Lecture 07 - Numerical Computing (`NumPy`)

Overview

`NumPy` (Numerical Python) is a fundamental library for scientific computing in Python. It provides support for arrays, matrices, and many mathematical functions that operate on these data structures.

This notebook covers:

- Creating and manipulating `NumPy` arrays
- Basic mathematical operations with `NumPy`
- Statistic calculations using `NumPy`
- Finance applications

Why Use NumPy?

`NumPy` offers significant advantages when working with numerical data:

- **Performance:** `NumPy` is much faster than traditional Python `lists` for numerical operations.
- **Functionality:** `NumPy` provides a wide range of mathematical functions and operations.
- **Convenience:** `NumPy` arrays are more convenient to work with, especially for large datasets.

1. Arrays of data

1.1 Definitions

General data structures in `Python`, especially `list`, are **flexible** but can be **inefficient** in terms of memory and performance.

For scientific and financial applications that demand high-performance operations on specialized data structures, `arrays` are crucial.

What are Arrays?

`Arrays` organize elements of the same data type in **rows** and **columns**, typically representing numbers like real values.

- A one-dimensional array represents a **vector**
- A multi-dimensional array forms **matrices**, **cubes**, etc.

To efficiently handle arrays, `NumPy` works with its `ndarray` class which provides powerful and specialized functionality.

1.2 Using `list` to handle arrays

Dimensionality

Python `list` can be of multiple dimensions: vector, matrices, cubes

In [ ]:

```
# 1-dimension list
v = [0.5, 0.75, 1.0, 1.5, 2.0]

# 2-dimensions list = matrix
m = [v, v, v]
```

Indexing

Reminder: `list` are indexed from 0 for each of their dimensions

In [ ]:
```
m
```

In [ ]:
```
m[1]
```

In [ ]:
```
m[1][0]
```

n-dimensions & deep copies

`list` of *n*-dimensions can be constructed as vectors of other lists.

In [ ]:
```
v1 = [0.5, 1.5]
v2 = [1, 2]
m = [v1, v2]
c = [m, m]
c
```

However, such construction leads to dependencies in terms of references and memory.

By default, a copy of a vector in a `list` points to the same original values.

Operations on the copy therefore also apply to the origin.

In [ ]:
```
v = [0.5, 0.75, 1.0, 1.5, 2.0]
m = [v, v, v]
v[0] = 'Python'
m
```

Deep copies allow to decouple origins and copies by creating different references in the memory.

In [ ]:
```
from copy import deepcopy
v = [0.5, 0.75, 1.0, 1.5, 2.0]
m = 3 * [deepcopy(v)]
m
```

In [ ]:
```
v[0] = 'Python'
m
```

In [ ]:
```
v
```

2. Regular `NumPy` arrays

2.1 The basics

Using `list` objects to compose array structures is possible, but **not very convenient** since the list class is designed for **broader, general purposes**.

A truly dedicated class for handling array-type structures is far more beneficial.

`numpy.array` is such a class, built with the specific goal of handling *n*-dimensional arrays both conveniently and efficiently.

In [ ]:
```
import numpy as np
a = np.array([0, 0.5, 1.0, 1.5, 2.0])
a
```

In [ ]:
```
type(a)
```

In [ ]:
```
a = np.array(['a', 'b', 'c'])
a
```

`arange()`

Similar to the `range` function

In [ ]:
```
a = np.arange(2, 20, 2)
a
```

In [ ]:
```
a = np.arange(8, dtype=float)
a
```

Indexing

Indexing works like in `list`.

In [ ]:
```
a[5:]
```

In [ ]:
```
a[:2]
```

`ndarray` built-in methods

In [ ]:
```
a.sum()
```

In [ ]:
```
a.std()
```

In [ ]:
```
a.cumsum()
```

(Vectorized) Math operations

In [ ]:
```
2 * a
```

In [ ]:
```
a ** 2
```

In [ ]:
```
a ** a
```

Universal functions

“Universal” because they operate on `ndarray` objects as well as on basic `Python` data types.

