What is Pandas?

- A Python library built on top of NumPy.
- Designed for working with **tabular data** (rows & columns).
- Widely used in **finance, data science, and statistics.**
- Two main objects:
  - **Series** → 1D labeled array (like one column in Excel).
  - **DataFrame** → 2D labeled data structure (like an Excel sheet).

NumPy vs Pandas

Feature	NumPy Array	Pandas DataFrame
Type of data	Homogeneous (all same type)	Heterogeneous (different types)
Labels	Indexed only by numbers	Columns and row labels
Structure	Mostly numeric	Tabular (rows + columns)
Use case	Mathematical operations	Data analysis & manipulation

Creating a Pandas Series

```python
import pandas as pd

# From a Python list
s = pd.Series([100, 102, 98, 105, 110], name="Stock Prices")
print(s)
```

Output:

```yaml
0    100
1    102
2     98
3    105
4    110
Name: Stock Prices, dtype: int64
```

👉 Explanation:

- The left column is the **index** (default: 0,1,2...).
- The right column is the **values**.
- The `name` attribute labels the Series.

Creating a Pandas DataFrame

```python
data = {
    "Stock": ["AAPL", "GOOG", "AMZN", "MSFT", "TSLA"],
    "Price": [150, 2800, 3400, 299, 720],
    "Volume": [100000, 150000, 120000, 90000, 110000]
}

df = pd.DataFrame(data)
print(df)
```

Output:

```yaml
  Stock  Price  Volume
0  AAPL    150  100000
1  GOOG   2800  150000
2  AMZN   3400  120000
3  MSFT    299   90000
4  TSLA    720  110000
```

👉 Explanation:

- Each **column has a name** ("Stock", "Price", "Volume").
- Each **row has an index** (0 → 4).
- Looks similar to an Excel sheet.

📝 Exercise – Pandas Basics with Series & DataFrame

1. Create a **Pandas Series** of 5 daily stock returns: `[0.01, -0.005, 0.02, -0.01, 0.015]`.
   - Each value corresponds to the **return of the stock on that day**.
   - Example: Day 1 → +1%, Day 2 → -0.5%, etc.
2. Create a **DataFrame** with two columns:
   - `"Day"` → numbers from 1 to 5
   - `"Return"` → the Series above
3. Print both the **Series** and the **DataFrame**.
4. Add a new column `"Cumulative Return"` to the DataFrame that shows the **cumulative product of returns** over time.
   - This shows the total growth of the investment if you reinvest profits each day.

💡 Hint:

- You can use the Pandas function `.cumprod()` to calculate the cumulative product.
- Since returns are percentages, remember to add 1 before applying `.cumprod()`.
  Example: `df["Cumulative Return"] = (1 + df["Return"]).cumprod()`

✅ Correction – Pandas Basics with Cumulative Return

python

```python
import pandas as pd

# 1. Create a Series of stock returns
returns = pd.Series([0.01, -0.005, 0.02, -0.01, 0.015], name="Return")
print("Series of Returns:")
print(returns)

# 2. Create a DataFrame with Day and Return
data = {
    "Day": [1, 2, 3, 4, 5],
    "Return": returns
}
df = pd.DataFrame(data)

print("\nDataFrame of Returns:")
print(df)

# 3. Add Cumulative Return column
df["Cumulative Return"] = (1 + df["Return"]).cumprod()

print("\nDataFrame with Cumulative Return:")
print(df)
```

Series of Returns:

vbnet

```
0    0.010
1   -0.005
2    0.020
3   -0.010
4    0.015
Name: Return, dtype: float64
```

DataFrame with Cumulative Return:

sql

```
   Day  Return  Cumulative Return
0    1   0.010           1.010000
1    2  -0.005           1.004950
2    3   0.020           1.025049
3    4  -0.010           1.014798
4    5   0.015           1.030020
```

Topic 2: Indexing & Selection in Pandas

Why Indexing Matters in Pandas?

- The **index** allows you to access rows and columns efficiently.
- Think of it as the “address” of your data.
- Pandas offers two main methods:
    - `.loc[]` → label-based indexing (by names, not numbers).
    - `.iloc[]` → position-based indexing (by row/column number).

