## 1. Introduction to Matplotlib

- **Matplotlib** is the fundamental Python library for creating plots and visualizations.
- Useful for visualizing financial data, time series, distributions, etc.
- Two main approaches:
  1. **Pyplot interface** → simple and quick, MATLAB-like.
  2. **Object-oriented (OO) interface** → more flexible for complex figures.

**Example:**

```python
import matplotlib.pyplot as plt

plt.plot([1,2,3,4], [10,20,25,30])
plt.title("Simple Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

**Output:**

- A simple line plot with axes and title.

## 2. Plot Types

| Type | Description | Example |
|---|---|---|
| Line | Time series or sequential data | `plt.plot()` |
| Scatter | Relationship between two variables | `plt.scatter()` |
| Bar | Compare categories | `plt.bar()` |
| Histogram | Distribution of values | `plt.hist()` |
| Boxplot | Summary statistics | `plt.boxplot()` |

## 3. Customization

- **Colors, markers, line styles:**

python Copier le code

```python
plt.plot(x, y, color="red", linestyle="--", marker="o")
```

- **Legend:**

python Copier le code

```python
plt.legend(["Asset 1"])
```

- **Grid:**

python Copier le code

```python
plt.grid(True)
```

- **Figure size:**

python Copier le code

```python
plt.figure(figsize=(10,6))
```

## 4. Plotting Financial Data (Time Series)

- Use the log returns or cumulative returns DataFrame from previous sessions.

**Example:**

python

Copier le code

```python
# Cumulative log returns for selected indices
cum_log_returns[["CAC 40", "S&P", "FTSE"]].plot(figsize=(12,6))
plt.title("Cumulative Log Returns")
plt.xlabel("Date")
plt.ylabel("Cumulative Log Return")
plt.grid(True)
plt.show()
```

**Output:**

- Line plot showing cumulative performance of multiple indices over time.

---

## 2. Bar Plot – Average Daily Returns per Index

python

Copier le code

```
avg_returns = log_returns.mean()
plt.figure(figsize=(10,5))
plt.bar(avg_returns.index, avg_returns.values, color="skyblue")
plt.title("Average Daily Log Returns")
plt.ylabel("Mean Daily Return")
plt.xticks(rotation=45)
plt.grid(axis='y')
plt.show()
```

**Explanation:**

- Compares mean daily return of each index.
- Bars make differences between indices obvious.

## 3. Histogram – Daily Log Returns Distribution

python

```
plt.figure(figsize=(10,5))
plt.hist(log_returns["S&P"], bins=20, color="orange", alpha=0.7)
plt.title("Histogram of S&P Daily Log Returns")
plt.xlabel("Daily Log Return")
plt.ylabel("Frequency")
plt.grid(True)
plt.show()
```

**Explanation:**

- Shows the distribution of daily returns.
- Useful to check volatility and extreme values (outliers).

## 4. Scatter Plot – Relationship Between Two Indices

python

Copier le code

```
plt.figure(figsize=(8,6))
plt.scatter(log_returns["CAC 40"], log_returns["S&P"], color="green", alpha=0.6)
plt.title("Daily Log Return: CAC 40 vs S&P")
plt.xlabel("CAC 40 Daily Log Return")
plt.ylabel("S&P Daily Log Return")
plt.grid(True)
plt.show()
```

**Explanation:**

- Each point = one day's return for both indices.
- Helps visualize correlation: upward trend → positive correlation.

# Exercise – Matplotlib Visualization of Financial Data

**Context:** Use the **log returns and cumulative returns DataFrame** from the previous sessions (`CAC 40`, `S&P`, `FTSE`, etc.).

**Tasks:**

1. **Line Plot:**
   - Plot **cumulative log returns** of `CAC 40` and `S&P` on the same figure.
   - Add **title, axis labels, legend, and grid.**
2. **Bar Plot:**
   - Compute **average daily log return** for all indices.
   - Plot a **bar chart** showing these mean returns.
3. **Histogram:**
   - Plot the **histogram of daily log returns** for `S&P`.
   - Use 20 bins and add gridlines.
4. **Scatter Plot:**
   - Plot a **scatter plot** of daily log returns between `CAC 40` (x-axis) and `S&P` (y-axis).
   - Add gridlines and axis labels.
5. **Optional Challenge:**
   - Plot the **portfolio cumulative returns** (equal-weight portfolio) alongside `CAC 40` and `S&P`.
   - Compare visually which line is smoother (diversification effect).

> 💡 **Hints:**

- Use `.plot()` for line plots, `plt.bar()` for bar charts, `plt.hist()` for histograms, and `plt.scatter()` for scatter plots.
- Customize with `title`, `xlabel`, `ylabel`, `legend()`, `grid()`, `figsize=(..)`

# Solution – Advanced Visualization & Analysis

python Copier le code

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# --- Data Preparation (from previous sessions) ---
df = pd.read_csv("indices.csv", parse_dates=["Date"], dayfirst=True)
df.set_index("Date", inplace=True)
df = df.apply(lambda x: x.str.replace(',', '.').astype(float) if x.dtype=='object' else x)
df.fillna(df.mean(), inplace=True)

# Compute daily log returns
log_returns = np.log(df / df.shift(1)).dropna()

# Equal-weight portfolio
weights = np.array([1/len(log_returns.columns)]*len(log_returns.columns))
portfolio_return = log_returns.dot(weights)
```

---

## 1. Rolling Volatility of S&P (20-day)

python Copier le code

```python
rolling_vol = log_returns["S&P"].rolling(window=20).std()

plt.figure(figsize=(12,5))
plt.plot(rolling_vol, color="purple")
plt.title("20-Day Rolling Volatility of S&P")
plt.xlabel("Date")
plt.ylabel("Volatility")
plt.grid(True)
plt.show()
```

## Introduction to Yahoo Finance (`yfinance`)

- `yfinance` is a Python library that allows easy access to **financial market data**.
- Data available:
  - Stock prices (daily, weekly, monthly)
  - OHLC (Open, High, Low, Close, Volume)
  - Dividends & stock splits
  - Company information
- Output is returned as **Pandas DataFrames** → easy to analyze.
- Widely used in **finance, data science, and trading**.

