Foundations of NumPy and Matplotlib

An Applied Python Mini-Chapter

Dr. Youssef Salman

Learning outcomes. By the end of this chapter, you will be able to:

- create and manipulate NumPy arrays (vectors and matrices);
- use vectorized operations and efficient slicing;
- generate synthetic data (linear grids, random samples, basic distributions);
- produce basic plots with Matplotlib.

0. Getting set up (Python environment)

Prerequisites. Basic Python (variables, lists, loops, functions) and an environment such as Jupyter (or Spyder/VS Code).

Quick install (terminal):

```
pip install --upgrade pip
pip install numpy matplotlib
```

Conventional imports in Python:

```
import numpy as np
import matplotlib.pyplot as plt
```

1 Meet NumPy: the ndarray

NumPy array (ndarray). A contiguous, fixed-size, homogeneous data container (all elements share the same type). Enables fast vectorized operations and broadcasting, often much faster than pure-Python loops.

1.1 Creation and basic properties

```
import numpy as np

# 1) From a Python list
a = np.array([1, 1.5, 2, 2.5])
print(a.dtype, a.shape, a.ndim)  # float64 (4,) 1

# 2) Common generators
```

```
z = np.zeros(5)           # [0., 0., 0., 0., 0.]
o = np.ones((2, 3))       # 2x3 matrix of ones
r = np.arange(0, 10, 2)   # 0, 2, 4, 6, 8
l = np.linspace(0, 1, 5)  # 5 points from 0 to 1 inclusive

# 3) Random numbers (seed for reproducibility)
rng = np.random.default_rng(seed=42)
u = rng.random(4)         # Uniform [0, 1)
n = rng.normal(0, 1, (2,2))  # Gaussian N(0,1) 2x2
```

Key attributes.

- `.shape`: tuple of dimensions, e.g. `(2, 3)` for a 2×3 matrix.
- `.ndim`: number of dimensions (axes).
- `.dtype`: data type (e.g. `float64`, `int32`, `bool`).
- `.size`: total number of elements.

Exercise (types and shapes). Create an array of integers from -3 to 3 included. Convert it to `float32`, then reshape it to a 2×3 matrix (discard or adapt one element if needed). Verify `dtype`, `shape` and `size`.

1.2 Vectorized arithmetic & broadcasting

Vectorized operations apply elementwise without explicit loops, and *broadcasting* aligns shapes when possible:

```
x = np.array([0., 1., 2., 3.])
y = np.array([10., 10., 10., 10.])

s = x + y          # [10., 11., 12., 13.]
p = x * 2          # [0., 2., 4., 6.]
q = (x + 1) / (y)  # [0.1, 0.2, 0.3, 0.4]

# Broadcasting: adding a scalar or compatible shape
M = np.arange(6).reshape(2, 3)   # [[0,1,2],[3,4,5]]
v = np.array([1, 0, -1])
B = M + v    # [[1,1,1],[4,4,4]]  (v broadcast across rows)
```

Why it is fast. NumPy offloads tight loops to optimized C/Fortran code and operates on contiguous memory blocks. Prefer vectorized code whenever possible.

1.3 Indexing and slicing

Slicing returns *views* whenever possible (no data copy), which is memory- and time-efficient.

```
a = np.arange(10)         # [0 1 2 3 4 5 6 7 8 9]
print(a[0], a[-1])        # first and last element
print(a[2:7])             # [2 3 4 5 6]
```

```
print(a[:5], a[5:])       # first five / from index 5 to end
print(a[::2])             # step of 2: [0 2 4 6 8]

# 2D indexing (row, col)
M = np.arange(12).reshape(3,4)
print(M[0, 1])            # element at row 0, col 1
print(M[1, :])            # entire row 1  (view)
print(M[:, 2])            # entire column 2 (view)

# Modifying a view modifies the original!
row1 = M[1, :]
row1[:] = -1
print(M)                  # second row becomes [-1 -1 -1 -1]
```

Exercise (views vs. copies). Create a 4×4 array with `np.arange`. Slice the central 2×2 block into `C`, set all entries of `C` to `999`, and inspect the original array. Then force a copy with `C = C.copy()` and repeat. What changes?

2 Array operations and reductions

2.1 Aggregations

NumPy provides fast aggregation functions that collapse an array to a scalar or along a given axis:

```
M = np.arange(1, 10).reshape(3, 3)

print(M.sum())          # total sum = 45
print(M.mean())         # average = 5.0
print(M.max(), M.min()) # 9, 1

# Axis-based reductions
print(M.sum(axis=0))  # column sums: [12 15 18]
print(M.sum(axis=1))  # row sums: [ 6 15 24]
```

Exercise. Create a random 5×5 array from the standard normal distribution. Compute: mean of each row, standard deviation of each column. Then normalize the array by subtracting the global mean and dividing by the global standard deviation.