Example: Our DataFrame

```python
import pandas as pd

data = {
    "Stock": ["AAPL", "GOOG", "AMZN", "MSFT", "TSLA"],
    "Price": [150, 2800, 3400, 299, 720],
    "Volume": [100000, 150000, 120000, 90000, 110000]
}
df = pd.DataFrame(data)
print(df)
```

Output:

```yaml
  Stock  Price  Volume
0  AAPL    150  100000
1  GOOG   2800  150000
2  AMZN   3400  120000
3  MSFT    299   90000
4  TSLA    720  110000
```

Selecting Columns

python

```python
print(df["Stock"])        # Single column
print(df[["Stock", "Price"]])  # Multiple columns
```

Output:

yaml

```
0    AAPL
1    GOOG
2    AMZN
3    MSFT
4    TSLA
Name: Stock, dtype: object

   Stock  Price
0   AAPL    150
1   GOOG   2800
2   AMZN   3400
3   MSFT    299
4   TSLA    720
```

Row Selection with loc (by labels)

```python
print(df.loc[2])          # Row with index label 2
print(df.loc[1:3])        # Rows from 1 to 3 (inclusive)
```

Output:

```yaml
Stock      AMZN
Price      3400
Volume    120000
Name: 2, dtype: object

   Stock  Price  Volume
1   GOOG   2800  150000
2   AMZN   3400  120000
3   MSFT    299   90000
```

Row Selection with iloc (by position)

```python
print(df.iloc[0])         # First row
print(df.iloc[0:2])       # First two rows
```

Output:

```yaml
Stock      AAPL
Price       150
Volume   100000
Name: 0, dtype: object

   Stock  Price  Volume
0   AAPL    150  100000
1   GOOG   2800  150000
```

Boolean Indexing

```python
print(df[df["Price"] > 1000])   # All stocks with Price > 1000
```

Output:

```yaml
   Stock  Price  Volume
1   GOOG   2800  150000
2   AMZN   3400  120000
```

📝 Mini-Exercise – Indexing with Returns DataFrame

Use the previous DataFrame of daily returns (with columns `Day`, `Return`, `Cumulative Return`) from the first exercise.

sql Copier le code

```sql
   Day  Return  Cumulative Return
0    1   0.010           1.010000
1    2  -0.005           1.004950
2    3   0.020           1.025049
3    4  -0.010           1.014798
4    5   0.015           1.030020
```

Tasks:

1. Select only the `"Return"` column.
2. Select the row corresponding to **Day = 3** using `.loc[]`.
3. Select the first 3 rows using `.iloc[]`.
4. Select all rows where `"Return"` is strictly positive (> 0).

```python
# 1. Select only the "Return" column
returns_only = df["Return"]
print("Return column:")
print(returns_only)

# 2. Select the row corresponding to Day = 3 using loc
day3_row = df.loc[df["Day"] == 3]
print("\nRow where Day = 3:")
print(day3_row)

# 3. Select the first 3 rows using iloc
first_three_rows = df.iloc[0:3]
print("\nFirst 3 rows:")
print(first_three_rows)

# 4. Select all rows where Return > 0
positive_returns = df[df["Return"] > 0]
print("\nRows with positive Return:")
print(positive_returns)
```

Return column:

```vbnet
0    0.010
1   -0.005
2    0.020
3   -0.010
4    0.015
Name: Return, dtype: float64
```

Row where Day = 3:

```sql
   Day  Return  Cumulative Return
2    3   0.020           1.025049
```

First 3 rows:

```sql
   Day  Return  Cumulative Return
0    1   0.010           1.010000
1    2  -0.005           1.004950
2    3   0.020           1.025049
```

Rows with positive Return:

```sql
   Day  Return  Cumulative Return
0    1   0.010           1.010000
2    3   0.020           1.025049
4    5   0.015           1.030020
```

Topic 3: Operations on DataFrame

---

1. Adding and Modifying Columns

- You can create new columns based on existing data.
- Examples in finance: daily returns, cumulative returns, log returns, rolling volatility, etc.
- Syntax:

python Copier le code

```python
df["NewColumn"] = ...
```

Example – Adding Rolling Volatility (3-day window)

python Copier le code

```python
import pandas as pd
import numpy as np

# Assume df is the existing DataFrame with Day, Return, Cumulative Return
df["Rolling Volatility"] = df["Return"].rolling(window=3).std()
print(df)
```

Output:

pgsql Copier le code

```pgsql
   Day  Return  Cumulative Return  Rolling Volatility
0    1   0.010           1.010000                 NaN
1    2  -0.005           1.004950                 NaN
2    3   0.020           1.025049            0.012583
3    4  -0.010           1.014798            0.015275
4    5   0.015           1.030020            0.015275
```

> 💡 Note: First two rows are NaN because a 3-day window is required to calculate the standard deviation.

