📘 Session 2 – NumPy, Financial Calculations & yfinance

Part 1 – Introduction to NumPy

Explanation

- **NumPy** (Numerical Python) is the backbone of numerical computing in Python.
- In finance, we often deal with **large datasets**: thousands of stock prices, long time series of returns, or matrices of correlations.
- A standard Python **list** is flexible, but slow when working with numerical data because each element is stored as an object.
- A **NumPy array** is stored in contiguous memory and supports **vectorized operations** (performing calculations on entire arrays at once without loops).
- This makes NumPy much faster and closer to **MATLAB or R performance**.

Key advantages of NumPy in finance:

1. **Efficiency** → Handles millions of price points quickly.
2. **Vectorization** → Allows clean formulas like `returns = (prices[1:] - prices[:-1]) / prices[:-1]`.
3. **Linear Algebra** → Used for covariance, optimization, portfolio weights.

Example – Create Arrays

python Copy code

```python
import numpy as np

prices = np.array([100, 102, 105, 103, 108])
print("Prices:", prices)
print("Type:", type(prices))
```

Result

vbnet Copy code

```vbnet
Prices: [100 102 105 103 108]
Type: <class 'numpy.ndarray'>
```

Comparison: Python List vs NumPy Array

Explication détaillée :

- Lists: flexible, can contain mixed types, but **slow for large calculations**.
- NumPy: homogeneous type, fast, vectorized, supports broadcasting.
- **Finance example:** summing 1 million stock prices or computing portfolio returns.

Code + Result:

python Copy code

```python
import time
lst = list(range(1000000))
arr = np.arange(1000000)
start = time.time(); sum(lst); print("List sum time:", time.time()-start)
start = time.time(); arr.sum(); print("NumPy sum time:", time.time()-start)
```

bash Copy code

```bash
List sum time: 0.05
NumPy sum time: 0.002
```

Creating Arrays – Step by Step

Explication très détaillée :

1. `np.array(list)`
   - Converts a Python list to a NumPy array.
   - Useful to store historical stock prices or returns.

```python
a = np.array([10,20,30])
print("Array from list:", a)  # [10 20 30]
```

2. `np.zeros(shape)`
   - Creates a matrix of zeros.
   - Useful to initialize portfolio weights before assigning values.

```python
z = np.zeros((2,3))
print("Zeros array:\n", z)
# [[0. 0. 0.]
#  [0. 0. 0.]]
```

Function	Syntax	Description	Example
`np.array`	`np.array([1,2,3])`	Convert a Python list to a NumPy array	`np.array([10,20,30]) → [10 20 30]`
`np.zeros`	`np.zeros((2,3))`	Create an array filled with zeros	`np.zeros((2,3)) → [ [0. 0. 0.] [0. 0. 0.] ]`
`np.ones`	`np.ones(4)`	Create an array filled with ones	`np.ones(4) → [1. 1. 1. 1.]`
`np.arange`	`np.arange(0,10,2)`	Create evenly spaced values	`np.arange(0,10,2) → [0 2 4 6 8]`
`np.linspace`	`np.linspace(0,1,5)`	Create fixed number of evenly spaced values	`np.linspace(0,1,5) → [0. 0.25 0.5 0.75 1.]`

`np.eye`	`np.eye(3)`	Create identity matrix	`np.eye(3) → [ [1. 0. 0.] [0. 1. 0.] [0. 0. 1.] ]`
`np.random.rand`	`np.random.rand(2,2)`	Uniform random numbers [0,1]	`np.random.rand(2,2) → [ [0.3745 0.9507] [0.7319 0.5986] ]`
`np.random.randn`	`np.random.randn(5)`	Standard normal random numbers	`np.random.randn(5) → [0.4967 -0.1382 0.6476 1.5230 -0.2341]`
`np.full`	`np.full((2,2),7)`	Create array filled with specific value	`np.full((2,2),7) → [ [7 7] [7 7] ]`
`np.empty`	`np.empty((2,2))`	Create uninitialized array	`np.empty((2,2)) → [ [1. 0.] [0. 1.] ]`

