.index, y=signal, mode='lines',
name='Signal Line'))

        # Bar plot for MACD histogram
        fig.add_trace(go.Bar(x=data.index, y=macd - signal, name='MACD
Histogram'))

        # Dark theme settings
        fig.update_layout(
            template="plotly_dark",
            title="MACD (Moving Average Convergence Divergence)",
            xaxis_title="Date",
            yaxis_title="MACD",
            font=dict(color="white"),
            plot_bgcolor="black",
            paper_bgcolor="black",
            hovermode="x",
        )
        return fig

    # Layout for the Dash app
    app.layout = html.Div(
        style={'backgroundColor': 'black', 'color': 'white', 'padding':
'20px'},
        children=[
```

```
            html.H1("MACD Dashboard", style={'textAlign': 'center'}),

            # Input for ticker symbol
            html.Label("Enter Stock Ticker:", style={'font-size': '20px'}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': 'orange'}),

            # Dropdown for duration
            html.Label("Select Duration:", style={'font-size': '20px',
'color': 'orange'}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Month', 'value': '1mo'},
                    {'label': '3 Months', 'value': '3mo'},
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': 'orange'}
            ),

            # Graph to display MACD chart
            dcc.Graph(id='macd-chart', style={'height': '600px'}),
        ]
    )

    # Callback to update the chart based on user inputs
    @app.callback(
        Output('macd-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch the stock data
        data = fetch_stock_data(ticker, period)

        # Generate the MACD chart
        fig = create_macd_chart(data)
        return fig

    # Run the Dash app
    app.run_server(debug=True)
```

Code Explanation

Code Breakdown

1. Dash App Initialization

python

Copy code

```
app = dash.Dash(__name__)
```

- Initializes the Dash app for creating the dashboard.

2. Fetch Stock Data

python

Copy code

```
def fetch_stock_data(ticker, period='1y'):
    stock = yf.Ticker(ticker)
    data = stock.history(period=period)
    return data
```

- Retrieves historical stock data from Yahoo Finance using the yfinance library.
- Inputs:
    - ticker: Stock ticker symbol (e.g., AAPL).
    - period: Time range for data (default: 1 year).
- Returns:
    - A DataFrame with historical stock data, including the Close price used to calculate MACD.

3. Calculate MACD

python

Copy code

```
def calculate_macd(data):
    short_ema = data['Close'].ewm(span=12, adjust=False).mean()  # 12-period EMA
    long_ema = data['Close'].ewm(span=26, adjust=False).mean()   # 26-period EMA
    macd = short_ema - long_ema  # MACD line
    signal = macd.ewm(span=9, adjust=False).mean()  # Signal line
    return macd, signal
```

- Calculates the MACD indicator:
    - Short EMA: 12-period Exponential Moving Average.
    - Long EMA: 26-period Exponential Moving Average.
    - MACD Line: Difference between the short EMA and long EMA.
    - Signal Line: 9-period EMA of the MACD line.
- Returns:
    - macd: The MACD line values.
    - signal: The Signal line values.

4. Create MACD Chart

python

Copy code

```
def create_macd_chart(data):
    macd, signal = calculate_macd(data)
    fig = go.Figure()

    # MACD line trace
    fig.add_trace(go.Scatter(x=data.index, y=macd, mode='lines', name='MACD Line'))

    # Signal line trace
    fig.add_trace(go.Scatter(x=data.index, y=signal, mode='lines', name='Signal Line'))

    # Bar plot for MACD histogram
    fig.add_trace(go.Bar(x=data.index, y=macd - signal, name='MACD Histogram'))

    # Dark theme settings
    fig.update_layout(
        template="plotly_dark",
        title="MACD (Moving Average Convergence Divergence)",
        xaxis_title="Date",
        yaxis_title="MACD",
        font=dict(color="white"),
```

```
    plot_bgcolor="black",
    paper_bgcolor="black",
    hovermode="x",
  )
  return fig
```

- Generates a Plotly chart to visualize the MACD:
  - MACD Line: Represented as a line graph.
  - Signal Line: Another line graph overlaid with the MACD line.
  - MACD Histogram: Bar chart showing the difference between the MACD and Signal lines.
- Customizes the chart with:
  - Dark theme for modern aesthetics.
  - Titles for the chart and axes.
  - Hover interaction for detailed data points.

5. App Layout

python

Copy code

```
app.layout = html.Div(
  style={'backgroundColor': 'black', 'color': 'white', 'padding': '20px'},
  children=[
    html.H1("MACD Dashboard", style={'textAlign': 'center'}),

    # Input for ticker symbol
    html.Label("Enter Stock Ticker:", style={'font-size': '20px'}),
    dcc.Input(id='ticker-input', type='text', value='AAPL', style={'font-size': '20px', 'color': 'orange'}),

    # Dropdown for duration
    html.Label("Select Duration:", style={'font-size': '20px', 'color': 'orange'}),
    dcc.Dropdown(
      id='duration-dropdown',
      options=[
```

```
                {'label': '1 Month', 'value': '1mo'},
                {'label': '3 Months', 'value': '3mo'},
                {'label': '6 Months', 'value': '6mo'},
                {'label': '1 Year', 'value': '1y'},
                {'label': '2 Years', 'value': '2y'},
                {'label': '5 Years', 'value': '5y'},
                {'label': '10 Years', 'value': '10y'},
            ],
            value='1y',
            style={'font-size': '20px', 'color': 'orange'}
        ),

        # Graph to display MACD chart
        dcc.Graph(id='macd-chart', style={'height': '600px'}),
    ]
)
```

- Ticker Input: Text box (dcc.Input) for entering the stock ticker (default: AAPL).
- Duration Dropdown: Dropdown menu (dcc.Dropdown) for selecting the time range (default: 1 Year).
- Graph Component: Displays the MACD chart (dcc.Graph).

6. Callback for Interactivity

python

Copy code

```
@app.callback(
    Output('macd-chart', 'figure'),
    Input('ticker-input', 'value'),
    Input('duration-dropdown', 'value')
)
def update_chart(ticker, period):
    # Fetch the stock data
    data = fetch_stock_data(ticker, period)
```

```
# Generate the MACD chart
fig = create_macd_chart(data)
return fig
```

- Triggers updates to the chart when the user modifies:
    - Stock Ticker: Fetches data for the specified stock.
    - Duration: Adjusts the time range of data.
- Dynamically generates the MACD chart with updated data.

7. Run the App

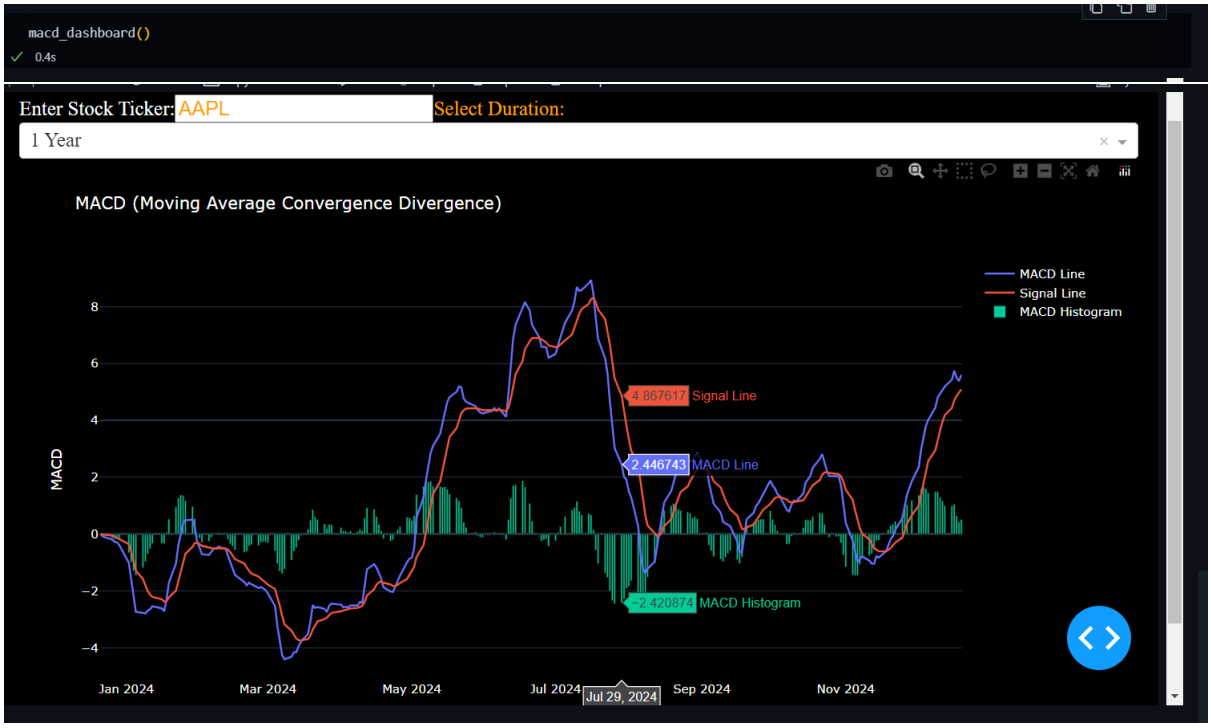
python

Copy code

```
app.run_server(debug=True)
```

- Launches the app in debug mode for live updates and error tracking.

Usage



rsi_graph()

This function, rsi_graph, creates an interactive Dash application to visualize the Relative Strength Index (RSI) of a stock. Here's a breakdown of its structure and functionality:

---

1. Purpose

The rsi_graph function builds a web-based dashboard to:

- Fetch stock price data using the yfinance library.
- Calculate the RSI, a popular momentum oscillator used in stock trading.
- Display the RSI as an interactive graph with visual indicators for overbought and oversold conditions.

2. User Experience

- The user opens the dashboard in a web browser.
- Enters a stock ticker (e.g., "AAPL") and selects a time duration.
- The app fetches the stock data, calculates the RSI, and updates the graph in real-time.
- The graph visually highlights overbought (RSI > 70) and oversold (RSI < 30) zones.

---

3. Key Features

- Interactive RSI calculation and visualization.
- Dynamic updates based on user inputs.
- Dark-themed interface for modern aesthetics.

Source Code

```
def rsi_graph():
    app = dash.Dash(__name__)

    # Function to fetch stock data using yfinance
    def fetch_stock_data(ticker, period='1y'):
        stock = yf.Ticker(ticker)
        data = stock.history(period=period)
        return data
```

```
    # Function to calculate RSI
    def calculate_rsi(data, period=14):
        delta = data['Close'].diff(1)
        gain = delta.where(delta > 0, 0)
        loss = -delta.where(delta < 0, 0)

        avg_gain = gain.rolling(window=period).mean()
        avg_loss = loss.rolling(window=period).mean()

        rs = avg_gain / avg_loss
        rsi = 100 - (100 / (1 + rs))
        return rsi

    # Function to generate RSI chart using plotly
    def create_rsi_chart(data):
        rsi = calculate_rsi(data)
        fig = go.Figure()

        # RSI line trace
        fig.add_trace(go.Scatter(x=data.index, y=rsi, mode='lines',
name='RSI'))

        # Dark theme settings
        fig.update_layout(
            template="plotly_dark",
            title="Relative Strength Index (RSI)",
            xaxis_title="Date",
            yaxis_title="RSI",
            font=dict(color="white"),
            plot_bgcolor="black",
            paper_bgcolor="black",
            hovermode="x",
        )

        # Add upper and lower bound lines
        fig.add_shape(
            type="line", x0=data.index[0], x1=data.index[-1], y0=70, y1=70,
            line=dict(color="red", dash="dash"), name='Overbought'
        )
        fig.add_shape(
            type="line", x0=data.index[0], x1=data.index[-1], y0=30, y1=30,
            line=dict(color="green", dash="dash"), name='Oversold'
        )
        return fig

    # Layout for the Dash app
    app.layout = html.Div(
```

```
        style={'backgroundColor': 'black', 'color': 'white', 'padding':
'20px'},
        children=[
            html.H1("RSI Dashboard", style={'textAlign': 'center'}),

            # Input for ticker symbol
            html.Label("Enter Stock Ticker:", style={'font-size': '20px'}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': 'orange'}),

            # Dropdown for duration
            html.Label("Select Duration:", style={'font-size': '20px',
'color': 'orange'}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Month', 'value': '1mo'},
                    {'label': '3 Months', 'value': '3mo'},
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': 'orange'}
            ),

            # Graph to display RSI chart
            dcc.Graph(id='rsi-chart', style={'height': '600px'}),
        ]
    )

    # Callback to update the chart based on user inputs
    @app.callback(
        Output('rsi-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch the stock data
        data = fetch_stock_data(ticker, period)

        # Generate the RSI chart
        fig = create_rsi_chart(data)
        return fig

    # Run the Dash app
```

```
    app.run_server(debug=True)
```

Code Explanation

2. Components

a. App Initialization

python

Copy code

app = dash.Dash(__name__)

- Creates a Dash app instance, which serves as the foundation for the dashboard.

---

b. Fetching Stock Data

python

Copy code

def fetch_stock_data(ticker, period='1y'):

    stock = yf.Ticker(ticker)

    data = stock.history(period=period)

    return data

- Uses the yfinance library to retrieve historical price data for a given stock ticker over a specified time period.

---

c. Calculating RSI

python

Copy code

def calculate_rsi(data, period=14):

    delta = data['Close'].diff(1)

    gain = delta.where(delta > 0, 0)

    loss = -delta.where(delta < 0, 0)

    avg_gain = gain.rolling(window=period).mean()

```
    avg_loss = loss.rolling(window=period).mean()


    rs = avg_gain / avg_loss
    rsi = 100 - (100 / (1 + rs))
    return rsi
```

- Input: Closing prices from the stock data.
- Steps:
    1. Computes the daily price change (delta).
    2. Separates gains (positive price changes) and losses (negative price changes).
    3. Calculates the rolling average of gains and losses over a 14-day period (default).
    4. Derives the RSI using the formula: RSI=100−(1001+RS)RSI = 100 - \left(\frac{100}{1 + \text{RS}} \right)RSI=100−(1+RS100)
        - RS (Relative Strength): The ratio of average gain to average loss.
- Output: A series of RSI values.

---

d. Generating the RSI Chart

python

Copy code

```
def create_rsi_chart(data):
    rsi = calculate_rsi(data)
    fig = go.Figure()


    fig.add_trace(go.Scatter(x=data.index, y=rsi, mode='lines', name='RSI'))


    fig.update_layout(
        template="plotly_dark",
        title="Relative Strength Index (RSI)",
        xaxis_title="Date",
        yaxis_title="RSI",
        font=dict(color="white"),
        plot_bgcolor="black",
```

```
    paper_bgcolor="black",
    hovermode="x",
  )

  fig.add_shape(
    type="line", x0=data.index[0], x1=data.index[-1], y0=70, y1=70,
    line=dict(color="red", dash="dash"), name='Overbought'
  )
  fig.add_shape(
    type="line", x0=data.index[0], x1=data.index[-1], y0=30, y1=30,
    line=dict(color="green", dash="dash"), name='Oversold'
  )
  return fig
```

- Creates a line chart using Plotly:
  - Plots the RSI values over time.
  - Adds two horizontal dashed lines:
    - At 70 (overbought condition).
    - At 30 (oversold condition).
- Applies a dark theme for better visual appeal.

---

e. Dashboard Layout

python

Copy code

```
app.layout = html.Div(
  style={'backgroundColor': 'black', 'color': 'white', 'padding': '20px'},
  children=[
    html.H1("RSI Dashboard", style={'textAlign': 'center'}),
    html.Label("Enter Stock Ticker:", style={'font-size': '20px'}),
    dcc.Input(id='ticker-input', type='text', value='AAPL', style={'font-size': '20px', 'color': 'orange'}),
    html.Label("Select Duration:", style={'font-size': '20px', 'color': 'orange'}),
    dcc.Dropdown(
```

```
        id='duration-dropdown',
        options=[{'label': '1 Month', 'value': '1mo'}, ...],
        value='1y',
        style={'font-size': '20px', 'color': 'orange'}
      ),
      dcc.Graph(id='rsi-chart', style={'height': '600px'}),
    ]
)
```

- Defines the layout with:
  - A header: Displays the dashboard title.
  - An input field: Lets the user specify the stock ticker symbol (default: AAPL).
  - A dropdown menu: Allows the user to select the data duration (e.g., 1 month, 1 year).
  - A graph component: Displays the RSI chart.

---

f. Callback to Update the Chart

python

Copy code

```
@app.callback(
    Output('rsi-chart', 'figure'),
    Input('ticker-input', 'value'),
    Input('duration-dropdown', 'value')
)
def update_chart(ticker, period):
    data = fetch_stock_data(ticker, period)
    fig = create_rsi_chart(data)
    return fig
```

- Updates the RSI chart dynamically based on:
  - Ticker input: Stock symbol entered by the user.
  - Duration dropdown: Selected time range.
- Fetches new data, recalculates RSI, and regenerates the chart.

g. Running the App

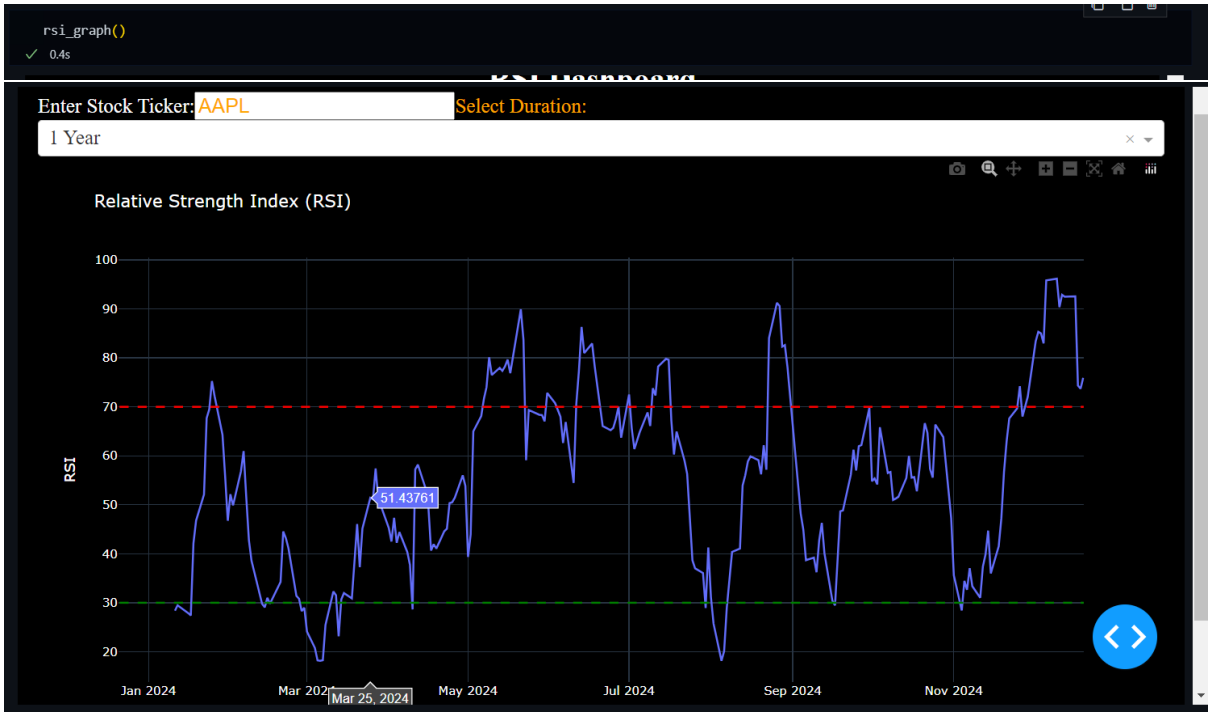
python

Copy code

app.run_server(debug=True)

- Starts the Dash app in debug mode for easy testing and troubleshooting.

Usage



bolinger_bands()

The bolinger_bands() function creates a Dash web application to visualize Bollinger Bands for a given stock. Bollinger Bands are a popular technical analysis tool that uses a moving average and standard deviations to define upper and lower price bounds. Here's a detailed explanation:

Primary Use of the Function

1. **Visualizing Stock Trends:**
    - Bollinger Bands are widely used in **technical analysis** to identify trends and potential buy/sell signals for stocks. This function visualizes:
        - **Closing prices** of the stock.
        - **Volatility** through upper and lower bands based on the stock's standard deviation.
2. **Interactive Stock Analysis Tool:**
    - Users can input a **stock ticker** (e.g., AAPL) and a **time period** (e.g., 1 year) to see how the stock has performed within the selected duration.
    - The app dynamically updates the visualization when the user changes the ticker or duration, making it a powerful exploratory tool.
3. **Decision-Making Aid for Traders/Investors:**
    - The app helps traders:
        - Identify **overbought** conditions when prices are near the **upper band**.
        - Spot **oversold** conditions when prices are near the **lower band**.
        - Analyze trends when prices consistently touch either the upper or lower bands.

Source Code

```
def bolinger_bands():
    app = dash.Dash(__name__)

    # Function to fetch stock data using yfinance
    def fetch_stock_data(ticker, period='1y'):
        stock = yf.Ticker(ticker)
        data = stock.history(period=period)
        return data

    # Function to calculate Bollinger Bands
    def calculate_bollinger_bands(data, window=20, num_std=2):
```

```
        rolling_mean = data['Close'].rolling(window).mean()
        rolling_std = data['Close'].rolling(window).std()
        upper_band = rolling_mean + (rolling_std * num_std)
        lower_band = rolling_mean - (rolling_std * num_std)
        return rolling_mean, upper_band, lower_band

    # Function to generate Bollinger Bands chart using plotly
    def create_bollinger_chart(data):
        rolling_mean, upper_band, lower_band = calculate_bollinger_bands(data)
        fig = go.Figure()

        # Closing price trace
        fig.add_trace(go.Scatter(x=data.index, y=data['Close'], mode='lines',
name='Close Price', line=dict(color='orange')))

        # Middle band (Moving Average) trace
        fig.add_trace(go.Scatter(x=data.index, y=rolling_mean, mode='lines',
name='Middle Band (SMA)', line=dict(color='blue')))

        # Upper band trace
        fig.add_trace(go.Scatter(x=data.index, y=upper_band, mode='lines',
name='Upper Band', line=dict(color='green')))

        # Lower band trace
        fig.add_trace(go.Scatter(x=data.index, y=lower_band, mode='lines',
name='Lower Band', line=dict(color='red')))

        # Dark theme settings
        fig.update_layout(
            template="plotly_dark",
            title="Bollinger Bands",
            xaxis_title="Date",
            yaxis_title="Price",
            font=dict(color="white"),
            plot_bgcolor="black",
            paper_bgcolor="black",
            hovermode="x",
        )
        return fig

    # Layout for the Dash app
    app.layout = html.Div(
        style={'backgroundColor': 'black', 'color': 'white', 'padding':
'20px'},
        children=[
            html.H1("Bollinger Bands Dashboard", style={'textAlign':
'center'}),
```

```
            # Input for ticker symbol
            html.Label("Enter Stock Ticker:", style={'font-size': '20px'}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': 'orange'}),

            # Dropdown for duration
            html.Label("Select Duration:", style={'font-size': '20px',
'color': 'orange'}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Month', 'value': '1mo'},
                    {'label': '3 Months', 'value': '3mo'},
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': 'orange'}
            ),

            # Graph to display Bollinger Bands chart
            dcc.Graph(id='bollinger-chart', style={'height': '600px'}),
        ]
    )

    # Callback to update the chart based on user inputs
    @app.callback(
        Output('bollinger-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch the stock data
        data = fetch_stock_data(ticker, period)

        # Generate the Bollinger Bands chart
        fig = create_bollinger_chart(data)
        return fig

    # Run the Dash app
    app.run_server(debug=True)
```

Code Explanation

1. Initializing the App

python

Copy code

```
app = dash.Dash(__name__)
```

- Initializes a Dash application that will display the Bollinger Bands chart.

---

2. Fetching Stock Data

python

Copy code

```
def fetch_stock_data(ticker, period='1y'):
    stock = yf.Ticker(ticker)
    data = stock.history(period=period)
    return data
```

- Uses the yfinance library to fetch historical stock data for a given ticker and time period.
- Input: Stock ticker (e.g., AAPL for Apple) and a time period (e.g., 1y for one year).
- Output: A DataFrame containing stock data, including the closing prices.

---

3. Calculating Bollinger Bands

python

Copy code

```
def calculate_bollinger_bands(data, window=20, num_std=2):
    rolling_mean = data['Close'].rolling(window).mean()
    rolling_std = data['Close'].rolling(window).std()
    upper_band = rolling_mean + (rolling_std * num_std)
    lower_band = rolling_mean - (rolling_std * num_std)
    return rolling_mean, upper_band, lower_band
```

- Bollinger Bands Formula:
    - Middle Band: Simple Moving Average (SMA) of the closing prices over a specified window (default: 20 days).

    - Upper Band: SMA + 2 standard deviations.
    - Lower Band: SMA - 2 standard deviations.
- Input: Stock data (closing prices), window size, and standard deviation multiplier.
- Output: Middle Band (rolling mean), Upper Band, and Lower Band.

---

4. Creating the Bollinger Bands Chart

python

Copy code

```
def create_bollinger_chart(data):
    rolling_mean, upper_band, lower_band = calculate_bollinger_bands(data)
    fig = go.Figure()


    # Closing Price
    fig.add_trace(go.Scatter(x=data.index, y=data['Close'], mode='lines', name='Close Price', line=dict(color='orange')))
    # Middle Band
    fig.add_trace(go.Scatter(x=data.index, y=rolling_mean, mode='lines', name='Middle Band (SMA)', line=dict(color='blue')))
    # Upper Band
    fig.add_trace(go.Scatter(x=data.index, y=upper_band, mode='lines', name='Upper Band', line=dict(color='green')))
    # Lower Band
    fig.add_trace(go.Scatter(x=data.index, y=lower_band, mode='lines', name='Lower Band', line=dict(color='red')))
```

- Uses Plotly to plot:
    - Closing price (orange line).
    - Middle Band (blue line).
    - Upper Band (green line).
    - Lower Band (red line).
- Sets a dark theme with black background and white text for better visualization.

---

5. Dash App Layout

python

Copy code

```
app.layout = html.Div(
    style={'backgroundColor': 'black', 'color': 'white', 'padding': '20px'},
    children=[
        html.H1("Bollinger Bands Dashboard", style={'textAlign': 'center'}),
        html.Label("Enter Stock Ticker:", style={'font-size': '20px'}),
        dcc.Input(id='ticker-input', type='text', value='AAPL', style={'font-size': '20px', 'color': 'orange'}),
        html.Label("Select Duration:", style={'font-size': '20px', 'color': 'orange'}),
        dcc.Dropdown(
            id='duration-dropdown',
            options=[...],  # Predefined durations
            value='1y',
            style={'font-size': '20px', 'color': 'orange'}
        ),
        dcc.Graph(id='bollinger-chart', style={'height': '600px'}),
    ]
)
```

- Layout Elements:
    - Header: "Bollinger Bands Dashboard."
    - Ticker Input: Text input to enter stock ticker.
    - Duration Dropdown: Dropdown menu to select a time period (e.g., 1 month, 1 year).
    - Chart Display: A placeholder for the Bollinger Bands chart.

---

6. Updating the Chart with User Input

python

Copy code

```
@app.callback(
    Output('bollinger-chart', 'figure'),
    Input('ticker-input', 'value'),
    Input('duration-dropdown', 'value')
```

)

```
def update_chart(ticker, period):
    data = fetch_stock_data(ticker, period)
    fig = create_bollinger_chart(data)
    return fig
```

- The callback function dynamically updates the chart based on:
    - Stock ticker entered by the user.
    - Duration selected from the dropdown.
- Fetches new stock data and calculates the Bollinger Bands, then generates the updated chart.

---

7. Running the App

python

Copy code

```
app.run_server(debug=True)
```

- Starts the Dash server in debug mode, allowing real-time updates when the user modifies inputs.

Use case

create_sma_dashboard()

The create_sma_dashboard() function is designed to create an interactive Simple Moving Averages (SMA) Dashboard using Dash and Plotly. Its primary significance lies in providing users with an intuitive way to visualize stock price trends and their corresponding SMAs over different time periods. This dashboard is particularly valuable for technical analysis, enabling traders and investors to identify market trends, potential buy/sell signals, and support/resistance levels.

Use Cases

1. **Stock Market Analysis for Traders**:
    - Helps traders analyze short-term, medium-term, and long-term trends.
    - Identifies entry and exit points based on SMA crossovers.
2. **Portfolio Management**:
    - Provides insights into the health and trends of portfolio stocks.
3. **Education and Training**:
    - Ideal for teaching technical analysis concepts, specifically SMAs.
4. **Customizable Framework**:
    - Can be extended to include additional technical indicators like **RSI**, **MACD**, or **Bollinger Bands**.
5. **Market Insights for Businesses**:
    - Useful for financial analysts and market researchers to present trends and insights.

By integrating interactive visualizations, the create_sma_dashboard() function becomes an invaluable tool for technical analysis, education, and market research.

Source Code

```
def create_sma_dashboard():
    # Initialize the Dash app with a dark theme
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors inspired by Bloomberg Terminal
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'sma_50': '#1f77b4',   # blue
        'sma_100': '#ff7f0e',  # orange
        'sma_200': '#2ca02c'   # green
    }
```

```
    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Simple Moving Averages (SMA) Dashboard",
style={'textAlign': 'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Month', 'value': '1mo'},
                    {'label': '3 Months', 'value': '3mo'},
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='sma-chart', style={'height': '600px'}),
        ]
    )

    # Define callback to update the chart
    @app.callback(
        Output('sma-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch the stock data
        data = yf.download(ticker, period=period)

        # Calculate SMAs
        data['SMA_50'] = data['Close'].rolling(window=50).mean()
        data['SMA_100'] = data['Close'].rolling(window=100).mean()
        data['SMA_200'] = data['Close'].rolling(window=200).mean()
```

```
        # Create the figure
        fig = go.Figure()
        fig.add_trace(go.Scatter(x=data.index, y=data['Close'], mode='lines',
name='Close Price', line=dict(color='orange')))
        fig.add_trace(go.Scatter(x=data.index, y=data['SMA_50'], mode='lines',
name='50-Day SMA', line=dict(color=colors['sma_50'])))
        fig.add_trace(go.Scatter(x=data.index, y=data['SMA_100'],
mode='lines', name='100-Day SMA', line=dict(color=colors['sma_100'])))
        fig.add_trace(go.Scatter(x=data.index, y=data['SMA_200'],
mode='lines', name='200-Day SMA', line=dict(color=colors['sma_200'])))