## Download Stock Data

python Copy code

```python
import yfinance as yf

# Example: Download Apple stock data
data = yf.download("AAPL", start="2023-01-01", end="2023-12-31")

print(data.head())
```

### Output Example:

mathematica Copy code

```
                 Open       High        Low      Close  Adj Close   Volume
Date
2023-01-03  130.2600  130.9000  124.1700  125.0700  124.116898  112117500
2023-01-04  126.8900  128.6550  125.0800  126.3600  125.396660   89113600
2023-01-05  127.1300  127.7700  124.7600  125.0200  124.068077   80962700
```

✅ The index is the **date**, and each column corresponds to a price type.

## Select Closing Prices Only

- Often in finance, we focus on the **closing price**.
- Extract the `Close` column directly.

```python
close_prices = data["Close"]
print(close_prices.head())
```

**Output Example:**

```yaml
Date
2023-01-03    125.0700
2023-01-04    126.3600
2023-01-05    125.0200
2023-01-06    129.6200
2023-01-09    130.1500
Name: Close, dtype: float64
```

✅ Returns a **Pandas Series** indexed by date.

## Download Multiple Stocks

- You can also download several assets at the same time.
- `yfinance` will return a **DataFrame with multiple columns**.

python Copy code

```python
tickers = ["AAPL", "MSFT", "TSLA"]
data = yf.download(tickers, start="2023-01-01", end="2023-12-31")["Close"]

print(data.head())
```

**Output Example:**

yaml Copy code

```yaml
                AAPL       MSFT       TSLA
Date
2023-01-03  125.0700  239.5800  108.1000
2023-01-04  126.3600  229.1000  113.6400
2023-01-05  125.0200  229.1000  110.3400
2023-01-06  129.6200  231.6600  113.0600
2023-01-09  130.1500  229.7300  118.4700
```

✅ Each column corresponds to one stock's closing price.

## Exercise – Portfolio Optimization with Sharpe Ratio

You are managing a portfolio of 3 stocks:

- **AAPL** (Apple)
- **MSFT** (Microsoft)
- **TSLA** (Tesla)

Using daily closing prices for **2023**, complete the following tasks:

1. Download the stock prices from Yahoo Finance.
2. Compute the **daily returns** of each stock.
3. Calculate the **mean returns** and the **covariance matrix**.
4. Build an **equal-weight portfolio** `[1/3, 1/3, 1/3]` and compute:
    - Expected return
    - Volatility (risk)
    - Sharpe ratio (assume a risk-free rate of **3% per year**)
5. Compute the **Sharpe ratio** for each stock individually.
6. Compare: which investment has the best **risk-adjusted performance**?

### 📌 Reminder: What is the Sharpe Ratio?

The **Sharpe Ratio** measures the **excess return per unit of risk**:

$$Sharpe = \frac{E[R_p] - R_f}{\sigma_p}$$

- $E[R_p]$ = expected return of the portfolio (or stock)
- $R_f$ = risk-free rate (here: 3% per year ≈ 0.000119 per day)
- $\sigma_p$ = portfolio volatility

