Goal. In these exercises you will learn to use NumPy to create, inspect, and manipulate arrays. The same operations appear directly in Mini-project A, where arrays represent images. Understanding them today will make the mini-project much easier to approach.

These exercises are self-contained. You only need NumPy and Pillow (for saving images). You do not need the mini-project files.

Setup. Open Visual Studio Code, open a terminal, and create a virtual environment:

```
python3 -m venv .venv
```

Activate the virtual environment depending on your operating system:

- **macOS/Linux:** `source .venv/bin/activate`
- **Windows:** `.venv\Scripts\activate`

Then install the required libraries:

```
python3 -m pip install numpy Pillow
```

Create a new file **`exercises.py`** with the following two lines at the top. Along with save_image function Every code snippet below assumes these imports are present.

```
import numpy as np
from PIL import Image
```

To save an array as an image file, use this helper throughout the session:

```
def save_image(arr, filename):
    """Save a numpy array as an image file."""
    Image.fromarray(arr.astype("uint8")).save(filename)
```

To run the code, make sure your virtual environment is activated, then execute:

```
python3 exercises.py
```

Exercise 1. Why NumPy?

Python lists are convenient, but they do not support mathematical operations the way we need for image processing.

(a) Run the following and observe the output:

```
l = [1, 2, 3, 4]
print(l * 2)
```

Is this what you expected? Are the values multiplied?

(b) Now try:

```
l + 5
```

What error do you get, and why?

(c) Try the same operations with NumPy:

```
a = np.array([1, 2, 3, 4])
print(a * 2)
print(a + 5)
```

Each element is processed individually. This is called an **element-wise operation** and is the foundation of image processing with NumPy.

(d) **Question.** In one sentence, explain why NumPy arrays are more useful than Python lists for numerical computations on images.

Exercise 2. Array Shape and Dimensions

Every NumPy array has a shape — a tuple describing how many elements it has in each dimension. In Mini-project A, a colour image has shape $(height, width, 3)$ and a grayscale image has shape $(height, width)$.

(a) Create a 1D array and inspect it:

```
a = np.array([10, 20, 30, 40, 50])
print(a.shape)   # how many elements?
print(a.dtype)   # what type are the values?
```

(b) Create a 2D array (a matrix) and check its shape:

```
m = np.array([[1, 2, 3],
              [4, 5, 6]])
print(m.shape)   # (rows, columns)
```

How many rows? How many columns?

(c) Create arrays pre-filled with values:

```
z = np.zeros((4, 5))
o = np.ones((3, 3))
f = np.full((3, 3), 128)
print(z)
print(o)
print(f)
```

What **`dtype`** does **`np.zeros`** use by default?

(d) In Mini-project A, all grayscale images use **`dtype=np.int16`**. Create one and verify:

```
gray = np.zeros((4, 6), dtype=np.int16)
print(gray.dtype)
print(gray.shape)
```

We use **`int16`** (not **`uint8`**) because intermediate computations in the blur may temporarily exceed the 0–255 range, and **`int16`** can hold negative values too.

Exercise 3. Indexing and Slicing

NumPy allows reading and writing individual elements or entire rectangular sub-regions with a compact syntax. This is the core tool for image manipulation: reading a pixel, extracting a region, or painting a zone.

(a) Access individual values in a 2D array:

```
m = np.array([[10, 20, 30],
              [40, 50, 60],
              [70, 80, 90]])

print(m[1, 1])   # row 1, column 1 => 50
print(m[0, 2])   # row 0, column 2 => 30
print(m[2, 0])   # row 2, column 0 => 70
```

Indexing starts at 0. **`m[row, col]`** — never **`m[col, row]`**.

(b) Access entire rows and columns using **`:`** (meaning "all"):

```
print(m[0, :])   # first row     => [10, 20, 30]
print(m[:, 1])   # second column => [20, 50, 80]
print(m[-1, :])  # last row      => [70, 80, 90]
```

(c) Extract a rectangular sub-region with **`m[a:b, c:d]`** (rows **`a`** to **`b-1`**, columns **`c`** to **`d-1`**):

```
sub = m[1:3, 1:3]
print(sub)
# expected:
# [[50, 60],
#  [80, 90]]
```

This is how you will extract a tile edge or a pattern from an image.

(d) **Write to a region.** Slices can appear on the left side of an assignment to overwrite a zone. This is how you will draw borders on images in Mini-project B:

```
m[0:2, 0:2] = 0
print(m)
```

Which values changed?