2. Using Built-in Aggregation Functions

- Pandas provides functions like `.mean()`, `.sum()`, `.min()`, `.max()`, `.std()` for quick statistics.

python

Copie

```python
mean_return = df["Return"].mean()
sum_return = df["Return"].sum()
min_return = df["Return"].min()
max_return = df["Return"].max()
std_return = df["Return"].std()

print(f"Mean Return: {mean_return}")
print(f"Sum of Returns: {sum_return}")
print(f"Min Return: {min_return}")
print(f"Max Return: {max_return}")
print(f"Std of Return: {std_return}")
```

Output:

mathematica

Copie

```mathematica
Mean Return: 0.006
Sum of Returns: 0.03
Min Return: -0.01
Max Return: 0.02
Std of Return: 0.012247448713915889
```

3. Using `apply()` for Custom Operations

- Apply a function to a column (or row) using `.apply()`.
- Example: calculate **squared returns**:

```python
df["Squared Return"] = df["Return"].apply(lambda x: x**2)
print(df)
```

Output:

```pgsql
   Day  Return  Cumulative Return  Rolling Volatility  Squared Return
0    1   0.010           1.010000                 NaN        0.000100
1    2  -0.005           1.004950                 NaN        0.000025
2    3   0.020           1.025049            0.012583        0.000400
3    4  -0.010           1.014798            0.015275        0.000100
4    5   0.015           1.030020            0.015275        0.000225
```

Mini-Exercise – Operations Practice

Using the **existing DataFrame** (`Day`, `Return`, `Cumulative Return`):

1. Add a column `"Absolute Return"` → absolute value of daily return.
2. Add a column `"Rolling Volatility"` → 3-day rolling standard deviation.
3. Add a column `"Return Cubed"` → cube of daily returns using `apply()`.
4. Compute **mean, sum, min, max, std** of `"Return"`.

> 💡 Hint:
>
> - "Absolute values: `abs()`"
> - "Rolling std: `.rolling(window=3).std()`"
> - "Cube: `apply(lambda x: x**3)`"

```python
import pandas as pd

# Assume df is the existing DataFrame with Day, Return, Cumulative Return

# 1. Add Absolute Return column
df["Absolute Return"] = df["Return"].abs()

# 2. Add Rolling Volatility (3-day window)
df["Rolling Volatility"] = df["Return"].rolling(window=3).std()

# 3. Add Return Cubed column using apply()
df["Return Cubed"] = df["Return"].apply(lambda x: x**3)

# 4. Compute basic statistics on Return
mean_return = df["Return"].mean()
sum_return = df["Return"].sum()
min_return = df["Return"].min()
max_return = df["Return"].max()
std_return = df["Return"].std()

# Display DataFrame
print("Updated DataFrame:")
print(df)

# Display statistics
print("\nStatistics on Return:")
print(f"Mean Return: {mean_return}")
print(f"Sum of Returns: {sum_return}")
print(f"Min Return: {min_return}")
print(f"Max Return: {max_return}")
print(f"Std of Return: {std_return}")
```

🔎 Output

Updated DataFrame:

```pgsql
   Day  Return  Cumulative Return  Absolute Return  Rolling Volatility  Return Cubed
0    1   0.010           1.010000            0.010                 NaN   0.000001
1    2  -0.005           1.004950            0.005                 NaN  -0.000000125
2    3   0.020           1.025049            0.020            0.012583    0.000008
3    4  -0.010           1.014798            0.010            0.015275   -0.000001
4    5   0.015           1.030020            0.015            0.015275    0.000003375
```

Statistics on Return:

```mathematica
Mean Return: 0.006
Sum of Returns: 0.03
Min Return: -0.01
Max Return: 0.02
Std of Return: 0.012247448713915890
```

Topic 5: Handling Missing Data & Cleaning (Using Existing DataFrame)

---

1. Display Existing DataFrame (Context Reminder)

python

Copier le code

```python
import pandas as pd
import numpy as np

# Assume 'df' is the existing DataFrame with all columns
print("Existing DataFrame:")
print(df)
```

Example Output:

pgsql

Copier le code

```
   Day  Return  Cumulative Return  Absolute Return  Rolling Volatility  Squared Return  Return Cub
0    1   0.010           1.010000            0.010                 NaN        0.000100    0.000001
1    2  -0.005           1.004950            0.005                 NaN        0.000025   -0.000000
2    3   0.020           1.025049            0.020            0.012583        0.000400    0.000008
3    4  -0.010           1.014798            0.010            0.015275        0.000100   -0.000001
4    5   0.015           1.030020            0.015            0.015275        0.000225    0.000003
```

> 💡 Note: The `Rolling Volatility` column contains `NaN` in the first rows due to the 3-day rolling window.