---

### 📌 How to compute the Sharpe Ratio of a portfolio

1. Compute **expected portfolio return**:

$$E[R_p] = w^T \cdot \mu$$

- $w$ = vector of portfolio weights
- $\mu$ = vector of mean returns

2. Compute **portfolio volatility**:

$$\sigma_p = \sqrt{w^T \cdot \Sigma \cdot w}$$

- $\Sigma$ = covariance matrix of returns

3. Apply the Sharpe formula:

$$Sharpe = \frac{E[R_p] - R_f}{\sigma_p}$$

👉 This shows why **both returns and risk** (via covariance) matter in portfolio optimization.

```python
import yfinance as yf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# ---- 1) Download closing prices ----
tickers = ["AAPL", "MSFT", "TSLA"]
data = yf.download(tickers, start="2023-01-01", end="2023-12-31")["Close"]

# If a single ticker is returned as Series, force DataFrame:
if isinstance(data, pd.Series):
    data = data.to_frame()

# ---- 2) Compute daily returns ----
returns = data.pct_change().dropna()

# ---- 3) Mean returns & covariance ----
mean_returns = returns.mean()            # daily mean returns (vector)
cov_matrix = returns.cov()               # daily covariance matrix

# ---- 4) Equal-weighted portfolio metrics ----
weights_eq = np.array([1/3, 1/3, 1/3])
port_return_eq = np.dot(weights_eq, mean_returns)                  # expected dai
port_vol_eq = np.sqrt(weights_eq.T @ cov_matrix.values @ weights_eq)  # daily vol

# Convert to annualized numbers for reporting (optional)
trading_days = 252
port_return_eq_annual = port_return_eq * trading_days
port_vol_eq_annual = port_vol_eq * np.sqrt(trading_days)

# ---- Risk-free rate (3% annual) converted to daily ----
rf_annual = 0.03
rf_daily = rf_annual / trading_days
```

```python
# ---- 5) Sharpe ratios: each stock and the equal-weight portfolio ----
sharpe_stocks = {}
for s in returns.columns:
    mu = mean_returns[s]
    sigma = returns[s].std()
    sharpe_stocks[s] = (mu - rf_daily) / sigma

sharpe_portfolio = (port_return_eq - rf_daily) / port_vol_eq

# ---- Print results ----
pd.set_option('display.float_format', '{:.6f}'.format)

print("=== Daily statistics (computed from downloaded data) ===\n")
print("Mean daily returns:\n", mean_returns, "\n")
print("Daily volatilities (std):\n", returns.std(), "\n")
print("Daily covariance matrix:\n", cov_matrix, "\n")

print("=== Equal-weighted portfolio (daily) ===")
print(f"Expected daily return: {port_return_eq:.6f}")
print(f"Daily volatility:      {port_vol_eq:.6f}")
print(f"Annualized return:     {port_return_eq_annual:.4f}")
print(f"Annualized volatility: {port_vol_eq_annual:.4f}\n")

print("=== Sharpe Ratios (using rf = 3% annual -> rf_daily = {0:.8f}) ===".format
for s, sr in sharpe_stocks.items():
    print(f"{s}: Sharpe (daily) = {sr:.6f}")

print(f"Equal-weight Portfolio Sharpe (daily): {sharpe_portfolio:.6f}")
```

```
# ---- 6) Visual outputs: cumulative returns and Sharpe comparison ----
# cumulative returns
cum_returns = (1 + returns).cumprod()

# portfolio cumulative (equal weights)
portfolio_cum = (cum_returns * weights_eq).sum(axis=1)

# Plot cumulative returns
plt.figure(figsize=(10,6))
for col in cum_returns.columns:
    plt.plot(cum_returns.index, cum_returns[col], label=col)
plt.plot(portfolio_cum.index, portfolio_cum, label="Equal-Weight Portfolio", colo
plt.title("Cumulative Returns: Stocks vs Equal-Weight Portfolio")
plt.ylabel("Cumulative return (growth of 1)")
plt.legend()
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()

# Bar chart of Sharpe ratios (annualized for readability)
sharpe_stocks_annual = {s: ((mean_returns[s]*trading_days - rf_annual) / (returns
                        for s in returns.columns}
sharpe_portfolio_annual = (port_return_eq_annual - rf_annual) / port_vol_eq_annua

plt.figure(figsize=(8,5))
names = list(sharpe_stocks_annual.keys()) + ["Portfolio"]
values = [sharpe_stocks_annual[n] for n in returns.columns] + [sharpe_portfolio_a
bars = plt.bar(names, values, color=["C0","C1","C2","k"])
plt.title("Annualized Sharpe Ratios (rf = 3% p.a.)")
plt.ylabel("Sharpe Ratio (annualized)")
plt.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.show()
```

# 📊 Markowitz Portfolio Optimization

## 🔷 Concept Recap (for a slide)

- **Markowitz Theory**: Find the best portfolio allocation between assets by balancing **return vs. risk**.
- **Expected Return**: Weighted average of asset returns.
- **Risk (Volatility)**: Standard deviation of portfolio returns.
- **Covariance Matrix**: Shows how assets move together.
- **Efficient Frontier**: The set of optimal portfolios offering the highest expected return for a given level of risk.
- **Sharpe Ratio**: Measures risk-adjusted return (portfolio return above risk-free rate divided by volatility).

Formula:

$$R_p = \sum_{i=1}^{n} w_i R_i$$

$$\sigma_p^2 = \mathbf{w}^T \Sigma \mathbf{w}$$

$$Sharpe = \frac{R_p - R_f}{\sigma_p}$$

## Exercise – Portfolio Optimization (Efficient Portfolio)

You are still working with the same 3 stocks:

- **AAPL** (Apple)
- **MSFT** (Microsoft)
- **TSLA** (Tesla)

You already have their **daily returns in 2023**.

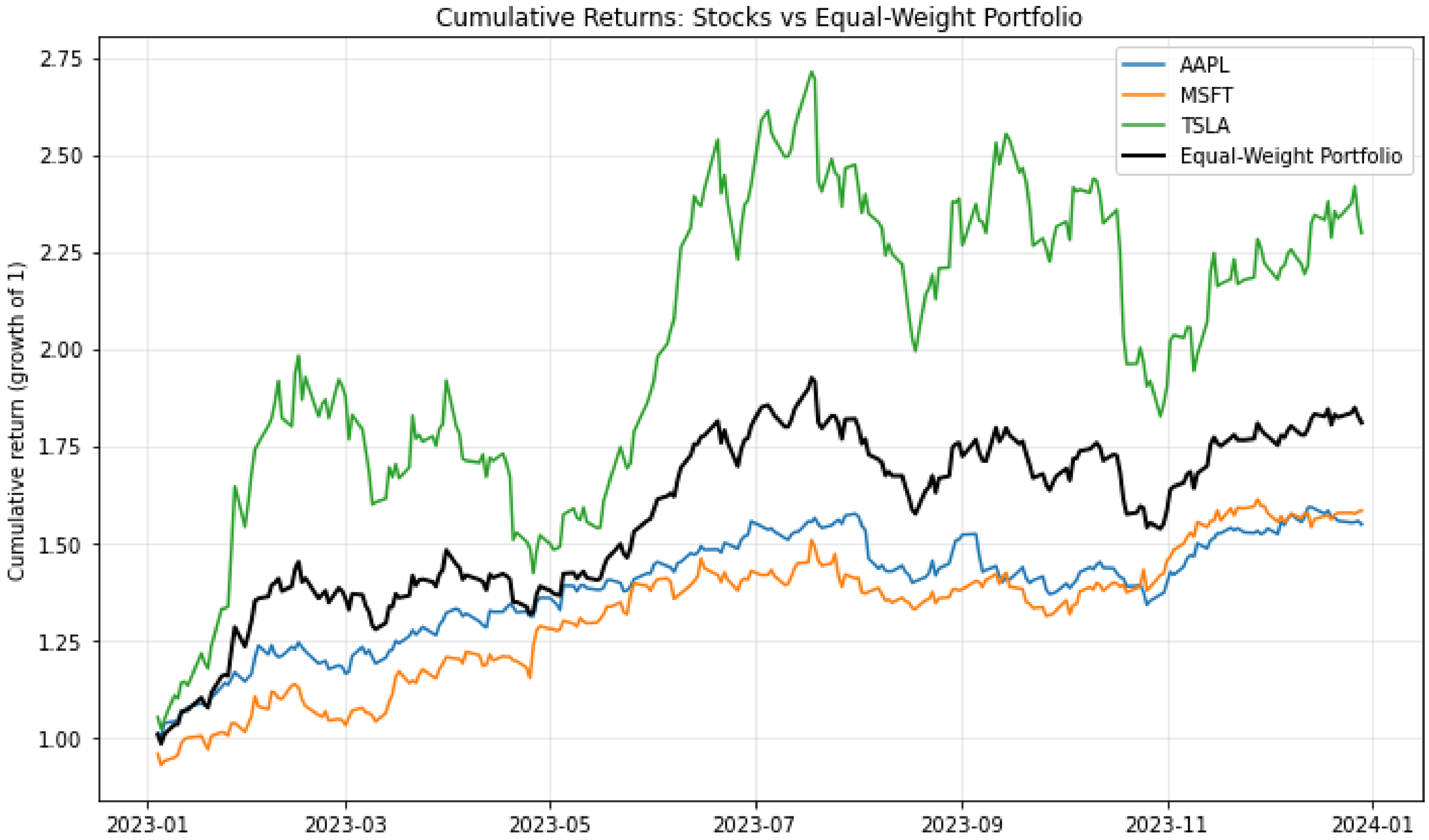
Now, instead of equal weights, you want to **find the optimal portfolio**.