        # Update layout with dark theme
        fig.update_layout(
            template="plotly_dark",
            title=f"{ticker} - Simple Moving Averages",
            xaxis_title="Date",
            yaxis_title="Price",
            font=dict(color="white"),
            plot_bgcolor=colors['background'],
            paper_bgcolor=colors['background'],
            hovermode="x",
        )
        return fig

    # Run the app
    app.run_server(debug=True)

# Call the function to launch the app
#create_sma_dashboard()
```

Code Explanation

Code Explanation

1. Initialization of the Dash App:

python

Copy code

app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

   - Initializes a Dash app with a dark theme for a professional and trader-friendly interface.

2. Definition of Colors:

python

Copy code

```
colors = {
    'background': '#0E1117',
    'text': '#E0E0E0',
    'sma_50': '#1f77b4',
    'sma_100': '#ff7f0e',
    'sma_200': '#2ca02c'
}
```

- Specifies a dark background and color palette for text and SMAs to make the interface visually appealing and easy to interpret.

3. Layout Definition:

```
python
Copy code
app.layout = html.Div(
    style={'backgroundColor': colors['background'], 'padding': '20px'},
    children=[
        html.H1("Simple Moving Averages (SMA) Dashboard", style={'textAlign': 'center', 'color': colors['text']}),
        html.Label("Enter Stock Ticker:", style={'font-size': '20px', 'color': colors['text']}),
        dcc.Input(id='ticker-input', type='text', value='AAPL', style={'font-size': '20px', 'color': colors['text']}),
        html.Label("Select Duration:", style={'font-size': '20px', 'color': colors['text']}),
        dcc.Dropdown(
            id='duration-dropdown',
            options=[
                {'label': '1 Month', 'value': '1mo'},
                {'label': '3 Months', 'value': '3mo'},
                {'label': '6 Months', 'value': '6mo'},
                {'label': '1 Year', 'value': '1y'},
                {'label': '2 Years', 'value': '2y'},
                {'label': '5 Years', 'value': '5y'},
                {'label': '10 Years', 'value': '10y'},
            ],
```

```
        value='1y',
        style={'font-size': '20px', 'color': colors['text']}
      ),
      dcc.Graph(id='sma-chart', style={'height': '600px'}),
  ]
)
```

- Defines the app layout with:
  - Input Field: Accepts a stock ticker (e.g., AAPL).
  - Dropdown Menu: Lets users select the duration for analysis.
  - Graph Component: Displays the stock price and SMAs.

4. Callback for Chart Updates:

```
python
Copy code
@app.callback(
  Output('sma-chart', 'figure'),
  Input('ticker-input', 'value'),
  Input('duration-dropdown', 'value')
)
def update_chart(ticker, period):
  # Fetch stock data and calculate SMAs
  data = yf.download(ticker, period=period)
  data['SMA_50'] = data['Close'].rolling(window=50).mean()
  data['SMA_100'] = data['Close'].rolling(window=100).mean()
  data['SMA_200'] = data['Close'].rolling(window=200).mean()

  # Create Plotly figure
  fig = go.Figure()
  fig.add_trace(go.Scatter(x=data.index, y=data['Close'], mode='lines', name='Close Price', line=dict(color='orange')))
  fig.add_trace(go.Scatter(x=data.index, y=data['SMA_50'], mode='lines', name='50-Day SMA', line=dict(color=colors['sma_50'])))
```

```
    fig.add_trace(go.Scatter(x=data.index, y=data['SMA_100'], mode='lines', name='100-Day SMA', line=dict(color=colors['sma_100'])))
    fig.add_trace(go.Scatter(x=data.index, y=data['SMA_200'], mode='lines', name='200-Day SMA', line=dict(color=colors['sma_200'])))

    # Update layout with a dark theme
    fig.update_layout(
        template="plotly_dark",
        title=f"{ticker} - Simple Moving Averages",
        xaxis_title="Date",
        yaxis_title="Price",
        font=dict(color="white"),
        plot_bgcolor=colors['background'],
        paper_bgcolor=colors['background'],
        hovermode="x",
    )
    return fig
```

- Input Processing: Dynamically updates the chart based on user inputs (stock ticker and duration).
- SMA Calculation: Calculates the 50-day, 100-day, and 200-day SMAs using the rolling average of the stock's closing prices.
- Visualization: Creates an interactive time-series graph showing stock price and SMAs with Plotly.

5. Run the App:

python

Copy code

```
app.run_server(debug=True)
```

- Starts the Dash server, allowing users to access the dashboard locally.

Usecase


create_sma_dashboard()
1 Year
AAPL - Simple Moving Averages
Close Price
50-Day SMA
100-Day SMA
200-Day SMA
Price
250
240
230
220
210
200
190
180
170
213.25 Close Price
188.9354 50-Day SMA
182.8645 100-Day SMA
Jan 2024
Mar 2024
May 2024
Jun 26, 2024
Sep 2024
Nov 2024

create_ema_dashboard()

Significance of the Function

The create_ema_dashboard() function creates an **Exponential Moving Averages (EMA) Dashboard** using Dash and Plotly, allowing users to analyze the performance of stock prices with an emphasis on recent data. By plotting the EMAs for different periods, it highlights trends and potential signals for traders and investors. EMAs are widely used in technical analysis to identify momentum, support/resistance levels, and market trends with higher weight on recent price movements compared to SMAs.

Use Cases

1. **Momentum Trading**:
    - Helps traders identify strong trends and their continuation or reversal based on EMA crossovers.
2. **Market Timing**:
    - Assists in timing entry and exit points by focusing on EMAs that adapt to recent market activity.
3. **Stock Analysis for Investors**:
    - Enables investors to monitor stock price trends and potential support/resistance levels over varying timeframes.
4. **Financial Education**:
    - Ideal for teaching the significance and calculation of EMAs in technical analysis.
5. **Customizable for Advanced Analysis**:
    - Can be extended to include additional indicators like **Volume Weighted Average Price (VWAP)**, **MACD**, or overlays like Bollinger Bands.

The create_ema_dashboard() function is a robust tool for dynamic and interactive stock analysis, making it invaluable for technical traders, educators, and financial analysts.

Source Code

```
def create_ema_dashboard():
    # Initialize the Dash app with a dark theme
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors inspired by Bloomberg Terminal
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
```

```
        'ema_20': '#1f77b4',   # blue
        'ema_50': '#ff7f0e',   # orange
        'ema_100': '#2ca02c'   # green
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Exponential Moving Averages (EMA) Dashboard",
style={'textAlign': 'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Month', 'value': '1mo'},
                    {'label': '3 Months', 'value': '3mo'},
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='ema-chart', style={'height': '600px'}),
        ]
    )

    # Define callback to update the chart
    @app.callback(
        Output('ema-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch the stock data
        data = yf.download(ticker, period=period)
```

```
        # Calculate EMAs
        data['EMA_20'] = data['Close'].ewm(span=20, adjust=False).mean()
        data['EMA_50'] = data['Close'].ewm(span=50, adjust=False).mean()
        data['EMA_100'] = data['Close'].ewm(span=100, adjust=False).mean()

        # Create the figure
        fig = go.Figure()
        fig.add_trace(go.Scatter(x=data.index, y=data['Close'], mode='lines',
name='Close Price', line=dict(color='orange')))
        fig.add_trace(go.Scatter(x=data.index, y=data['EMA_20'], mode='lines',
name='20-Day EMA', line=dict(color=colors['ema_20'])))
        fig.add_trace(go.Scatter(x=data.index, y=data['EMA_50'], mode='lines',
name='50-Day EMA', line=dict(color=colors['ema_50'])))
        fig.add_trace(go.Scatter(x=data.index, y=data['EMA_100'],
mode='lines', name='100-Day EMA', line=dict(color=colors['ema_100'])))

        # Update layout with dark theme
        fig.update_layout(
            template="plotly_dark",
            title=f"{ticker} - Exponential Moving Averages",
            xaxis_title="Date",
            yaxis_title="Price",
            font=dict(color="white"),
            plot_bgcolor=colors['background'],
            paper_bgcolor=colors['background'],
            hovermode="x",
        )
        return fig

    # Run the app
    app.run_server(debug=True)

# Call the function to launch the app
#create_ema_dashboard()
```

Code Explanation

 Initialization of the Dash App:

python

Copy code

app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

- Initializes a Dash app with a dark theme for a sleek, professional interface.

 Definition of Colors:

python

Copy code

```
colors = {
   'background': '#0E1117',
   'text': '#E0E0E0',
   'ema_20': '#1f77b4',  # blue
   'ema_50': '#ff7f0e',  # orange
   'ema_100': '#2ca02c'  # green
}
```

- Provides a dark color scheme inspired by professional trading platforms like Bloomberg Terminal, with distinct colors for each EMA line.

 Dashboard Layout:

python

Copy code

```
app.layout = html.Div(
   style={'backgroundColor': colors['background'], 'padding': '20px'},
   children=[
      html.H1("Exponential Moving Averages (EMA) Dashboard", style={'textAlign': 'center', 'color': colors['text']}),

      html.Label("Enter Stock Ticker:", style={'font-size': '20px', 'color': colors['text']}),
      dcc.Input(id='ticker-input', type='text', value='AAPL', style={'font-size': '20px', 'color': colors['text']}),

      html.Label("Select Duration:", style={'font-size': '20px', 'color': colors['text']}),
      dcc.Dropdown(
         id='duration-dropdown',
         options=[
            {'label': '1 Month', 'value': '1mo'},
            {'label': '3 Months', 'value': '3mo'},
            {'label': '6 Months', 'value': '6mo'},
```

```
            {'label': '1 Year', 'value': '1y'},
            {'label': '2 Years', 'value': '2y'},
            {'label': '5 Years', 'value': '5y'},
            {'label': '10 Years', 'value': '10y'},
        ],
        value='1y',
        style={'font-size': '20px', 'color': colors['text']}
    ),

    dcc.Graph(id='ema-chart', style={'height': '600px'}),
  ]
)
```

- Stock Ticker Input: Lets users specify a stock symbol (e.g., AAPL).
- Duration Dropdown: Allows users to select the timeframe for analysis.
- Graph Component: Displays the stock price and calculated EMAs dynamically.

 Callback to Update the Chart:

```
python
Copy code
@app.callback(
  Output('ema-chart', 'figure'),
  Input('ticker-input', 'value'),
  Input('duration-dropdown', 'value')
)
def update_chart(ticker, period):
  # Fetch the stock data
  data = yf.download(ticker, period=period)

  # Calculate EMAs
  data['EMA_20'] = data['Close'].ewm(span=20, adjust=False).mean()
  data['EMA_50'] = data['Close'].ewm(span=50, adjust=False).mean()
  data['EMA_100'] = data['Close'].ewm(span=100, adjust=False).mean()
```

```
    # Create the figure
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=data.index, y=data['Close'], mode='lines', name='Close Price', line=dict(color='orange')))
    fig.add_trace(go.Scatter(x=data.index, y=data['EMA_20'], mode='lines', name='20-Day EMA', line=dict(color=colors['ema_20'])))
    fig.add_trace(go.Scatter(x=data.index, y=data['EMA_50'], mode='lines', name='50-Day EMA', line=dict(color=colors['ema_50'])))
    fig.add_trace(go.Scatter(x=data.index, y=data['EMA_100'], mode='lines', name='100-Day EMA', line=dict(color=colors['ema_100'])))

    # Update layout with dark theme
    fig.update_layout(
        template="plotly_dark",
        title=f"{ticker} - Exponential Moving Averages",
        xaxis_title="Date",
        yaxis_title="Price",
        font=dict(color="white"),
        plot_bgcolor=colors['background'],
        paper_bgcolor=colors['background'],
        hovermode="x",
    )
    return fig
```

- Data Fetching: Retrieves stock price data for the specified ticker and duration using Yahoo Finance (yfinance).
- EMA Calculation: Uses Pandas' ewm() method to compute 20-day, 50-day, and 100-day EMAs, giving more weight to recent prices.
- Chart Creation: Builds a Plotly figure with time-series plots for stock prices and EMAs, applying the defined color scheme.

 Run the Application:

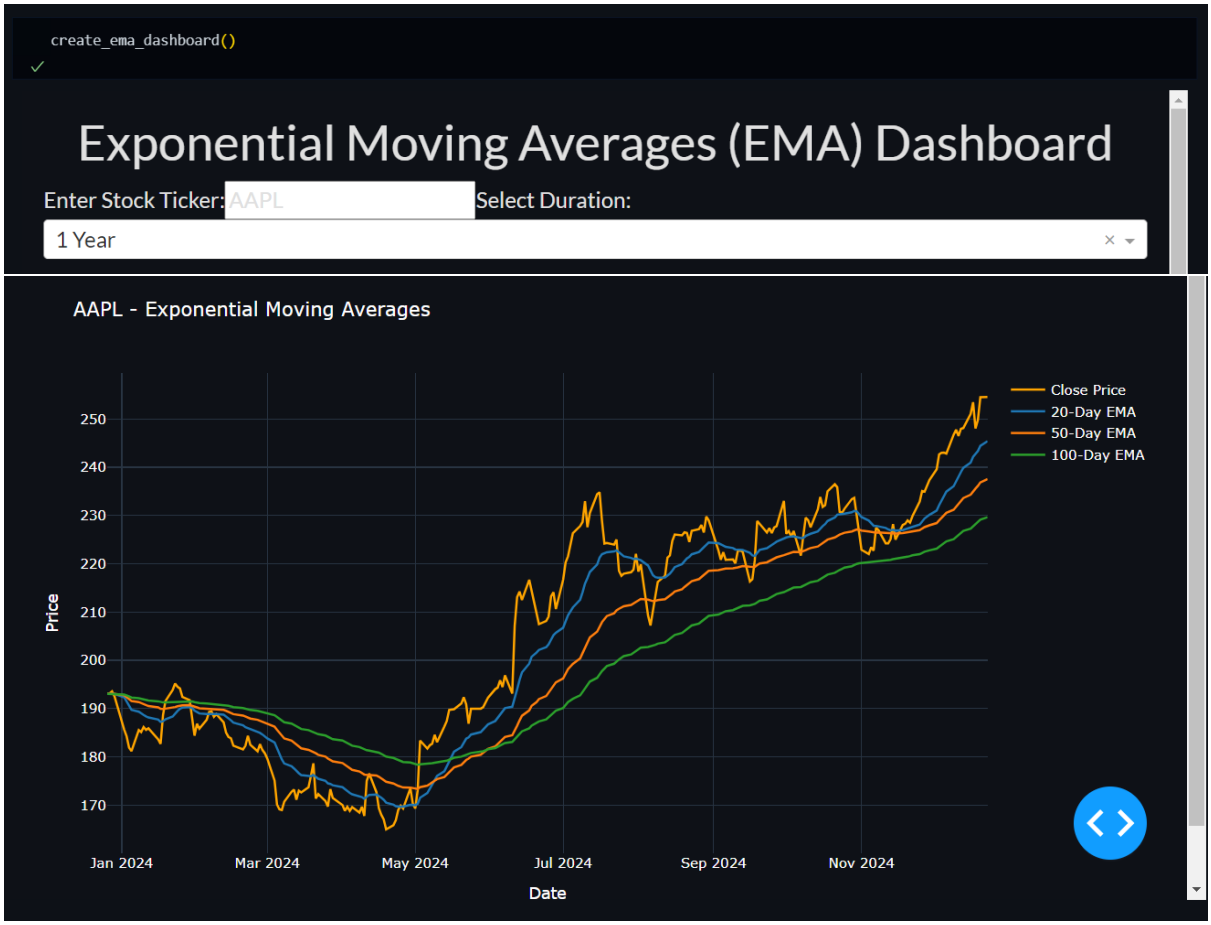
python

Copy code

app.run_server(debug=True)

- Launches the Dash server locally with debugging enabled for easier troubleshooting.

Usecase



create_ema_dashboard()

Significance of the Function

The create_ema_dashboard() function creates an **Exponential Moving Averages (EMA) Dashboard** using Dash and Plotly, allowing users to analyze the performance of stock prices with an emphasis on recent data. By plotting the EMAs for different periods, it highlights trends and potential signals for traders and investors. EMAs are widely used in technical analysis to identify momentum, support/resistance levels, and market trends with higher weight on recent price movements compared to SMAs.

Use Cases

1. **Momentum Trading**:
    - Helps traders identify strong trends and their continuation or reversal based on EMA crossovers.
2. **Market Timing**:
    - Assists in timing entry and exit points by focusing on EMAs that adapt to recent market activity.
3. **Stock Analysis for Investors**:
    - Enables investors to monitor stock price trends and potential support/resistance levels over varying timeframes.
4. **Financial Education**:
    - Ideal for teaching the significance and calculation of EMAs in technical analysis.
5. **Customizable for Advanced Analysis**:
    - Can be extended to include additional indicators like **Volume Weighted Average Price (VWAP)**, **MACD**, or overlays like Bollinger Bands.

The create_ema_dashboard() function is a robust tool for dynamic and interactive stock analysis, making it invaluable for technical traders, educators, and financial analysts.

Source Code

```
def create_stochastic_dashboard():
    # Initialize the Dash app with a dark theme
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors inspired by Bloomberg Terminal
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
```

```
        'k_line': '#1f77b4',   # blue
        'd_line': '#ff7f0e',   # orange
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Stochastic Oscillator Dashboard", style={'textAlign':
'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Month', 'value': '1mo'},
                    {'label': '3 Months', 'value': '3mo'},
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='stochastic-chart', style={'height': '600px'}),
        ]
    )

    # Define callback to update the chart
    @app.callback(
        Output('stochastic-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch the stock data
        data = yf.download(ticker, period=period)

        # Calculate the Stochastic Oscillator
```

```
        high_14 = data['High'].rolling(window=14).max()
        low_14 = data['Low'].rolling(window=14).min()
        data['%K'] = 100 * ((data['Close'] - low_14) / (high_14 - low_14))
        data['%D'] = data['%K'].rolling(window=3).mean()

        # Create the figure
        fig = go.Figure()

        # Plot %K line
        fig.add_trace(go.Scatter(x=data.index, y=data['%K'], mode='lines',
name='%K Line (14-day)', line=dict(color=colors['k_line'])))

        # Plot %D line (3-day moving average of %K)
        fig.add_trace(go.Scatter(x=data.index, y=data['%D'], mode='lines',
name='%D Line (3-day MA)', line=dict(color=colors['d_line'])))

        # Add 20 and 80 reference lines for overbought/oversold levels
        fig.add_hline(y=80, line_dash="dash", line_color="red",
annotation_text="Overbought", annotation_position="top right")
        fig.add_hline(y=20, line_dash="dash", line_color="green",
annotation_text="Oversold", annotation_position="bottom right")

        # Update layout with dark theme
        fig.update_layout(
            template="plotly_dark",
            title=f"{ticker} - Stochastic Oscillator",
            xaxis_title="Date",
            yaxis_title="Stochastic Oscillator (%)",
            font=dict(color="white"),
            plot_bgcolor=colors['background'],
            paper_bgcolor=colors['background'],
            hovermode="x",
            yaxis=dict(range=[0, 100])
        )
        return fig

    # Run the app
    app.run_server(debug=True)

# Call the function to launch the app
#create_stochastic_dashboard()
```

Code Explanation

 Initialization of the Dash App:

python

Copy code

app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

- Initializes a Dash app with a dark theme for a sleek, professional interface.

 Definition of Colors:

python

Copy code

colors = {

  'background': '#0E1117',

  'text': '#E0E0E0',

  'ema_20': '#1f77b4', # blue

  'ema_50': '#ff7f0e', # orange

  'ema_100': '#2ca02c' # green

}

- Provides a dark color scheme inspired by professional trading platforms like Bloomberg Terminal, with distinct colors for each EMA line.

 Dashboard Layout:

python

Copy code

app.layout = html.Div(

  style={'backgroundColor': colors['background'], 'padding': '20px'},

  children=[

    html.H1("Exponential Moving Averages (EMA) Dashboard", style={'textAlign': 'center', 'color': colors['text']}),

    html.Label("Enter Stock Ticker:", style={'font-size': '20px', 'color': colors['text']}),

    dcc.Input(id='ticker-input', type='text', value='AAPL', style={'font-size': '20px', 'color': colors['text']}),

    html.Label("Select Duration:", style={'font-size': '20px', 'color': colors['text']}),

    dcc.Dropdown(

      id='duration-dropdown',

```
            options=[
                {'label': '1 Month', 'value': '1mo'},
                {'label': '3 Months', 'value': '3mo'},
                {'label': '6 Months', 'value': '6mo'},
                {'label': '1 Year', 'value': '1y'},
                {'label': '2 Years', 'value': '2y'},
                {'label': '5 Years', 'value': '5y'},
                {'label': '10 Years', 'value': '10y'},
            ],
            value='1y',
            style={'font-size': '20px', 'color': colors['text']}
        ),

        dcc.Graph(id='ema-chart', style={'height': '600px'}),
    ]
)
```

- Stock Ticker Input: Lets users specify a stock symbol (e.g., AAPL).
- Duration Dropdown: Allows users to select the timeframe for analysis.
- Graph Component: Displays the stock price and calculated EMAs dynamically.

 Callback to Update the Chart:

python

Copy code

```
@app.callback(
    Output('ema-chart', 'figure'),
    Input('ticker-input', 'value'),
    Input('duration-dropdown', 'value')
)
def update_chart(ticker, period):
    # Fetch the stock data
    data = yf.download(ticker, period=period)
```

```
    # Calculate EMAs
    data['EMA_20'] = data['Close'].ewm(span=20, adjust=False).mean()
    data['EMA_50'] = data['Close'].ewm(span=50, adjust=False).mean()
    data['EMA_100'] = data['Close'].ewm(span=100, adjust=False).mean()

    # Create the figure
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=data.index, y=data['Close'], mode='lines', name='Close Price', line=dict(color='orange')))
    fig.add_trace(go.Scatter(x=data.index, y=data['EMA_20'], mode='lines', name='20-Day EMA', line=dict(color=colors['ema_20'])))
    fig.add_trace(go.Scatter(x=data.index, y=data['EMA_50'], mode='lines', name='50-Day EMA', line=dict(color=colors['ema_50'])))
    fig.add_trace(go.Scatter(x=data.index, y=data['EMA_100'], mode='lines', name='100-Day EMA', line=dict(color=colors['ema_100'])))

    # Update layout with dark theme
    fig.update_layout(
        template="plotly_dark",
        title=f"{ticker} - Exponential Moving Averages",
        xaxis_title="Date",
        yaxis_title="Price",
        font=dict(color="white"),
        plot_bgcolor=colors['background'],
        paper_bgcolor=colors['background'],
        hovermode="x",
    )
    return fig
```

- Data Fetching: Retrieves stock price data for the specified ticker and duration using Yahoo Finance (yfinance).
- EMA Calculation: Uses Pandas' ewm() method to compute 20-day, 50-day, and 100-day EMAs, giving more weight to recent prices.

- Chart Creation: Builds a Plotly figure with time-series plots for stock prices and EMAs, applying the defined color scheme.

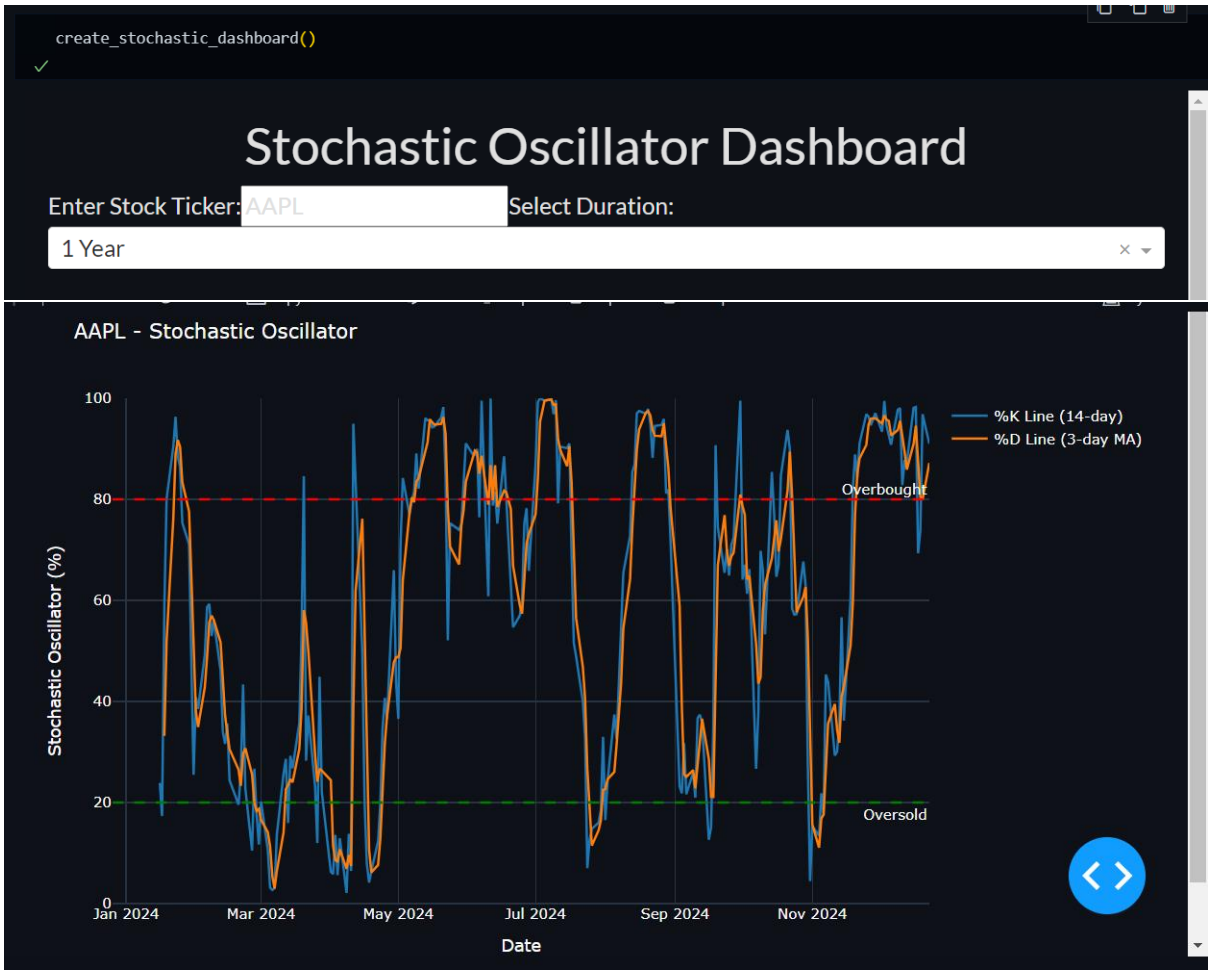
 Run the Application:

python

Copy code

app.run_server(debug=True)

Usecase



create_obv_dashboard()

Significance of the Function

The create_obv_dashboard function is a comprehensive solution for visualizing the On-Balance Volume (OBV) of stocks over a user-specified time period. OBV is a momentum indicator that uses volume flow to predict stock price changes. By providing an interactive and visually appealing dashboard, this function empowers users to analyze stock trends and make data-driven decisions. Its design is particularly useful for traders, investors, or financial analysts.

Use Cases

1. **Stock Trend Analysis**:
   Traders can use the OBV dashboard to identify accumulation and distribution phases in a stock, helping to predict price breakouts or reversals.
2. **Investment Decision-Making**:
   Investors can gauge the overall market sentiment for a stock by analyzing the OBV trend relative to price changes.
3. **Educational Tool**:
   Financial educators can use the dashboard as a visual aid to teach concepts like OBV and volume-based analysis.
4. **Market Research**:
   Financial analysts can include OBV charts as part of their market research presentations to stakeholders.

The create_obv_dashboard function offers an elegant, interactive, and informative approach to financial data visualization, suitable for both professional and educational contexts.

Source Code

```
def create_obv_dashboard():
    # Initialize the Dash app with a dark theme
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors inspired by Bloomberg Terminal
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'obv_line': '#1f77b4',   # blue
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
```

```
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("On-Balance Volume (OBV) Dashboard", style={'textAlign':
'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Month', 'value': '1mo'},
                    {'label': '3 Months', 'value': '3mo'},
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='obv-chart', style={'height': '600px'}),
        ]
    )

    # Define callback to update the chart
    @app.callback(
        Output('obv-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch the stock data
        data = yf.download(ticker, period=period)

        # Calculate On-Balance Volume (OBV)
        obv = [0]  # Start with the first OBV value at 0
        for i in range(1, len(data)):
            if data['Close'][i] > data['Close'][i - 1]:
                obv.append(obv[-1] + data['Volume'][i])
            elif data['Close'][i] < data['Close'][i - 1]:
                obv.append(obv[-1] - data['Volume'][i])
```

```
            else:
                obv.append(obv[-1])
        data['OBV'] = obv

        # Create the figure
        fig = go.Figure()

        # Plot OBV line
        fig.add_trace(go.Scatter(x=data.index, y=data['OBV'], mode='lines',
name='OBV', line=dict(color=colors['obv_line'])))

        # Update layout with dark theme
        fig.update_layout(
            template="plotly_dark",
            title=f"{ticker} - On-Balance Volume (OBV)",
            xaxis_title="Date",
            yaxis_title="On-Balance Volume",
            font=dict(color="white"),
            plot_bgcolor=colors['background'],
            paper_bgcolor=colors['background'],
            hovermode="x"
        )
        return fig