2. Detect Missing Values

python Copier le code

```python
# Detect NaN
print("\nDetect missing values (bool):")
print(df.isna())

# Count missing values per column
print("\nCount of missing values per column:")
print(df.isna().sum())
```

Output:

mathematica Copier le code

```
Detect missing values (bool):
     Day  Return  Cumulative Return  Absolute Return  Rolling Volatility  Squared Return  Return C
0  False   False              False            False                True           False       Fal
1  False   False              False            False                True           False       Fal
2  False   False              False            False               False           False       Fal
3  False   False              False            False               False           False       Fal
4  False   False              False            False               False           False       Fal

Count of missing values per column:
Day                   0
Return                0
Cumulative Return     0
Absolute Return       0
Rolling Volatility    2
Squared Return        0
Return Cubed          0
dtype: int64
```

3. Filling Missing Values

- Fill NaN in `Rolling Volatility` with `0` (or column mean if preferred):

python Copier le code

```python
df["Rolling Volatility"] = df["Rolling Volatility"].fillna(0)
print("\nDataFrame after filling NaN in Rolling Volatility:")
print(df)
```

Output:

sql Copier le code

```sql
   Day  Return  Cumulative Return  Absolute Return  Rolling Volatility  Squared Return  Return Cub
0    1   0.010           1.010000            0.010               0.000        0.000100     0.00000
1    2  -0.005           1.004950            0.005               0.000        0.000025    -0.00000
2    3   0.020           1.025049            0.020            0.012583       0.000400     0.00000
3    4  -0.010           1.014798            0.010            0.015275       0.000100    -0.000001
4    5   0.015           1.030020            0.015            0.015275       0.000225     0.00000
```

4. Dropping Missing Values

- If there were NaN in critical columns (e.g., `Return` or `Cumulative Return`), drop them:

```python
df_clean = df.dropna(subset=["Return", "Cumulative Return"])
print("\nCleaned DataFrame (after dropping critical NaN):")
print(df_clean)
```

> Ici, aucune ligne n'est supprimée car `Return` et `Cumulative Return` n'ont pas de NaN.

Mini-Exercise – Cleaning & Log Return Analysis

Context:

We have the existing DataFrame `df` with:

`Day`, `Return`, `Cumulative Return`, `Absolute Return`, `Rolling Volatility`, `Squared Return`, `Return Cubed`.

- Some values in `Rolling Volatility` are `NaN`.

Tasks:

1. Detect missing values per column.
2. Fill missing values in `Rolling Volatility` with the mean of the column.
3. Create a new column `Log Return` = `np.log(1 + Return)` using NumPy.
4. Identify the day with the maximum Log Return.
5. Compute mean and standard deviation of Log Return.
6. Identify the day with maximum Absolute Return.

> 💡 Hint:
>
> - " `.isna()`, `.sum()` → detect missing values"
> - " `.fillna(value=df["Rolling Volatility"].mean())` → fill NaN"
> - " `np.log(1 + df["Return"])` → log return"
> - " `.idxmax()` → find row of max value"
> - " `.mean()`, `.std()` → statistics"

```python
import numpy as np
import pandas as pd

# Assume 'df' is the existing DataFrame
print("Original DataFrame:")
print(df)

# 1. Detect missing values per column
print("\nMissing values per column:")
print(df.isna().sum())

# 2. Fill missing Rolling Volatility with column mean
rolling_mean = df["Rolling Volatility"].mean()
df["Rolling Volatility"] = df["Rolling Volatility"].fillna(rolling_mean)

# 3. Create Log Return column
df["Log Return"] = np.log(1 + df["Return"])

# 4. Identify the day with maximum Log Return
max_log_return_day = df["Log Return"].idxmax()

# 5. Compute mean and standard deviation of Log Return
mean_log_return = df["Log Return"].mean()
std_log_return = df["Log Return"].std()

# 6. Identify the day with maximum Absolute Return
max_abs_return_day = df["Absolute Return"].idxmax()

# 7. Display final DataFrame
print("\nCleaned DataFrame with Log Return:")
print(df)

# 8. Display results
print(f"\nDay with maximum Log Return: {df.loc[max_log_return_day, 'Day']}")
print(f"Mean of Log Return: {mean_log_return}")
print(f"Std of Log Return: {std_log_return}")
print(f"Day with maximum Absolute Return: {df.loc[max_abs_return_day, 'Day']}")
```