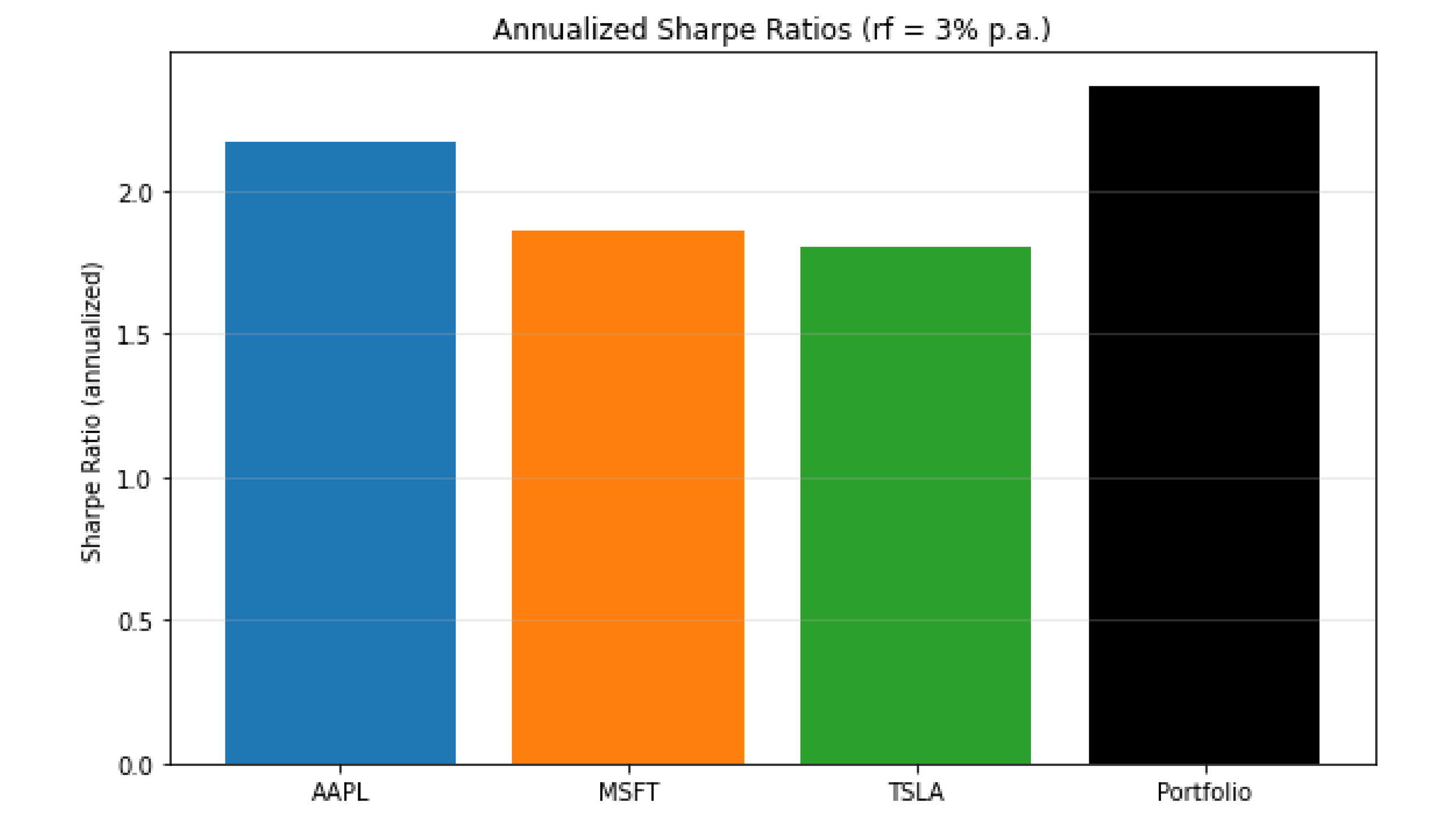
### Your tasks:

1. Generate **1000 random portfolios** by randomly assigning weights to the 3 stocks (weights must sum to 1).
2. For each portfolio, compute:
   - Expected return
   - Volatility
   - Sharpe ratio (use the **3% annual risk-free rate**)
3. Plot all portfolios on a graph:
   - X-axis = Volatility
   - Y-axis = Expected return
   - Color = Sharpe ratio
4. Identify and highlight:
   - The **Maximum Sharpe Ratio portfolio** (tangent portfolio)
   - The **Minimum Volatility portfolio**
5. Compare the results with the **equal-weight portfolio** from the previous exercise.