In [ ]:
```
np.exp(a)
```

In [ ]:
```
np.sqrt(a)
```

In [ ]:
```
np.sqrt(2.5)
```

Matrices

In [ ]:
```
b = np.array([a, a * 2])
b
```

In [ ]:
```
b[0]
```

In [ ]:
```
b[0,2]
```

In [ ]:
```
b[:,1]
```

In [ ]:
```
b.sum()
```

In [ ]:
```
# column-wise operation
b.sum(axis = 0)
```

In [ ]:
```
# row-wise operation
b.sum(axis = 1)
```

2.2 Initialization

Initialization functions allow to construct *n*-dimensional arrays with pre-instructed procedures to fill each data point:

- `np.zeros()`, `np.zeros_like()`
- `np.ones()`, `np.ones_like()`
- `np.empty()`, `np.empty_like()`
- `np.eye()`
- `np.linspace()`

For initialization functions, one can provide the following parameters:

- `shape`: Either an `int`, a sequence of `int` objects, or a reference to another `ndarray`
- `dtype` (*optional*): A `dtype`—these are `NumPy`-specific data types for `ndarray` objects

dtype	Description
?	Boolean
i	Signed integer
u	Unsigned integer
f	Floating point
c	Complex floating point
m	timedelta
M	datetime
O	Object
U	Unicode
V	Raw data (void)

`np.zeros()`

The `np.zeros()` function creates a 2-dimensional array of zeros with specific characteristics.

In [ ]:
```
c = np.zeros((2, 3), dtype='i')
c
```

In [ ]:
```
d = np.zeros_like(c, dtype='f')
d
```

`np.ones()`

The `np.ones()` function creates a 2-dimensional array of ones with specific characteristics.

In [ ]:

```
c = np.ones((2, 3, 4), dtype='i')
c
```

This example generates an array filled with ones. The shape of the array is (2, 3, 4), which means it will have:

- 2 blocks (first dimension),
- each block contains 3 rows (second dimension),
- and each row contains 4 elements (third dimension).

The result is a 3D array like:

```
[
  [[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]],  # First block (2x3x4)
  [[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]   # Second block (2x3x4)
]
```

In [ ]:

```
d = np.ones_like(c, dtype='f')
d
```

`np.empty()`

Unlike `np.ones()` or `np.zeros()`, `np.empty()` does not initialize the array with any specific values like 1 or 0.

Instead, it leaves the array filled with whatever data is already in memory at that location. This means the array will have arbitrary values, often referred to as “garbage” values. This makes `np.empty()` faster than functions that initialize the array with specific values because it skips that step.

Empty view

```
[
  [[?, ?], [?, ?], [?, ?]],  # First block (2x3x2)
  [[?, ?], [?, ?], [?, ?]]   # Second block (2x3x2)
]
```

Where the `?` symbols represent random or garbage values, because the array is uninitialized.

In [ ]:
```
e = np.empty((2, 3, 2))
e
```

In [ ]:
```
f = np.empty_like(c)
f
```

`np.eye()`

The `np.eye(n)` function creates a 2D identity matrix of size nxn

In [ ]:
```
np.eye(5)
```

`np.linspace()`

The `np.linspace()` function creates a 1-dimensional array of evenly spaced numbers.

In [ ]:

```
g = np.linspace(5, 15, 12)
g
```

2.3 Metainformation

Arrays deliver several meta-information.

- `size` returns the total number of elements in the array

In [ ]:
```
g.size
```

- `itemsize` returns the size (in bytes) of each element in the array

In [ ]:
```
g.itemsize
```

- `ndim` returns the number of dimensions of the array

In [ ]:
```
g
```

In [ ]:
```
g.ndim
```

- `shape` returns the shape of the array

In [ ]:
```
g.shape
```

- `dtype` returns the data type of the elements in the array

In [ ]:
```
g.dtype
```

- `nbytes` returns the total number of bytes consumed by the elements of the array

In [ ]:
```
g.nbytes
```

2.4 Structural operations

In [ ]:

```
# Initialise an array
g = np.arange(15)
print (g)
print (g.shape)
```

Reshaping

Reshaping provides another view on the same data.

During a reshaping operation, the total number of elements in the ndarray object is unchanged.

- `np.reshape(a, newshape)`

In [ ]:
```
g.reshape((3,5))
```

In [ ]:
```
h = g.reshape((5,3))
h
```

Transposing

Transposing creates a view of the original array (i.e., it does not create a new object unless explicitly copied).

The dimensions (axes) of the `ndarray` object are swapped.

For 2D arrays, this means rows become columns and columns become rows. For higher-dimensional arrays, the axes are reordered according to the shape.

- `a.transpose(*axes)`
- `a.T`
- `np.transpose(a, axes)`

In [ ]:
```
h
```

In [ ]:
```
h.transpose()
```

In [ ]:
```
h.T
```

Resizing

Resizing creates a new (temporary) object.

During a resizing operation, the total number of elements in the `ndarray` object changes: it either decreases ("down-sizing") or increases ("up-sizing").