    # Run the app
    app.run_server(debug=True)
```

Code Explanation

 Initializing the App:
The Dash application is initialized with a dark theme using an external stylesheet (DARKLY). This ensures a professional and modern interface for the dashboard.

 Defining Colors:
A color scheme is specified for the dashboard's elements, including the background, text, and the OBV line. These colors are chosen for aesthetic coherence, inspired by Bloomberg Terminal's design.

 Defining the Layout:
The layout of the dashboard includes:

- A header displaying the title "On-Balance Volume (OBV) Dashboard," styled for prominence and readability.
- An input field where users can enter a stock ticker symbol (e.g., AAPL).
- A dropdown menu offering various time periods, such as 1 month, 1 year, or 5 years, to define the range of historical data.

- A graphical display area (dcc.Graph) to visualize the OBV line chart.

 Callback Function for Chart Updates:
A callback function updates the OBV chart dynamically based on the stock ticker and time period entered by the user.

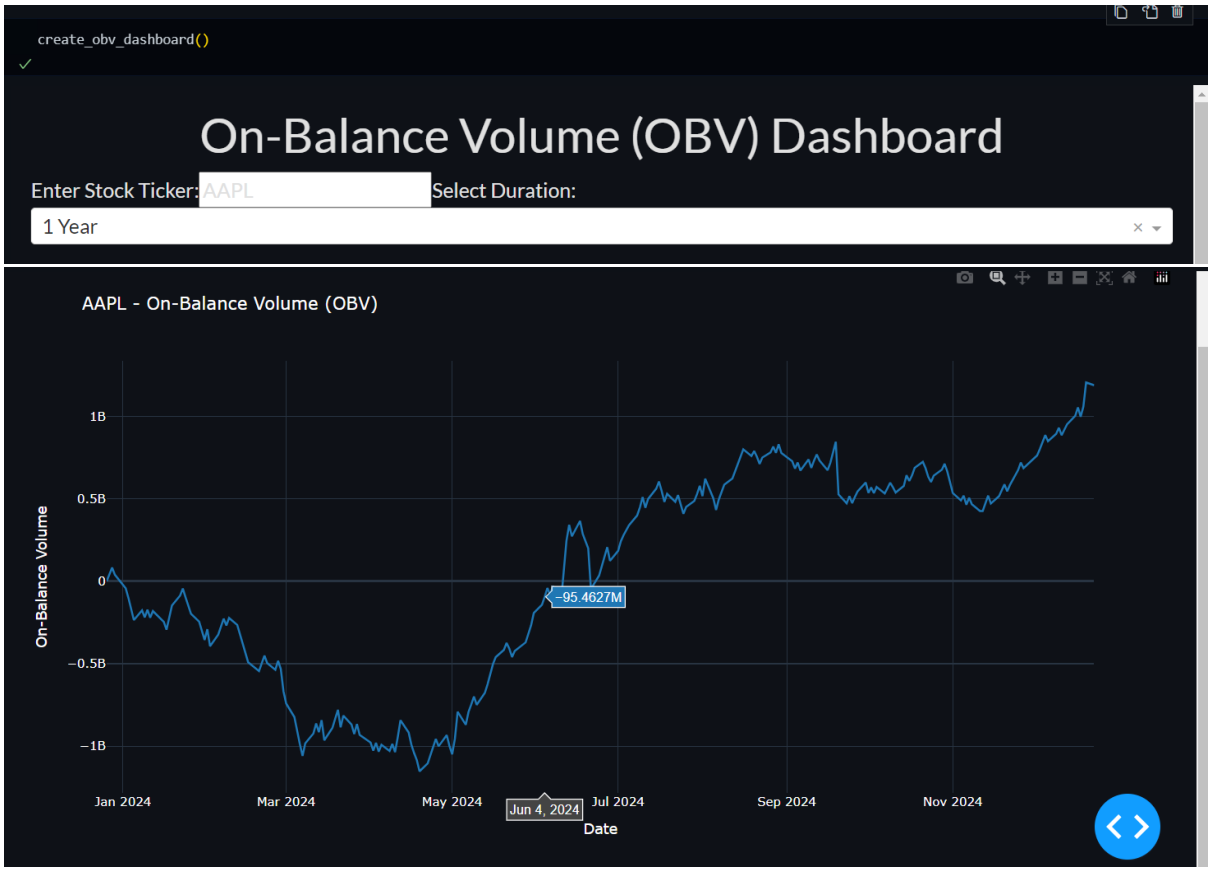
- The stock data is fetched using the yfinance library for the specified ticker and duration.
- OBV is calculated by iterating through the stock data.
    - If the closing price increases compared to the previous day, the day's volume is added to the OBV.
    - If the closing price decreases, the day's volume is subtracted.
    - If the closing price remains the same, the OBV remains unchanged.
- The OBV values are added as a new column to the stock data.

 Creating the Chart:
A Plotly figure is created, with the OBV plotted as a blue line. The chart includes:

- X-axis labels showing dates.
- Y-axis labels for OBV values.
- A title incorporating the selected stock ticker.
  The chart is styled with a dark theme, matching the dashboard's overall design.

 Running the App:
The app is run in debug mode, making it accessible in a web browser with live updates as changes are made.

Usecase


create_obv_dashboard()
On-Balance Volume (OBV) Dashboard
Enter Stock Ticker: AAPL
Select Duration:
1 Year
AAPL - On-Balance Volume (OBV)
On-Balance Volume
1B
0.5B
0
−0.5B
−1B
−95.4627M
Jan 2024
Mar 2024
May 2024
Jun 4, 2024
Jul 2024
Sep 2024
Nov 2024
Date

calculate_atr()

Significance of the Functions

The calculate_atr function computes the Average True Range (ATR), a widely used volatility indicator in technical analysis. It helps traders and analysts measure market volatility over a specified period. The create_atr_dashboard function complements this by creating an interactive dashboard for visualizing ATR, allowing users to input a stock ticker and a duration for analysis.

Use Cases

1. **Volatility Analysis**:
   Traders can use the dashboard to measure and compare the volatility of different stocks over specific time periods.
2. **Risk Management**:
   Investors can use ATR to adjust position sizes, setting stop-loss levels based on a stock's volatility.
3. **Market Research**:
   Financial analysts can include ATR visualizations in presentations to stakeholders, offering insights into market conditions.
4. **Educational Tool**:
   The dashboard can be used in training programs to demonstrate the importance of ATR in technical analysis.

The combination of calculate_atr and create_atr_dashboard provides a robust framework for analyzing stock market volatility interactively and effectively.

Source Code

```
def calculate_atr(data, period=14):
    """Calculate the Average True Range (ATR)"""
    high = data['High']
    low = data['Low']
    close = data['Close']

    # Calculate the True Range (TR)
    tr1 = high - low
    tr2 = abs(high - close.shift(1))
    tr3 = abs(low - close.shift(1))

    tr = pd.concat([tr1, tr2, tr3], axis=1)
    tr = tr.max(axis=1)  # Maximum of the three True Range calculations

    # Calculate ATR
```

```
    atr = tr.rolling(window=period).mean()
    return atr

def create_atr_dashboard():
    # Initialize the Dash app with a dark theme
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors inspired by Bloomberg Terminal
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'atr_line': '#FF6347',  # Tomato color for ATR line
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Average True Range (ATR) Dashboard", style={'textAlign':
'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Month', 'value': '1mo'},
                    {'label': '3 Months', 'value': '3mo'},
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='atr-chart', style={'height': '600px'}),
        ]
    )

    # Define callback to update the chart
```

```
    @app.callback(
        Output('atr-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch the stock data
        data = yf.download(ticker, period=period)

        # Calculate ATR
        atr = calculate_atr(data)

        # Create the figure
        fig = go.Figure()

        # Plot ATR line
        fig.add_trace(go.Scatter(x=data.index, y=atr, mode='lines',
name='ATR', line=dict(color=colors['atr_line'])))

        # Update layout with dark theme
        fig.update_layout(
            template="plotly_dark",
            title=f"{ticker} - Average True Range (ATR)",
            xaxis_title="Date",
            yaxis_title="ATR",
            font=dict(color="white"),
            plot_bgcolor=colors['background'],
            paper_bgcolor=colors['background'],
            hovermode="x"
        )
        return fig

    # Run the app
    app.run_server(debug=True)

# Call the function to launch the app
#create_atr_dashboard()
```

Code Explanation

calculate_atr:

This function computes the ATR for a stock dataset, providing insights into the stock's volatility:

- Input Data: The function expects a dataset containing High, Low, and Close prices.
- True Range (TR) Calculation:
    - The True Range is calculated using three components:

        - The difference between the daily high and low prices.
        - The absolute difference between the daily high and the previous day's close.
        - The absolute difference between the daily low and the previous day's close.
    - These three values are combined into a single DataFrame, and the maximum of the three is selected for each day as the True Range.
- ATR Calculation:
    - The ATR is computed as the rolling mean of the True Range over a specified period (default: 14 days).
- Output: The function returns the ATR as a pandas Series.

---

create_atr_dashboard:

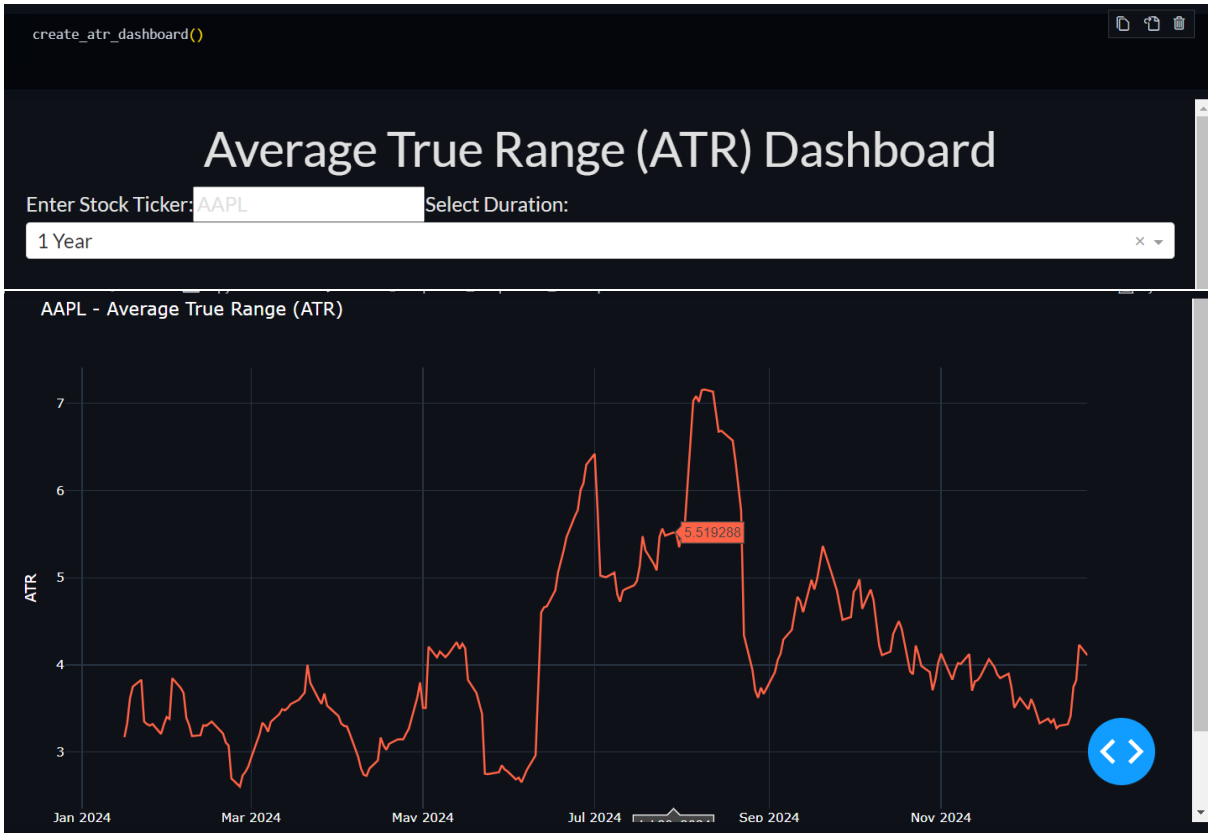
This function creates a dashboard to visualize the ATR for a given stock and time period:

- App Initialization:
  A Dash application is created using a dark theme for a professional and visually appealing interface.
- Defining Colors:
  A color scheme is specified for the dashboard, including a unique "Tomato" color for the ATR line to make it visually distinct.
- Defining the Layout:
  The dashboard includes:
    - A title displayed at the center of the page.
    - An input field where users can enter a stock ticker symbol.
    - A dropdown menu offering time durations for analysis, such as 1 month, 1 year, or 5 years.
    - A graphical display area (dcc.Graph) to visualize the ATR chart.
- Callback Function for Chart Updates:
  A callback function dynamically updates the ATR chart based on the user's inputs for stock ticker and time period.
    - Stock data is fetched using yfinance.
    - The ATR is calculated using the calculate_atr function.
    - A Plotly figure is created to display the ATR:
        - X-axis represents dates.
        - Y-axis represents ATR values.
        - A title includes the stock ticker and ATR label.

- The chart is styled with a dark theme to match the dashboard's design.

- Running the App:
  The application is launched in debug mode, making it accessible in a web browser and supporting live updates during development.

Usecase



fibonacci_dashboard()

Significance of the Function

The fibonacci_dashboard function creates an interactive web-based dashboard for analyzing Fibonacci retracement levels. Fibonacci levels are a popular technical analysis tool used to identify potential support and resistance levels in a stock's price movements.

Use Cases

1. **Technical Analysis for Traders**:
    - Identify potential support and resistance levels to make informed trading decisions.
2. **Educational Insights**:
    - Provide a visual demonstration of how Fibonacci levels can help predict price movements.
3. **Market Research and Reporting**:
    - Generate retracement level charts for stock analysis presentations.
4. **Interactive Exploration**:
    - Allow users to explore different stocks and durations dynamically.

The fibonacci_dashboard function offers a robust, visually appealing, and interactive tool for applying Fibonacci retracement techniques to stock price analysis.

Source Code

```
def fibonacci_dashboard():
    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'fib_lines': '#FF6347',  # Tomato color for Fibonacci lines
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
```

```
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Fibonacci Retracement Dashboard", style={'textAlign':
'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Month', 'value': '1mo'},
                    {'label': '3 Months', 'value': '3mo'},
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='fibonacci-chart', style={'height': '600px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('fibonacci-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch the stock data
        data = yf.download(ticker, period=period)

        # Calculate Fibonacci retracement levels
        high_price = data['High'].max()
        low_price = data['Low'].min()
        diff = high_price - low_price
        levels = {
            '0%': high_price,
            '23.6%': high_price - 0.236 * diff,
```

```
            '38.2%': high_price - 0.382 * diff,
            '50%': high_price - 0.5 * diff,
            '61.8%': high_price - 0.618 * diff,
            '100%': low_price,
        }

        # Create the figure
        fig = go.Figure()

        # Add the stock's closing price line
        fig.add_trace(go.Scatter(x=data.index, y=data['Close'], mode='lines',
name='Close Price', line=dict(color='cyan')))

        # Plot Fibonacci retracement levels as horizontal lines
        for level_name, level_value in levels.items():
            fig.add_hline(y=level_value, line_dash="dash",
line_color=colors['fib_lines'], annotation_text=level_name,
annotation_position="top right")

        # Update layout with dark theme
        fig.update_layout(
            template="plotly_dark",
            title=f"{ticker} - Fibonacci Retracement Levels",
            xaxis_title="Date",
            yaxis_title="Price",
            font=dict(color="white"),
            plot_bgcolor=colors['background'],
            paper_bgcolor=colors['background'],
            hovermode="x"
        )
        return fig

    # Run the app
    app.run_server(debug=True)
```

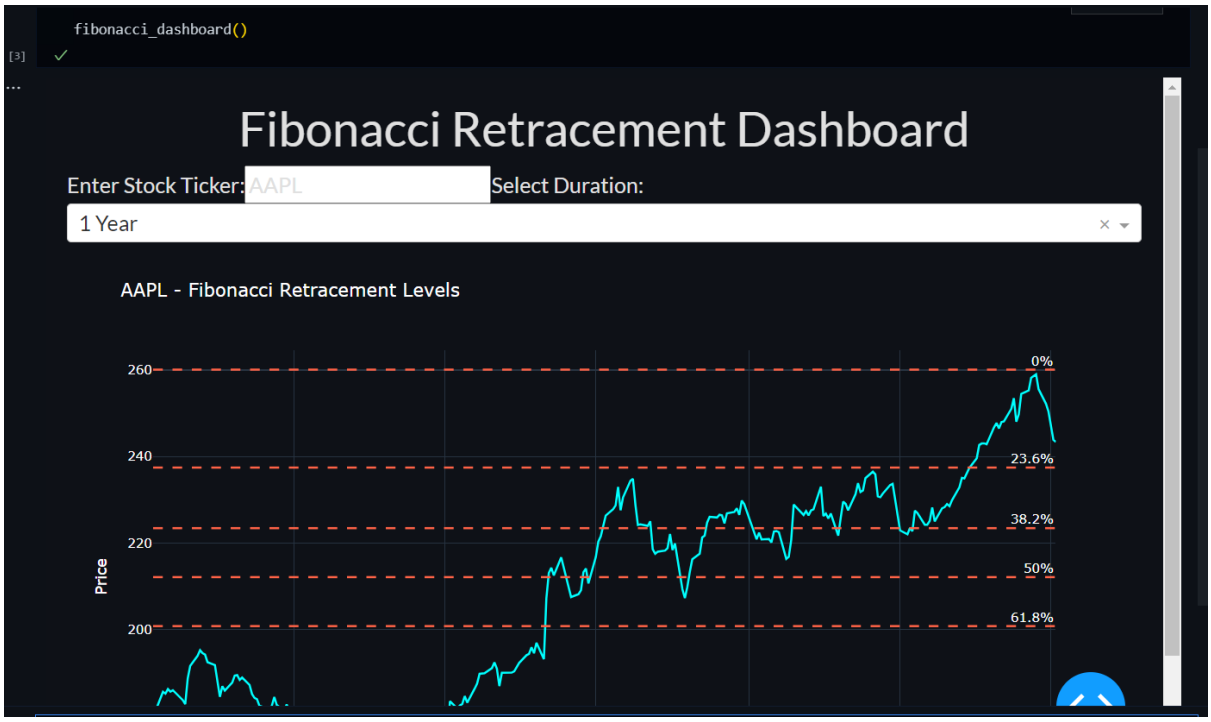
Code Explanation

Key Features:

1. App Initialization:
    - The Dash app is created with the DARKLY theme from Bootstrap, giving it a professional and visually appealing dark interface.
2. Color Scheme:
    - A custom color scheme is defined, including a "Tomato" color (#FF6347) for the Fibonacci retracement lines, ensuring visual clarity.

3. Layout Design:
    - Title: The dashboard's title, "Fibonacci Retracement Dashboard," is displayed prominently at the top, centered, and styled with the specified text color.
    - Stock Input Field: A text input allows users to enter a stock ticker symbol, with AAPL as the default value.
    - Duration Dropdown: A dropdown menu provides options for selecting the duration of historical stock data, ranging from 1 month to 10 years.
    - Graph Display: A graph area (dcc.Graph) is included to render the chart displaying Fibonacci retracement levels and stock prices.
4. Callback Function for Chart Updates:
    - Inputs: The callback takes user inputs for the stock ticker and selected duration.
    - Stock Data Retrieval: Stock data is fetched using the yfinance library based on the selected inputs.
    - Fibonacci Level Calculation:
        - The highest (High) and lowest (Low) stock prices are identified from the historical data.
        - The price difference between these levels is used to calculate Fibonacci retracement levels (0%, 23.6%, 38.2%, 50%, 61.8%, and 100%).
    - Chart Creation:
        - A line plot is created for the stock's closing prices.
        - Horizontal dashed lines represent Fibonacci retracement levels, annotated with their respective percentages.
5. Styling:
    - A dark theme is applied to the graph, with white text and cyan-colored closing price lines for improved readability.
    - The background and hover modes are customized for a seamless user experience.
6. App Execution:
    - The app.run_server(debug=True) statement launches the dashboard in debug mode, enabling live updates during development.

Example Usage



pe_ratio_dashboard()

Purpose

The pe_ratio_dashboard function creates an interactive dashboard to analyze the Price-to-Earnings (P/E) ratio of a selected stock over a specified time period. The P/E ratio is a key financial metric used to evaluate the relative valuation of a company.

Code Breakdown

Functionality:

The function initializes a Dash application and provides an interface for visualizing the P/E ratio of a stock, allowing users to explore its valuation trends dynamically.

Overview

The pe_ratio_dashboard function offers a versatile, visually appealing, and informative tool for analyzing a stock's valuation through its P/E ratio. It is designed to cater to investors, analysts, and anyone interested in financial metrics.

Use Cases

1. **Valuation Analysis for Investors**:
    - Evaluate a stock's valuation trends over time to inform investment decisions.
2. **Educational Resource**:
    - Provide an intuitive way to understand how P/E ratios fluctuate with stock prices and earnings.
3. **Corporate Reporting**:
    - Generate P/E ratio trends for presentations or reports.
4. **Interactive Financial Exploration**:
    - Empower users to dynamically explore the P/E ratio of any publicly traded stock.

Source Code

```
def pe_ratio_dashboard():
    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])
```

```
    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'pe_line': '#FF6347',  # Tomato color for P/E ratio line
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Price-to-Earnings (P/E) Ratio Dashboard",
style={'textAlign': 'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Month', 'value': '1mo'},
                    {'label': '3 Months', 'value': '3mo'},
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='pe-ratio-chart', style={'height': '600px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('pe-ratio-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
```

```
        # Fetch the stock data
        data = yf.download(ticker, period=period)

        # Get the trailing twelve-month (TTM) EPS for P/E calculation
        stock = yf.Ticker(ticker)
        ttm_eps = stock.info.get('trailingEps')

        if ttm_eps:
            # Calculate the P/E ratio by dividing close price by TTM EPS
            data['P/E Ratio'] = data['Close'] / ttm_eps
        else:
            # If no EPS data, P/E ratio cannot be calculated
            data['P/E Ratio'] = None

        # Create the figure
        fig = go.Figure()

        # Add the P/E ratio line
        fig.add_trace(go.Scatter(x=data.index, y=data['P/E Ratio'],
mode='lines', name='P/E Ratio', line=dict(color=colors['pe_line'])))

        # Update layout with dark theme
        fig.update_layout(
            template="plotly_dark",
            title=f"{ticker} - Price-to-Earnings (P/E) Ratio Over Time",
            xaxis_title="Date",
            yaxis_title="P/E Ratio",
            font=dict(color="white"),
            plot_bgcolor=colors['background'],
            paper_bgcolor=colors['background'],
            hovermode="x"
        )
        return fig

    # Run the app
    app.run_server(debug=True)
```

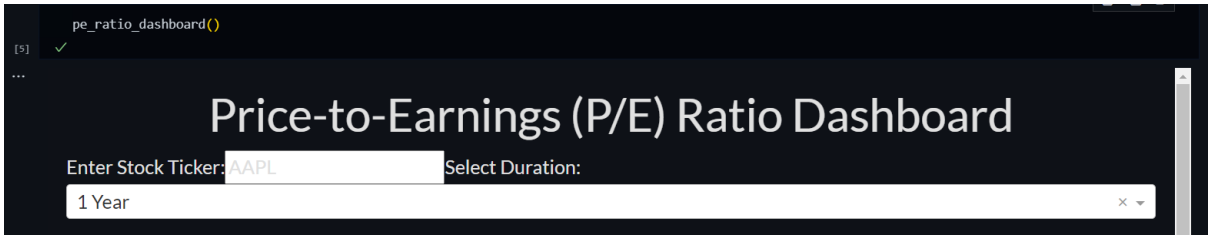
Code Explanation

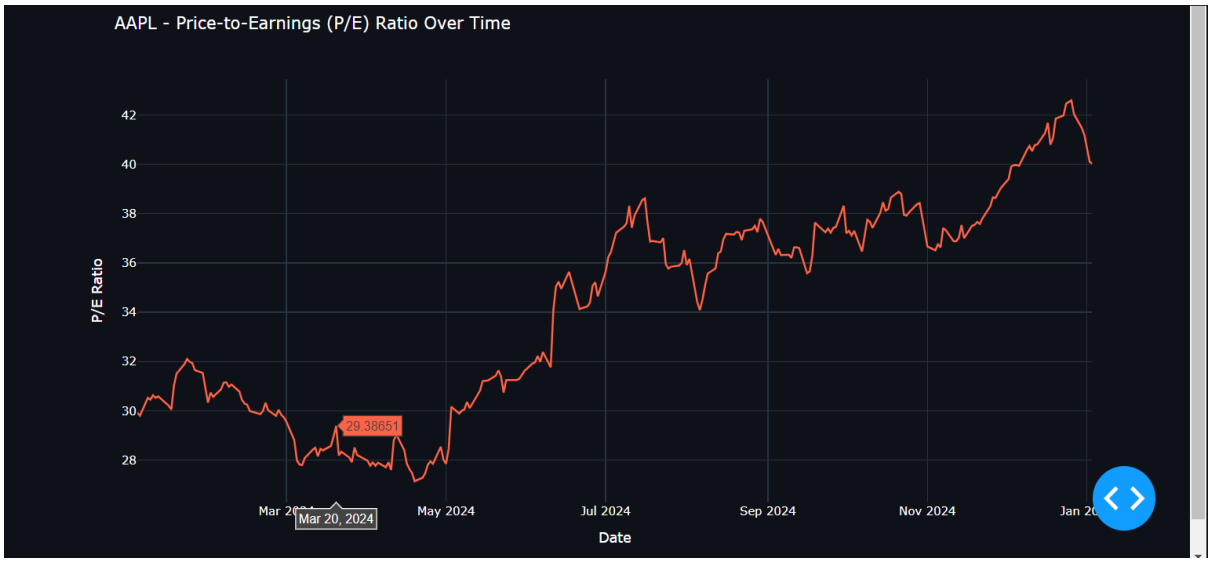
Key Features:

1. **App Initialization:**
    - **The Dash app is created using the DARKLY Bootstrap theme, ensuring a clean, dark-themed design suitable for financial dashboards.**

2. **Color Scheme:**
    - **Custom colors are defined, including a "Tomato" color (#FF6347) for the P/E ratio line to make it visually distinct.**
3. **Dashboard Layout:**
    - **Title: The dashboard title, "Price-to-Earnings (P/E) Ratio Dashboard," is displayed at the top, centered, and styled for prominence.**
    - **Stock Input Field: A text input field allows users to specify the stock ticker symbol, with a default value set to AAPL.**
    - **Duration Dropdown: A dropdown menu lets users choose the time range for analysis, with options from 1 month to 10 years.**
    - **Graph Area: A graph component (dcc.Graph) is included for displaying the P/E ratio trends.**
4. **Callback Functionality:**
    - **Inputs: The callback takes two user inputs — the stock ticker symbol and the time period.**
    - **Stock Data Retrieval: Historical stock data is fetched using the yfinance library.**
    - **P/E Ratio Calculation:**
        - **The trailing twelve-month (TTM) earnings per share (EPS) is retrieved using the stock's metadata.**
        - **The P/E ratio is calculated by dividing the stock's closing price by the TTM EPS.**
        - **If TTM EPS is unavailable, the P/E ratio is set to None, and the chart will reflect missing data.**
    - **Graph Creation:**
        - **A line plot is generated to display the P/E ratio over time.**
        - **The chart uses a dark theme for readability and a modern design aesthetic.**
5. **Styling:**
    - **The layout features a consistent dark background with white text, ensuring high contrast.**
    - **The graph's hover mode is customized for smooth user interaction.**
6. **App Execution:**
    - **The app.run_server(debug=True) statement starts the server in debug mode, allowing for live updates during development.**

Example Usage





pb_ratio_dashboard()

Purpose

The pb_ratio_dashboard function creates an interactive dashboard for analyzing the Price-to-Book (P/B) ratio of a selected stock over a user-specified time period. The P/B ratio compares a stock's market price to its book value, making it a valuable tool for assessing company valuation.

Use Cases

1. **Investor Valuation Analysis**:
    - Examine trends in a stock's P/B ratio to evaluate whether it is undervalued or overvalued compared to its book value.
2. **Educational Tool**:
    - Offer a dynamic way to understand how P/B ratios change with stock prices and book value.
3. **Financial Reporting**:
    - Generate P/B ratio charts for presentations or financial discussions.
4. **Interactive Exploration**:
    - Allow users to analyze the P/B ratio of any publicly traded stock dynamically.

Overview

The pb_ratio_dashboard function is an interactive and visually appealing tool for analyzing the P/B ratio of stocks. It caters to investors, financial analysts, and anyone interested in understanding company valuations through a simple, user-friendly interface.

Source Code

```
def pb_ratio_dashboard():
    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'pb_line': '#1E90FF',  # DodgerBlue color for P/B ratio line
```

```
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Price-to-Book (P/B) Ratio Dashboard", style={'textAlign':
'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Month', 'value': '1mo'},
                    {'label': '3 Months', 'value': '3mo'},
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='pb-ratio-chart', style={'height': '600px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('pb-ratio-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch the stock data
        data = yf.download(ticker, period=period)

        # Get the book value per share (BVPS) for P/B calculation
        stock = yf.Ticker(ticker)
        bvps = stock.info.get('bookValue')
```

```
        if bvps:
            # Calculate the P/B ratio by dividing close price by BVPS
            data['P/B Ratio'] = data['Close'] / bvps
        else:
            # If no BVPS data, P/B ratio cannot be calculated
            data['P/B Ratio'] = None

        # Create the figure
        fig = go.Figure()

        # Add the P/B ratio line
        fig.add_trace(go.Scatter(x=data.index, y=data['P/B Ratio'],
mode='lines', name='P/B Ratio', line=dict(color=colors['pb_line'])))

        # Update layout with dark theme
        fig.update_layout(
            template="plotly_dark",
            title=f"{ticker} - Price-to-Book (P/B) Ratio Over Time",
            xaxis_title="Date",
            yaxis_title="P/B Ratio",
            font=dict(color="white"),
            plot_bgcolor=colors['background'],
            paper_bgcolor=colors['background'],
            hovermode="x"
        )
        return fig