📌 **Reminder: Optimization logic**

- **Diversification**: combining stocks reduces overall volatility.
- **Efficient frontier**: the set of portfolios that offer the best return for a given level of risk.
- **Optimal portfolio**: the one with the **highest Sharpe ratio** (best risk-adjusted return).


Cumulative Returns: Stocks vs Equal-Weight Portfolio
Cumulative return (growth of 1)
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
2023-01
2023-03
2023-05
2023-07
2023-09
2023-11
2024-01
AAPL
MSFT
TSLA
Equal-Weight Portfolio


Annualized Sharpe Ratios (rf = 3% p.a.)
Sharpe Ratio (annualized)
0.0
0.5
1.0
1.5
2.0
AAPL
MSFT
TSLA
Portfolio

```python
# Correction: Portfolio Optimization (3 assets, Monte Carlo Simulation)
import yfinance as yf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# ---- 1) Download data ----
tickers = ["AAPL", "MSFT", "TSLA"]
data = yf.download(tickers, start="2023-01-01", end="2023-12-31")["Close"]

# ---- 2) Compute daily returns ----
returns = data.pct_change().dropna()
mean_returns = returns.mean()
cov_matrix = returns.cov()

# ---- 3) Parameters ----
n_portfolios = 10000
trading_days = 252
rf_annual = 0.03
rf_daily = rf_annual / trading_days

results = np.zeros((3+3, n_portfolios))
# rows: return, vol, sharpe, weight1, weight2, weight3
```

```python
# ---- 4) Monte Carlo Simulation ----
for i in range(n_portfolios):
    # random weights that sum to 1
    weights = np.random.random(3)
    weights /= np.sum(weights)

    # portfolio return & vol
    port_return = np.dot(weights, mean_returns)
    port_vol = np.sqrt(weights.T @ cov_matrix.values @ weights)

    # sharpe ratio
    sharpe = (port_return - rf_daily) / port_vol

    # store results
    results[0,i] = port_return
    results[1,i] = port_vol
    results[2,i] = sharpe
    results[3:,i] = weights

# ---- 5) Extract best portfolios ----
max_sharpe_idx = np.argmax(results[2])
min_vol_idx = np.argmin(results[1])

max_sharpe_port = results[:, max_sharpe_idx]
min_vol_port = results[:, min_vol_idx]
```

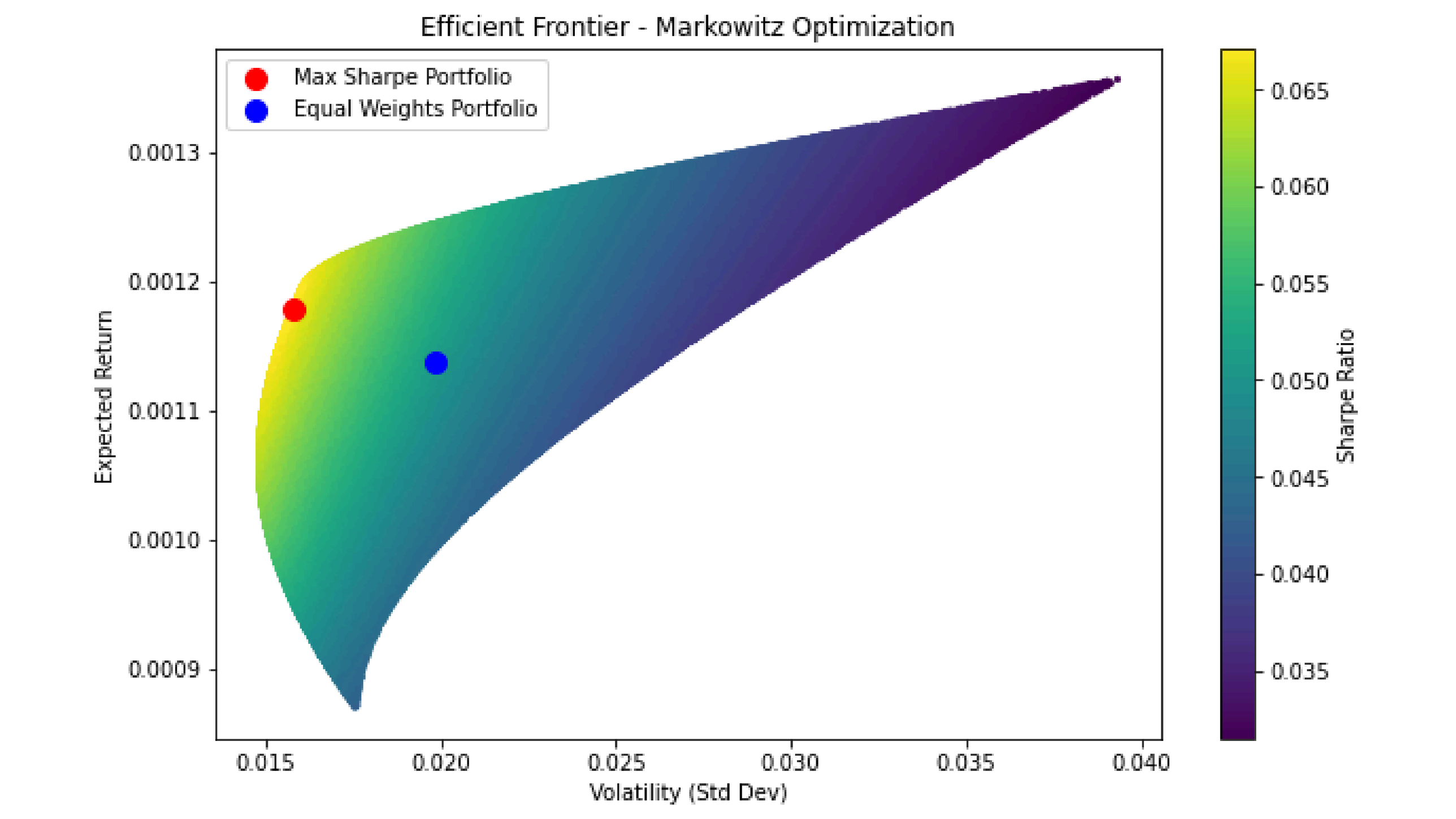
```python
# ---- 6) Plot efficient frontier ----
plt.figure(figsize=(10,6))
plt.scatter(results[1,:], results[0,:], c=results[2,:], cmap="viridis", s=10)
plt.colorbar(label="Sharpe Ratio")
plt.xlabel("Volatility (Daily)")
plt.ylabel("Expected Return (Daily)")
plt.title("Efficient Frontier - Monte Carlo Simulation")

# Highlight best portfolios
plt.scatter(max_sharpe_port[1], max_sharpe_port[0],
            c="red", s=80, marker="*", label="Max Sharpe Ratio")
plt.scatter(min_vol_port[1], min_vol_port[0],
            c="blue", s=80, marker="*", label="Min Volatility")

plt.legend()
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()

# ---- 7) Print results ----
print("Max Sharpe Ratio Portfolio (daily stats):")
print("Return:", max_sharpe_port[0])
print("Volatility:", max_sharpe_port[1])
print("Sharpe Ratio:", max_sharpe_port[2])
print("Weights: AAPL={:.2f}, MSFT={:.2f}, TSLA={:.2f}".format(*max_sharpe_port[3:

print("\nMin Volatility Portfolio (daily stats):")
print("Return:", min_vol_port[0])
print("Volatility:", min_vol_port[1])
print("Sharpe Ratio:", min_vol_port[2])
print("Weights: AAPL={:.2f}, MSFT={:.2f}, TSLA={:.2f}".format(*min_vol_port[3:]))
```