NumPy Functions Table – Finance Context

Function	Description	Syntax / Example	Output / Explanation
`np.array()`	Create a NumPy array	`prices = np.array([100,102,105])`	`[100 102 105]`
`np.mean()`	Compute mean (average)	`np.mean(returns)`	Average return of series
`np.std()`	Compute standard deviation	`np.std(returns)`	Volatility of returns
`np.var()`	Compute variance	`np.var(returns)`	Risk measure, variance = std^2
`np.sum()`	Sum all elements	`np.sum(prices)`	Total of prices or returns
`np.min()`	Minimum value	`np.min(returns)`	Worst return
`np.max()`	Maximum value	`np.max(returns)`	Best return
`np.cov()`	Covariance matrix	`np.cov(returns1, returns2)`	Covariance between assets
`np.corrcoef()`	Correlation matrix	`np.corrcoef(returns1, returns2)`	Correlation (-1 to 1)
`np.cumprod()`	Cumulative product	`np.cumprod(1 + returns)`	Simulate price path from returns

		`returns)`	returns
`np.diff()`	Difference between consecutive elements	`np.diff(prices)`	`[102-100, 105-102]`
`np.round()`	Round numbers	`np.round(returns, 3)`	`[0.020, 0.029, -0.019]`
`np.arange()`	Create array with steps	`np.arange(0, 10, 2)`	`[0,2,4,6,8]`
`np.linspace()`	Create evenly spaced array	`np.linspace(0,1,5)`	`[0., 0.25, 0.5, 0.75, 1.]`
`np.random.normal()`	Generate random numbers (normal)	`np.random.normal(0.01,0.05,10)`	Simulate returns or shocks
`np.random.rand()`	Uniform random numbers	`np.random.rand(5)`	Random weights for portfolio

Correction

python

Copy code

```python
prices = np.array([100, 101, 103, 104, 106, 105, 107, 110, 108, 111])
print("Average price:", np.mean(prices))
print("Volatility:", np.std(prices))
```

Correction Result

yaml

Copy code

```yaml
Average price: 105.5
Volatility: 3.24
```

Broadcasting

Explication détaillée :

- Allows arithmetic between arrays of different shapes without loops.
- Finance: apply fees, exchange rates, or scale returns across multiple assets.

python Copy code

```python
x = np.array([1,2,3])
print("Add 10:", x+10)          # [11 12 13]
print("Multiply by vector:", x*[2,3,4])  # [2 6 12]
```

Exercise – Daily Returns with NumPy Array (No Vectorization)

Task:

1. Create a **NumPy array** of daily stock prices over 10 days: `[100, 102, 101, 105, 107, 106, 108, 110, 109, 111]`.
2. Using a **for loop**, compute the **daily returns**: $\mathrm{return}_t = \frac{P_t - P_{t-1}}{P_{t-1}}$.
3. Store the returns in a **NumPy array**.
4. Compute the **mean return** and **standard deviation** manually using **loops**, not NumPy vectorized functions.

```python
import numpy as np

# Step 1: Prices as NumPy array
prices = np.array([100, 102, 101, 105, 107, 106, 108, 110, 109, 111])
print("Prices:", prices)

# Step 2: Initialize returns array
returns = np.zeros(len(prices)-1)

# Step 3: Compute daily returns using a for loop
for i in range(1, len(prices)):
    returns[i-1] = (prices[i] - prices[i-1]) / prices[i-1]
print("Daily returns:", returns)

# Step 4: Compute mean manually
mean_return = 0
for r in returns:
    mean_return += r
mean_return /= len(returns)
print("Mean return:", mean_return)

# Step 5: Compute standard deviation manually
variance = 0
for r in returns:
    variance += (r - mean_return)**2
variance /= len(returns)
std_dev = variance**0.5
print("Standard deviation:", std_dev)
```

```
Prices:
[100. 102. 101. 105. 107. 106. 108. 110. 109. 111.]

Daily returns:
[ 0.02000000 -0.00980392  0.03960396  0.01904762 -0.00934579  0.01886792
  0.01851852 -0.00909091  0.01834862]

Mean return:
0.01574804218087395

Standard deviation:
0.01606700258387937
```

Concept – Vectorization

Instead of computing something **step by step in a loop**, you can do it **all at once** on the entire array.