    # Run the app
    app.run_server(debug=True)
```

Code Explanation

Key Features:

1. App Initialization:
    - A Dash app is created using the DARKLY theme, offering a dark, professional look suitable for financial dashboards.
2. Color Scheme:
    - Custom colors are defined, with "DodgerBlue" (#1E90FF) for the P/B ratio line to ensure clarity and visual appeal.
3. Dashboard Layout:

- Title: Displays the dashboard title, "Price-to-Book (P/B) Ratio Dashboard," at the top, styled for visibility and centered alignment.
- Stock Input Field: Allows users to specify the stock ticker symbol, with a default value of AAPL.
- Duration Dropdown: Enables users to select the time range for analysis, ranging from 1 month to 10 years.
- Graph Area: Includes a graph component (dcc.Graph) to visualize P/B ratio trends.

4. Callback Functionality:
    - Inputs: Accepts the stock ticker and time range as user inputs.
    - Stock Data Retrieval: Fetches historical stock data using the yfinance library.
    - P/B Ratio Calculation:
        - Retrieves the book value per share (BVPS) from the stock metadata.
        - Calculates the P/B ratio by dividing the closing price by BVPS.
        - If BVPS is unavailable, assigns None to the P/B ratio, resulting in missing data points in the chart.
    - Graph Creation:
        - Generates a line plot to display the P/B ratio over time.
        - Employs a dark theme for readability and aesthetic consistency.
5. Styling:
    - The layout features a cohesive dark background with white text for high contrast.
    - The graph's hover mode is optimized for smooth and intuitive interaction.
6. App Execution:
    - The app.run_server(debug=True) statement starts the Dash server in debug mode, enabling real-time updates during development.

Example Usage

dividend_yield_dashboard()

Purpose

The dividend_yield_dashboard function creates an interactive dashboard that visualizes the dividend yield of a selected stock over a user-defined time period. Dividend yield is an essential metric for income-focused investors as it represents the annual return on investment through dividends relative to the stock price.

Use Cases

1. **Income Investing**:
    - Analyze dividend yield trends to identify stocks with consistent or increasing returns.
2. **Portfolio Evaluation**:
    - Compare dividend yields of different stocks over time to make informed investment decisions.
3. **Educational Insights**:
    - Provide a practical tool for teaching dividend yield concepts and their importance in investing.
4. **Dynamic Analysis**:
    - Allow users to explore the impact of stock price changes on dividend yield.

Overview

The dividend_yield_dashboard function is an engaging and practical tool for analyzing dividend yield trends. It caters to income investors, financial educators, and analysts, offering an intuitive interface for exploring the relationship between stock prices and dividend returns.

Source Code

```
def dividend_yield_dashboard():
    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
```

```
        'dividend_line': '#FFD700',  # Gold color for dividend yield line
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Dividend Yield Dashboard", style={'textAlign': 'center',
'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Month', 'value': '1mo'},
                    {'label': '3 Months', 'value': '3mo'},
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='dividend-yield-chart', style={'height': '600px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('dividend-yield-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch the stock data
        data = yf.download(ticker, period=period)

        # Fetch dividend yield data
        stock = yf.Ticker(ticker)
```

```
        annual_dividend = stock.info.get('dividendRate', 0)

        if annual_dividend and not data.empty:
            # Calculate dividend yield as a percentage
            data['Dividend Yield (%)'] = (annual_dividend / data['Close']) *
100
        else:
            data['Dividend Yield (%)'] = None

        # Create the figure
        fig = go.Figure()

        # Add the Dividend Yield line
        fig.add_trace(go.Scatter(x=data.index, y=data['Dividend Yield (%)'],
mode='lines', name='Dividend Yield',
line=dict(color=colors['dividend_line'])))

        # Update layout with dark theme
        fig.update_layout(
            template="plotly_dark",
            title=f"{ticker} - Dividend Yield Over Time",
            xaxis_title="Date",
            yaxis_title="Dividend Yield (%)",
            font=dict(color="white"),
            plot_bgcolor=colors['background'],
            paper_bgcolor=colors['background'],
            hovermode="x"
        )
        return fig

    # Run the app
    app.run_server(debug=True)
```

Code Explanation

Key Features:

1. App Initialization:
    - The Dash app is initialized with the DARKLY theme, offering a sleek dark interface.
2. Color Scheme:
    - Custom colors are defined, with "Gold" (#FFD700) used for the dividend yield line, emphasizing key data points.
3. Dashboard Layout:
    - Title: A central, bold title "Dividend Yield Dashboard" to establish the context of the visualization.

- Stock Input Field: Allows users to specify the stock ticker, with a default value of AAPL.
- Duration Dropdown: Enables users to select the time period for analysis, ranging from 1 month to 10 years.
- Graph Area: Contains a dcc.Graph element where the dividend yield trends are displayed dynamically.

4. Callback Functionality:
    - Inputs: Takes the stock ticker and the duration as user inputs.
    - Stock Data Retrieval:
        - Fetches historical stock price data using yfinance.
        - Extracts the stock's annual dividend rate from its metadata.
    - Dividend Yield Calculation:
        - Computes the dividend yield as a percentage using the formula: Dividend Yield (%)=(Annual DividendClosing Price)×100\text{Dividend Yield (\%)} = \left(\frac{\text{Annual Dividend}}{\text{Closing Price}}\right) \times 100Dividend Yield (%)=(Closing PriceAnnual Dividend)×100
        - If the dividend data or stock data is unavailable, assigns None to the dividend yield, resulting in missing data points in the chart.
    - Graph Creation:
        - Generates a line plot to display the dividend yield over time.
        - The line uses a distinct gold color for visual prominence.
        - A dark theme enhances readability and creates a professional aesthetic.
5. Styling:
    - The layout uses a consistent dark background with white text for contrast.
    - The hover mode is optimized for a smooth and informative user experience.
6. App Execution:
    - The app.run_server(debug=True) statement launches the Dash app in debug mode for development and testing.

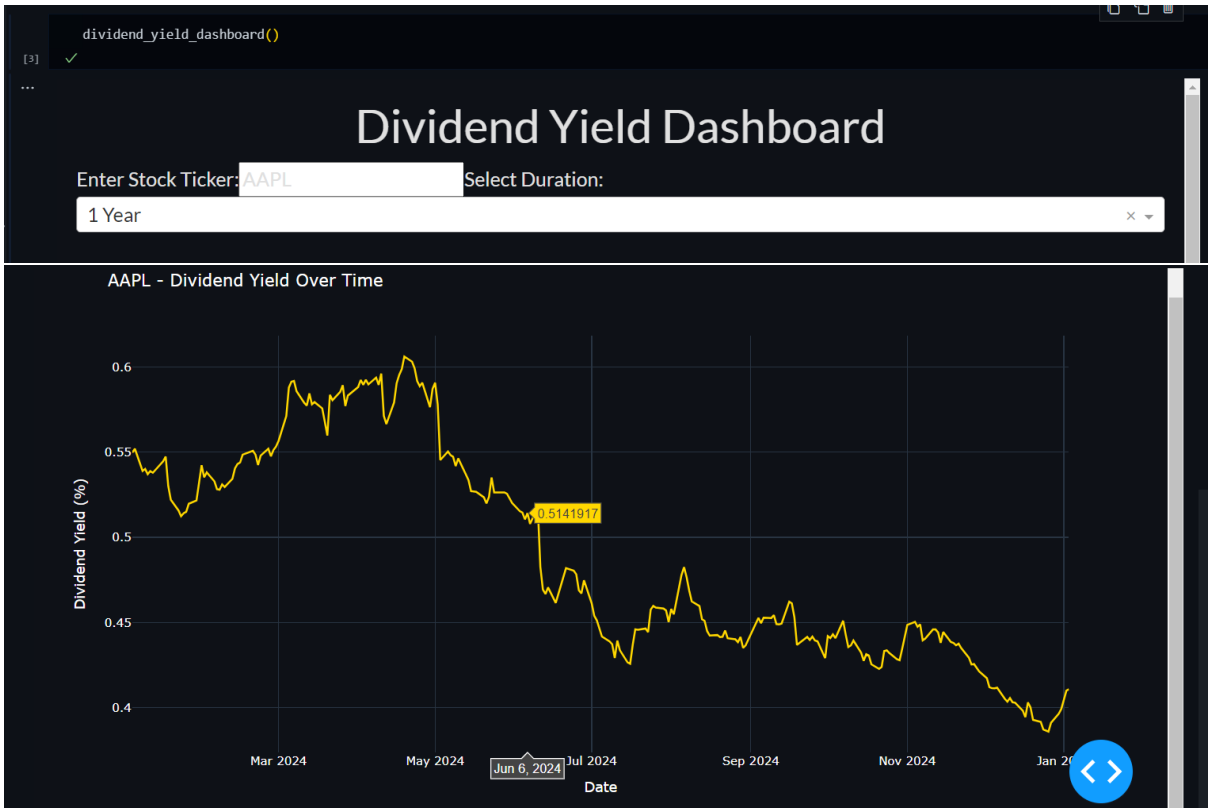
Use Cases

1. Income Investing:
    - Analyze dividend yield trends to identify stocks with consistent or increasing returns.
2. Portfolio Evaluation:

- Compare dividend yields of different stocks over time to make informed investment decisions.

3. Educational Insights:
    - Provide a practical tool for teaching dividend yield concepts and their importance in investing.
4. Dynamic Analysis:
    - Allow users to explore the impact of stock price changes on dividend yield.

Example Usage



revenue_growth_dashboard()

Purpose

The revenue_growth_dashboard function creates an interactive dashboard to visualize the revenue growth of a selected stock over a specified duration. Revenue growth, calculated as a year-over-year (YoY) percentage change, is a key metric for assessing a company's financial health and growth trajectory.

Code Breakdown

Functionality

The function initializes a Dash application, allowing users to explore and analyze revenue growth trends for publicly traded companies.

Use Cases

1. **Growth Analysis**:
    - Understand how a company's revenue has evolved over time.
2. **Investment Decision-Making**:
    - Identify companies with consistent or accelerating revenue growth.
3. **Comparative Analysis**:
    - Compare revenue growth trends across multiple stocks by changing the ticker.
4. **Education**:
    - Provide a hands-on tool to learn about revenue growth and its importance in fundamental analysis.

Key Metrics

- **Revenue Growth (%)**: Measures YoY changes in revenue, reflecting a company's ability to expand its business and attract new customers.

Overview

The revenue_growth_dashboard is a user-friendly tool for visualizing revenue growth trends. It offers a clean and interactive way to analyze a critical financial metric, helping investors, analysts, and educators make data-driven decisions.

Source Code

```
def revenue_growth_dashboard():
    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'revenue_line': '#FFA500',  # Orange color for Revenue Growth line
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Revenue Growth Dashboard", style={'textAlign': 'center',
'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='revenue-growth-chart', style={'height': '600px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('revenue-growth-chart', 'figure'),
        Input('ticker-input', 'value'),
```

```
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch financial data
        stock = yf.Ticker(ticker)
        financials = stock.financials

        if financials is not None:
            # Calculate revenue growth (year-over-year)
            financials = financials.transpose()
            financials['Revenue Growth'] = financials['Total
Revenue'].pct_change() * 100  # Convert to percentage

            # Filter data for the selected duration
            financials = financials.tail(int(period.replace('y', '')))

            # Create the figure
            fig = go.Figure()

            # Add the Revenue Growth line
            fig.add_trace(go.Scatter(x=financials.index, y=financials['Revenue
Growth'], mode='lines+markers',
                                     name='Revenue Growth (%)',
line=dict(color=colors['revenue_line'])))

            # Update layout with dark theme
            fig.update_layout(
                template="plotly_dark",
                title=f"{ticker} - Revenue Growth Over Time",
                xaxis_title="Date",
                yaxis_title="Revenue Growth (%)",
                font=dict(color="white"),
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background'],
                hovermode="x"
            )
        else:
            # Display a message if financial data is unavailable
            fig = go.Figure()
            fig.add_annotation(
                text="Revenue data not available",
                xref="paper", yref="paper",
                showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
```

```
            )

        return fig

    # Run the app
    app.run_server(debug=True)
```

Code Explanation

Key Features:

1. App Initialization:
    - The Dash app is initialized with the DARKLY theme, providing a modern, dark-themed interface.
2. Color Scheme:
    - Custom colors are defined, with "Orange" (#FFA500) used for the revenue growth line, making it visually prominent.
3. Dashboard Layout:
    - Title: Displays "Revenue Growth Dashboard" as a centered heading.
    - Stock Input Field: A text input box for entering the stock ticker, defaulting to AAPL.
    - Duration Dropdown: Allows users to select a time period (1 to 10 years) for analyzing revenue growth.
    - Graph Area: Contains a dcc.Graph element for dynamically displaying the revenue growth chart.
4. Callback Functionality:
    - Inputs: Takes the stock ticker and the selected duration as user inputs.
    - Data Retrieval:
        - Retrieves financial data using the yfinance library.
        - Extracts and processes the Total Revenue field from the company's financial statements.
    - Revenue Growth Calculation:
        - Computes revenue growth as a percentage using the formula: Revenue Growth (%)=(Current Revenue−Previous RevenuePrevious Revenue) ×100\text{Revenue Growth (\%)} = \left(\frac{\text{Current Revenue} - \text{Previous Revenue}}{\text{Previous Revenue}}\right) \times 100Revenue Growth (%)=(Previous RevenueCurrent Revenue−Previous Reve nue)×100

- Chart Construction:
    - Plots the revenue growth data as a line chart with markers, emphasizing data points.
    - Filters data to match the selected duration.

5. Error Handling:
    - If the financial data is unavailable or incomplete, the app displays a message ("Revenue data not available") in place of the graph.
6. Styling:
    - Dark-themed background and white text for visual consistency.
    - Tooltips and hover features enhance the user experience.
7. App Execution:
    - The app.run_server(debug=True) statement launches the dashboard in debug mode, enabling live updates and testing.

Example Usecase

profit_margin_trend_dashboard()

Purpose

The profit_margin_trend_dashboard function creates an interactive dashboard to visualize the profit margin trends of a selected stock over a specified duration. The profit margin, expressed as a percentage, is a critical measure of a company's profitability relative to its revenue.

Use Cases

1. **Profitability Analysis**:
    - Examine how effectively a company converts its revenue into profits over time.
2. **Investment Research**:
    - Identify companies with stable or improving profit margins, indicative of efficient cost management or increasing revenue quality.
3. **Comparative Study**:
    - Compare profit margins of different companies by changing the ticker.
4. **Educational Tool**:
    - Provide insights into the importance of profit margins in financial analysis.

Key Metric

- **Profit Margin (%)**: A measure of a company's profitability, calculated as the ratio of net income to total revenue, expressed as a percentage.

Overview

The profit_margin_trend_dashboard function delivers a user-friendly and visually appealing tool to track profit margin trends. It allows users to explore an essential financial metric interactively, facilitating better decision-making for investors, analysts, and learners.

Source Code

```
def profit_margin_trend_dashboard():
    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])
```

```
    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'margin_line': '#32CD32',  # LimeGreen color for Profit Margin line
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Profit Margin Trend Dashboard", style={'textAlign':
'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='profit-margin-chart', style={'height': '600px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('profit-margin-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch financial data
        stock = yf.Ticker(ticker)
        financials = stock.financials
```

```
        if financials is not None:
            # Calculate profit margin (Net Income / Total Revenue) * 100 for
percentage
            financials = financials.transpose()
            financials['Profit Margin'] = (financials['Net Income'] /
financials['Total Revenue']) * 100

            # Filter data for the selected duration
            financials = financials.tail(int(period.replace('y', '')))

            # Create the figure
            fig = go.Figure()

            # Add the Profit Margin line
            fig.add_trace(go.Scatter(x=financials.index, y=financials['Profit
Margin'], mode='lines+markers',
                                     name='Profit Margin (%)',
line=dict(color=colors['margin_line'])))

            # Update layout with dark theme
            fig.update_layout(
                template="plotly_dark",
                title=f"{ticker} - Profit Margin Trend Over Time",
                xaxis_title="Date",
                yaxis_title="Profit Margin (%)",
                font=dict(color="white"),
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background'],
                hovermode="x"
            )
        else:
            # Display a message if financial data is unavailable
            fig = go.Figure()
            fig.add_annotation(
                text="Profit margin data not available",
                xref="paper", yref="paper",
                showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)
```

```
# Call the function to launch the app
#profit_margin_trend_dashboard()
```

Code Explanation

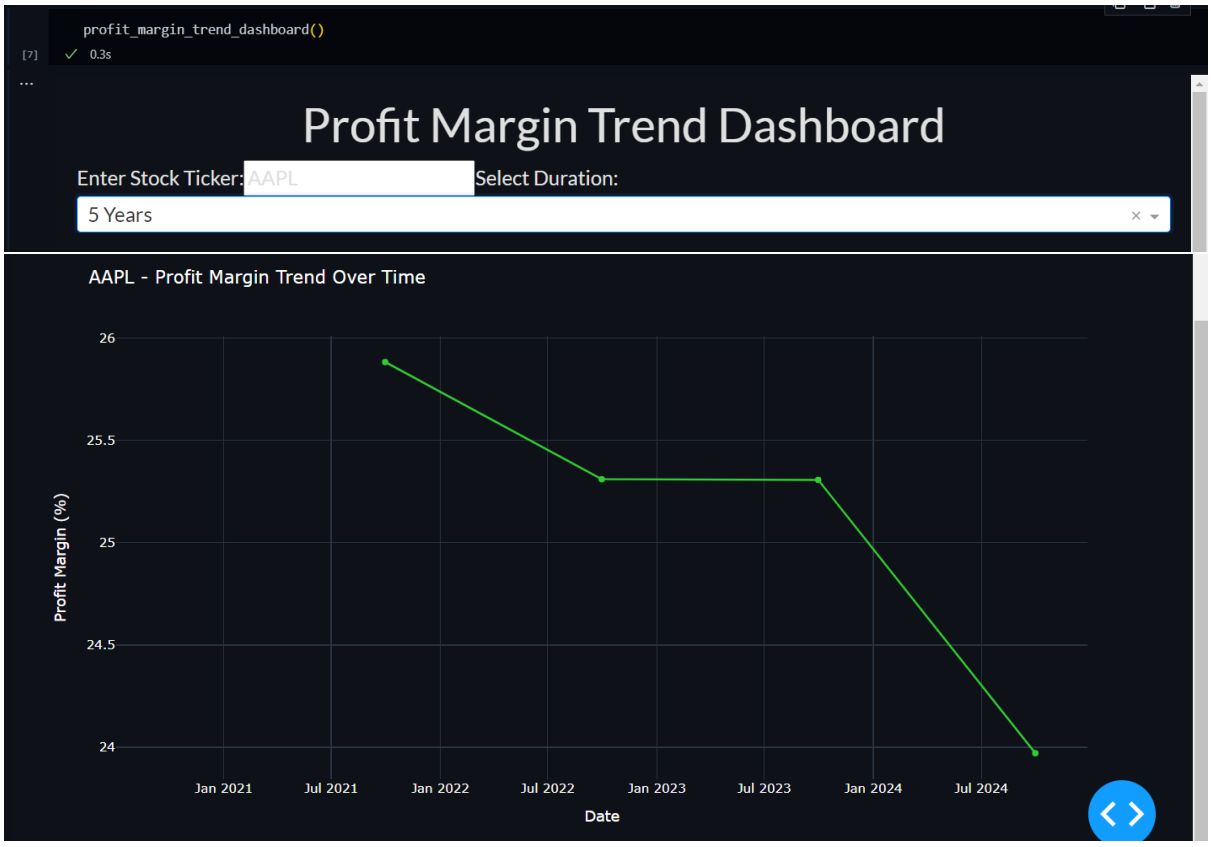
Key Features:

1. App Initialization:
    - The Dash app is initialized with the DARKLY theme, offering a sleek, dark visual design.
2. Color Scheme:
    - LimeGreen (#32CD32) is used for the profit margin line, providing a visually appealing contrast against the dark background.
3. Dashboard Layout:
    - Title: Displays "Profit Margin Trend Dashboard" as a central heading.
    - Stock Input Field: A text box for entering the stock ticker, defaulting to AAPL.
    - Duration Dropdown: Allows users to select a time range (1 to 10 years) for trend analysis.
    - Graph Area: Contains a dcc.Graph for dynamically visualizing profit margin trends.
4. Callback Functionality:
    - Inputs: Accepts the stock ticker and the selected duration as user inputs.
    - Data Retrieval:
        - Uses the yfinance library to fetch financial data.
        - Extracts and processes the Net Income and Total Revenue fields.
    - Profit Margin Calculation:
        - Computes profit margin as a percentage using the formula:
          Profit Margin (%)=(Net IncomeTotal Revenue)×100\text{Profit Margin (\%)} = \left(\frac{\text{Net Income}}{\text{Total Revenue}}\right) \times 100Profit Margin (%)=(Total RevenueNet Income)×100
    - Chart Construction:
        - Plots profit margin trends as a line chart with markers for better readability.
        - Filters data based on the selected time duration.
5. Error Handling:

- If the financial data is unavailable, the app displays an informative message ("Profit margin data not available") in place of the graph.

6. Styling:
   - Dark-themed background and white text ensure visual coherence.
   - Tooltips and hover features provide an enhanced user experience.
7. App Execution:
   - The app.run_server(debug=True) statement launches the dashboard, enabling live updates and debugging during development.

Example Usecase



free_cash_flow_trend_dashboard()

Purpose

The free_cash_flow_trend_dashboard function creates an interactive dashboard to visualize the Free Cash Flow (FCF) trends of a selected stock over a specified duration. Free Cash Flow is a crucial financial metric that indicates the cash generated by a company after accounting for capital expenditures, providing insights into its financial health and ability to generate shareholder value.

Use Cases

Financial Health Analysis

- Evaluate a company's ability to generate cash after accounting for operational and capital expenses.

Investment Research

- Identify companies with consistent or improving Free Cash Flow trends, indicating strong financial stability and growth potential.

Comparative Analysis

- Compare Free Cash Flow trends of different companies by entering different stock tickers.

Educational Tool

- Provide insights into the significance of Free Cash Flow in assessing company performance.

Key Metric

Free Cash Flow (FCF)

- A measure of a company's financial performance calculated as:
  FCF=Operating Cash Flow−Capital ExpendituresFCF = \text{Operating Cash Flow} - \text{Capital Expenditures}FCF=Operating Cash Flow−Capital Expenditures

---

Overview

The free_cash_flow_trend_dashboard function delivers a user-friendly and visually appealing tool to track Free Cash Flow trends interactively. It empowers investors, analysts, and learners to better understand this critical financial metric, facilitating informed decision-making.

Source Code

```
def free_cash_flow_trend_dashboard():
    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])
```

```
    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'fcf_line': '#1E90FF',  # DodgerBlue color for Free Cash Flow line
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Free Cash Flow Trend Dashboard", style={'textAlign':
'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='fcf-chart', style={'height': '600px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('fcf-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch cash flow data
        stock = yf.Ticker(ticker)
        cash_flow = stock.cashflow
```

```
        if cash_flow is not None:
            # Transpose to have dates as rows
            cash_flow = cash_flow.transpose()

            # Check if 'Free Cash Flow' column is available
            if 'Free Cash Flow' in cash_flow.columns:
                # Filter data for the selected duration
                cash_flow = cash_flow.tail(int(period.replace('y', '')))

                # Create the figure
                fig = go.Figure()

                # Add the Free Cash Flow line
                fig.add_trace(go.Scatter(x=cash_flow.index, y=cash_flow['Free
Cash Flow'], mode='lines+markers',
                                         name='Free Cash Flow',
line=dict(color=colors['fcf_line'])))

                # Update layout with dark theme
                fig.update_layout(
                    template="plotly_dark",
                    title=f"{ticker} - Free Cash Flow Trend Over Time",
                    xaxis_title="Date",
                    yaxis_title="Free Cash Flow (USD)",
                    font=dict(color="white"),
                    plot_bgcolor=colors['background'],
                    paper_bgcolor=colors['background'],
                    hovermode="x"
                )
            else:
                # Display a message if the column is unavailable
                fig = go.Figure()
                fig.add_annotation(
                    text="Free Cash Flow data not available for this stock",
                    xref="paper", yref="paper",
                    showarrow=False,
                    font=dict(size=20, color=colors['text'])
                )
                fig.update_layout(
                    plot_bgcolor=colors['background'],
                    paper_bgcolor=colors['background']
                )
        else:
            # Display a message if cash flow data is unavailable
            fig = go.Figure()
            fig.add_annotation(
                text="Cash flow data not available",
```

```
                xref="paper", yref="paper",
                showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)

# Call the function to launch the app
#free_cash_flow_trend_dashboard()
```

Code Explanation

Key Features

App Initialization

- **The Dash app is initialized with the DARKLY theme, offering a sleek, modern dark design that enhances readability.**

Color Scheme

- **DodgerBlue (#1E90FF) is used for the Free Cash Flow line, providing a vibrant contrast against the dark background.**

Dashboard Layout

- **Title: Displays "Free Cash Flow Trend Dashboard" as a central heading.**
- **Stock Input Field: A text box for entering the stock ticker symbol, defaulting to AAPL.**
- **Duration Dropdown: Allows users to select a time range (1 to 10 years) for trend analysis.**
- **Graph Area: A dcc.Graph component dynamically visualizes Free Cash Flow trends.**

Callback Functionality

- **Inputs: Accepts the stock ticker and selected duration as user inputs.**
- **Data Retrieval:**
    - **Leverages the yfinance library to fetch cash flow data.**
    - **Extracts the "Free Cash Flow" field from the company's financial statements.**

- **Data Filtering:**
    - **Filters the financial data based on the selected time duration (e.g., 1, 2, 5, or 10 years).**
- **Chart Construction:**
    - **Plots Free Cash Flow trends as a line chart with markers for better readability.**
    - **Includes axis labels and hover features for an enhanced user experience.**

Error Handling

- **If the required financial data is unavailable, the app gracefully handles the error by displaying an informative message, such as:**
    - **"Free Cash Flow data not available for this stock."**
    - **"Cash flow data not available."**

Styling

- **Dark-themed background with light text ensures visual coherence and readability.**
- **Interactive tooltips and hover features add to the user experience.**

App Execution

- **The app is launched with app.run_server(debug=True), enabling live updates and debugging during development.**

Example Usage

beta_trend_dashboard()

Purpose

The beta_trend_dashboard function creates an interactive dashboard to visualize the Beta Coefficient trend for a selected stock over a specified duration. The Beta Coefficient measures a stock's volatility relative to the broader market, providing valuable insights for risk assessment and portfolio management.

Use Cases

Risk Analysis

- Evaluate a stock's market risk relative to the broader market.

Investment Research

- Identify stocks with Beta Coefficients aligned with specific risk preferences (e.g., low Beta for risk-averse investors).

Educational Tool

- Provide insights into the Beta Coefficient's role in financial analysis and portfolio management.

Key Metric

Beta Coefficient

- A measure of a stock's volatility relative to the market:
    - **Beta > 1**: Stock is more volatile than the market.
    - **Beta < 1**: Stock is less volatile than the market.
    - **Beta = 1**: Stock moves in line with the market.

Overview

The beta_trend_dashboard function delivers a user-friendly and visually appealing tool to track Beta Coefficient trends interactively. It empowers investors, analysts, and learners to understand this key financial metric, facilitating informed decision-making in portfolio and risk management.

Source Code

```
def beta_trend_dashboard():
    # Initialize the Dash app
```

```
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'beta_line': '#FF4500',  # OrangeRed color for Beta line
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Beta Coefficient Trend Dashboard", style={'textAlign':
'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                    {'label': '10 Years', 'value': '10y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='beta-chart', style={'height': '600px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('beta-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        if not ticker:
            fig = go.Figure()
```

```
            fig.add_annotation(
                text="Please enter a valid stock ticker symbol",
                xref="paper", yref="paper",
                showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )
            return fig

        # Fetch stock data
        try:
            stock = yf.Ticker(ticker)
            beta = stock.info.get('beta')
        except Exception as e:
            fig = go.Figure()
            fig.add_annotation(
                text="Failed to retrieve data. Please check the stock ticker
symbol.",
                xref="paper", yref="paper",
                showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )
            return fig

        if beta is not None:
            # Simulate Beta trend as a constant line
            date_range = pd.date_range(end=pd.Timestamp.today(),
periods=int(period.replace('y', '')) * 12, freq='ME')

            # Create the figure
            fig = go.Figure()
            fig.add_trace(go.Scatter(x=date_range, y=[beta] * len(date_range),
mode='lines',
                                     name='Beta Coefficient',
line=dict(color=colors['beta_line'])))

            # Update layout with dark theme
            fig.update_layout(
                template="plotly_dark",
                title=f"{ticker} - Beta Coefficient Trend Over Time",
                xaxis_title="Date",
```

```
                yaxis_title="Beta Coefficient",
                font=dict(color="white"),
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background'],
                hovermode="x"
            )
        else:
            # Display a message if Beta data is unavailable
            fig = go.Figure()
            fig.add_annotation(
                text="Beta coefficient data not available",
                xref="paper", yref="paper",
                showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)