Efficient Frontier - Markowitz Optimization
Max Sharpe Portfolio
Equal Weights Portfolio
Expected Return
0.0013
0.0012
0.0011
0.0010
0.0009
Volatility (Std Dev)
0.015
0.020
0.025
0.030
0.035
0.040
Sharpe Ratio
0.065
0.060
0.055
0.050
0.045
0.040
0.035

## 🔷 1. Maximum Sharpe Ratio Portfolio (MSP)

- Also called the **Tangency Portfolio**.
- It's the portfolio that maximizes the **Sharpe Ratio**.
- Best for investors who want the **highest return per unit of risk**.
- This is what we just calculated above.

---

## 🔷 2. Minimum Variance Portfolio (MVP)

- It's the portfolio that minimizes **risk (variance / volatility)** regardless of return.
- Formula:

$$w_{MVP} = \frac{\Sigma^{-1}\mathbf{1}}{\mathbf{1}^T\Sigma^{-1}\mathbf{1}}$$

Where:

- $\Sigma$ = covariance matrix
- $\mathbf{1}$ = vector of ones
- $w_{MVP}$ = weights of the MVP
- It gives you the **least risky portfolio**, but not necessarily the best returns.
- Often used as a benchmark to compare with other portfolios.

## 💵 What is a Bond?

- A **bond** is a type of loan made by an investor to a borrower (usually a company or government).
- The borrower promises to:
  - Pay **regular interest** (coupon payments).
  - Repay the **principal (face value)** at maturity.
- Bonds are considered **fixed income securities** because they provide predictable cash flows.

---

## 🔷 Key Characteristics

- **Face Value (Par Value)**: Amount repaid at maturity (e.g., $1,000).
- **Coupon Rate**: Annual interest paid to the investor (e.g., 5%).
- **Maturity Date**: Date when the face value is returned.
- **Issuer**: The borrower (government, corporation).
- **Price**: Market value of the bond, which may differ from face value.

# Zero-Coupon Bonds

## Definition

- A **Zero-Coupon Bond** pays **no periodic coupons**.
- The investor only receives **a single payment at maturity** = **Face Value (Par Value)**.
- They are sold at a **discount** today and redeemed at **full face value** in the future.

---

## Pricing Formula

The price is simply the **present value of the face value**:

$$P = \frac{FV}{(1+r)^T}$$

Where:

- $P$ = Price of the bond today
- $FV$ = Face Value (amount repaid at maturity)
- $r$ = discount rate (yield to maturity)
- $T$ = time to maturity (in years)

---

## 🔷 Example

- Face Value = **$1,000**
- Maturity = **5 years**
- Interest rate $r = 5\%$

$$P = \frac{1000}{(1+0.05)^5} = 783.53$$

👉 Investor pays **$783.53 today** and receives **$1,000 in 5 years**.

---

# Exercise – Zero-Coupon Bonds

You are provided with a CSV file called `zcbonds.csv`.