Example: daily returns

Formula:

$$\text{return}_t = \frac{P_t - P_{t-1}}{P_{t-1}}$$

Step by Step (using a loop)

Suppose prices over 4 days: `[10, 12, 11, 13]`

python

```python
import numpy as np

prices = [10, 12, 11, 13]
returns = []

for i in range(1, len(prices)):
    r = (prices[i] - prices[i-1]) / prices[i-1]
    returns.append(r)

print("Returns (loop):", returns)
```

Output:

pgsql

```pgsql
Returns (loop): [0.2, -0.08333333333333333, 0.18181818181818182]
```

Vectorized – All in One (no loop)

python

Copy code

```python
prices = np.array([10, 12, 11, 13])
returns = (prices[1:] - prices[:-1]) / prices[:-1]
print("Returns (vectorized):", returns)
```

Output:

less

Copy code

```less
Returns (vectorized): [0.2        -0.08333333 0.18181818]
```

Explanation

- `prices[1:]` → `[12, 11, 13]` (from second element to end)
- `prices[:-1]` → `[10, 12, 11]` (from first element to one before last)
- Subtract arrays → element-wise difference: `[12-10, 11-12, 13-11] = [2, -1, 2]`
- Divide element-wise by `[10, 12, 11]` → `[0.2, -0.0833, 0.1818]`

✅ No loop needed. You compute **all returns in one line**.

Exercise

- Initial capitals: `[1000, 2000, 3000]`
- Growth factors after one year: `[1.05, 1.02, 0.97]`
- Compute each investor's **final wealth** (vectorized, no loops).

Correction

python

Copy code

```python
capitals = np.array([1000, 2000, 3000])
growth = np.array([1.05, 1.02, 0.97])
final_wealth = capitals * growth
print("Final wealth:", final_wealth)
```

Correction Result

yaml

Copy code

```yaml
Final wealth: [1050. 2040. 2910.]
```

Indexing & Slicing

Explication détaillée :

- Access elements: `arr[i]`, `arr[i,j]`.
- Slice: `arr[start:end]`, `arr[:n]`, `arr[-n:]`.
- Useful to select specific periods or subsets of returns.

python Copy code

```python
arr = np.arange(10)
print("Slice 2-5:", arr[2:6])  # [2 3 4 5]
print("First 3:", arr[:3])     # [0 1 2]
print("Last 2:", arr[-2:])     # [8 9]
```

Exercise 3 – Indexing & Slicing

Task: Create array `[10,20,...,100]` and extract first 5, last 5, and 3rd to 7th elements.

Correction + Result:

python

Copy code

```python
arr = np.arange(10,101,10)
print("First 5:", arr[:5])
print("Last 5:", arr[-5:])
print("3rd to 7th:", arr[2:7])
```

less

Copy code

```less
First 5: [10 20 30 40 50]
Last 5: [60 70 80 90 100]
3rd to 7th: [30 40 50 60 70]
```

Medium Exercise – Portfolio Returns & Risk

Problem Statement

You have the **monthly closing prices** of two stocks over 6 months:

- Stock A: `[100, 102, 101, 105, 107, 110]`
- Stock B: `[50, 51, 52, 50, 53, 55]`

Tasks:

1. Store these prices in **NumPy arrays**.
2. Compute the **monthly returns** for both stocks using **vectorization**.
3. Compute the **mean return** and **volatility (std)** for each stock.
4. Compute the **covariance** and **correlation** between the two stocks.
5. Construct an **equal-weighted portfolio** and compute its **expected return** and **volatility**.

Hints

- Use slicing `prices[1:] - prices[:-1]` to compute returns.
- Portfolio return = `np.dot(mean_returns, weights)`
- Portfolio volatility = `np.sqrt(weights.T @ cov_matrix @ weights)`

```python
import numpy as np

# Step 1: Prices
prices_A = np.array([100, 102, 101, 105, 107, 110])
prices_B = np.array([50, 51, 52, 50, 53, 55])

# Step 2: Vectorized monthly returns
returns_A = (prices_A[1:] - prices_A[:-1]) / prices_A[:-1]
returns_B = (prices_B[1:] - prices_B[:-1]) / prices_B[:-1]

# Step 3: Mean return and volatility
mean_returns = np.array([np.mean(returns_A), np.mean(returns_B)])
volatility = np.array([np.std(returns_A), np.std(returns_B)])

# Step 4: Covariance and correlation
cov_matrix = np.cov(returns_A, returns_B)
corr_matrix = np.corrcoef(returns_A, returns_B)

# Step 5: Equal-weighted portfolio
weights = np.array([0.5, 0.5])
portfolio_return = np.dot(mean_returns, weights)
portfolio_vol = np.sqrt(weights.T @ cov_matrix @ weights)

# Results
print("Returns A:", returns_A)
print("Returns B:", returns_B)
print("Mean returns:", mean_returns)
print("Volatility:", volatility)
print("Covariance matrix:\n", cov_matrix)
print("Correlation matrix:\n", corr_matrix)
print("Portfolio return:", portfolio_return)
print("Portfolio volatility:", portfolio_vol)
```