# Call the function to launch the app
#beta_trend_dashboard()
```

Code Explanation

Functionality

This function initializes a Dash application, enabling users to analyze the Beta Coefficient trends of publicly traded stocks.

Key Features

App Initialization

- The Dash app is initialized with the DARKLY theme, offering a modern dark-themed design for enhanced readability and aesthetic appeal.

Color Scheme

- Background: Dark gray (#0E1117)
- Text: Light gray (#E0E0E0)
- Beta Line: OrangeRed (#FF4500), adding a vibrant contrast to the dark background.

Dashboard Layout

- Title: Displays "Beta Coefficient Trend Dashboard" as a central heading.
- Stock Input Field: A text input box for entering the stock ticker symbol, defaulting to AAPL.
- Duration Dropdown: Allows users to select a time range (1 Year, 2 Years, 5 Years, or 10 Years) for trend visualization.
- Graph Area: A dcc.Graph component dynamically displays the Beta Coefficient trends.

Callback Functionality

- Inputs: Accepts the stock ticker and selected duration as user inputs.
- Data Retrieval:
    - Leverages the yfinance library to fetch stock information, including the Beta Coefficient.
- Simulated Beta Trend:
    - Constructs a simulated Beta trend line, treating Beta as a constant over time.
    - Uses a date range proportional to the selected time duration to plot the trend.
- Error Handling:
    - Displays an informative message if:
        - The stock ticker is invalid.
        - Beta data is unavailable.
- Chart Construction:
    - Uses a line chart to represent Beta trends with labeled axes and hover features.
    - Includes annotations for error messages when applicable.

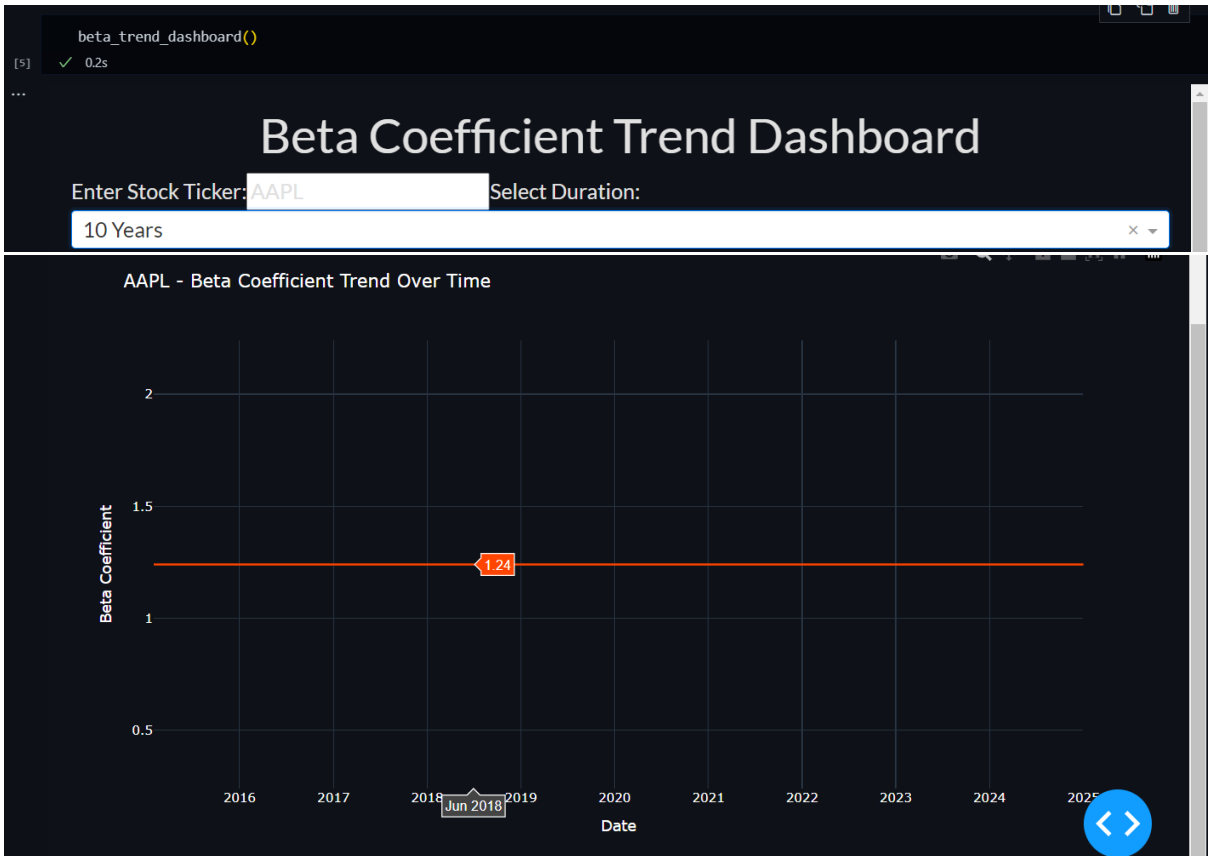
Styling

- Dark Theme: A consistent dark-themed background with light-colored text ensures visual coherence and readability.
- Interactive Elements: Hover features and annotations enhance the user experience.

App Execution

- The app is launched with app.run_server(debug=True), allowing live updates and debugging during development.

Example Usage



correlation_with_market_dashboard()

Purpose

The correlation_with_market_dashboard function creates an interactive dashboard to visualize the **rolling correlation** between a selected stock's monthly returns and a market index (e.g., S&P 500) over a specified duration. Rolling correlation helps identify how closely a stock's performance aligns with the market over time, providing valuable insights for investment strategies and market analysis.

Use Cases

Market Analysis

Understand how a stock's performance aligns with the market during specific periods.

Investment Research

Identify stocks with correlations that match specific portfolio strategies.

Educational Tool

Illustrate the concept of correlation in financial analysis for students and professionals.

Key Metric

Rolling Correlation

A measure of how closely a stock's returns align with the market over a rolling period:

- **Correlation > 0**: Stock and market move in the same direction.
- **Correlation < 0**: Stock and market move in opposite directions.
- **Correlation ≈ 0**: No significant relationship.

Overview

The correlation_with_market_dashboard function delivers a user-friendly tool for tracking and analyzing rolling correlation trends between stocks and the market index. It empowers investors, analysts, and educators to explore this key relationship dynamically, aiding in portfolio strategy, market analysis, and financial education.

Source Code

```
def correlation_with_market_dashboard():
    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])
```

```
    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'correlation_line': '#FF6347',  # Tomato color for correlation line
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Stock and Market Index Correlation Dashboard",
style={'textAlign': 'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='correlation-chart', style={'height': '600px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('correlation-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Define the market index ticker (S&P 500 in this example)
        market_index_ticker = '^GSPC'  # S&P 500

        # Fetch historical data for the stock and the market index
```

```
        stock_data = yf.download(ticker, period=period, interval='1mo')
        market_index_data = yf.download(market_index_ticker, period=period,
interval='1mo')

        # Ensure we have data for both the stock and the index
        if not stock_data.empty and not market_index_data.empty:
            # Calculate the monthly returns
            stock_returns = stock_data['Adj Close'].pct_change().dropna()
            market_index_returns = market_index_data['Adj
Close'].pct_change().dropna()

            # Align data by index (date)
            returns_df = pd.DataFrame({
                'Stock': stock_returns,
                'Market Index': market_index_returns
            }).dropna()

            # Calculate the rolling correlation (using a 6-month window)
            returns_df['Correlation'] =
returns_df['Stock'].rolling(window=6).corr(returns_df['Market Index'])

            # Create the figure
            fig = go.Figure()

            # Add the Correlation line
            fig.add_trace(go.Scatter(x=returns_df.index,
y=returns_df['Correlation'], mode='lines+markers',
                                     name='Rolling Correlation',
line=dict(color=colors['correlation_line'])))

            # Update layout with dark theme
            fig.update_layout(
                template="plotly_dark",
                title=f"{ticker} - Rolling Correlation with
{market_index_ticker} Over Time",
                xaxis_title="Date",
                yaxis_title="Correlation",
                font=dict(color="white"),
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background'],
                hovermode="x"
            )
        else:
            # Display a message if data is unavailable
            fig = go.Figure()
            fig.add_annotation(
                text="Data not available",
                xref="paper", yref="paper",
```

```
                showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)

# Call the function to launch the app
#correlation_with_market_dashboard()
```

Code Explanation

Key Features

App Initialization

The Dash app is initialized with the DARKLY theme, offering a modern dark-themed design for enhanced readability and a professional look.

Color Scheme

- Background: Dark gray (#0E1117) for a cohesive dark theme.
- Text: Light gray (#E0E0E0) for optimal contrast and readability.
- Correlation Line: Tomato (#FF6347), providing a vibrant highlight for the rolling correlation trend.

Dashboard Layout

1. Title: Displays "Stock and Market Index Correlation Dashboard" as a central heading.
2. Stock Input Field: A text input box for entering the stock ticker symbol, defaulting to AAPL.
3. Duration Dropdown: Allows users to select a time range (1 Year, 2 Years, or 5 Years) for correlation analysis.
4. Graph Area: A dcc.Graph component dynamically displays the rolling correlation trends.

Callback Functionality

Inputs:

- Stock ticker entered by the user.
- Selected duration for correlation analysis.

Data Retrieval

1. Stock and Market Data:
    - Leverages the yfinance library to fetch historical monthly adjusted closing prices for both the stock and the S&P 500 market index.
2. Data Validation:
    - Ensures data for both the stock and market index is available.

Correlation Calculation

- Returns Calculation:
  Computes monthly percentage changes (returns) for both the stock and the market index.
- Rolling Correlation:
  Calculates the rolling correlation using a 6-month window to observe trend changes over time.

Error Handling

- Displays a message if:
    - The stock ticker is invalid.
    - Data for the selected duration is unavailable.

Chart Construction

- Rolling Correlation Line:
  Uses a line chart with labeled axes and hover features to represent rolling correlation trends dynamically.
- Dark Theme:
  Ensures consistent visual styling with a dark background, light-colored text, and hover enhancements.

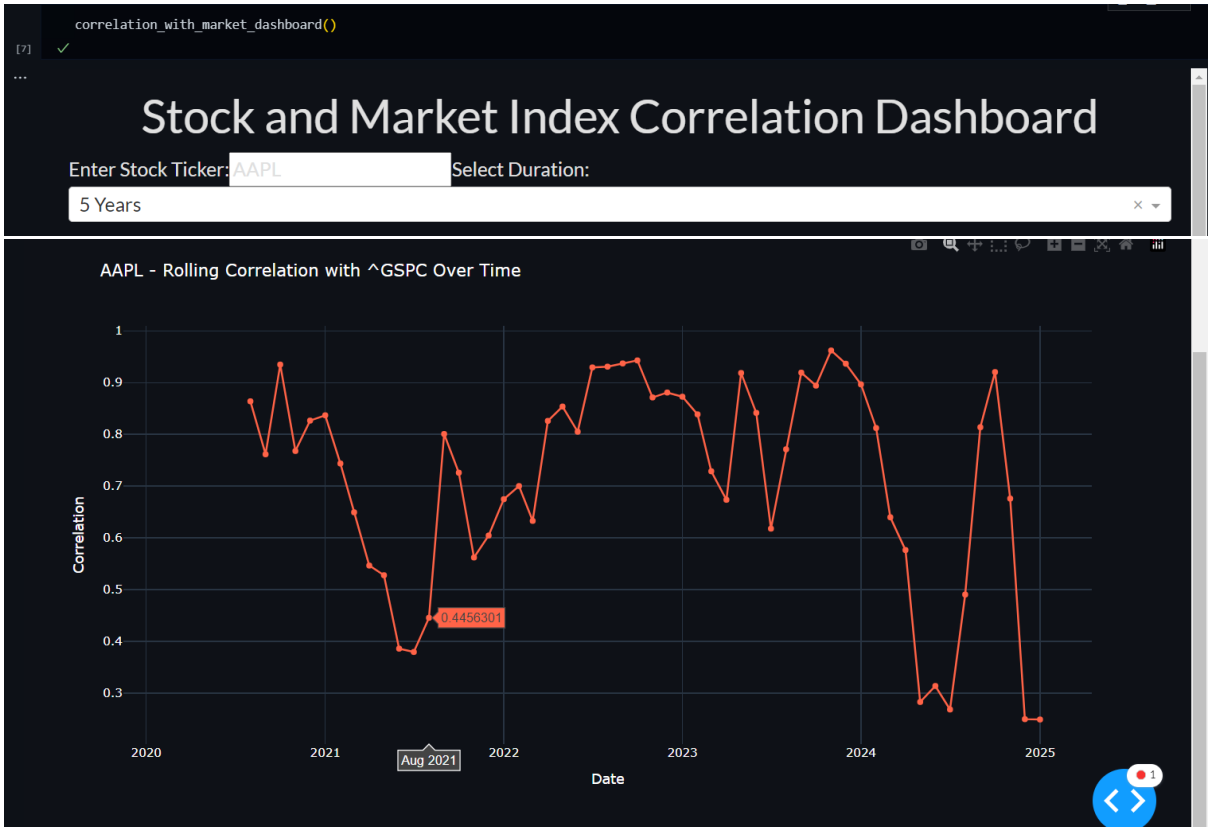
Styling

- Interactive Elements:
  Hover features and annotations enrich the user experience.

App Execution

- The app is launched with app.run_server(debug=True), enabling live updates and debugging during development.

Example Usage



volatility_trend_dashboard()

Purpose

The volatility_trend_dashboard function creates an interactive dashboard to visualize the rolling volatility (annualized standard deviation) of a selected stock's daily returns over a specified duration. Volatility is a key metric in assessing market risk and understanding the price fluctuations of a stock.

Use Cases

Risk Management

Assess the level of price fluctuation (volatility) of a stock for better portfolio risk management.

Investment Research

Identify stocks with high or low volatility trends to align with specific investment strategies.

Educational Tool

Provide an interactive way to understand the concept of volatility in stock price movements.

Key Metric

Rolling Volatility

A measure of the stock's price fluctuation over time:

- **Higher Volatility**: Indicates larger price swings and higher risk.
- **Lower Volatility**: Suggests smaller price swings and lower risk.

Overview

The volatility_trend_dashboard function offers a user-friendly, visually appealing platform for analyzing stock price volatility trends. It equips investors, analysts, and learners with the tools to explore and interpret volatility data dynamically, fostering informed decision-making in portfolio and risk management.

Source Code

```
def volatility_trend_dashboard():
    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
```

```
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'volatility_line': '#FFA500',  # Orange color for volatility line
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Stock Volatility Dashboard", style={'textAlign':
'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='volatility-chart', style={'height': '600px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('volatility-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch historical data for the stock
        stock_data = yf.download(ticker, period=period, interval='1d')

        if not stock_data.empty:
            # Calculate daily returns
            stock_returns = stock_data['Adj Close'].pct_change().dropna()
```

```
            # Calculate rolling volatility (standard deviation) with a 20-day
window (approximately 1 month)
            rolling_volatility = stock_returns.rolling(window=20).std() *
(252**0.5)  # Annualize the volatility

            # Create the figure
            fig = go.Figure()

            # Add the Volatility line
            fig.add_trace(go.Scatter(x=rolling_volatility.index,
y=rolling_volatility, mode='lines',
                                     name='Rolling Volatility (Annualized)',
line=dict(color=colors['volatility_line'])))

            # Update layout with dark theme
            fig.update_layout(
                template="plotly_dark",
                title=f"{ticker} - Rolling Volatility Over Time",
                xaxis_title="Date",
                yaxis_title="Volatility (Standard Deviation)",
                font=dict(color="white"),
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background'],
                hovermode="x"
            )
        else:
            # Display a message if data is unavailable
            fig = go.Figure()
            fig.add_annotation(
                text="Data not available",
                xref="paper", yref="paper",
                showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)

# Call the function to launch the app
#volatility_trend_dashboard()
```

Code Explanation

Functionality

This function initializes a Dash application, enabling users to analyze the rolling volatility trends of publicly traded stocks dynamically.

Key Features

App Initialization

The Dash app is initialized with the DARKLY theme, providing a sleek and professional dark-themed interface.

Color Scheme

- Background: Dark gray (#0E1117) for consistency in the dark theme.
- Text: Light gray (#E0E0E0) for improved readability.
- Volatility Line: Orange (#FFA500), offering a bright, visually distinct line to represent the rolling volatility trend.

Dashboard Layout

1. Title: Displays "Stock Volatility Dashboard" as a central heading.
2. Stock Input Field: A text input box for entering the stock ticker symbol, defaulting to AAPL.
3. Duration Dropdown: Allows users to select a time range (1 Year, 2 Years, or 5 Years) for volatility analysis.
4. Graph Area: A dcc.Graph component dynamically visualizes the rolling volatility trends over the selected duration.

Callback Functionality

Inputs:

- Stock ticker entered by the user.
- Selected time period for volatility analysis.

Data Retrieval

1. Stock Data:

    - Utilizes the yfinance library to fetch historical daily adjusted closing prices for the selected stock over the specified period.

2. Data Validation:
    - Ensures that valid stock data is available for analysis.

Volatility Calculation

- Daily Returns:
  Calculates daily percentage changes in the adjusted closing prices.
- Rolling Volatility:
  Computes rolling volatility (annualized standard deviation) using a 20-day rolling window, which approximates one month of trading days.

Error Handling

- Displays a message in the chart if:
    - The stock ticker is invalid.
    - No data is available for the specified duration.

Chart Construction

- Volatility Line:
  Uses a line chart to represent rolling volatility trends with labeled axes and hover features.
- Dark Theme:
  Maintains visual consistency with a dark background, light-colored text, and hover enhancements.

Styling

- Interactive Elements:
  Hover features and annotations enhance the user experience, making the chart more informative and engaging.

App Execution

- The app is launched with app.run_server(debug=True), enabling real-time updates and debugging during development.

Example Usecase

sharpe_ratio_trend_dashboard()

Purpose

The sharpe_ratio_trend_dashboard function creates an interactive dashboard that visualizes the rolling Sharpe Ratio of a stock over a specified period. The Sharpe Ratio is a widely used metric in finance to evaluate the risk-adjusted return of an investment.

Use Cases

Portfolio Management

Evaluate the risk-adjusted performance of a stock to make informed portfolio decisions.

Investment Research

Identify periods of high or low Sharpe Ratio to assess whether the stock's returns justify its risk.

Educational Tool

Provide an interactive way to understand the Sharpe Ratio and its significance in investment analysis.

Key Metric

Sharpe Ratio

A measure of risk-adjusted return:

- **Higher Sharpe Ratio**: Indicates better risk-adjusted performance.
- **Lower Sharpe Ratio**: Suggests poor risk-adjusted performance.

Overview

The sharpe_ratio_trend_dashboard function offers an interactive, user-friendly platform for analyzing the Sharpe Ratio trends of stocks. It equips investors, analysts, and learners with the tools to explore risk-adjusted returns dynamically, fostering better investment decisions and financial insights.

Source Code

```
def sharpe_ratio_trend_dashboard():
    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])
```

```
    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'sharpe_ratio_line': '#00FF00',  # Green color for Sharpe Ratio line
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Stock Sharpe Ratio Dashboard", style={'textAlign':
'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '2 Years', 'value': '2y'},
                    {'label': '5 Years', 'value': '5y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='sharpe-ratio-chart', style={'height': '600px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('sharpe-ratio-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        # Fetch historical data for the stock
        stock_data = yf.download(ticker, period=period, interval='1d')

        if not stock_data.empty:
            # Calculate daily returns
```

```
            stock_returns = stock_data['Adj Close'].pct_change().dropna()

            # Assume a risk-free rate of 0% for simplicity
            risk_free_rate = 0

            # Calculate rolling average return and rolling volatility
(standard deviation) with a 20-day window
            rolling_avg_return = stock_returns.rolling(window=20).mean()
            rolling_volatility = stock_returns.rolling(window=20).std()

            # Calculate rolling Sharpe Ratio
            rolling_sharpe_ratio = (rolling_avg_return - risk_free_rate) /
rolling_volatility

            # Create the figure
            fig = go.Figure()

            # Add the Sharpe Ratio line
            fig.add_trace(go.Scatter(
                x=rolling_sharpe_ratio.index,
                y=rolling_sharpe_ratio,
                mode='lines',
                name='Rolling Sharpe Ratio',
                line=dict(color=colors['sharpe_ratio_line'])
            ))

            # Update layout with dark theme
            fig.update_layout(
                template="plotly_dark",
                title=f"{ticker} - Rolling Sharpe Ratio Over Time",
                xaxis_title="Date",
                yaxis_title="Sharpe Ratio",
                font=dict(color="white"),
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background'],
                hovermode="x"
            )
        else:
            # Display a message if data is unavailable
            fig = go.Figure()
            fig.add_annotation(
                text="Data not available",
                xref="paper", yref="paper",
                showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
```

```
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)
```

Code Explanation

Key Features

App Initialization

The Dash app is initialized with the DARKLY theme, providing a visually appealing dark-themed interface suitable for financial data visualization.

Color Scheme

- Background: Dark gray (#0E1117) for consistency in the dark theme.
- Text: Light gray (#E0E0E0) for readability.
- Sharpe Ratio Line: Bright green (#00FF00), symbolizing positive performance and easily distinguishable on a dark background.

Dashboard Layout

1. Title: Displays "Stock Sharpe Ratio Dashboard" as a central heading.
2. Stock Input Field: A text input box for entering the stock ticker symbol, defaulting to AAPL.
3. Duration Dropdown: Allows users to select a time range (1 Year, 2 Years, or 5 Years) for the Sharpe Ratio analysis.
4. Graph Area: A dcc.Graph component dynamically visualizes the rolling Sharpe Ratio trends over the selected duration.

Callback Functionality

Inputs:

- Stock ticker entered by the user.
- Selected time period for the analysis.

Data Retrieval

1. Stock Data:
    - Utilizes the yfinance library to fetch historical daily adjusted closing prices for the selected stock over the specified period.
2. Data Validation:
    - Ensures that valid stock data is available for the analysis.

Sharpe Ratio Calculation

1. Daily Returns:
   Calculates the percentage change in the adjusted closing prices.
2. Assumption:
   Assumes a risk-free rate of 0%, simplifying the calculation.
3. Rolling Calculations:
    - Rolling Average Return: Mean of daily returns over a 20-day window (approximately one month).
    - Rolling Volatility: Standard deviation of daily returns over the same window.
4. Rolling Sharpe Ratio:
   Computes the rolling Sharpe Ratio using the formula:

Sharpe Ratio=Rolling Average Return−Risk-Free RateRolling Volatility\text{Sharpe Ratio} = \frac{\text{Rolling Average Return} - \text{Risk-Free Rate}}{\text{Rolling Volatility}}Sharpe Ratio=Rolling VolatilityRolling Average Return−Risk-Free Rate

---

Error Handling

- Displays a message in the chart if:
    - The stock ticker is invalid.
    - No data is available for the specified duration.

Chart Construction

- Sharpe Ratio Line:
  Uses a line chart to represent rolling Sharpe Ratio trends with labeled axes and hover features.
- Dark Theme:
  Ensures visual consistency with a dark background, light-colored text, and interactive hover enhancements.

Styling

- Interactive Elements:
  Hover features and annotations enhance user experience, making the chart more informative and engaging.

App Execution

- The app is launched with app.run_server(debug=True), enabling real-time updates and debugging during development.

Example Usage

trading_volume_price_dashboard()

Purpose
The trading_volume_price_dashboard function creates an interactive dashboard to visualize the relationship between a stock's trading volume and its price over a specified period. It enables users to explore historical trends and analyze how price movements correspond to trading activity.

Use Cases

1. **Investment Analysis**
   Assess how trading volume correlates with price movements, aiding in investment decisions.
2. **Market Research**
   Identify periods of high trading activity and their impact on stock prices.
3. **Educational Tool**
   Provide a hands-on approach for understanding trading volume and its relationship with stock price trends.

Overview
The trading_volume_price_dashboard function delivers a feature-rich, interactive platform for exploring trading volume and price trends. It caters to investors, market analysts, and learners seeking insights into stock market dynamics. This tool enhances decision-making and offers a deeper understanding of trading behavior.

Source Code

```
def trading_volume_price_dashboard():


    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'price_line': '#00FF00',  # Green for Price line
        'volume_bar': '#007BFF',  # Blue for Volume bars
    }

    # Define the layout of the dashboard
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
```

```
            html.H1("Trading Volume vs. Price Dashboard", style={'textAlign':
'center', 'color': colors['text']}),

            html.Label("Enter Stock Ticker:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Input(id='ticker-input', type='text', value='AAPL',
style={'font-size': '20px', 'color': colors['text']}),

            html.Label("Select Duration:", style={'font-size': '20px',
'color': colors['text']}),
            dcc.Dropdown(
                id='duration-dropdown',
                options=[
                    {'label': '6 Months', 'value': '6mo'},
                    {'label': '1 Year', 'value': '1y'},
                    {'label': '5 Years', 'value': '5y'},
                ],
                value='1y',
                style={'font-size': '20px', 'color': colors['text']}
            ),

            dcc.Graph(id='volume-price-chart', style={'height': '600px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('volume-price-chart', 'figure'),
        Input('ticker-input', 'value'),
        Input('duration-dropdown', 'value')
    )
    def update_chart(ticker, period):
        try:
            # Fetch historical data for the stock
            stock_data = yf.download(ticker, period=period, interval='1d')

            if not stock_data.empty:
                # Create the figure
                fig = go.Figure()

                # Add the price line (Adjusted Close)
                fig.add_trace(go.Scatter(
                    x=stock_data.index,
                    y=stock_data['Adj Close'],
                    mode='lines',
                    name='Price',
                    line=dict(color=colors['price_line'], width=2),
                    yaxis='y1'  # Use the left y-axis
```

```
            ))

            # Add the volume bar chart
            fig.add_trace(go.Bar(
                x=stock_data.index,
                y=stock_data['Volume'],
                name='Volume',
                marker=dict(color=colors['volume_bar']),
                yaxis='y2'  # Use the right y-axis
            ))

            # Update layout
            fig.update_layout(
                template="plotly_dark",
                title=f"{ticker} - Trading Volume vs. Price",
                xaxis_title="Date",
                yaxis=dict(
                    title="Price",
                    titlefont=dict(color=colors['price_line']),
                    tickfont=dict(color=colors['price_line'])
                ),
                yaxis2=dict(
                    title="Volume",
                    titlefont=dict(color=colors['volume_bar']),
                    tickfont=dict(color=colors['volume_bar']),
                    overlaying='y',
                    side='right'
                ),
                font=dict(color="white"),
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background'],
                legend=dict(x=0, y=1, xanchor='left', yanchor='top')
            )
        else:
            # Handle empty data case
            fig = go.Figure()
            fig.add_annotation(
                text="Data not available",
                xref="paper", yref="paper",
                showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

    except Exception as e:
```

```
            # Handle errors (e.g., invalid ticker, API issues)
            fig = go.Figure()
            fig.add_annotation(
                text=f"Error: {str(e)}",
                xref="paper", yref="paper",
                showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)


#Example Use Case
#trading_volume_price_dashboard()
```

Code Explanation

Functionality
This function initializes a Dash application, allowing dynamic visualization of trading volume and price trends for publicly traded stocks.

Key Features

App Initialization
The Dash app is built using the DARKLY theme, providing a sleek dark-themed interface ideal for financial data visualization.

Color Scheme

- Background: Dark gray (#0E1117) for consistency in the dark theme.
- Text: Light gray (#E0E0E0) for readability.
- Price Line: Bright green (#00FF00) to symbolize growth and trends.
- Volume Bars: Blue (#007BFF) for clear differentiation from price trends.

Dashboard Layout

- Title: Displays "Trading Volume vs. Price Dashboard" as the central heading.
- Stock Input Field: A text input box for entering the stock ticker symbol, defaulting to AAPL.

- Duration Dropdown: Allows users to select a time range (6 Months, 1 Year, or 5 Years) for analysis.
- Graph Area: A dcc.Graph component dynamically visualizes price trends and trading volume over the selected period.

Callback Functionality
Inputs:

- Stock ticker entered by the user.
- Selected time period for analysis.

Data Retrieval

- Uses the yfinance library to fetch historical daily adjusted closing prices and volume data.
- Validates stock data availability for the selected ticker and duration.

Chart Construction

- Price Line: A line chart represents the stock's adjusted closing price trends.
- Volume Bars: A bar chart overlays trading volume data on a secondary y-axis.

Layout Customization

- Dual Y-Axis: The left y-axis displays the stock price, while the right y-axis shows trading volume.
- Dark Theme: Ensures a visually cohesive dark background with clear and interactive chart elements.

Error Handling

- Displays an error message if the stock ticker is invalid or data is unavailable.

Styling
Interactive hover features and annotations enhance the user experience, making the chart engaging and informative.

App Execution
The app is launched with app.run_server(debug=True), enabling real-time updates and debugging during development.

Example Usecase

multi_ticker_dashboard()

The multi_ticker_dashboard() function creates an interactive Dash web application that allows users to compare stock values of two tickers over different periods. Below is a breakdown of the function's purpose and workflow:

Purpose

The function is designed to:

1. Compare the performance of two stock tickers using data from Yahoo Finance.
2. Visualize the comparison for selected value types (Open, Close, High, Low, or Average).
3. Allow users to customize inputs such as tickers, value types, and time duration.

Source Code

```
def multi_ticker_dashboard():
    # Import necessary libraries
    import dash
    import dash_core_components as dcc
    import dash_html_components as html
    from dash.dependencies import Input, Output
    import plotly.graph_objects as go
    import yfinance as yf
    import dash_bootstrap_components as dbc

    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'line1': '#FF4500',  # Line for Ticker 1
        'line2': '#1E90FF',  # Line for Ticker 2
    }

    # Define the layout
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Multi-Ticker Value Comparison Dashboard",
style={'textAlign': 'center', 'color': colors['text']}),
```

```
            html.Div([
                html.Label("Enter First Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker1-input', type='text', value='AAPL',
style={'margin-right': '20px'}),
                html.Label("Enter Second Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker2-input', type='text', value='MSFT'),
            ], style={'margin-bottom': '20px'}),

            html.Div([
                html.Label("Select Value Type:", style={'color':
colors['text']}),
                dcc.Dropdown(
                    id='value-type-dropdown',
                    options=[
                        {'label': 'Open', 'value': 'Open'},
                        {'label': 'Close', 'value': 'Close'},
                        {'label': 'High', 'value': 'High'},
                        {'label': 'Low', 'value': 'Low'},
                        {'label': 'Average', 'value': 'Average'},
                    ],
                    value='Close',
                    style={'margin-bottom': '20px'}
                ),
                html.Label("Select Time Duration:", style={'color':
colors['text']}),
                dcc.Dropdown(
                    id='duration-dropdown',
                    options=[
                        {'label': '6 Months', 'value': '6mo'},
                        {'label': '1 Year', 'value': '1y'},
                        {'label': '5 Years', 'value': '5y'},
                    ],
                    value='1y',
                )
            ]),

            dcc.Graph(id='comparison-chart', style={'height': '600px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('comparison-chart', 'figure'),
        [
            Input('ticker1-input', 'value'),
```

```
            Input('ticker2-input', 'value'),
            Input('value-type-dropdown', 'value'),
            Input('duration-dropdown', 'value')
        ]
    )
    def update_chart(ticker1, ticker2, value_type, period):
        try:
            # Fetch historical data for both tickers
            data1 = yf.download(ticker1, period=period, interval='1d')
            data2 = yf.download(ticker2, period=period, interval='1d')

            # Calculate average if selected
            if value_type == 'Average':
                data1['Average'] = (data1['High'] + data1['Low']) / 2
                data2['Average'] = (data2['High'] + data2['Low']) / 2

            # Create the figure
            fig = go.Figure()

            # Add line for Ticker 1
            if not data1.empty:
                fig.add_trace(go.Scatter(
                    x=data1.index,
                    y=data1[value_type],
                    mode='lines',
                    name=f'{ticker1} {value_type}',
                    line=dict(color=colors['line1'], width=2),
                ))
            else:
                fig.add_annotation(
                    text=f"Data not available for {ticker1}",
                    xref="paper", yref="paper",
                    x=0.5, y=0.8, showarrow=False,
                    font=dict(size=18, color=colors['text'])
                )