🔎 Example Output

Original DataFrame (with NaN in Rolling Volatility):

```pgsql
   Day  Return  Cumulative Return  Absolute Return  Rolling Volatility  Squared Return  Return Cub
0    1   0.010           1.010000            0.010                 NaN        0.000100    0.000001
1    2  -0.005           1.004950            0.005                 NaN        0.000025   -0.000000
2    3   0.020           1.025049            0.020            0.012583        0.000400    0.000008
3    4  -0.010           1.014798            0.010            0.015275        0.000100   -0.000001
4    5   0.015           1.030020            0.015            0.015275        0.000225    0.000003
```

Missing values per column:

```sql
Day                   0
Return                0
Cumulative Return     0
Absolute Return       0
Rolling Volatility    2
Squared Return        0
Return Cubed          0
dtype: int64
```

Cleaned DataFrame with Log Return:

mathematica

Copier le code

```
   Day  Return  Cumulative Return  Absolute Return  Rolling Volatility  Squared Return  Return Cub
0    1   0.010           1.010000            0.010            0.014378        0.000100    0.000001
1    2  -0.005           1.004950            0.005            0.014378        0.000025   -0.000000
2    3   0.020           1.025049            0.020            0.012583        0.000400    0.000008
3    4  -0.010           1.014798            0.010            0.015275        0.000100   -0.000001
4    5   0.015           1.030020            0.015            0.015275        0.000225    0.000003
```

Results:

mathematica

Copier le code

```
Day with maximum Log Return: 3
Mean of Log Return: 0.009918
Std of Log Return: 0.011231
Day with maximum Absolute Return: 3
```

Topic 6: Grouping & Merge in Pandas

---

1. Grouping Data

- `groupby()` allows you to aggregate data by one or more columns.
- Common operations: `mean()`, `sum()`, `count()`, `std()`.
- Financial examples:
  - Average return per stock
  - Total volume traded per sector

```python
import pandas as pd

# Sample DataFrame
data = {
    "Stock": ["A", "A", "A", "B", "B", "B"],
    "Day": [1, 2, 3, 1, 2, 3],
    "Return": [0.01, -0.005, 0.02, 0.015, -0.01, 0.005]
}
df = pd.DataFrame(data)

# Group by Stock and compute mean return
grouped = df.groupby("Stock")["Return"].mean()
print("Mean Return by Stock:")
print(grouped)
```

Output:

```css
Stock
A    0.008333
B    0.003333
Name: Return, dtype: float64
```

Explanation:

- `df.groupby("Stock")` → splits the data into groups by Stock.
- `["Return"].mean()` → computes the mean of `Return` for each group.

Merge in Pandas – Detailed Explanation

---

1. What is Merge?

- `merge()` allows you to combine two DataFrames based on one or more columns (keys).
- Similar to **SQL joins**: inner, left, right, outer.
- **Key idea:** you define which column(s) to match and what kind of join you want.

2. Example 1: Inner Join

- **Inner join** keeps only rows that **exist in both DataFrames.**

```python
import pandas as pd

# Prices DataFrame
df_prices = pd.DataFrame({
    "Stock": ["A", "B", "C"],
    "Price": [100, 200, 300]
})

# Sectors DataFrame
df_sector = pd.DataFrame({
    "Stock": ["A", "B", "D"],
    "Sector": ["Tech", "Finance", "Healthcare"]
})

# Inner merge
df_inner = pd.merge(df_prices, df_sector, on="Stock", how="inner")
print(df_inner)
```

Output:

```css
  Stock  Price   Sector
0     A    100     Tech
1     B    200  Finance
```

Explanation:

- Only `A` and `B` exist in **both DataFrames**, so only these rows appear.
- `C` is in `df_prices` only → ignored.
- `D` is in `df_sector` only → ignored.

3. Example 2: Left Join

- Left join keeps all rows from the left DataFrame, missing matches from the right → `NaN`.

```python
df_left = pd.merge(df_prices, df_sector, on="Stock", how="left")
print(df_left)
```

Output:

```css
  Stock  Price   Sector
0     A    100     Tech
1     B    200  Finance
2     C    300      NaN
```

Explanation:

- All rows from `df_prices` are kept.
- Stock `C` has no corresponding sector → Sector = NaN.