2.2 Boolean indexing and masks

Boolean arrays can be used to select or filter values:

```
x = np.arange(10)
mask = x % 2 == 0        # True for even numbers
print(x[mask])           # [0 2 4 6 8]
```

```

x[x > 5] = 99          # in-place modification
print(x)               # [ 0 1 2 3 4 5 99 99 99 99]
```

Tip. Boolean masks and vectorized comparisons are much faster and clearer than looping with `if` statements in Python.

3 Introduction to `Matplotlib`

3.1 Basic plotting

The most common interface is `matplotlib.pyplot`, which mimics MATLAB-style plotting.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 200)
y = np.sin(x)

plt.plot(x, y, label="sin(x)")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Basic sine curve")
plt.legend()
plt.show()
```

Exercise. Plot both $\sin(x)$ and $\cos(x)$ on the same axes from 0 to 2π. Use different line styles or colors, add a legend, and a grid.

3.2 Figures and subplots

Plots live inside a *figure*, which can contain multiple subplots arranged in a grid.

```
x = np.linspace(0, 1, 100)
y1 = x**2
y2 = np.sqrt(x)

fig, axes = plt.subplots(1, 2, figsize=(10, 4))

axes[0].plot(x, y1, color="red")
axes[0].set_title("y = x^2")

axes[1].plot(x, y2, color="blue")
axes[1].set_title("y = sqrt(x)")

plt.tight_layout()
```

```
plt.show()
```

Note. The object-oriented API (`fig, ax = plt.subplots(...)`) is recommended for complex plots, since it gives more explicit control.

3.3 Scatter plots and styles

```
rng = np.random.default_rng(0)
x = rng.normal(size=200)
y = rng.normal(size=200)

plt.scatter(x, y, alpha=0.6, c="purple", marker="x")
plt.title("Random scatter plot")
plt.xlabel("X")
plt.ylabel("Y")
plt.axis("equal")
plt.show()
```

Exercise. Generate 300 random points with coordinates (x, y) uniformly sampled in $[-1, 1]^2$. Plot them with a scatter plot, then highlight in a different color the points inside the unit circle $x^2 + y^2 \leq 1$.

4 Customizing plots

4.1 Colors, markers, and line styles

You can customize the appearance of curves with concise codes or full keywords.

```
x = np.linspace(0, 2*np.pi, 200)

plt.plot(x, np.sin(x), "r--", label="red dashed")
plt.plot(x, np.cos(x), color="green", marker="o",
         linestyle=":", label="green dotted with circles")

plt.legend()
plt.title("Custom styles")
plt.show()
```

Style codes.

- Colors: `'r'` red, `'b'` blue, `'g'` green, or named colors like `"orange"`.
- Line styles: `'-'` solid, `'--'` dashed, `':'` dotted, `'-.'` dash-dot.
- Markers: `'o'` circle, `'x'` cross, `'s'` square, etc.

4.2 Labels, grids, and legends

```
x = np.linspace(0, 10, 100)
y1 = np.exp(-0.1*x) * np.sin(x)
y2 = np.exp(-0.1*x) * np.cos(x)

plt.plot(x, y1, label="exp(-0.1x) * sin(x)")
plt.plot(x, y2, label="exp(-0.1x) * cos(x)")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Damped oscillations")
plt.legend(loc="upper right")
plt.grid(True, linestyle="--", alpha=0.6)
plt.show()
```

4.3 Histograms

Histograms display the distribution of data.

```
rng = np.random.default_rng(123)
data = rng.normal(loc=0, scale=1, size=1000)

plt.hist(data, bins=30, color="skyblue",
         edgecolor="black", alpha=0.7)
plt.title("Histogram of N(0,1)")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```

Exercise. Generate 10,000 samples from a uniform distribution on $[0, 1]$. Plot a histogram with 20 bins. Overlay a vertical line at the theoretical mean 0.5.

5 Practical session: Bringing NumPy and Matplotlib together

5.1 Example: projectile motion

We can simulate simple physics using NumPy arrays and visualize with Matplotlib.

```
g = 9.81     # gravity
v0 = 20.0    # initial speed (m/s)
theta = 45 * np.pi/180  # angle in radians

t = np.linspace(0, 3, 200)
x = v0 * np.cos(theta) * t
y = v0 * np.sin(theta) * t - 0.5*g*t**2

plt.plot(x, y)
```

```python
plt.title("Projectile trajectory")
plt.xlabel("x (m)")
plt.ylabel("y (m)")
plt.axhline(0, color="black", linewidth=0.8)
plt.show()
```

Exercise. Modify the initial angle and velocity, and observe the changes in trajectory. Can you compute the maximum height and range analytically and compare with the plot?