            # Add line for Ticker 2
            if not data2.empty:
                fig.add_trace(go.Scatter(
                    x=data2.index,
                    y=data2[value_type],
                    mode='lines',
                    name=f'{ticker2} {value_type}',
                    line=dict(color=colors['line2'], width=2),
                ))
            else:
                fig.add_annotation(
                    text=f"Data not available for {ticker2}",
```

```
                    xref="paper", yref="paper",
                    x=0.5, y=0.6, showarrow=False,
                    font=dict(size=18, color=colors['text'])
                )

            # Update layout
            fig.update_layout(
                template="plotly_dark",
                title=f"{ticker1} vs {ticker2} - {value_type} Comparison",
                xaxis_title="Date",
                yaxis_title=value_type,
                font=dict(color=colors['text']),
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background'],
                legend=dict(x=0, y=1, xanchor='left', yanchor='top')
            )

        except Exception as e:
            # Handle errors
            fig = go.Figure()
            fig.add_annotation(
                text=f"Error: {str(e)}",
                xref="paper", yref="paper",
                x=0.5, y=0.5, showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)


#Example Usecase
#multi_ticker_dashboard()
```

Code Explanation

 Import Necessary Libraries:

- dash, dash_core_components, and dash_html_components: For creating the web interface.
- plotly.graph_objects: To build the chart visualizations.

- yfinance: For retrieving stock data.
- dash_bootstrap_components: To apply the DARKLY theme for styling.

 Initialize the Dash App:

- The app is created with external_stylesheets set to a dark theme using Dash Bootstrap.

 Set Color Scheme:

- A colors dictionary defines the background, text, and line colors for consistency in the app's theme.

 Define the Layout:

- Header: Displays the app title.
- Input Fields: Provides text inputs for the stock tickers, dropdowns for value type and time duration.
- Graph: A placeholder for the comparison chart, dynamically updated based on user input.

 Callback for Interactivity:

- The @app.callback decorator connects user inputs (tickers, value type, and duration) to the chart's output.
- The callback fetches stock data, processes it, and updates the graph.

 Fetch Stock Data:

- The app uses yfinance.download to retrieve historical data for the entered tickers.
- If Average is selected, it calculates the average of High and Low prices.

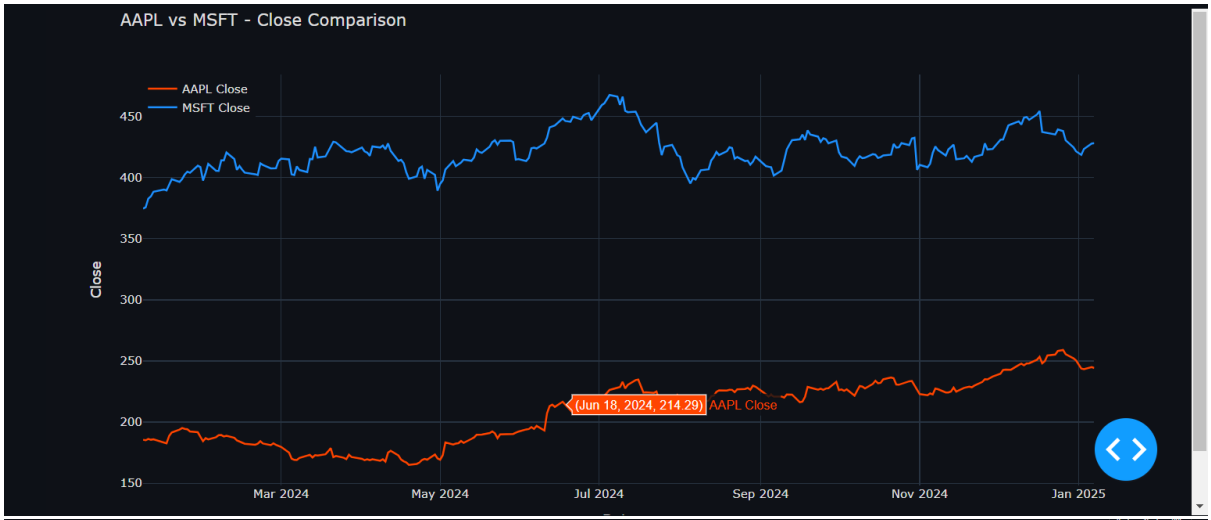
 Create the Comparison Chart:

- Adds lines for each ticker, color-coded for clarity.
- Displays error messages or warnings if data is unavailable for a ticker.

 Run the App:

- Starts the Dash app server with debugging enabled for real-time error tracking.

Example Usage




```
multi_ticker_dashboard()
```

[5] ✓

```
<ipython-input-4-9cbeafa1e5f6>:4: UserWarning:

The dash_core_components package is deprecated. Please replace
`import dash_core_components as dcc` with `from dash import dcc`

<ipython-input-4-9cbeafa1e5f6>:5: UserWarning:

The dash_html_components package is deprecated. Please replace
`import dash_html_components as html` with `from dash import html`
```

Multi-Ticker Value Comparison Dashboard

Enter First Stock Ticker: AAPL Enter Second Stock Ticker: MSFT

Select Value Type:

Close

Select Time Duration:

1 Year

price_moving_average_dashboard()

The price_moving_average_dashboard() function creates an interactive **Dash web application** to compare the stock prices of two tickers with their respective moving averages (Simple Moving Average (SMA) or Exponential Moving Average (EMA)) over a chosen period. It enables users to analyze trends and patterns using financial data.

Purpose

The function is designed to:

1. Compare the **adjusted closing prices** of two stock tickers.
2. Visualize **moving averages (SMA/EMA)** of the stock prices over a customizable time window.
3. Allow users to adjust parameters such as tickers, moving average type, duration, and window size.

Source Code

```
def price_moving_average_dashboard():


    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'price1': '#FF4500',  # Price line for Ticker 1
        'price2': '#1E90FF',  # Price line for Ticker 2
        'sma_ema1': '#FFD700',  # SMA/EMA line for Ticker 1
        'sma_ema2': '#32CD32',  # SMA/EMA line for Ticker 2
    }

    # Define the layout
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Price vs. Moving Average (SMA/EMA) Dashboard",
style={'textAlign': 'center', 'color': colors['text']}),

            html.Div([
```

```
                html.Label("Enter First Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker1-input', type='text', value='AAPL',
style={'margin-right': '20px'}),
                html.Label("Enter Second Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker2-input', type='text', value='MSFT'),
            ], style={'margin-bottom': '20px'}),

            html.Div([
                html.Label("Select Moving Average Type:", style={'color':
colors['text']}),
                dcc.Dropdown(
                    id='ma-type-dropdown',
                    options=[
                        {'label': 'Simple Moving Average (SMA)', 'value':
'SMA'},
                        {'label': 'Exponential Moving Average (EMA)', 'value':
'EMA'},
                    ],
                    value='SMA',
                    style={'margin-bottom': '20px'}
                ),
                html.Label("Select Time Duration:", style={'color':
colors['text']}),
                dcc.Dropdown(
                    id='duration-dropdown',
                    options=[
                        {'label': '6 Months', 'value': '6mo'},
                        {'label': '1 Year', 'value': '1y'},
                        {'label': '5 Years', 'value': '5y'},
                    ],
                    value='1y',
                ),
                html.Label("Moving Average Window (Days):", style={'color':
colors['text']}),
                dcc.Input(id='ma-window-input', type='number', value=20,
style={'margin-top': '10px'}),
            ]),

            dcc.Graph(id='ma-comparison-chart', style={'height': '700px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('ma-comparison-chart', 'figure'),
        [
```

```
            Input('ticker1-input', 'value'),
            Input('ticker2-input', 'value'),
            Input('ma-type-dropdown', 'value'),
            Input('duration-dropdown', 'value'),
            Input('ma-window-input', 'value')
        ]
    )
    def update_chart(ticker1, ticker2, ma_type, period, window):
        try:
            # Fetch historical data for both tickers
            data1 = yf.download(ticker1, period=period, interval='1d')
            data2 = yf.download(ticker2, period=period, interval='1d')

            # Calculate moving averages
            if not data1.empty:
                if ma_type == 'SMA':
                    data1['MA'] = data1['Adj
Close'].rolling(window=window).mean()
                elif ma_type == 'EMA':
                    data1['MA'] = data1['Adj Close'].ewm(span=window,
adjust=False).mean()

            if not data2.empty:
                if ma_type == 'SMA':
                    data2['MA'] = data2['Adj
Close'].rolling(window=window).mean()
                elif ma_type == 'EMA':
                    data2['MA'] = data2['Adj Close'].ewm(span=window,
adjust=False).mean()

            # Create the figure
            fig = go.Figure()

            # Add price and moving average for Ticker 1
            if not data1.empty:
                fig.add_trace(go.Scatter(
                    x=data1.index,
                    y=data1['Adj Close'],
                    mode='lines',
                    name=f'{ticker1} Price',
                    line=dict(color=colors['price1'], width=2),
                ))
                fig.add_trace(go.Scatter(
                    x=data1.index,
                    y=data1['MA'],
                    mode='lines',
                    name=f'{ticker1} {ma_type}',
                    line=dict(color=colors['sma_ema1'], dash='dash', width=2),
```

```
                ))

            # Add price and moving average for Ticker 2
            if not data2.empty:
                fig.add_trace(go.Scatter(
                    x=data2.index,
                    y=data2['Adj Close'],
                    mode='lines',
                    name=f'{ticker2} Price',
                    line=dict(color=colors['price2'], width=2),
                ))
                fig.add_trace(go.Scatter(
                    x=data2.index,
                    y=data2['MA'],
                    mode='lines',
                    name=f'{ticker2} {ma_type}',
                    line=dict(color=colors['sma_ema2'], dash='dash', width=2),
                ))

            # Update layout
            fig.update_layout(
                template="plotly_dark",
                title=f"{ticker1} vs {ticker2} - Price and {ma_type}
Comparison",
                xaxis_title="Date",
                yaxis_title="Price",
                font=dict(color=colors['text']),
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background'],
                legend=dict(x=0, y=1, xanchor='left', yanchor='top')
            )

        except Exception as e:
            # Handle errors
            fig = go.Figure()
            fig.add_annotation(
                text=f"Error: {str(e)}",
                xref="paper", yref="paper",
                x=0.5, y=0.5, showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig
```

```
    # Run the app
    app.run_server(debug=True)


#Example Usecase
#price_moving_average_dashboard()
```

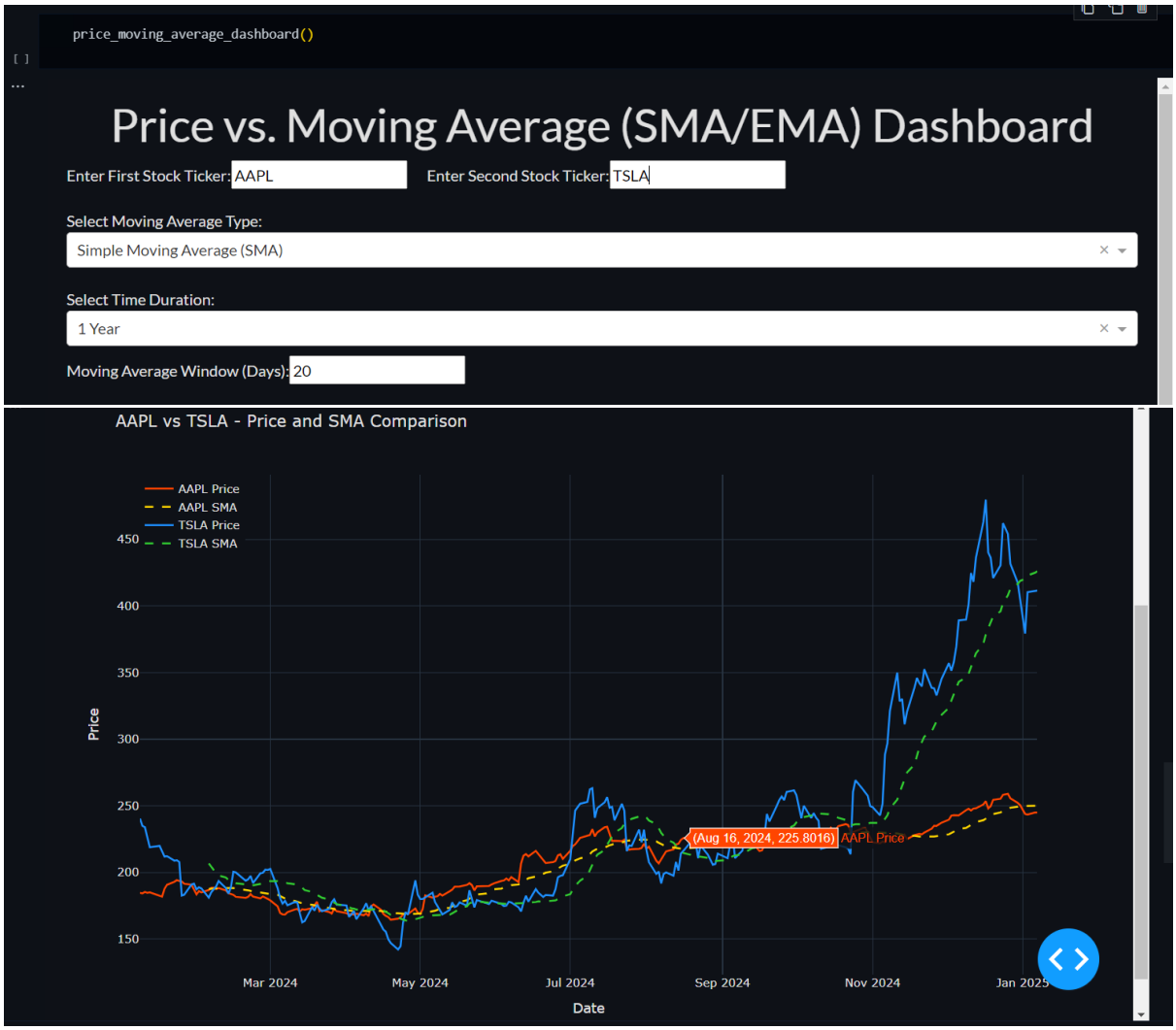
Code Explanation

1. Import Necessary Libraries:
    - dash, dash_core_components, and dash_html_components: To build the web application interface.
    - plotly.graph_objects: For creating dynamic line charts.
    - yfinance: To fetch stock market data.
    - dash_bootstrap_components: For applying the DARKLY theme for styling.
2. Initialize the Dash App:
    - The app is initialized with a dark theme (DARKLY) for a visually appealing UI.
3. Set Color Scheme:
    - A colors dictionary defines custom colors for background, text, and line styles for easy customization.
4. Define the Layout:
    - Header: Displays the title of the dashboard.
    - Input Fields:
        - Text fields for entering stock tickers.
        - Dropdowns for selecting the moving average type (SMA or EMA) and duration (e.g., 6 months, 1 year).
        - Numeric input for specifying the moving average window size in days.
    - Graph Component: Displays the price and moving average comparison chart.
5. Define Callback for Interactivity:
    - The @app.callback decorator binds user inputs (tickers, moving average type, duration, and window) to the chart's output.
    - The callback dynamically fetches stock data, computes the chosen moving average, and updates the graph.
6. Fetch and Process Stock Data:

- The app uses yfinance.download to fetch historical stock data for the given tickers.
- The moving average is computed based on user selections:
  - SMA: Rolling mean over the specified window.
  - EMA: Exponentially weighted mean over the specified window.

7. Create the Comparison Chart:
   - The chart includes:
     - Lines for the adjusted closing price of both tickers.
     - Lines for the selected moving average (SMA/EMA) of both tickers.
   - Color-coded lines distinguish between the two tickers and their respective moving averages.
8. Error Handling:
   - Displays an error message on the chart if stock data cannot be fetched or processed.
9. Run the App:
   - The app is launched using app.run_server(debug=True) for real-time updates and error tracking.

Example Usage



rsi_comparison_dashboard()

The rsi_comparison_dashboard() function creates an interactive Dash web application to compare the Relative Strength Index (RSI) of two stock tickers over a chosen time period. It allows users to analyze momentum and overbought/oversold conditions in financial markets.

Purpose

The function is designed to:

- Compare the RSI values of two stock tickers.
- Allow users to adjust parameters such as stock tickers, time duration, and RSI calculation window size.
- Provide visual insights into RSI trends and highlight critical levels (overbought and oversold thresholds).

Source Code

```
def rsi_comparison_dashboard():
    # Import necessary libraries
    import dash
    import dash_core_components as dcc
    import dash_html_components as html
    from dash.dependencies import Input, Output
    import plotly.graph_objects as go
    import yfinance as yf
    import pandas as pd
    import dash_bootstrap_components as dbc

    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'line1': '#FF4500',  # RSI for Ticker 1
        'line2': '#1E90FF',  # RSI for Ticker 2
        'overbought': '#FF6347',  # Overbought threshold line
        'oversold': '#32CD32',  # Oversold threshold line
    }

    # Define the layout
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
```

```
        children=[
            html.H1("Relative Strength Index (RSI) Comparison Dashboard",
style={'textAlign': 'center', 'color': colors['text']}),

            html.Div([
                html.Label("Enter First Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker1-input', type='text', value='AAPL',
style={'margin-right': '20px'}),
                html.Label("Enter Second Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker2-input', type='text', value='MSFT'),
            ], style={'margin-bottom': '20px'}),

            html.Div([
                html.Label("Select Time Duration:", style={'color':
colors['text']}),
                dcc.Dropdown(
                    id='duration-dropdown',
                    options=[
                        {'label': '6 Months', 'value': '6mo'},
                        {'label': '1 Year', 'value': '1y'},
                        {'label': '5 Years', 'value': '5y'},
                    ],
                    value='1y',
                    style={'margin-bottom': '20px'}
                ),
                html.Label("RSI Window (Days):", style={'color':
colors['text']}),
                dcc.Input(id='rsi-window-input', type='number', value=14,
style={'margin-top': '10px'}),
            ]),

            dcc.Graph(id='rsi-comparison-chart', style={'height': '700px'}),
        ]
    )

    # Define the RSI calculation function
    def calculate_rsi(data, window):
        """Calculate the Relative Strength Index (RSI)."""
        delta = data['Adj Close'].diff()
        gain = (delta.where(delta > 0, 0)).rolling(window=window).mean()
        loss = (-delta.where(delta < 0, 0)).rolling(window=window).mean()

        rs = gain / loss
        rsi = 100 - (100 / (1 + rs))
        return rsi
```

```
    # Define the callback to update the chart
    @app.callback(
        Output('rsi-comparison-chart', 'figure'),
        [
            Input('ticker1-input', 'value'),
            Input('ticker2-input', 'value'),
            Input('duration-dropdown', 'value'),
            Input('rsi-window-input', 'value')
        ]
    )
    def update_chart(ticker1, ticker2, period, window):
        try:
            # Fetch historical data for both tickers
            data1 = yf.download(ticker1, period=period, interval='1d')
            data2 = yf.download(ticker2, period=period, interval='1d')

            # Calculate RSI
            data1['RSI'] = calculate_rsi(data1, window) if not data1.empty
else pd.Series(dtype='float64')
            data2['RSI'] = calculate_rsi(data2, window) if not data2.empty
else pd.Series(dtype='float64')

            # Create the figure with subplots for side-by-side RSI comparison
            fig = go.Figure()

            # Add RSI chart for Ticker 1
            if not data1.empty:
                fig.add_trace(go.Scatter(
                    x=data1.index,
                    y=data1['RSI'],
                    mode='lines',
                    name=f'{ticker1} RSI',
                    line=dict(color=colors['line1'], width=2),
                ))

            # Add RSI chart for Ticker 2
            if not data2.empty:
                fig.add_trace(go.Scatter(
                    x=data2.index,
                    y=data2['RSI'],
                    mode='lines',
                    name=f'{ticker2} RSI',
                    line=dict(color=colors['line2'], width=2),
                ))

            # Add overbought and oversold threshold lines
```

```
            fig.add_hline(y=70, line=dict(color=colors['overbought'],
dash='dash'), annotation_text="Overbought", annotation_position="bottom
right")
            fig.add_hline(y=30, line=dict(color=colors['oversold'],
dash='dash'), annotation_text="Oversold", annotation_position="top right")

            # Update layout
            fig.update_layout(
                template="plotly_dark",
                title=f"{ticker1} vs {ticker2} - RSI Comparison",
                xaxis_title="Date",
                yaxis_title="RSI",
                font=dict(color=colors['text']),
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background'],
                legend=dict(x=0, y=1, xanchor='left', yanchor='top')
            )

        except Exception as e:
            # Handle errors
            fig = go.Figure()
            fig.add_annotation(
                text=f"Error: {str(e)}",
                xref="paper", yref="paper",
                x=0.5, y=0.5, showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)
```

Code Explanation

1. Import Necessary Libraries

- dash, dash_core_components, and dash_html_components: Build the web application interface.
- plotly.graph_objects: Create dynamic line charts for visualization.

- yfinance: Fetch historical stock market data.
- dash_bootstrap_components: Apply the DARKLY theme for a sleek UI.

2. Initialize the Dash App

The app is initialized using dash.Dash with the DARKLY theme to create an intuitive and visually appealing user interface.

3. Set Color Scheme

A colors dictionary defines a consistent color scheme for the dashboard, including:

- Background and text colors.
- Line colors for the two tickers' RSI.
- Colors for overbought (70) and oversold (30) thresholds.

4. Define the Layout

The layout comprises:

- Header: Displays the title of the dashboard.
- Input Fields:
    - Text inputs for entering two stock tickers.
    - A dropdown to select the time duration (e.g., 6 months, 1 year).
    - A numeric input for the RSI calculation window size (e.g., 14 days).
- Graph Component: A dynamic chart that visualizes the RSI comparison.

5. Define the RSI Calculation Function

A helper function, calculate_rsi, computes the RSI:

- Uses price changes (Adj Close).
- Computes average gains and losses over the specified window.
- Calculates RSI using the formula: RSI=100−1001+Average GainAverage LossRSI = 100 - \frac{100}{1 + \frac{\text{Average Gain}}{\text{Average Loss}}}RSI=100−1+Average LossAverage Gain100

6. Define Callback for Interactivity

The @app.callback decorator binds user inputs to the output graph:

- Inputs:
    - Tickers for comparison.
    - Time duration for historical data.
    - RSI calculation window size.
- Output:

- An updated graph displaying RSI trends for the selected tickers.

7. Fetch and Process Stock Data

- Stock data is retrieved using yfinance.download.
- The RSI is calculated for both tickers using the calculate_rsi function.

8. Create the Comparison Chart

The chart includes:

- RSI lines for each ticker, color-coded for distinction.
- Horizontal lines at RSI levels 70 (overbought) and 30 (oversold), with annotations.
- A dark-themed layout for readability and aesthetics.

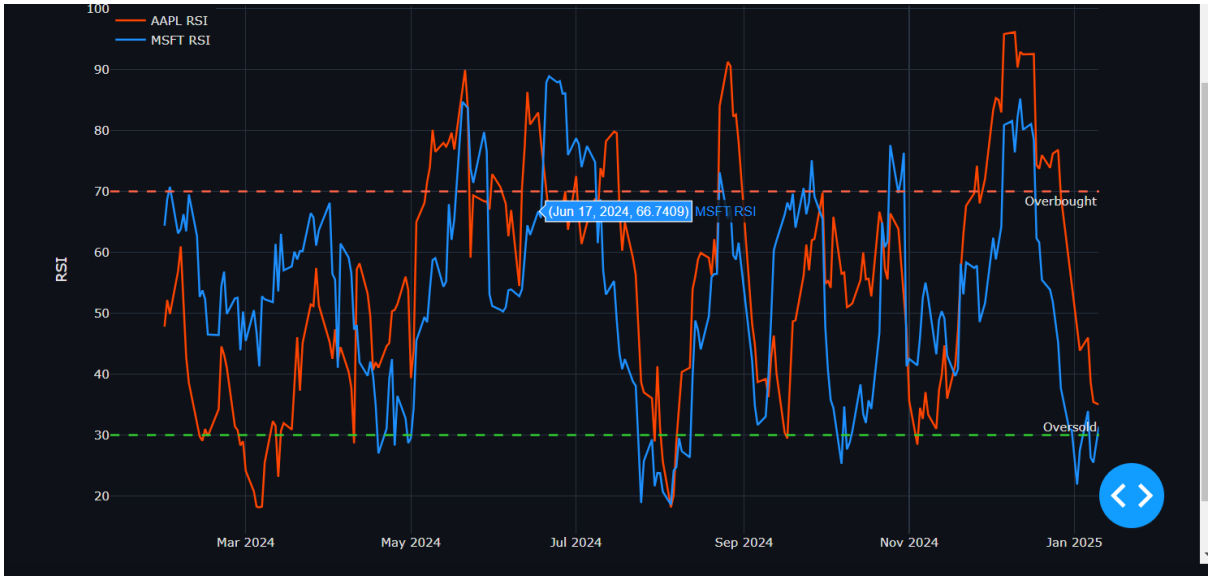
9. Error Handling

If data cannot be fetched or processed, an error message is displayed on the chart.

10. Run the App

The dashboard is launched using app.run_server(debug=True), allowing real-time updates and debugging.

Example Usage


AAPL RSI
MSFT RSI
RSI
100
90
80
70
60
50
40
30
20
Overbought
Oversold
(Jun 17, 2024, 66.7409) MSFT RSI
Mar 2024
May 2024
Jul 2024
Sep 2024
Nov 2024
Jan 2025

bollinger_bands_comparison_dashboard()

The bollinger_bands_comparison_dashboard() function creates an interactive Dash web application to compare the stock prices of two tickers along with their Bollinger Bands over a chosen period. Bollinger Bands are a technical analysis tool that uses a moving average and standard deviation to indicate volatility and potential price trends.

Purpose

The function is designed to:

- Compare the adjusted closing prices of two stock tickers.
- Display Bollinger Bands (upper and lower bounds) for both tickers.
- Allow users to adjust parameters such as stock tickers, duration, and Bollinger Bands window size.

Source Code

```
def bollinger_bands_comparison_dashboard():
    # Import necessary libraries
    import dash
    import dash_core_components as dcc
    import dash_html_components as html
    from dash.dependencies import Input, Output
    import plotly.graph_objects as go
    import yfinance as yf
    import pandas as pd
    import dash_bootstrap_components as dbc

    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'stock1': '#FF4500',  # Stock 1 Bollinger Bands
        'stock2': '#1E90FF',  # Stock 2 Bollinger Bands
        'band_fill': 'rgba(255, 69, 0, 0.2)',  # Semi-transparent fill for
Stock 1 bands
        'band_fill2': 'rgba(30, 144, 255, 0.2)'  # Semi-transparent fill for
Stock 2 bands
```

```
    }

    # Define the layout
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Bollinger Bands Comparison Dashboard",
style={'textAlign': 'center', 'color': colors['text']}),

            html.Div([
                html.Label("Enter First Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker1-input', type='text', value='AAPL',
style={'margin-right': '20px'}),
                html.Label("Enter Second Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker2-input', type='text', value='MSFT'),
            ], style={'margin-bottom': '20px'}),

            html.Div([
                html.Label("Select Time Duration:", style={'color':
colors['text']}),
                dcc.Dropdown(
                    id='duration-dropdown',
                    options=[
                        {'label': '6 Months', 'value': '6mo'},
                        {'label': '1 Year', 'value': '1y'},
                        {'label': '5 Years', 'value': '5y'},
                    ],
                    value='1y',
                    style={'margin-bottom': '20px'}
                ),
                html.Label("Bollinger Bands Window (Days):", style={'color':
colors['text']}),
                dcc.Input(id='bollinger-window-input', type='number',
value=20, style={'margin-top': '10px'}),
            ]),

            dcc.Graph(id='bollinger-bands-comparison-chart', style={'height':
'700px'}),
        ]
    )

    # Define the Bollinger Bands calculation function
    def calculate_bollinger_bands(data, window):
        """Calculate Bollinger Bands."""
        sma = data['Adj Close'].rolling(window=window).mean()  # Simple Moving
Average
```

```
        std = data['Adj Close'].rolling(window=window).std()   # Standard
Deviation
        upper_band = sma + (2 * std)  # Upper Band
        lower_band = sma - (2 * std)  # Lower Band
        return sma, upper_band, lower_band

    # Define the callback to update the chart
    @app.callback(
        Output('bollinger-bands-comparison-chart', 'figure'),
        [
            Input('ticker1-input', 'value'),
            Input('ticker2-input', 'value'),
            Input('duration-dropdown', 'value'),
            Input('bollinger-window-input', 'value')
        ]
    )
    def update_chart(ticker1, ticker2, period, window):
        try:
            # Fetch historical data for both tickers
            data1 = yf.download(ticker1, period=period, interval='1d')
            data2 = yf.download(ticker2, period=period, interval='1d')

            # Calculate Bollinger Bands
            if not data1.empty:
                data1['SMA'], data1['Upper Band'], data1['Lower Band'] =
calculate_bollinger_bands(data1, window)
            if not data2.empty:
                data2['SMA'], data2['Upper Band'], data2['Lower Band'] =
calculate_bollinger_bands(data2, window)

            # Create the figure with subplots for side-by-side comparison
            fig = go.Figure()

            # Add Bollinger Bands for Ticker 1
            if not data1.empty:
                fig.add_trace(go.Scatter(
                    x=data1.index,
                    y=data1['Upper Band'],
                    mode='lines',
                    name=f'{ticker1} Upper Band',
                    line=dict(color=colors['stock1'], dash='dot')
                ))
                fig.add_trace(go.Scatter(
                    x=data1.index,
                    y=data1['Lower Band'],
                    mode='lines',
                    name=f'{ticker1} Lower Band',
                    line=dict(color=colors['stock1'], dash='dot')
```

```
            ))
            fig.add_trace(go.Scatter(
                x=data1.index,
                y=data1['Adj Close'],
                mode='lines',
                name=f'{ticker1} Price',
                line=dict(color=colors['stock1'])
            ))

        # Add Bollinger Bands for Ticker 2
        if not data2.empty:
            fig.add_trace(go.Scatter(
                x=data2.index,
                y=data2['Upper Band'],
                mode='lines',
                name=f'{ticker2} Upper Band',
                line=dict(color=colors['stock2'], dash='dot')
            ))
            fig.add_trace(go.Scatter(
                x=data2.index,
                y=data2['Lower Band'],
                mode='lines',
                name=f'{ticker2} Lower Band',
                line=dict(color=colors['stock2'], dash='dot')
            ))
            fig.add_trace(go.Scatter(
                x=data2.index,
                y=data2['Adj Close'],
                mode='lines',
                name=f'{ticker2} Price',
                line=dict(color=colors['stock2'])
            ))