---

4. Example 3: Right Join

- Right join keeps all rows from the right DataFrame, missing matches from left → `NaN`.

```python
df_right = pd.merge(df_prices, df_sector, on="Stock", how="right")
print(df_right)
```

Output:

```css
  Stock  Price    Sector
0     A  100.0      Tech
1     B  200.0   Finance
2     D    NaN  Healthcare
```

Explanation:

- All rows from `df_sector` are kept.
- Stock `D` is not in `df_prices` → Price = NaN.

5. Example 4: Outer Join

- Outer join keeps all rows from both DataFrames, missing matches → `NaN`.

```python
df_outer = pd.merge(df_prices, df_sector, on="Stock", how="outer")
print(df_outer)
```

Output:

```css
  Stock  Price      Sector
0     A  100.0        Tech
1     B  200.0     Finance
2     C  300.0         NaN
3     D    NaN  Healthcare
```

Explanation:

- Includes all stocks from both DataFrames.
- Missing values are filled with `NaN`.

7. Summary Table

Join Type	Keeps Rows From	Missing Matches
inner	Both DataFrames	Dropped
left	Left DataFrame	NaN
right	Right DataFrame	NaN
outer	Both DataFrames	NaN

Opening Files in Pandas – CSV & Excel

---

1. Reading a CSV File

- CSV = **Comma-Separated Values**, very common for datasets.
- Use `pd.read_csv()` to load into a DataFrame.

Example:

```python
import pandas as pd

# Read CSV file
df_csv = pd.read_csv("data.csv")

# Show first 5 rows
print(df_csv.head())
```

Explanation:

- `"data.csv"` → path to your CSV file.
- `.head()` → displays the first 5 rows.
- Optional parameters:
  - `sep=";"` → if values are separated by `;` instead of `,`
  - `index_col=0` → set first column as index
  - `parse_dates=["Date"]` → automatically convert Date column to datetime

2. Reading an Excel File

- Excel files can have multiple sheets.
- Use `pd.read_excel()` to load.

Example:

```python
# Read Excel file, sheet named "Prices"
df_excel = pd.read_excel("data.xlsx", sheet_name="Prices")

# Show first 5 rows
print(df_excel.head())
```

Explanation:

- `"data.xlsx"` → path to your Excel file.
- `sheet_name="Prices"` → choose which sheet to read.
- Optional parameters:
  - `index_col=0` → set first column as index
  - `parse_dates=["Date"]` → convert Date column to datetime

Function	Description	Example Code	Example Output

`pd.read_csv()`	Read a CSV file into a DataFrame	`df = pd.read_csv("data.csv")`	`Date Stock Return`
			`0 2025-01-01 A 0.01`
			`1 2025-01-02 A 0.02`

`pd.read_excel()`	Read an Excel file	`df = pd.read_excel("data.xlsx", sheet_name="Prices")`	`Date Stock Price`
			`0 2025-01-01 A 100`
			`1 2025-01-02 A 101`

`df.head(n)`	Display first n rows	`df.head(3)`	`Date Stock Return`
			`0 2025-01-01 A 0.01`
			`1 2025-01-02 A 0.02`
			`2 2025-01-03 B -0.01`

`df.tail(n)`	Display last n rows	`df.tail(2)`	`Date Stock Return`
			`3 2025-01-04 B 0.015`
			`4 2025-01-05 A -0.005`

`df.info()`	Summary of DataFrame	`df.info()`	`<class 'pandas.DataFrame'>...` `RangeIndex: 5 entries, 0 to 4`
`df.describe()`	Summary statistics of numeric columns	`df.describe()`	`Return count 5 mean 0.005 std 0.012 min -0.01 max 0.02`
`df.isna()` / `df.isna().sum()`	Detect missing values	`df.isna().sum()`	`Date 0 Stock 0 Return 1`
`df.fillna(value)`	Fill missing values	`df["Return"].fillna(0, inplace=True)`	`Return 0.01 0.02 0.00 0.015 -0.005`

`df.groupby()`	Group data by column(s)	`df.groupby("Stock")["Return"].mean()`	`Stock A 0.008333 B 0.003333`
`df.merge()`	Merge two DataFrames	`pd.merge(df1, df2, on="Stock", how="left")`	`Stock Price Sector A 100 Tech B 200 Finance C 300 NaN`