- `np.resize(a, newshape)`

In [ ]:
```
g
```

In [ ]:
```
g.size
```

In [ ]:
```
np.resize(g,(3,1))
```

In [ ]:
```
np.resize(g, (1, 5))
```

In [ ]:
```
np.resize(g, (2, 5))
```

In [ ]:
```
n = np.resize(g, (5, 4))
n
```

Stacking

Stacking joins a sequence of arrays along a new axis.

This is used to combine arrays into a single array with a new dimension. The arrays must have the same shape, except along the axis being stacked.

- `np.stack(arrays, axis=0)`
- `np.hstack(tuple)` (for horizontal stacking)
- `np.vstack(tuple)` (for vertical stacking)

In [ ]:
```
a = np.array([1, 2])
b = np.array([3, 4])
```

In [ ]:
```
# Stacking along a new axis
np.stack((a, b), axis=0)
```

In [ ]:
```
np.stack((a, b), axis=1)
```

In [ ]:
```
# Horizontal and vertical stacking
np.hstack((a, b))
```

In [ ]:
```
np.vstack((a, b))
```

In [ ]:
```
h
```

In [ ]:
```
# horizontal stacking
np.hstack((h,2*h))
```

In [ ]:
```
# vertical stacking
np.vstack((h, 0.5 * h))
```

Flattening

Flattening refers to converting a multi-dimensional array into a 1-dimensional array. This is useful when processing the array as a sequence of elements, regardless of its original shape.

- `a.flatten()`
- `np.ravel(a)`

In [ ]:
```
a = np.array([[1, 2], [3, 4]])

# Flattening the array
a.flatten()
```

In [ ]:
```
np.ravel(a)
```

In [ ]:
```
h
```

In [ ]:
```
h.flatten()
```

In [ ]:
```
for i in h.flat:
    print(i, end=',')
```

2.5 Boolean arrays

Logical operations work on `ndarray` objects the same way, element-wise, as on standard Python data types.

In [ ]:
```
h
```

Basics

In [ ]:
```
h > 8
```

In [ ]:
```
h <= 7
```

In [ ]:
```
h == 5
```

In [ ]:
```
(h == 5).astype(int)
```

In [ ]:
```
(h > 4 ) & (h <= 12)
```

Boolean filtering

In [ ]:
```
h[h > 8]
```

In [ ]:
```
h[(h > 4) & (h <= 12)]
```

In [ ]:
```
h[(h < 4) | (h >= 12)]
```

`np.where()` returns the indices of elements in an input array that satisfy a given condition.

It can also be used to return values from one of two arrays, depending on a condition (conditional selection).

- `np.where(condition, [x, y])`:
    - If only `condition` is provided, `np.where()` returns the indices where the condition is `True`.
    - If `x` and `y` are also provided, `np.where()` returns elements from `x` where the condition is `True`, and from `y` where the condition is `False`.

In [ ]:
```
a = np.array([1, 2, 3, 4, 5])
np.where(a > 3)
```

In [ ]:
```
np.where(np.eye(10)>0)
```

In [ ]:
```
# use the np.where() function
np.where(h > 7, 1, 0)
```

In [ ]:
```
np.where(h % 2 == 0, 'even', 'odd')
```

In [ ]:
```
np.where(h <= 7, h * 2, h / 2)
```

3. Structured `NumPy` arrays

Structured `ndarray` objects allow to have a different `dtype` per column.

The construction is similar to the operation for initializing tables in a database (e.g., SQL): one has column names and column data types, with maybe some additional information (e.g., maximum number of characters per str object).

In [ ]:
```
dt = np.dtype([('Name', 'S10'), ('Age', 'i'),
                                ('Height', 'f'), ('Children/Pets', 'i', 2)])
dt
```

In [ ]:
```
# alternative
dt = np.dtype({'names': ['Name', 'Age', 'Height', 'Children/Pets'],
                        'formats':'O int float int,int'.split()})
dt
```

In [ ]:
```
s = np.array([('Smith', 45, 1.83, (0, 1)),
                         ('Jones', 53, 1.72, (2, 2))], dtype=dt)
s
```

In [ ]:
```
type(s)
```

Indexing

The single columns can now be easily accessed by their names and the rows by their index values.

In [ ]:
```
s['Name']
```

In [ ]:
```
s['Height'].mean()
```

In [ ]:
```
s[0]
```

In [ ]:
```
s[1]['Age']
```

4. Vectorized programming

Vectorization is a strategy to get **more compact code** that is possibly executed **faster** by simplifying the treatment and operations over `array` (vector) objects.

The fundamental idea is to conduct an operation on or to apply a function to a complex `array` object "**at once**" and **not by looping** over the single elements of the object.

`NumPy` has vectorization built in deep down in its core.