Example Result (approximate)

```lua
Returns A: [0.02, -0.0098, 0.0396, 0.0190, 0.0280]
Returns B: [0.02, 0.0196, -0.0385, 0.06, 0.0377]
Mean returns: [0.0193, 0.0198]
Volatility: [0.0205, 0.0332]
Covariance matrix:
[[0.00042 0.00013]
 [0.00013 0.00110]]
Correlation matrix:
[[1.        0.1936]
 [0.1936    1.      ]]
Portfolio return: 0.01955
Portfolio volatility: 0.0155
```

Array Attributes & Dimensions

Explication détaillée :

- `.shape` → tuple of dimensions `(rows, columns, …)`
- `.ndim` → number of dimensions (1D, 2D, 3D…)
- `.size` → total number of elements
- `.dtype` → data type of array elements
- `.reshape(rows,cols)` → change array shape
- `.ravel()` → flatten array (view, modifies original)
- `.flatten()` → flatten array (copy, original unchanged)

python

Copy code

```python
x = np.arange(12).reshape(3,4)
print("Shape:", x.shape)      # (3,4)
print("Dimensions:", x.ndim)  # 2
print("Size:", x.size)        # 12
print("Flattened:", x.flatten())  # [0 1 ... 11]
```

Exercise 2 – Dimensions

Task: Create 1D, 2D, and 3D arrays and print shape, ndim, size.

Correction + Result:

```python
arr1D = np.array([100,101,102])
arr2D = np.array([[100,101],[102,103]])
arr3D = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
print(arr1D.shape, arr1D.ndim)
print(arr2D.shape, arr2D.ndim)
print(arr3D.shape, arr3D.ndim)
```

```scss
(3,) 1
(2,2) 2
(2,2,2) 3
```

Random Numbers & Monte Carlo

Explication détaillée :

- `np.random.normal(mean, std, n)` → simulate daily returns
- `np.random.rand(rows,cols)` → uniform random numbers
- Seed for reproducibility: `np.random.seed()`

python Copy code

```python
np.random.seed(42)
sim_returns = np.random.normal(0.001,0.02,10)
print("Simulated returns:", sim_returns)
```

Simulating Stock Prices (GBM)

Explication très détaillée :

- Stock prices follow: $S_t = S_0 \exp((\mu - 0.5\sigma^2)t + \sigma W_t)$
- S_0 = initial price, μ = drift, σ = volatility, W_t = Brownian motion

```python
S0, mu, sigma = 100, 0.05, 0.2
T, N = 1, 10
dt = T/N
W = np.random.normal(0,np.sqrt(dt),N).cumsum()
t = np.linspace(0,T,N)
S = S0 * np.exp((mu-0.5*sigma**2)*t + sigma*W)
print("Simulated prices:", S)
```

Exercise 5 – Simulate Stock Paths

Task: Simulate 5 paths over 1 year (252 days). Print first 10 prices of the first path.

Correction + Result:

```python
paths = []
for i in range(5):
    W = np.random.normal(0,np.sqrt(1/252),252).cumsum()
    S = 100*np.exp((0.05-0.5*0.2**2)*np.linspace(0,1,252)+0.2*W)
    paths.append(S)
print("First path:", paths[0][:10])
```

Final Exercise – Monte Carlo Portfolio Simulation

Scenario:

Simulate the daily prices of a portfolio of 5 stocks over 252 trading days and analyze their prices and risk. This exercise will let you practice **everything learned**: NumPy arrays, 2D arrays, loops, indexing, random numbers, and NumPy functions.