`df.sort_values()`	Sort DataFrame by column(s)	`df.sort_values("Return", ascending=False)`	`Stock Return`
			`2 A 0.02`
			`0 A 0.01`

`df.set_index()`	Set a column as index	`df.set_index("Date", inplace=True)`	Index = Date column
`df.reset_index()`	Reset index to default	`df.reset_index(inplace=True)`	Index = 0,1,2,...
`df.apply()`	Apply a function to column/row	`df["Return"].apply(lambda x: x**2)`	`Return 0 0.0001 1 0.0004 2 0.000225`
`df.loc[]` / `df.iloc[]`	Indexing by label / position	`df.loc[0, "Return"]` → 0.01	`0.01`
`df.cumsum()`	Cumulative sum	`df["Return"].cumsum()`	`0 0.01 1 0.03 2 0.02`
`df.shift()`	Shift data for lag/lead	`df["Return"].shift(1)`	`0 NaN 1 0.01 2 0.02`
`df.resample()`	Resample time series	`df.resample("W").mean()`	Weekly average return

Exercise – Portfolio Statistical Analysis with Equal Weights

Dataset:

`Date`, `CAC 40`, `DAX`, `FTSE`, `S&P`, `IBOVESPA`, `IGPA`, `KOSPI`, `US T-Bill`, `SSE`

Tasks

Step 1: Load & Prepare Data

1. Load the CSV/Excel file into a Pandas DataFrame.
2. Convert the `Date` column to **datetime** format.
3. Set `Date` as the **index** of the DataFrame.
4. Ensure all numeric columns are **floats** (replace commas `,` with dots `.` if needed).
5. Fill any **missing values** with the **mean of the corresponding column.**

Step 2: Compute Log Returns

1. Calculate **daily log returns** for each index:

$$r_t = \ln\left(\frac{P_t}{P_{t-1}}\right)$$

2. Calculate **cumulative log returns** for each index:

$$R_{cum} = \sum r_t$$

> 💡 Hint: Use `np.log()` and `.shift(1)`

Step 3: Asset Statistics

1. For each index, compute:
   - **Mean daily log return**
   - **Standard deviation** (volatility)
   - **Minimum daily return**
   - **Maximum daily return**
   - **Sharpe ratio** (assume risk-free rate = 0):

$$\text{Sharpe} = \frac{\text{Mean}}{\text{Std}}$$

2. Identify the **most attractive asset** in terms of risk-adjusted return (highest Sharpe ratio).
3. Identify the **most volatile asset** (highest standard deviation).

Step 4: Portfolio with Equal Weights

1. Assume a portfolio including all indices with **equal weights.**
2. Compute the **portfolio daily log return:**

$$R_{p,t} = \sum_i w_i r_{i,t}$$

3. Compute the **cumulative portfolio log return.**
4. Compute **portfolio statistics:** mean, standard deviation, min, max, Sharpe ratio.

Step 5: Insights

- Compare individual assets' Sharpe ratios with the portfolio Sharpe ratio.
- Discuss the **diversification benefits** of combining multiple assets.
- Comment on which indices contribute most to portfolio volatility.

> 💡 Hint: Use `.mean()`, `.std()`, `.min()`, `.max()`, `.cumsum()`, and `np.dot(weights, log_returns.T)`

Solution – Portfolio Statistical Analysis with Equal Weights

python

Copier le code

```python
import pandas as pd
import numpy as np

# --- Step 1: Load & Prepare Data ---
df = pd.read_csv("indices.csv", parse_dates=["Date"], dayfirst=True)
df.set_index("Date", inplace=True)

# Convert numeric columns to float
df = df.apply(lambda x: x.str.replace(',', '.').astype(float) if x.dtype=='object' else x)

# Fill missing values with column mean
df.fillna(df.mean(), inplace=True)

print("DataFrame head:\n", df.head())
```

Output (simulated):

yaml

Copier le code

```yaml
            CAC 40       DAX     FTSE      S&P  IBOVESPA      IGPA    KOSPI  US T-Bill      SSE
Date
2016-10-28  4548.58  10526.16  6845.4  2088.66     64158  20792.95  1974.40     216.40  3104.27
2016-10-31  4509.26  10370.93  6790.5  2085.18     63258  20587.41  1967.53     216.40  3100.49
2016-11-01  4470.28  10325.88  6693.3  2131.52     61201  20801.85  1979.65     216.39  3122.44
2016-11-02  4414.67  10259.13  6806.9  2139.56     59184  20915.76  1980.55     216.39  3102.73
2016-11-03  4411.68  10456.95  6843.1  2163.26     59657  20939.45  1974.58     216.39  3128.94
```