        # Update layout
        fig.update_layout(
            template="plotly_dark",
            title=f"{ticker1} vs {ticker2} - Bollinger Bands Comparison",
            xaxis_title="Date",
            yaxis_title="Price",
            font=dict(color=colors['text']),
            plot_bgcolor=colors['background'],
            paper_bgcolor=colors['background'],
            legend=dict(x=0, y=1, xanchor='left', yanchor='top')
        )

    except Exception as e:
        # Handle errors
        fig = go.Figure()
```

```
            fig.add_annotation(
                text=f"Error: {str(e)}",
                xref="paper", yref="paper",
                x=0.5, y=0.5, showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)
```

Code Explanation

1. Import Necessary Libraries

- **Dash:** To create the web application interface.
- **Plotly Graph Objects:** For creating dynamic and interactive line charts.
- **yfinance:** To fetch historical stock market data.
- **pandas:** For data manipulation and calculating Bollinger Bands.
- **dash_bootstrap_components:** For styling the dashboard with the "DARKLY" theme.

2. Initialize the Dash App

- The app is initialized with a dark theme (DARKLY) for a sleek and modern UI.

3. Set Color Scheme

- A custom colors dictionary is defined to style the text, background, and Bollinger Bands for both stock tickers.

4. Define the Layout

- **Header:** Displays the title of the dashboard.
- **Input Fields:**
    - Text fields for entering two stock tickers.
    - Dropdown for selecting the time duration (e.g., 6 months, 1 year).
    - Numeric input for specifying the Bollinger Bands window size in days.
- **Graph Component:** Displays the comparison chart with Bollinger Bands for both tickers.

5. Calculate Bollinger Bands

The calculate_bollinger_bands function computes:

- **SMA (Simple Moving Average):** Rolling mean of the adjusted closing prices.
- **Upper Band:** SMA + (2 × Standard Deviation).
- **Lower Band:** SMA - (2 × Standard Deviation).

6. Define the Callback for Interactivity

The @app.callback decorator dynamically updates the chart based on user inputs:

- Fetches historical stock data for the entered tickers using yfinance.download.
- Computes Bollinger Bands for the chosen window size.
- Updates the chart with:
    - Price trends for both tickers.
    - Upper and lower Bollinger Bands for both tickers.

7. Create the Comparison Chart

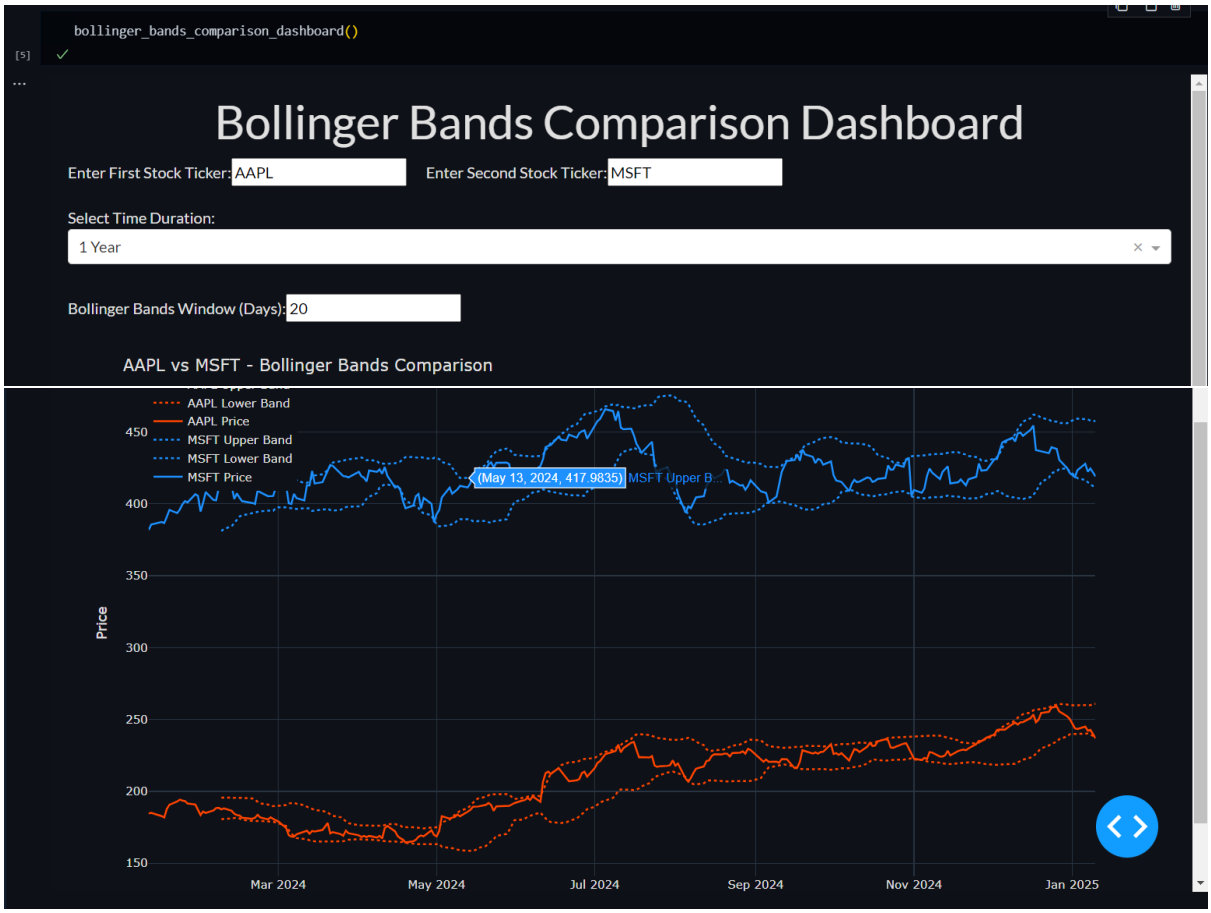
The chart includes:

- **Price Lines:** For adjusted closing prices of both tickers.
- **Bollinger Bands:** Upper and lower bands, distinguished by color and dashed lines.
- **Error Handling:** Displays an error message on the chart if data cannot be fetched or processed.

8. Run the App

- The app is launched using app.run_server(debug=True) for real-time updates and debugging.

Example Usecase



macd_comparison_dashboard()

The macd_comparison_dashboard function creates an interactive financial dashboard using the Dash framework, which allows users to compare the MACD (Moving Average Convergence Divergence) indicators for two stocks over a selected period. Here's a breakdown of its functionality:

Summary

This function provides a powerful tool for financial analysis, making it easy to compare trends and signals for two stocks based on the MACD indicator. It emphasizes customization, interactivity, and error handling for an optimal user experience.

Source Code

```
def macd_comparison_dashboard():
    # Import necessary libraries
    import dash
    import dash_core_components as dcc
    import dash_html_components as html
    from dash.dependencies import Input, Output
    import plotly.graph_objects as go
    import yfinance as yf
    import pandas as pd
    import dash_bootstrap_components as dbc

    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'macd1': '#FF4500',  # Stock 1 MACD line
        'signal1': '#FFA07A',  # Stock 1 Signal line
        'hist1': '#FF6347',  # Stock 1 Histogram
        'macd2': '#1E90FF',  # Stock 2 MACD line
        'signal2': '#87CEEB',  # Stock 2 Signal line
        'hist2': '#4682B4',  # Stock 2 Histogram
    }

    # Define the layout
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("MACD Comparison Dashboard", style={'textAlign': 'center',
'color': colors['text']}),
```

```
            html.Div([
                html.Label("Enter First Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker1-input', type='text', value='AAPL',
style={'margin-right': '20px'}),
                html.Label("Enter Second Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker2-input', type='text', value='MSFT'),
            ], style={'margin-bottom': '20px'}),

            html.Div([
                html.Label("Select Time Duration:", style={'color':
colors['text']}),
                dcc.Dropdown(
                    id='duration-dropdown',
                    options=[
                        {'label': '6 Months', 'value': '6mo'},
                        {'label': '1 Year', 'value': '1y'},
                        {'label': '5 Years', 'value': '5y'},
                    ],
                    value='1y',
                    style={'margin-bottom': '20px'}
                ),
                html.Label("MACD Short Window (Fast):", style={'color':
colors['text']}),
                dcc.Input(id='macd-fast-window', type='number', value=12,
style={'margin-right': '20px'}),
                html.Label("MACD Long Window (Slow):", style={'color':
colors['text']}),
                dcc.Input(id='macd-slow-window', type='number', value=26,
style={'margin-right': '20px'}),
                html.Label("Signal Line Window:", style={'color':
colors['text']}),
                dcc.Input(id='signal-line-window', type='number', value=9),
            ], style={'margin-bottom': '20px'}),

            dcc.Graph(id='macd-comparison-chart', style={'height': '700px'}),
        ]
    )

    # Define the MACD calculation function
    def calculate_macd(data, fast, slow, signal):
        """Calculate MACD, Signal Line, and Histogram."""
        ema_fast = data['Adj Close'].ewm(span=fast, adjust=False).mean()
        ema_slow = data['Adj Close'].ewm(span=slow, adjust=False).mean()
        macd_line = ema_fast - ema_slow
        signal_line = macd_line.ewm(span=signal, adjust=False).mean()
        histogram = macd_line - signal_line
```

```
        return macd_line, signal_line, histogram

    # Define the callback to update the chart
    @app.callback(
        Output('macd-comparison-chart', 'figure'),
        [
            Input('ticker1-input', 'value'),
            Input('ticker2-input', 'value'),
            Input('duration-dropdown', 'value'),
            Input('macd-fast-window', 'value'),
            Input('macd-slow-window', 'value'),
            Input('signal-line-window', 'value')
        ]
    )
    def update_chart(ticker1, ticker2, period, fast, slow, signal):
        try:
            # Fetch historical data for both tickers
            data1 = yf.download(ticker1, period=period, interval='1d')
            data2 = yf.download(ticker2, period=period, interval='1d')

            # Calculate MACD for both stocks
            if not data1.empty:
                data1['MACD'], data1['Signal'], data1['Histogram'] =
calculate_macd(data1, fast, slow, signal)
            if not data2.empty:
                data2['MACD'], data2['Signal'], data2['Histogram'] =
calculate_macd(data2, fast, slow, signal)

            # Create the figure
            fig = go.Figure()

            # Add MACD for Ticker 1
            if not data1.empty:
                fig.add_trace(go.Scatter(
                    x=data1.index,
                    y=data1['MACD'],
                    mode='lines',
                    name=f'{ticker1} MACD Line',
                    line=dict(color=colors['macd1'])
                ))
                fig.add_trace(go.Scatter(
                    x=data1.index,
                    y=data1['Signal'],
                    mode='lines',
                    name=f'{ticker1} Signal Line',
                    line=dict(color=colors['signal1'])
                ))
                fig.add_trace(go.Bar(
```

```
                x=data1.index,
                y=data1['Histogram'],
                name=f'{ticker1} Histogram',
                marker=dict(color=colors['hist1']),
                opacity=0.5
            ))

        # Add MACD for Ticker 2
        if not data2.empty:
            fig.add_trace(go.Scatter(
                x=data2.index,
                y=data2['MACD'],
                mode='lines',
                name=f'{ticker2} MACD Line',
                line=dict(color=colors['macd2'])
            ))
            fig.add_trace(go.Scatter(
                x=data2.index,
                y=data2['Signal'],
                mode='lines',
                name=f'{ticker2} Signal Line',
                line=dict(color=colors['signal2'])
            ))
            fig.add_trace(go.Bar(
                x=data2.index,
                y=data2['Histogram'],
                name=f'{ticker2} Histogram',
                marker=dict(color=colors['hist2']),
                opacity=0.5
            ))

        # Update layout
        fig.update_layout(
            template="plotly_dark",
            title=f"{ticker1} vs {ticker2} - MACD Comparison",
            xaxis_title="Date",
            yaxis_title="Value",
            font=dict(color=colors['text']),
            plot_bgcolor=colors['background'],
            paper_bgcolor=colors['background'],
            legend=dict(x=0, y=1, xanchor='left', yanchor='top')
        )

    except Exception as e:
        # Handle errors
        fig = go.Figure()
        fig.add_annotation(
            text=f"Error: {str(e)}",
```

```
                xref="paper", yref="paper",
                x=0.5, y=0.5, showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)
```

Code Explanation

1. Import Required Libraries

The function starts by importing libraries essential for creating the app (dash, dash_core_components, dash_html_components, plotly, etc.) and fetching financial data (yfinance).

---

2. Initialize the App

- A Dash app is initialized with a dark theme (DARKLY) from dash_bootstrap_components for an aesthetically pleasing interface.

---

3. Define Color Scheme

- A dictionary colors is defined to maintain a consistent color scheme for text, background, and various chart elements.

---

4. Layout of the Dashboard

The app's layout includes:

- Title: A centered header displaying "MACD Comparison Dashboard."
- Inputs:
    - Two text inputs for stock tickers (default: "AAPL" and "MSFT").
    - A dropdown for selecting the time duration (e.g., 6 months, 1 year, or 5 years).
    - Numeric inputs for MACD parameters (fast, slow, and signal line windows).
- Graph Area: A large plot (dcc.Graph) to display the MACD comparison chart.

---

5. Define the MACD Calculation

The calculate_macd function computes the MACD, signal line, and histogram values for a stock's adjusted closing prices:

- MACD Line: Difference between two exponentially weighted moving averages (fast and slow windows).
- Signal Line: Exponential moving average of the MACD line.
- Histogram: Difference between the MACD line and the signal line.

---

6. Update Chart via Callback

A callback function dynamically updates the MACD comparison chart based on user inputs:

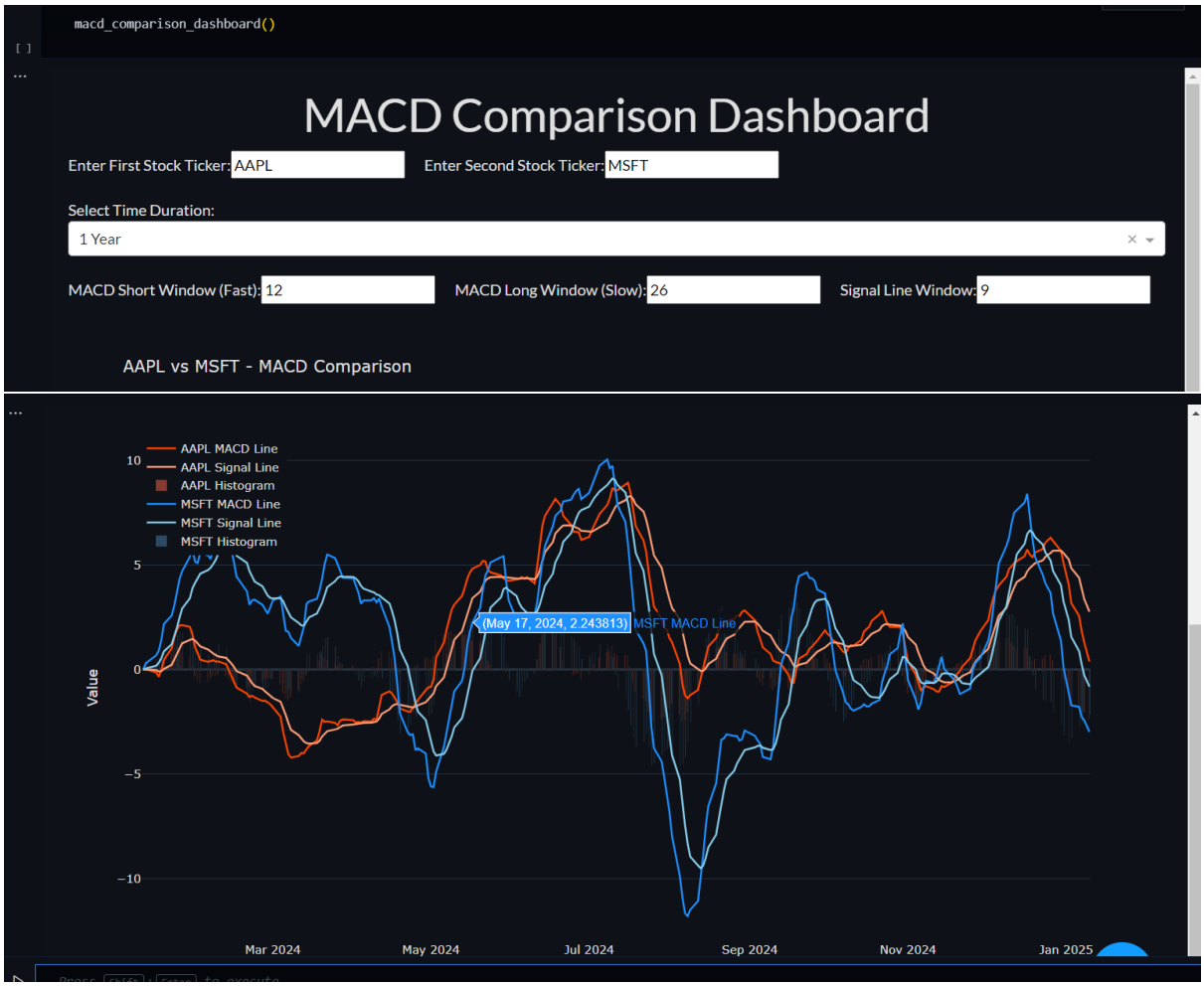
- Inputs Monitored:
    - Stock tickers (ticker1, ticker2).
    - Time period for data.
    - MACD parameters (fast, slow, and signal line windows).
- Data Fetching: Uses yfinance to retrieve historical data for both stocks over the specified period.
- MACD Calculation: Applies the calculate_macd function to compute MACD indicators for each stock.
- Visualization:
    - Uses plotly.graph_objects to plot:
        - MACD line, signal line, and histogram for each stock in different colors.
    - Ensures the layout matches the defined dark theme and colors.
- Error Handling: Displays an error message in case of any issues (e.g., invalid ticker).

---

7. Run the App

The app runs locally, displaying an interactive dashboard that allows users to compare MACD indicators of two stocks visually.

Example Usecase



comparative_return_chart()

The comparative_return_chart function creates an interactive financial dashboard using the Dash framework, enabling users to compare cumulative returns of two stocks over a selected time period. Here's a breakdown of its functionality:

Summary

This function provides an interactive financial analysis tool, enabling users to compare the performance of two stocks visually. It emphasizes:

- Customization (user input for stock tickers and time period).
- Interactivity (dynamic chart updates based on inputs).
- Robustness (error handling for invalid inputs).

Source Code

```
def comparative_return_chart():
    # Import necessary libraries
    import dash
    import dash_core_components as dcc
    import dash_html_components as html
    from dash.dependencies import Input, Output
    import plotly.graph_objects as go
    import yfinance as yf
    import pandas as pd
    import dash_bootstrap_components as dbc

    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'stock1': '#FF6347',  # Stock 1
        'stock2': '#1E90FF',  # Stock 2
    }

    # Define the layout
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
```

```
            html.H1("Comparative Return Chart", style={'textAlign': 'center',
'color': colors['text']}),

            html.Div([
                html.Label("Enter First Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker1-input', type='text', value='AAPL',
style={'margin-right': '20px'}),
                html.Label("Enter Second Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker2-input', type='text', value='MSFT'),
            ], style={'margin-bottom': '20px'}),

            html.Div([
                html.Label("Select Time Duration:", style={'color':
colors['text']}),
                dcc.Dropdown(
                    id='duration-dropdown',
                    options=[
                        {'label': '1 Month', 'value': '1mo'},
                        {'label': '6 Months', 'value': '6mo'},
                        {'label': '1 Year', 'value': '1y'},
                        {'label': '5 Years', 'value': '5y'},
                    ],
                    value='1y',
                    style={'margin-bottom': '20px'}
                ),
            ], style={'margin-bottom': '20px'}),

            dcc.Graph(id='return-comparison-chart', style={'height':
'700px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('return-comparison-chart', 'figure'),
        [
            Input('ticker1-input', 'value'),
            Input('ticker2-input', 'value'),
            Input('duration-dropdown', 'value')
        ]
    )
    def update_chart(ticker1, ticker2, period):
        try:
            # Fetch historical data for both tickers
            data1 = yf.download(ticker1, period=period, interval='1d')
            data2 = yf.download(ticker2, period=period, interval='1d')
```

```
            # Calculate cumulative returns
            data1['Daily Return'] = data1['Adj Close'].pct_change()
            data2['Daily Return'] = data2['Adj Close'].pct_change()

            data1['Cumulative Return'] = (1 + data1['Daily Return']).cumprod()
- 1
            data2['Cumulative Return'] = (1 + data2['Daily Return']).cumprod()
- 1


            # Create the figure
            fig = go.Figure()

            # Add cumulative returns for Stock 1
            if not data1.empty:
                fig.add_trace(go.Scatter(
                    x=data1.index,
                    y=data1['Cumulative Return'] * 100,
                    mode='lines',
                    name=f'{ticker1} Cumulative Return',
                    line=dict(color=colors['stock1'])
                ))

            # Add cumulative returns for Stock 2
            if not data2.empty:
                fig.add_trace(go.Scatter(
                    x=data2.index,
                    y=data2['Cumulative Return'] * 100,
                    mode='lines',
                    name=f'{ticker2} Cumulative Return',
                    line=dict(color=colors['stock2'])
                ))

            # Update layout
            fig.update_layout(
                template="plotly_dark",
                title=f"{ticker1} vs {ticker2} - Cumulative Return
Comparison",
                xaxis_title="Date",
                yaxis_title="Cumulative Return (%)",
                font=dict(color=colors['text']),
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background'],
                legend=dict(x=0, y=1, xanchor='left', yanchor='top')
            )

        except Exception as e:
            # Handle errors
```

```
            fig = go.Figure()
            fig.add_annotation(
                text=f"Error: {str(e)}",
                xref="paper", yref="paper",
                x=0.5, y=0.5, showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)
```

Code Explanation

1. Import Required Libraries

The function starts by importing libraries essential for creating the app and fetching financial data:

- **Dash** and its components (dash_core_components, dash_html_components) for building the interface.
- **Plotly Graph Objects (go)** for visualizing data.
- **Yahoo Finance (yfinance)** to retrieve stock data.
- **Pandas** for data manipulation.
- **Dash Bootstrap Components (dbc)** for applying the dark-themed styling.

---

2. Initialize the App

A Dash app is initialized using the DARKLY theme from dash_bootstrap_components to provide a modern, visually appealing interface.

---

3. Define Color Scheme

A dictionary colors defines a consistent color scheme for the app's background, text, and chart elements (e.g., colors for each stock).

---

4. Layout of the Dashboard

The app's layout includes the following components:

- **Title**: A header, "Comparative Return Chart," styled with the defined color scheme and centered.
- **Input Fields**:
    - Two text inputs for stock tickers (default values: "AAPL" and "MSFT").
- **Dropdown**: A dropdown menu to select the time period (e.g., 1 month, 6 months, 1 year, 5 years).
- **Graph Area**: A large dcc.Graph component to display the cumulative return comparison chart.

---

5. Data Processing

The cumulative returns are calculated as follows:

- Fetch historical stock data using yfinance for the selected period.
- Compute daily returns as the percentage change in adjusted closing prices.
- Calculate cumulative returns as the compounded growth of daily returns.

---

6. Update Chart via Callback

A callback function dynamically updates the chart based on user inputs:

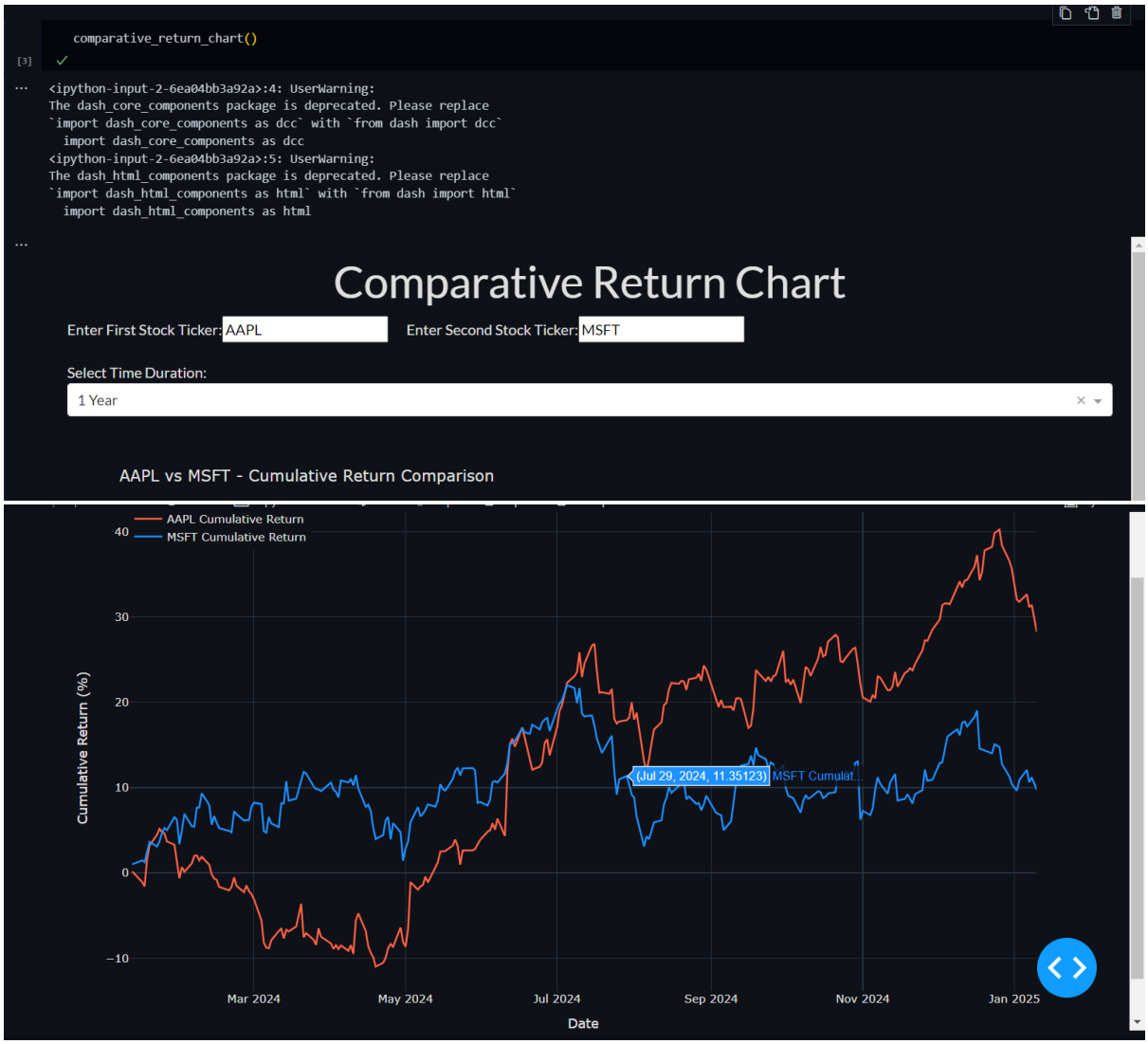
- **Inputs Monitored**:
    - Stock tickers (ticker1 and ticker2).
    - Selected time period from the dropdown.
- **Data Fetching and Calculation**:
    - Retrieves historical stock data using yfinance.
    - Calculates daily and cumulative returns for both stocks.
- **Visualization**:
    - Plots cumulative returns for each stock using plotly.graph_objects with different colors.
    - The layout adheres to the dark theme, ensuring consistency with the overall design.
- **Error Handling**:
    - If any issue arises (e.g., invalid stock ticker), an error message is displayed on the chart.

---

7. Run the App

The app runs locally with app.run_server(debug=True), allowing users to interact with the dashboard.

Example Usage



volatility_comparison_chart()

The volatility_comparison_chart function creates an interactive financial dashboard using the Dash framework, enabling users to compare the volatility metrics of two stocks over a selected time period. Here's a detailed breakdown of its functionality:

Summary

The volatility_comparison_chart function is a powerful tool for financial analysis, enabling users to compare volatility metrics (standard deviation or ATR) for two stocks visually. It features:

- Customization options for tickers, metrics, and time periods.
- Interactive and visually appealing design with error handling for a seamless user experience.

Source Code

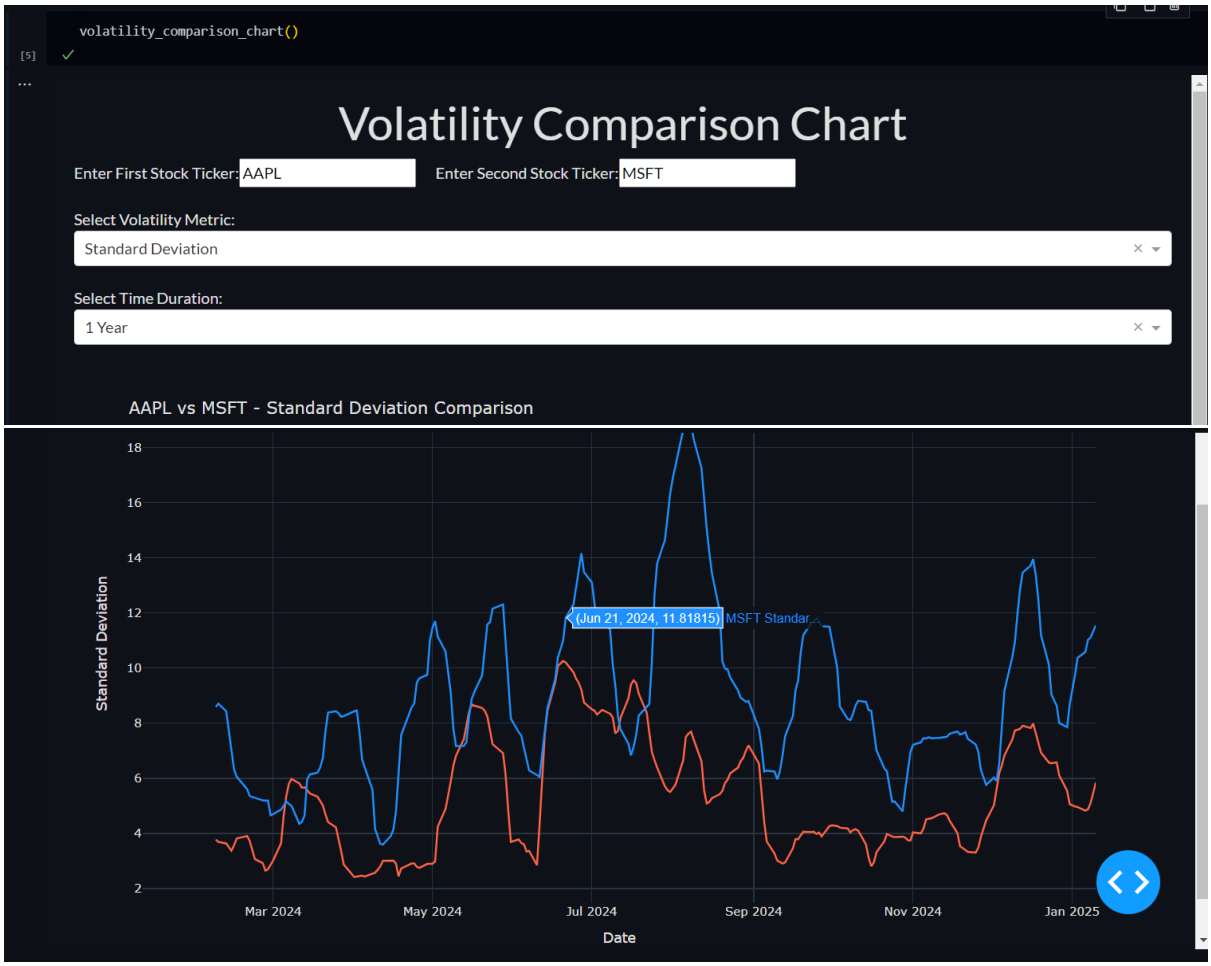
```
def volatility_comparison_chart():
    # Import necessary libraries
    import dash
    import dash_core_components as dcc
    import dash_html_components as html
    from dash.dependencies import Input, Output
    import plotly.graph_objects as go
    import yfinance as yf
    import pandas as pd
    import numpy as np
    import dash_bootstrap_components as dbc