Operations

In [ ]:
```
r = np.arange(12).reshape((4, 3))
s = np.arange(12).reshape((4, 3)) * 0.5
```

In [ ]:
```
r
```

In [ ]:
```
s
```

In [ ]:
```
r + s
```

In [ ]:
```
r + 3
```

In [ ]:
```
2 * r
```

In [ ]:
```
2 * r + 3
```

Role of shapes

In [ ]:
```
r.shape
```

In [ ]:
```
r
```

In [ ]:
```
s = np.arange(0, 12, 4)
s
```

In [ ]:
```
s.shape
```

In [ ]:
```
r + s
```

In [ ]:
```
s = np.arange(0, 12, 3)
```

In [ ]:
```
s.shape
```

In [ ]:
```
r + s
```

In [ ]:
```
r.transpose() + s
```

Applying functions

In [ ]:
```
def f(x):
    return 3 * x + 5
```

In [ ]:
```
f(0.5)
```

In [ ]:
```
f(r)
```

5. Random number generation

Random number generation is often essential, particularly in simulations like Monte Carlo methods.

`NumPy` provides a convenient and powerful way to generate random numbers through the `np.random` module.

- `rand()`: generates numbers between 0 and 1.

In [ ]:
```
np.random.rand(5)
```

- `randn()`: generates numbers with a mean of 0 and a standard deviation of 1 by default.

In [ ]:
```
np.random.randn(5)
```

- `normal(mean, std, n)` : generates random *n* numbers from a normal distribution with mean and std

In [ ]:

```
# Simulate 10 days of daily returns with a mean of 0.001 and std deviat
daily_returns = np.random.normal(0.001, 0.02, 10)
daily_returns
```

- `randint()` : generates random integers

In [ ]:
```
np.random.randint(1, 100, 5)
```

6. Applications

6.1 Stock price simulations

Geometric brownian motion

In finance, the Geometric Brownian Motion model is commonly used to simulate the future path of stock prices. The model is represented by the following equation:

$$S_{t+1} = S_t \times e^{(\mu - 0.5 \times \sigma^2)\Delta t + \sigma \times \epsilon \times \sqrt{\Delta t}}$$

Where:

- S_t is the stock price at time t
- μ is the expected return (mean)
- σ is the volatility (standard deviation)
- Δt is the time step (in days, for example)
- ϵ is a random variable from a normal distribution

Expressing as Log Returns

Taking logs makes this additive:

$$\ln \frac{S_{t+\Delta t}}{S_t} = (\mu - 0.5\sigma^2)\Delta t + \sigma\epsilon_t\sqrt{\Delta t} = r_t$$

This shows that log returns are normally distributed with:

- Mean:
  $(\mu - 0.5\sigma^2$
  $)\Delta t$
- Std. deviation: $\sigma\sqrt{\Delta t}$

Price path

Then, cumulative summation and exponentiation reconstruct the price path from $t = 0$ to $t = T$:

$$S_T = S_0 \, e^{\sum r_t}$$

Simulating daily returns with `NumPy`

Let's simulate daily stock price returns over a one-year period (252 trading days).

Using `NumPy`, we can generate random normal values for the stock price fluctuations and compute the simulated prices.

In [ ]:

```python
import numpy as np

# Parameters
S0 = 100  # Initial stock price
mu = 0.05  # Expected annual return
sigma = 0.2  # Annual volatility
T = 1  # Time horizon (in years)
N = 252  # Number of trading days
dt = T / N  # Time step (1 day)

# Simulate random daily returns
daily_returns = np.random.normal((mu - 0.5 * sigma**2) * dt, sigma * np

# Simulate stock prices
prices = S0 * np.exp(np.cumsum(daily_returns))
```

```
daily_returns = np.random.normal((mu - 0.5 * sigma**2) * dt, sigma *
np.sqrt(dt), N)
```

This line generates random daily returns based on a normal distribution.

- The mean of the distribution is adjusted by the term $\mu - 0.5\sigma^2$ to account for the fact that stock prices follow a geometric Brownian motion.
- The standard deviation of the returns is $\sigma\sqrt{dt}$, which scales the annual volatility to daily volatility.

```
prices = S0 * np.exp(np.cumsum(daily_returns))
```

This line calculates the simulated stock prices using the exponential function.

It does this by:

- Using `np.cumsum(daily_returns)` to get the cumulative returns over time. The cumulative sum allows for the aggregation of daily returns into total returns.
- The `np.exp()` function then exponentiates these cumulative returns, which translates them into price levels.
- Finally, the initial stock price `S0` is multiplied by the exponentiated cumulative returns to get the actual stock prices.