5.2 Example: population growth (logistic model)

The logistic growth model is widely used in biology and computer science (e.g. spreading of information or viruses).

$$P(t) = \frac{K}{1 + Ae^{-rt}}$$

where K is the carrying capacity, r the growth rate, and A depends on the initial condition.

```python
K = 1000      # carrying capacity
r = 0.5       # growth rate
P0 = 10       # initial population
A = (K - P0) / P0

t = np.linspace(0, 20, 200)
P = K / (1 + A * np.exp(-r*t))

plt.plot(t, P, label="Logistic growth")
plt.xlabel("Time")
plt.ylabel("Population")
plt.title("Logistic model")
plt.legend()
plt.show()
```

Exercise. Try different values of K, r, and P_0. What happens if P_0 is close to K? What if r is very small?

6 Practical exercises

1. **Vector operations.** Create two random vectors of size 100. Compute:
   - dot product,
   - elementwise product,
   - cosine similarity.
2. **Matrix manipulations.** Create a 5×5 matrix with values $1, 2, \ldots, 25$. Extract:
   - the main diagonal,

- the last column,
- the submatrix of the four central elements.

3. **Visualization of functions.** Plot on the same figure:

$$f(x) = e^{-x^2}, \quad g(x) = \sin(5x)\, e^{-x^2}$$

on $x \in [-3, 3]$. Add labels, legend, and grid.

4. **Random data and histograms.** Generate 5000 samples from the exponential distribution with parameter $\lambda = 2$. Plot the histogram and compare the empirical mean with the theoretical mean $1/\lambda$.
5. **Real-world application (informatics).** Suppose a server records the number of requests per minute, modeled as a Poisson$(\lambda = 5)$. Simulate 1000 minutes, plot the histogram, and discuss what the distribution suggests about server load.

Tip. Always label axes and use legends in plots. Visual clarity is as important as numerical accuracy.

7 Selected solutions and hints

1. Vector operations

```
rng = np.random.default_rng(0)
u = rng.random(100)
v = rng.random(100)

dot = np.dot(u, v)
elemwise = u * v
cos_sim = dot / (np.linalg.norm(u) * np.linalg.norm(v))

print("dot =", dot)
print("cosine similarity =", cos_sim)
```

2. Matrix manipulations

```
M = np.arange(1, 26).reshape(5, 5)
diag = np.diag(M)
last_col = M[:, -1]
central = M[1:4, 1:4]

print("Diagonal:", diag)
print("Last column:", last_col)
print("Central block:\n", central)
```

3. Visualization of functions

```
x = np.linspace(-3, 3, 400)
f = np.exp(-x**2)
g = np.sin(5*x) * np.exp(-x**2)

plt.plot(x, f, label="exp(-x^2)")
plt.plot(x, g, label="sin(5x) * exp(-x^2)")
plt.xlabel("x"); plt.ylabel("y")
plt.legend(); plt.grid(True)
plt.title("Function comparison")
plt.show()
```

4. Random data and histograms

```
lam = 2
samples = rng.exponential(1/lam, 5000)

plt.hist(samples, bins=40, color="lightcoral",
         edgecolor="black", alpha=0.7, density=True)
plt.axvline(samples.mean(), color="blue", linestyle="--",
            label=f"empirical mean {samples.mean():.2f}")
plt.axvline(1/lam, color="green", linestyle="-.",
            label="theoretical mean 0.5")
plt.legend()
plt.title("Exponential distribution")
plt.show()
```

5. Real-world application: server load

```
lam = 5
data = rng.poisson(lam, 1000)

plt.hist(data, bins=range(0, 15), align="left",
         rwidth=0.8, color="skyblue", edgecolor="black")
plt.xlabel("Requests per minute")
plt.ylabel("Frequency")
plt.title("Poisson simulation (lambda=5)")
plt.show()

print("Mean load =", data.mean())
```

Interpretation. The histogram is centered around 5, with most values between 2 and 8. This indicates moderate variability but stable average server load, consistent with a Poisson process.

Conclusion

In this mini-chapter you have learned:

- how to create and manipulate arrays with NumPy;
- how vectorization, slicing, and broadcasting simplify code and improve performance;
- how to use Matplotlib for line plots, scatter plots, histograms, and subplots;
- how to connect mathematical models to data visualization in applied contexts.

Next step. In the following chapter we extend these foundations to linear algebra (matrix operations, decompositions) and their applications in computer science.