👉 Task:

1. Load the CSV file into Python.
2. Compute the **price** of each bond.
3. Add a new column `"Price"`.
4. Save the results into a new file `zcbonds_priced.csv`.
5. Print the final table.

```
df = pd.read_csv("zcbonds.csv")

# ---------------------------------------------------------
# 2) Convert columns to numeric types
# ---------------------------------------------------------
df['NominalValue']    = pd.to_numeric(df['NominalValue'], errors='coerce')
df['MaturityYears']   = pd.to_numeric(df['MaturityYears'], errors='coerce')
df['DiscountRatePct'] = pd.to_numeric(df['DiscountRatePct'], errors='coerce')

# ---------------------------------------------------------
# 3) Compute the bond price
# Formula: Price = NominalValue / (1 + DiscountRate)^MaturityYears
# ---------------------------------------------------------
df['Price'] = df.apply(
    lambda row: row['NominalValue'] / ((1 + row['DiscountRatePct']/100) ** row['MaturityYears']),
    axis=1
)

# ---------------------------------------------------------
# 4) Round the prices to 2 decimal places
# ---------------------------------------------------------
df['Price'] = df['Price'].round(2)

# ---------------------------------------------------------
# 5) Save the results to a new CSV
# ---------------------------------------------------------
df.to_csv("zcbonds_priced.csv", index=False)

# ---------------------------------------------------------
# 6) Display the final table
# ---------------------------------------------------------
print(df[['NominalValue', 'MaturityYears', 'DiscountRatePct', 'Price']])
```

$$P = \sum_{t=1}^{N} \frac{C}{(1+y)^t} + \frac{F}{(1+y)^N}$$

---

**Where:**

- $P$ = bond price
- $C$ = coupon payment each period = $\text{coupon rate} \times F$
- $F$ = face value (e.g., 100 or 1000)
- $y$ = yield to maturity (YTM) per period
- $N$ = number of periods until maturity

---

### Example

- Face Value $F = 100$
- Annual Coupon Rate = **5%** → Coupon $C = 5$
- Maturity = **3 years** ($N = 3$)
- Yield to Maturity (YTM) = **4%** ($y = 0.04$)

---

### Price Calculation

$$P = \frac{5}{(1.04)^1} + \frac{5}{(1.04)^2} + \frac{5}{(1.04)^3} + \frac{100}{(1.04)^3}$$

Step by step:

- Year 1 coupon: $\frac{5}{1.04} = 4.81$
- Year 2 coupon: $\frac{5}{(1.04)^2} = 4.62$
- Year 3 coupon: $\frac{5}{(1.04)^3} = 4.44$
- Face value: $\frac{100}{(1.04)^3} = 88.90$

---

### Final Price

$$P = 4.81 + 4.62 + 4.44 + 88.90 = 102.77$$

👉 **Task**:

1. Import the CSV file into a Pandas DataFrame.
2. For each bond, compute its **price** using the standard bond pricing formula:

$$P = \sum_{t=1}^{N} \frac{C}{(1+y)^t} + \frac{F}{(1+y)^N}$$

where:

- $F$ = Nominal Value
- $C$ = Coupon per period = $\frac{\text{Coupon Rate} \times F}{\text{Coupon Periodicity}}$
- $y$ = Discount Rate / Coupon Periodicity
- $N$ = Maturity × Coupon Periodicity

3. Create a **new column** in the DataFrame called `"Price"` that stores the computed price for each bond.

---

```python
df = pd.read_csv("bond.csv", sheet_name="Bond With Coupon")

# Function to compute bond price
def bond_price(row):
    F = row["Nominal Value"]
    r = row["Discount Rate (%)"] / 100
    T = row["Maturity (Years)"]
    freq = row["Coupon Periodicity"]
    c_rate = row["Coupon rate In Pourcentage"] / 100

    # Case 1: Zero-coupon bond
    if freq == 0:
        return F / ((1 + r) ** T)

    # Case 2: Coupon bond
    C = (c_rate * F) / freq       # Coupon per period
    y = r / freq                  # Discount rate per period
    N = int(T * freq)             # Total number of coupon payments

    # --- Step by step ---
    # Present value of coupons
    pv_coupons = 0
    for t in range(1, N + 1):
        pv_coupons += C / ((1 + y) ** t)

    # Present value of face value
    pv_face = F / ((1 + y) ** N)

    # Final price
    price = pv_coupons + pv_face
    return price

# Apply to DataFrame
df["Price"] = df.apply(bond_price, axis=1)
```

## 📌 What is Yield to Maturity (YTM)?

**Definition:**

- The **Yield to Maturity (YTM)** is the **annualized rate of return** an investor earns if they buy a bond today **at its current price** and hold it until maturity, assuming:
  1. All coupon payments are made on time.
  2. All coupons are **reinvested at the same rate** (the YTM).
  3. The bond is held until its maturity date.

---

### 🔹 Interpretation

- It is the **internal rate of return (IRR)** of a bond.
- It answers the question:

  👉 *"What constant discount rate makes the present value of all future cash flows (coupons + face value) equal to the bond's current market price?"*

### 🔹 Formula (implicit)

There is no closed-form formula for YTM (except for zero-coupon bonds).

It is the solution to:

$$P = \sum_{t=1}^{N} \frac{C}{(1+\text{YTM})^t} + \frac{F}{(1+\text{YTM})^N}$$

- $P$ = bond price (market price)
- $C$ = coupon payment per period
- $F$ = face value
- $N$ = number of periods

Here, **YTM is the unknown**.

---

## Yield to Maturity (YTM) – Approximation

### Zero-Coupon Bond

$$\mathrm{YTM} = \left(\frac{F}{P}\right)^{\frac{1}{N}} - 1$$

### Coupon Bond

$$\mathrm{YTM} \approx \frac{C + \frac{F-P}{N}}{\frac{F+P}{2}}$$

**Task:**

1. Import the dataset into a Pandas DataFrame.
2. For each bond, compute its **approximate Yield to Maturity (YTM)** using the formulas:

- **Zero-coupon bond (no coupons)**:

$$\text{YTM} = \left(\frac{F}{P}\right)^{\frac{1}{N}} - 1$$

- **Coupon bond**:

$$\text{YTM} \approx \frac{C + \frac{F-P}{N}}{\frac{F+P}{2}}$$

where:

- $F$ = Face Value
- $P$ = Price
- $N$ = Maturity – Current Year
- $C$ = Annual coupon payment = $\text{Coupon Rate} \times F$

3. Create a new column `"Yield to Maturity"` in the DataFrame storing the calculated YTM for each bond.

# Finding YTM Using Optimization

## 1 Concept

- Yield to Maturity (YTM) is **the discount rate that makes the present value of all future cash flows equal to the bond's price.**
- Mathematically:

$$P = \sum_{t=1}^{N} \frac{C_t}{(1+\text{YTM})^t} + \frac{F}{(1+\text{YTM})^N}$$

- Here, **YTM is unknown.**
- For coupon bonds, there's no closed-form solution → we need **numerical methods.**

---

## 2 Optimization Approach

- Treat the **difference between the present value and the market price** as an objective function:

$$f(y) = \left| P_{\text{market}} - \left( \sum_{t=1}^{N} \frac{C}{(1+y)^t} + \frac{F}{(1+y)^N} \right) \right|$$

- Goal: **find y (YTM) that minimizes** $f(y)$.