In [ ]:

```
# Plot the simulated stock prices
import matplotlib.pyplot as plt
plt.plot(prices)
plt.title('Simulated Stock Price Using Geometric Brownian Motion')
plt.xlabel('Day')
plt.ylabel('Price')
plt.show()
```

6.2 Portfolio optimization

In portfolio management, finance professionals must calculate the returns, risks, and correlations of assets to construct optimal portfolios.

`NumPy` makes this task efficient and straightforward by offering matrix operations and linear algebra functions.

Let's assume a portfolio with three assets. We can calculate the expected portfolio return and risk using `NumPy`.

- The function `np.dot()` returns the product of matrices.

Defining the portfolio

In [ ]:

```
# Asset returns (annualized)
asset_returns = np.array([0.08, 0.12, 0.15])

# Covariance matrix of asset returns
cov_matrix = np.array([[0.005, -0.002, 0.004],
                       [-0.002, 0.010, -0.003],
                       [0.004, -0.003, 0.015]])

# Portfolio weights
weights = np.array([0.4, 0.3, 0.3])
```

Computing risk and return

In [ ]:

```
# Calculate portfolio return
portfolio_return = np.dot(weights, asset_returns)

# Calculate portfolio risk (standard deviation)
portfolio_risk = np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))

print(f"Expected Portfolio Return: {portfolio_return:.2%}")
print(f"Portfolio Risk (Standard Deviation): {portfolio_risk:.2%}")
```

```
portfolio_return = np.dot(weights, asset_returns)
```

The expected portfolio return is calculated using the dot product of the portfolio weights and the asset returns.

This calculation effectively gives the weighted average return of the assets in the portfolio.

The formula is:

$$\text{Portfolio Return} = w_1 \cdot r_1 + w_2 \cdot r_2 + w_3 \cdot r_3$$

In this case:

$$\begin{aligned}\text{Portfolio Return} &= (0.4 \times 0.08) + (0.3 \times 0.12) + (0.3 \times 0.15) \\ &= 0.104 \text{ or } 10.4\%\end{aligned}$$

```
portfolio_risk = np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))
```

The portfolio risk is calculated using the covariance matrix and the portfolio weights.

This formula computes the standard deviation of the portfolio, reflecting its total risk:

$$\text{Portfolio Risk} = \sqrt{W^T \Sigma W}$$

where:

- W is the vector of weights.
- Σ is the covariance matrix.
- The inner product `np.dot(cov_matrix, weights)` calculates the weighted covariance for each asset, and then the outer product with `weights.T` provides the total variance of the portfolio. The square root of this value gives the standard deviation.

6.3 Risk analysis

Monte Carlo simulations are widely used in finance to model the probability of different outcomes in uncertain environments.

`NumPy` can be used to run a Monte Carlo simulation for portfolio returns.

In [ ]:

```
# Parameters for simulation
n_simulations = 10000  # Number of simulations
n_days = 252  # One year of trading days
mean_return = 0.001  # Mean daily return
std_dev = 0.02  # Daily standard deviation

# Simulate random daily returns for one year
simulated_returns = np.random.normal(mean_return, std_dev, (n_simulatic

# Calculate cumulative returns for each simulation
cumulative_returns = np.cumprod(1 + simulated_returns, axis=1)

# Final value of each simulation
final_values = cumulative_returns[:, -1]
```

```
simulated_returns = np.random.normal(mean_return, std_dev,
(n_simulations, n_days))
```

This line generates a 2D array of random daily returns using a normal distribution.

- Each row represents a separate simulation (one potential outcome for the year), and each column represents the daily returns for that simulation across 252 trading days.
- The `np.random.normal` function takes the mean and standard deviation, generating random returns for each day in each simulation.

```
cumulative_returns = np.cumprod(1 + simulated_returns, axis=1)
```

The cumulative returns are calculated by taking the product of the daily returns over time.

- The expression `1 + simulated_returns` adjusts the returns to represent growth factors (e.g., if the return is 0.01, the growth factor is 1.01).
- The `np.cumprod` function computes the cumulative product along the specified axis (here, `axis=1`, which means across each simulation).

```
final_values = cumulative_returns[:, -1]
```

This line extracts the final value of the portfolio at the end of the simulation period (after 252 days) for each of the 10,000 simulations.

- `cumulative_returns[:, -1]` retrieves the last column of the cumulative returns matrix, representing the portfolio value after one year of trading.

In [ ]:

```
# Plot distribution of final values
plt.hist(final_values, bins=50, edgecolor='k')
plt.title('Monte Carlo Simulation of Portfolio Returns')
plt.xlabel('Final Portfolio Value')
plt.ylabel('Frequency')
plt.show()
```