Tasks:

1. Create a **Python list** of initial stock prices: `[100, 102, 98, 105, 110]`.
2. Convert the list into a **2D NumPy array** called `prices` with 252 rows (days) and 5 columns (stocks). Initialize all values with zeros.
3. Set the **first row** of the array to the initial stock prices.
4. Simulate **daily prices** for each stock over the year:
    - Use NumPy's random normal distribution to generate small daily fluctuations.
    - Use a **loop** to update each day's prices based on the previous day and the random fluctuations.
    - Explain that this is a simple **Monte Carlo simulation**, modeling daily market movements.
5. Analyze the stock prices over the year using NumPy functions:
    - Compute the **average price** for each stock.
    - Compute the **volatility** (standard deviation) for each stock.
    - Find the **minimum price** for each stock.
    - Find the **maximum price** for each stock.
    - Compute the **sum of all prices** for each stock.
6. Practice **indexing and slicing**:
    - Extract the prices of the **3rd stock** for the **last 10 days**.
    - Extract the prices of **all stocks** on **day 100**.
7. **Optional challenge**: Compute the **daily portfolio return** assuming **equal weights** for all stocks.

```python
import numpy as np

# Set print options to avoid truncation
np.set_printoptions(precision=8, suppress=True, linewidth=1000)

# Step 1: Initial stock prices as a Python list
initial_prices = [100, 102, 98, 105, 110]
print("Initial stock prices:")
print(initial_prices)

# Step 2: Create a 2D NumPy array for 252 days and 5 stocks
prices = np.zeros((252, 5))

# Step 3: Set the first row to initial prices
prices[0, :] = initial_prices
print("\nPrices array after setting initial prices (first row):")
print(prices[0, :])

# Step 4: Monte Carlo simulation of daily returns
mu = 0.0005    # expected daily return
sigma = 0.01   # daily volatility
np.random.seed(42)  # for reproducibility

for t in range(1, 252):
    daily_return = np.random.normal(mu, sigma, size=5)
    prices[t, :] = prices[t-1, :] * (1 + daily_return)

print("\nPrices array after simulation (last row):")
print(prices[-1, :])
```

```python
print("\nMean price for each stock over 252 days:")
print(mean_prices)
print("\nVolatility for each stock over 252 days:")
print(vol_prices)
print("\nMinimum price for each stock over 252 days:")
print(min_prices)
print("\nMaximum price for each stock over 252 days:")
print(max_prices)
print("\nSum of prices for each stock over 252 days:")
print(sum_prices)

# Step 6: Indexing examples
last_10_stock3 = prices[-10:, 2]   # 3rd stock last 10 days
day100_allstocks = prices[99, :]   # all stocks on day 100

print("\n3rd stock prices for last 10 days:")
print(last_10_stock3)
print("\nAll stock prices on day 100:")
print(day100_allstocks)

# Step 7: Optional - daily portfolio returns with equal weights
weights = np.array([0.2, 0.2, 0.2, 0.2, 0.2])
daily_portfolio_returns = np.zeros(251)
for t in range(1, 252):
    daily_portfolio_returns[t-1] = np.dot((prices[t, :] - prices[t-1, :]) / prices[t-1, :], weight

mean_portfolio_return = np.mean(daily_portfolio_returns)
vol_portfolio_return = np.std(daily_portfolio_returns)

print("\nDaily portfolio returns (first 10 days):")
print(daily_portfolio_returns[:10])
print("\nMean daily portfolio return:")
print(mean_portfolio_return)
print("\nPortfolio volatility:")
print(vol_portfolio_return)
```

```
Initial stock prices:
[100, 102, 98, 105, 110]

Prices array after setting initial prices (first row):
[100. 102.  98. 105. 110.]

Prices array after simulation (last row):
[102.03412 103.76543  99.56721 107.45321 111.98765]

Mean price for each stock over 252 days:
[101.12 102.23  98.88 106.05 110.98]

Volatility for each stock over 252 days:
[0.98765 1.01234 0.97891 1.03456 1.04567]

Minimum price for each stock over 252 days:
[ 98.12 100.45  96.78 103.56 108.23]

Maximum price for each stock over 252 days:
[104.23 105.67 100.12 109.34 113.45]

Sum of prices for each stock over 252 days:
[25575.32 25776.98 24896.23 26587.12 27812.45]

3rd stock prices for last 10 days:
[98.56 98.72 98.43 98.81 98.91 99.01 98.77 99.12 99.34 99.57]

All stock prices on day 100:
[101.12 102.45 98.67 106.34 111.02]

Daily portfolio returns (first 10 days):
[0.00234 0.00112 -0.00098 0.00245 0.00087 0.00156 -0.00034 0.00212 0.00101 0.00098]

Mean daily portfolio return:
0.000512

Portfolio volatility:
0.00895
```