Step 2: Daily Log Returns

python

Copier le code

```python
log_returns = np.log(df / df.shift(1))
log_returns = log_returns.dropna()

cum_log_returns = log_returns.cumsum()

print("Daily Log Returns:\n", log_returns.head())
print("\nCumulative Log Returns:\n", cum_log_returns.head())
```

Output (simulated):

yaml

Copier le code

```
Daily Log Returns:
               CAC 40       DAX      FTSE       S&P  IBOVESPA      IGPA     KOSPI  US T-Bill
Date
2016-10-31 -0.00864 -0.01468 -0.00806 -0.00172 -0.01405 -0.00992 -0.00348  0.00000 -0.00123
2016-11-01 -0.00865 -0.00432 -0.01478  0.02203 -0.03333  0.00687  0.00613 -0.00005  0.00715
2016-11-02 -0.01295 -0.00648  0.01691  0.00377 -0.03454  0.00545  0.00045 -0.00005 -0.00638
2016-11-03 -0.00068  0.01915  0.00543  0.00355  0.00804  0.00114 -0.00298  0.00000  0.00850
2016-11-04 -0.00769  0.00245  0.01001  0.00191  0.01809 -0.00018 -0.00440  0.00005 -0.00119

Cumulative Log Returns:
               CAC 40       DAX      FTSE       S&P  IBOVESPA      IGPA     KOSPI  US T-Bill
Date
2016-10-31 -0.00864 -0.01468 -0.00806 -0.00172 -0.01405 -0.00992 -0.00348  0.00000 -0.00123
2016-11-01 -0.01729 -0.01899 -0.02284  0.02031 -0.04738 -0.00305  0.00265 -0.00005  0.00592
2016-11-02 -0.03024 -0.02547 -0.00593  0.02408 -0.08191  0.00240  0.00310 -0.00010 -0.00046
2016-11-03 -0.03092 -0.00632  0.00050  0.02763 -0.07387  0.00354  0.00012  0.00010  0.00804
2016-11-04 -0.03861 -0.00387  0.01051  0.02954 -0.05578  0.00336 -0.00428  0.00015  0.00685
```

Step 3: Asset Statistics

python

```python
stats_assets = pd.DataFrame(index=df.columns)
stats_assets["Mean"] = log_returns.mean()
stats_assets["Std"] = log_returns.std()
stats_assets["Min"] = log_returns.min()
stats_assets["Max"] = log_returns.max()
stats_assets["Sharpe"] = stats_assets["Mean"] / stats_assets["Std"]

print("\nStatistics for Each Asset:\n", stats_assets)
```

Output (simulated):

mathematica

```mathematica
            Mean       Std      Min      Max    Sharpe
CAC 40   -0.00345  0.0123 -0.0256  0.0187  -0.280
DAX      -0.00175  0.0118 -0.0221  0.0192  -0.148
FTSE     -0.00095  0.0105 -0.0182  0.0168  -0.090
S&P       0.00214  0.0098 -0.0153  0.0203   0.218
IBOVESPA -0.00223  0.0182 -0.0353  0.0301  -0.123
IGPA      0.00105  0.0109 -0.0186  0.0200   0.096
KOSPI     0.00042  0.0112 -0.0195  0.0184   0.037
US T-Bill 0.00000  0.0000  0.0000  0.0000   NaN
SSE       0.00145  0.0120 -0.0205  0.0221   0.121
```

Step 4: Equal-Weight Portfolio

```python
n_assets = len(log_returns.columns)
weights = np.array([1/n_assets]*n_assets)

portfolio_return = log_returns.dot(weights)

portfolio_stats = {
    "Mean": portfolio_return.mean(),
    "Std": portfolio_return.std(),
    "Min": portfolio_return.min(),
    "Max": portfolio_return.max(),
    "Sharpe": portfolio_return.mean() / portfolio_return.std()
}

print("\nPortfolio Statistics (Equal Weights):\n", portfolio_stats)
```

Output (simulated):

```csharp
Portfolio Statistics (Equal Weights):
{'Mean': 0.00054, 'Std': 0.0085, 'Min': -0.0182, 'Max': 0.0167, 'Sharpe': 0.064}
```