# SciPy Library (Scientific Python)

## 1️⃣ What is SciPy?

- **SciPy** is a **Python library for scientific and technical computing**.
- It builds on **NumPy** and provides **advanced mathematical functions**.
- Very useful in **optimization, integration, interpolation, linear algebra, statistics**, and more.

---

## 2️⃣ Key Submodules

1. `scipy.optimize` – optimization and root-finding
   - `minimize`, `minimize_scalar`, `root`, etc.
   - Solve equations, find minima/maxima, fit models.
2. `scipy.integrate` – numerical integration
   - `quad`, `dblquad`, `odeint` (for differential equations)
3. `scipy.stats` – statistics
   - Probability distributions, hypothesis tests, random variables
4. `scipy.linalg` – linear algebra
   - Matrices, eigenvalues, decompositions
5. `scipy.signal` – signal processing
   - Filters, convolution, Fourier transforms

---

## 3️⃣ Why use SciPy for finance?

- **Optimization**: finding YTM, portfolio optimization, calibration of models
- **Statistics**: compute distributions, risk measures, correlation, regression
- **Numerical methods**: solving equations that don't have closed-form solutions

## Understanding `scipy.optimize.minimize_scalar`

### 1️⃣ Purpose

- `minimize_scalar` is a function in Python that **finds the minimum value of a single-variable function**.
- You give it a **function** $f(x)$ and a **range of possible values**, and it finds the $x$ where $f(x)$ is smallest.

---

### 2️⃣ How it works

- You define a **scalar function**: it takes one input (a number) and returns one output (a number).
- You provide **bounds** for the input variable (optional, but recommended for bounded problems).
- The algorithm **searches for the value of the input** that minimizes the function output.

## 3 Syntax

python

Copy code

```python
from scipy.optimize import minimize_scalar

result = minimize_scalar(fun, bounds=(a, b), method='bounded')
```

- `fun` → the function to minimize
- `bounds=(a, b)` → lower and upper limits for the input
- `method='bounded'` → ensures the solution stays inside the bounds
- `result.x` → the input value that minimizes the function
- `result.fun` → the minimum function value at `result.x`

---

## 4 Key Points

- Works **only for single-variable functions**.
- The function should be **continuous** for best results.
- Can be used to **solve equations indirectly** by minimizing the difference between two expressions.
- Very useful for **optimization problems**, root approximations, or parameter estimation.

**Task:**

1. Import the dataset into a Pandas DataFrame.
2. For each bond, compute its **Yield to Maturity using optimization**:
   - Define an objective function as the **absolute difference** between the bond's present value (based on a guessed YTM) and its market price.
   - Use `scipy.optimize.minimize_scalar` to **find the YTM that minimizes this difference**.
3. Fill the column `"Yield to Maturity using optimization"` with the computed values.
4. Compare the **optimized YTM** with the **approximate YTM** in the previous column, and observe the differences.

---

```python
import pandas as pd
from scipy.optimize import minimize_scalar

# Load dataset
df = pd.read_csv("bonds.csv")  # make sure the CSV has the columns as in your table

# Function to compute approximate YTM
def ytm_approx(row, current_year=2023):
    F = row["Face Value (€)"]
    P = row["Price"]
    N = row["Maturity"] - current_year
    coupon_rate = row["Coupon Rate (%)"] / 100
    C = coupon_rate * F
    freq = row["Coupon Periodicity"]

    # Zero-coupon bond
    if freq == 0 or freq == '-':
        return (F / P) ** (1/N) - 1

    # Coupon bond approximation
    return (C + (F - P)/N) / ((F + P)/2)

# Compute approximate YTM column if not already done
df["Yield to Maturity"] = df.apply(ytm_approx, axis=1)
```

```python
# Function to compute optimized YTM
def ytm_optimized(row, current_year=2023):
    F = row["Face Value (€)"]
    P = row["Price"]
    N = row["Maturity"] - current_year
    coupon_rate = row["Coupon Rate (%)"] / 100
    freq = row["Coupon Periodicity"]

    # Zero-coupon bond
    if freq == 0 or freq == '-':
        return (F / P) ** (1/N) - 1

    # Annual coupon payment
    C = coupon_rate * F / (freq if freq != '-' else 1)
    periods = int(N * (freq if freq != '-' else 1))

    # Objective function: PV - Price
    def objective(y):
        pv_coupons = sum([C / ((1 + y)**t) for t in range(1, periods+1)])
        pv_face = F / ((1 + y)**periods)
        return abs(P - (pv_coupons + pv_face))

    result = minimize_scalar(objective, bounds=(0, 1), method='bounded')
    return result.x

# Compute optimized YTM column
df["Yield to Maturity using optimization"] = df.apply(ytm_optimized, axis=1)

# Comparison
df["Difference"] = df["Yield to Maturity using optimization"] - df["Yield to Maturity"]

# Display results
print(df[["ISIN", "Yield to Maturity", "Yield to Maturity using optimization", "Difference"]])
```