    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'stock1': '#FF6347',  # Stock 1
        'stock2': '#1E90FF',  # Stock 2
    }

    # Define the layout
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Volatility Comparison Chart", style={'textAlign':
'center', 'color': colors['text']}),
```

```
            html.Div([
                html.Label("Enter First Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker1-input', type='text', value='AAPL',
style={'margin-right': '20px'}),
                html.Label("Enter Second Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker2-input', type='text', value='MSFT'),
            ], style={'margin-bottom': '20px'}),

            html.Div([
                html.Label("Select Volatility Metric:", style={'color':
colors['text']}),
                dcc.Dropdown(
                    id='volatility-metric-dropdown',
                    options=[
                        {'label': 'Standard Deviation', 'value': 'std_dev'},
                        {'label': 'Average True Range (ATR)', 'value': 'atr'},
                    ],
                    value='std_dev',
                    style={'margin-bottom': '20px'}
                ),
            ], style={'margin-bottom': '20px'}),

            html.Div([
                html.Label("Select Time Duration:", style={'color':
colors['text']}),
                dcc.Dropdown(
                    id='duration-dropdown',
                    options=[
                        {'label': '1 Month', 'value': '1mo'},
                        {'label': '6 Months', 'value': '6mo'},
                        {'label': '1 Year', 'value': '1y'},
                        {'label': '5 Years', 'value': '5y'},
                    ],
                    value='1y',
                    style={'margin-bottom': '20px'}
                ),
            ], style={'margin-bottom': '20px'}),

            dcc.Graph(id='volatility-comparison-chart', style={'height':
'700px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
```

```
        Output('volatility-comparison-chart', 'figure'),
        [
            Input('ticker1-input', 'value'),
            Input('ticker2-input', 'value'),
            Input('volatility-metric-dropdown', 'value'),
            Input('duration-dropdown', 'value')
        ]
    )
    def update_chart(ticker1, ticker2, metric, period):
        try:
            # Fetch historical data for both stocks
            data1 = yf.download(ticker1, period=period, interval='1d')
            data2 = yf.download(ticker2, period=period, interval='1d')

            # Calculate volatility metrics
            if metric == 'std_dev':  # Standard Deviation
                data1['Volatility'] = data1['Adj
Close'].rolling(window=20).std()
                data2['Volatility'] = data2['Adj
Close'].rolling(window=20).std()
                metric_label = "Standard Deviation"
            elif metric == 'atr':  # Average True Range (ATR)
                data1['High-Low'] = data1['High'] - data1['Low']
                data1['High-Close'] = abs(data1['High'] -
data1['Close'].shift())
                data1['Low-Close'] = abs(data1['Low'] -
data1['Close'].shift())
                data1['True Range'] = data1[['High-Low', 'High-Close', 'Low-
Close']].max(axis=1)
                data1['Volatility'] = data1['True
Range'].rolling(window=14).mean()

                data2['High-Low'] = data2['High'] - data2['Low']
                data2['High-Close'] = abs(data2['High'] -
data2['Close'].shift())
                data2['Low-Close'] = abs(data2['Low'] -
data2['Close'].shift())
                data2['True Range'] = data2[['High-Low', 'High-Close', 'Low-
Close']].max(axis=1)
                data2['Volatility'] = data2['True
Range'].rolling(window=14).mean()
                metric_label = "Average True Range (ATR)"

            # Create the figure
            fig = go.Figure()

            # Add volatility for Stock 1
            if not data1.empty:
```

```
                fig.add_trace(go.Scatter(
                    x=data1.index,
                    y=data1['Volatility'],
                    mode='lines',
                    name=f'{ticker1} {metric_label}',
                    line=dict(color=colors['stock1'])
                ))

            # Add volatility for Stock 2
            if not data2.empty:
                fig.add_trace(go.Scatter(
                    x=data2.index,
                    y=data2['Volatility'],
                    mode='lines',
                    name=f'{ticker2} {metric_label}',
                    line=dict(color=colors['stock2'])
                ))

            # Update layout
            fig.update_layout(
                template="plotly_dark",
                title=f"{ticker1} vs {ticker2} - {metric_label} Comparison",
                xaxis_title="Date",
                yaxis_title=metric_label,
                font=dict(color=colors['text']),
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background'],
                legend=dict(x=0, y=1, xanchor='left', yanchor='top')
            )

        except Exception as e:
            # Handle errors
            fig = go.Figure()
            fig.add_annotation(
                text=f"Error: {str(e)}",
                xref="paper", yref="paper",
                x=0.5, y=0.5, showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)
```

Example Usage



correlation_comparison_chart()

The correlation_comparison_chart function defines a Dash-based web application that visualizes the correlation between the daily returns of two selected stock tickers over a specified time period. Here's a detailed breakdown:

Output:

When the app runs:

- Users can input two stock tickers and select a time period.
- A scatter plot is generated, showing the correlation between the stocks' daily returns.
- The chart includes the correlation coefficient and interactive tooltips for detailed analysis.

Source Code

```
def correlation_comparison_chart():
    # Import necessary libraries
    import dash
    import dash_core_components as dcc
    import dash_html_components as html
    from dash.dependencies import Input, Output
    import plotly.graph_objects as go
    import yfinance as yf
    import pandas as pd
    import dash_bootstrap_components as dbc

    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'ticker1': '#FF6347',  # Tomato for Ticker 1
        'ticker2': '#1E90FF',  # Dodger Blue for Ticker 2
    }

    # Define the layout
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Correlation Comparison Chart", style={'textAlign':
'center', 'color': colors['text']}),
```

```
            html.Div([
                html.Label("Enter First Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker1-input', type='text', value='AAPL',
style={'margin-right': '20px'}),
                html.Label("Enter Second Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='ticker2-input', type='text', value='MSFT'),
            ], style={'margin-bottom': '20px'}),

            html.Div([
                html.Label("Select Time Duration:", style={'color':
colors['text']}),
                dcc.Dropdown(
                    id='duration-dropdown',
                    options=[
                        {'label': '1 Month', 'value': '1mo'},
                        {'label': '6 Months', 'value': '6mo'},
                        {'label': '1 Year', 'value': '1y'},
                        {'label': '5 Years', 'value': '5y'},
                    ],
                    value='1y',
                    style={'margin-bottom': '20px'}
                ),
            ], style={'margin-bottom': '20px'}),

            dcc.Graph(id='correlation-comparison-chart', style={'height':
'700px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('correlation-comparison-chart', 'figure'),
        [
            Input('ticker1-input', 'value'),
            Input('ticker2-input', 'value'),
            Input('duration-dropdown', 'value')
        ]
    )
    def update_chart(ticker1, ticker2, period):
        try:
            # Fetch historical data for both stocks
            data1 = yf.download(ticker1, period=period, interval='1d')
            data2 = yf.download(ticker2, period=period, interval='1d')

            # Calculate daily returns
            data1['Return'] = data1['Adj Close'].pct_change()
```

```
            data2['Return'] = data2['Adj Close'].pct_change()

            # Align data by date and drop NaN values
            merged_data = pd.concat([data1['Return'], data2['Return']],
axis=1, keys=[ticker1, ticker2]).dropna()

            # Calculate correlation
            correlation = merged_data.corr().iloc[0, 1]

            # Create the figure
            fig = go.Figure()

            # Add scatter plot for Ticker 1
            fig.add_trace(go.Scatter(
                x=merged_data[ticker1],
                y=merged_data[ticker2],
                mode='markers',
                name=f"Correlation: {ticker1} vs {ticker2}",
                marker=dict(color=colors['ticker1'], size=8, opacity=0.7),
                hovertemplate=f"<b>{ticker1} Return:</b> {{x:.2%}}<br>" +
                              f"<b>{ticker2} Return:</b>
{{y:.2%}}<extra></extra>"
            ))

            # Update layout
            fig.update_layout(
                template="plotly_dark",
                title=f"Correlation of Returns: {ticker1} vs
{ticker2}<br>Correlation Coefficient: {correlation:.2f}",
                xaxis_title=f"Daily Returns ({ticker1})",
                yaxis_title=f"Daily Returns ({ticker2})",
                font=dict(color=colors['text']),
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background'],
                legend=dict(x=0, y=1, xanchor='left', yanchor='top'),
            )

        except Exception as e:
            # Handle errors
            fig = go.Figure()
            fig.add_annotation(
                text=f"Error: {str(e)}",
                xref="paper", yref="paper",
                x=0.5, y=0.5, showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
```

```
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)
```

Code Explanation

Key Steps in the Function:

1. Import Libraries:
    - Libraries like dash, dash_core_components (dcc), dash_html_components (html), dash.dependencies, plotly.graph_objects, yfinance, and pandas are imported.
    - dash_bootstrap_components is used for styling with a dark theme.
2. App Initialization:
    - The Dash application is initialized using dash.Dash and configured to use the "DARKLY" Bootstrap theme for a modern, dark-mode aesthetic.
3. Color Configuration:
    - A dictionary defines the app's color scheme:
        - background: Background color of the app.
        - text: Text color.
        - ticker1 and ticker2: Colors for scatterplot points representing the two stock tickers.
4. App Layout:
    - Title: Displays the title "Correlation Comparison Chart" at the center.
    - Input Fields:
        - Two dcc.Input fields for users to enter stock tickers (default: AAPL and MSFT).
        - A dcc.Dropdown for users to select a time period (1 Month, 6 Months, 1 Year, 5 Years).
    - Chart Area:
        - A dcc.Graph element to display the correlation scatterplot.
5. Callback Function:
    - A callback updates the chart dynamically when the user changes inputs.

- Inputs: Stock tickers (ticker1, ticker2) and the time period (period).
- Output: Updates the figure displayed in the dcc.Graph.

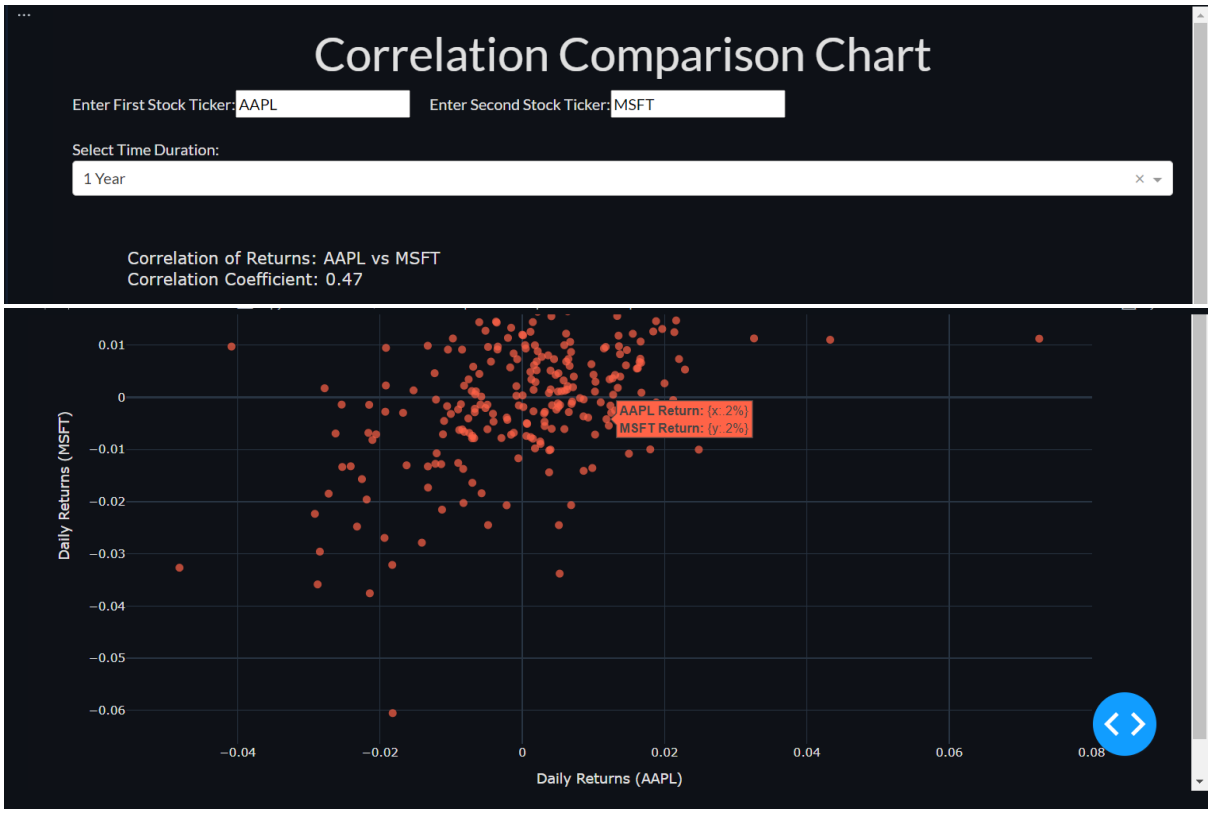
Steps in the Callback (update_chart):

1. Fetch Data:
    - Uses yfinance to download historical stock data for the selected tickers over the specified time period.
2. Calculate Daily Returns:
    - Computes percentage changes in adjusted closing prices to calculate daily returns for both stocks.
3. Data Alignment and Cleaning:
    - Combines daily returns of both stocks into a single DataFrame, aligning by date and removing NaN values.
4. Correlation Calculation:
    - Computes the correlation coefficient between the daily returns of the two stocks.
5. Visualization:
    - Creates a scatter plot of the daily returns of the two stocks using Plotly.
    - The chart displays:
        - Points representing the correlation.
        - Hover text showing individual daily returns for both tickers.
        - The correlation coefficient in the title.
6. Error Handling:
    - If an error occurs (e.g., invalid tickers), a placeholder figure with an error message is displayed.
7. Styling:
    - Dark-themed layout with customized colors for consistency with the app theme.

App Execution:

- The app is launched in debug mode, enabling live reloading and detailed error reporting.

Example Usecase



sector_industry_comparison()

This function creates a Dash web application to compare the performance of two stocks and their respective sector or industry indices over a selected time duration. Here's a breakdown of its components and functionality:

Use Case

This app is useful for **financial analysts**, **investors**, or anyone interested in comparing stock and sector performance. For example:

- Compare the performance of Apple (AAPL) and Microsoft (MSFT) against their respective technology and communication services indices (e.g., XLK and XLC).
- Analyze data over different periods, such as 1 year or 5 years, to observe trends and patterns.

Source Code

```
def sector_industry_comparison():
    # Import necessary libraries
    import dash
    import dash_core_components as dcc
    import dash_html_components as html
    from dash.dependencies import Input, Output
    import plotly.graph_objects as go
    import yfinance as yf
    import dash_bootstrap_components as dbc

    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'stock1': '#FF6347',  # Tomato for Stock 1
        'stock2': '#1E90FF',  # Dodger Blue for Stock 2
        'sector1': '#32CD32',  # Lime Green for Sector 1
        'sector2': '#FFD700',  # Gold for Sector 2
    }

    # Define the layout
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
```

```
            html.H1("Sector/Industry Comparison", style={'textAlign':
'center', 'color': colors['text']}),

            html.Div([
                html.Label("Enter First Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='stock1-input', type='text', value='AAPL',
style={'margin-right': '20px'}),
                html.Label("Enter Second Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='stock2-input', type='text', value='MSFT'),
            ], style={'margin-bottom': '20px'}),

            html.Div([
                html.Label("Enter Proxy Sector Index for First Stock:",
style={'color': colors['text']}),
                dcc.Input(id='sector1-input', type='text', value='XLK',
style={'margin-right': '20px'}),
                html.Label("Enter Proxy Sector Index for Second Stock:",
style={'color': colors['text']}),
                dcc.Input(id='sector2-input', type='text', value='XLC'),
            ], style={'margin-bottom': '20px'}),

            html.Div([
                html.Label("Select Time Duration:", style={'color':
colors['text']}),
                dcc.Dropdown(
                    id='duration-dropdown',
                    options=[
                        {'label': '1 Month', 'value': '1mo'},
                        {'label': '6 Months', 'value': '6mo'},
                        {'label': '1 Year', 'value': '1y'},
                        {'label': '5 Years', 'value': '5y'},
                    ],
                    value='1y',
                    style={'margin-bottom': '20px'}
                ),
            ], style={'margin-bottom': '20px'}),

            dcc.Graph(id='sector-industry-comparison-chart', style={'height':
'700px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('sector-industry-comparison-chart', 'figure'),
        [
```

```
            Input('stock1-input', 'value'),
            Input('stock2-input', 'value'),
            Input('sector1-input', 'value'),
            Input('sector2-input', 'value'),
            Input('duration-dropdown', 'value')
        ]
    )
    def update_chart(stock1, stock2, sector1, sector2, period):
        try:
            # Fetch historical data for stocks and sectors
            stock1_data = yf.download(stock1, period=period, interval='1d')
            stock2_data = yf.download(stock2, period=period, interval='1d')
            sector1_data = yf.download(sector1, period=period, interval='1d')
            sector2_data = yf.download(sector2, period=period, interval='1d')

            # Calculate cumulative returns
            stock1_data['Cumulative Return'] = (1 + stock1_data['Adj
Close'].pct_change()).cumprod()
            stock2_data['Cumulative Return'] = (1 + stock2_data['Adj
Close'].pct_change()).cumprod()
            sector1_data['Cumulative Return'] = (1 + sector1_data['Adj
Close'].pct_change()).cumprod()
            sector2_data['Cumulative Return'] = (1 + sector2_data['Adj
Close'].pct_change()).cumprod()

            # Create the figure
            fig = go.Figure()

            # Add traces for stocks
            fig.add_trace(go.Scatter(
                x=stock1_data.index,
                y=stock1_data['Cumulative Return'],
                mode='lines',
                name=f"{stock1} (Stock)",
                line=dict(color=colors['stock1'], width=2),
            ))
            fig.add_trace(go.Scatter(
                x=stock2_data.index,
                y=stock2_data['Cumulative Return'],
                mode='lines',
                name=f"{stock2} (Stock)",
                line=dict(color=colors['stock2'], width=2),
            ))

            # Add traces for sectors
            fig.add_trace(go.Scatter(
                x=sector1_data.index,
                y=sector1_data['Cumulative Return'],
```

```
                mode='lines',
                name=f"{sector1} (Sector)",
                line=dict(color=colors['sector1'], width=2, dash='dash'),
            ))
            fig.add_trace(go.Scatter(
                x=sector2_data.index,
                y=sector2_data['Cumulative Return'],
                mode='lines',
                name=f"{sector2} (Sector)",
                line=dict(color=colors['sector2'], width=2, dash='dash'),
            ))

            # Update layout
            fig.update_layout(
                template="plotly_dark",
                title=f"Sector/Industry Comparison: {stock1} vs {stock2}",
                xaxis_title="Date",
                yaxis_title="Cumulative Return",
                font=dict(color=colors['text']),
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background'],
                legend=dict(x=0, y=1, xanchor='left', yanchor='top'),
            )

        except Exception as e:
            # Handle errors
            fig = go.Figure()
            fig.add_annotation(
                text=f"Error: {str(e)}",
                xref="paper", yref="paper",
                x=0.5, y=0.5, showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)


#Usecase Example
#sector_industry_comparison()
```

Code Explanation

1. Importing Required Libraries

The function imports libraries for:

- Dash: For building the web application.
- Dash Core Components: For input fields and graphical components.
- Dash HTML Components: For structuring the UI with HTML elements.
- Plotly: For creating interactive plots.
- Yahoo Finance (yfinance): To fetch stock and sector historical data.
- Dash Bootstrap Components: To style the app using Bootstrap themes.

---

2. App Initialization

The app is initialized with a dark-themed layout using the DARKLY Bootstrap stylesheet. A color scheme for different elements (background, text, stocks, and sectors) is defined.

---

3. App Layout

The layout consists of:

1. Title: A centered header displaying "Sector/Industry Comparison."
2. Inputs:
    - Stock tickers for the two companies.
    - Proxy sector indices for the two stocks.
    - A dropdown menu to select the time duration for comparison (e.g., 1 month, 6 months, 1 year, etc.).
3. Graph: A placeholder for a Plotly graph to visualize the cumulative returns of stocks and sectors.

---

4. Callback Functionality

The callback function dynamically updates the graph based on user input:

- Inputs: User-provided values for stock tickers, sector indices, and the selected time duration.
- Outputs: A graph (dcc.Graph) showing cumulative returns.

Steps in the Callback:

1. Fetch Historical Data:

- Uses yfinance.download to get daily adjusted closing prices for the two stocks and their sector indices over the selected period.

2. Calculate Cumulative Returns:
    - Computes cumulative returns using percentage changes in adjusted closing prices.
3. Create the Graph:
    - Adds separate traces for each stock and sector, with distinct colors and line styles (solid for stocks, dashed for sectors).
4. Handle Errors:
    - If data fetching or processing fails, displays an error message on the graph.

---

5. Interactive Graph

The generated graph:

- Compares cumulative returns for the selected stocks and sectors.
- Features a dark background with Plotly's dark theme.
- Includes a legend and dynamic labels for better understanding.

---

6. Running the App

The app runs locally with debug=True, allowing real-time updates and error tracking during development.

Example Usecase

dividend_yield_comparison()

The dividend_yield_comparison function is a complete Dash web application that allows users to compare the dividend yields of two stocks over a specified period. Here's a detailed breakdown of its components:

Key Features

- **Interactive Inputs:** Users can customize stock tickers and time durations.
- **Dynamic Updates:** Changes in inputs reflect instantly in the graph.
- **Error Handling:** Provides clear feedback if an error occurs.
- **Aesthetic Design:** A dark, modern theme with customizable colors.

Source Code

```
def dividend_yield_comparison():
    # Import necessary libraries
    import dash
    import dash_core_components as dcc
    import dash_html_components as html
    from dash.dependencies import Input, Output
    import plotly.graph_objects as go
    import yfinance as yf
    import dash_bootstrap_components as dbc
    import pandas as pd

    # Initialize the Dash app
    app = dash.Dash(__name__, external_stylesheets=[dbc.themes.DARKLY])

    # Define colors
    colors = {
        'background': '#0E1117',
        'text': '#E0E0E0',
        'stock1': '#FF6347',  # Tomato for Stock 1
        'stock2': '#1E90FF',  # Dodger Blue for Stock 2
    }

    # Define the layout
    app.layout = html.Div(
        style={'backgroundColor': colors['background'], 'padding': '20px'},
        children=[
            html.H1("Dividend Yield Comparison", style={'textAlign': 'center',
'color': colors['text']}),
```

```
            html.Div([
                html.Label("Enter First Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='stock1-input', type='text', value='AAPL',
style={'margin-right': '20px'}),
                html.Label("Enter Second Stock Ticker:", style={'color':
colors['text']}),
                dcc.Input(id='stock2-input', type='text', value='MSFT'),
            ], style={'margin-bottom': '20px'}),

            html.Div([
                html.Label("Select Time Duration:", style={'color':
colors['text']}),
                dcc.Dropdown(
                    id='duration-dropdown',
                    options=[
                        {'label': '1 Year', 'value': '1y'},
                        {'label': '2 Years', 'value': '2y'},
                        {'label': '5 Years', 'value': '5y'},
                    ],
                    value='1y',
                    style={'margin-bottom': '20px'}
                ),
            ], style={'margin-bottom': '20px'}),

            dcc.Graph(id='dividend-yield-chart', style={'height': '700px'}),
        ]
    )

    # Define the callback to update the chart
    @app.callback(
        Output('dividend-yield-chart', 'figure'),
        [
            Input('stock1-input', 'value'),
            Input('stock2-input', 'value'),
            Input('duration-dropdown', 'value')
        ]
    )
    def update_chart(stock1, stock2, period):
        try:
            # Fetch historical data for stocks
            stock1_data = yf.Ticker(stock1)
            stock2_data = yf.Ticker(stock2)

            stock1_dividends = stock1_data.dividends
            stock2_dividends = stock2_data.dividends

            # Fetch historical close prices
```

```
            stock1_prices = yf.download(stock1, period=period,
interval='1mo')['Adj Close']
            stock2_prices = yf.download(stock2, period=period,
interval='1mo')['Adj Close']

            # Standardize datetime index to be timezone naive
            stock1_dividends.index = stock1_dividends.index.tz_localize(None)
            stock2_dividends.index = stock2_dividends.index.tz_localize(None)
            stock1_prices.index = stock1_prices.index.tz_localize(None)
            stock2_prices.index = stock2_prices.index.tz_localize(None)

            # Align the data to the same index
            stock1_dividends = stock1_dividends.reindex(stock1_prices.index,
method='pad')
            stock2_dividends = stock2_dividends.reindex(stock2_prices.index,
method='pad')

            # Calculate dividend yield
            stock1_yield = (stock1_dividends / stock1_prices) * 100
            stock2_yield = (stock2_dividends / stock2_prices) * 100

            # Create the figure
            fig = go.Figure()

            # Add traces for stocks
            fig.add_trace(go.Scatter(
                x=stock1_yield.index,
                y=stock1_yield,
                mode='lines+markers',
                name=f"{stock1} Dividend Yield",
                line=dict(color=colors['stock1'], width=2),
            ))
            fig.add_trace(go.Scatter(
                x=stock2_yield.index,
                y=stock2_yield,
                mode='lines+markers',
                name=f"{stock2} Dividend Yield",
                line=dict(color=colors['stock2'], width=2),
            ))

            # Update layout
            fig.update_layout(
                template="plotly_dark",
                title=f"Dividend Yield Comparison: {stock1} vs {stock2}",
                xaxis_title="Date",
                yaxis_title="Dividend Yield (%)",
                font=dict(color=colors['text']),
                plot_bgcolor=colors['background'],
```

```
                paper_bgcolor=colors['background'],
                legend=dict(x=0, y=1, xanchor='left', yanchor='top'),
            )

        except Exception as e:
            # Handle errors
            fig = go.Figure()
            fig.add_annotation(
                text=f"Error: {str(e)}",
                xref="paper", yref="paper",
                x=0.5, y=0.5, showarrow=False,
                font=dict(size=20, color=colors['text'])
            )
            fig.update_layout(
                plot_bgcolor=colors['background'],
                paper_bgcolor=colors['background']
            )

        return fig

    # Run the app
    app.run_server(debug=True)


#Example Usecase
#dividend_yield_comparison()
```

Code Explanation

1. Imports Necessary Libraries

- Dash: Used for creating the web application.
- Dash Core Components (dcc): Provides interactive elements like inputs, dropdowns, and graphs.
- Dash HTML Components (html): Facilitates building HTML structures like headings and divs.
- Plotly Graph Objects (go): Enables creating visualizations.
- yFinance (yf): Retrieves stock market data such as historical prices and dividends.
- Dash Bootstrap Components (dbc): Adds pre-designed themes and styles for the app.
- Pandas (pd): Handles data manipulation and alignment.

---

2. Initializes the Dash App

- The app is initialized using the Dash class.

- The external_stylesheets parameter loads the DARKLY Bootstrap theme to give the app a dark, visually appealing style.

---

3. Color Theme

A dictionary, colors, defines color schemes for:

- Background
- Text
- Plot lines for Stock 1 and Stock 2.

---

4. App Layout

The layout of the app is defined using Dash components:

- Title: Displays "Dividend Yield Comparison."
- Inputs:
    - Text fields for two stock tickers (stock1-input and stock2-input).
    - Dropdown for selecting a time duration (duration-dropdown) with options for 1 year, 2 years, and 5 years.
- Graph: A Plotly graph (dividend-yield-chart) to display the comparison.

---

5. Callback for Dynamic Updates

The app uses a callback to dynamically update the graph based on user inputs. This function:

1. Fetches Data: Retrieves historical price and dividend data for the input stocks using the yfinance library.
2. Aligns Data:
    - Aligns dividends and prices to the same dates for accurate calculations.
    - Handles timezones to ensure consistency.
3. Calculates Dividend Yield:
    - Dividend Yield = (Dividends / Stock Prices) * 100
    - Yields are calculated for each stock over the selected period.
4. Creates Plotly Visualization:
    - Plots the dividend yields of the two stocks using line and marker plots.
    - Customizes the layout with dark themes, titles, and legends.

5. Handles Errors: Displays a clear error message in the graph if issues arise during data retrieval or calculation.

---

6. Running the App

The app is launched with app.run_server(debug=True), enabling it to run locally in development mode with automatic reloading for code changes.

Example Usage