(e) **Write the slice expressions yourself.** Given **`a = np.arange(25).reshape(5, 5)`**, write the expression that extracts each of the following and verify by running it:

- the entire second row
- the last two columns
- a 3×3 block starting at row 1, column 1

Exercise 4. Element-Wise Operations and Comparing Arrays

NumPy operations apply to every element simultaneously, without loops. This is critical for image processing: you cannot afford a Python loop over every pixel of a large image.

(a) Arithmetic between two arrays of the same shape:

```
a = np.array([1, 2, 3])
b = np.array([10, 20, 30])

print(a + b)
print(a * b)
print(b - a)
```

(b) **Subtracting two image regions.**

```
img1 = np.array([[100, 150],
                 [200,  50]], dtype=np.int16)
img2 = np.array([[ 80, 160],
                 [210,  40]], dtype=np.int16)

diff = img1 - img2
print(diff)      # values can be negative
```

Why would this go wrong if we used **`dtype=np.uint8`** instead of **`int16`**? (Hint: what happens when you subtract 160 from 150 in an unsigned 8-bit integer?)

(c) **Absolute value.**

```
abs_diff = np.abs(img1 - img2)
print(abs_diff)   # all values ≥ 0
```

(d) **Mean.**

```
print(np.mean(abs_diff))

m = np.array([[1, 2, 3],
              [4, 5, 6]])
print(np.mean(m))         # mean of all 6 values
print(np.mean(m[0, :]))   # mean of first row only
```

What values do you expect?

(e) **Similarity score — write it yourself.** Using **`np.abs`**, **`np.mean`**, and the arrays below, compute a similarity score in $[0, 1]$ where 0 means identical and 1 means maximally different. You have seen all the pieces in parts (b)–(d); now combine them.

```
zone1 = np.array([[100, 120, 130]], dtype=np.int16)
zone2 = np.array([[ 90, 115, 135]], dtype=np.int16)

score = ...   # your code here
print(score)

# Also check your formula on these two cases:
# case 1: zone1 == zone2                => expected score: 0.0
# case 2: all-zeros vs all-255 (1×2)    => expected score: 1.0
```

Exercise 5. Creating and Drawing on Grayscale Images

A grayscale image is a 2D array of shape (*height*, *width*) with integer values from 0 (black) to 255 (white).

(a) Create a black 10×10 image and save it:

```
img = np.zeros((10, 10), dtype=np.uint8)
print(img)
save_image(img, "black.png")
```

Open **`black.png`**. What do you see?

(b) Create a vertical gradient — each row gets a brighter shade of grey:

```
img = np.zeros((10, 10), dtype=np.uint8)
for row in range(10):
    for col in range(10):
        img[row, col] = (row + 1) * 20
print(img)
save_image(img, "gradient.png")
```

Open **`gradient.png`**. Which direction is the gradient? What single change would make it go left to right instead?

(c) **Draw a rectangle outline — write it yourself.** Starting from a black 10×10 image, draw a white rectangle outline with interior corners at rows 2–7, columns 2–7. Use loops (one loop per edge). Save the result as **`box_loops.png`**.

(d) **Draw the same rectangle using slices.** Now repeat (c) using only four numpy slice assignments — no loops. Save the result as **`box_slices.png`** and verify it looks identical.

Hint: **`img[2, 2:8] = 255`** draws the top edge. Write the remaining three lines yourself.

This slice-based approach is how you will implement **`highlight_rectangle`** in Mini-project B.

Exercise 6. Colour Images

A colour image is a 3D array of shape (*height*, *width*, 3). The three values at each pixel are the red, green, and blue intensities, each between 0 and 255.

(a) Create a colour image (all black) and inspect a pixel:

```
img = np.zeros((250, 250, 3), dtype=np.uint8)
print(img.shape)
print(img[125, 125])
```

(b) Paint a gradient: red increases downward, blue increases rightward:

```
for row in range(250):
    for col in range(250):
        img[row, col] = [row, 0, col]
save_image(img, "gradient_rgb.png")
```

Open **`gradient_rgb.png`**. What colour is the top-left corner? The top-right? The bottom-left?

(c) **`img[:, :, 0]`** extracts all red values as a 2D array of shape **`(250, 250)`**. Write the expression to extract the blue channel and print its shape to verify.

(d) Set an entire channel to zero using a single slice assignment:

```
img[:, :, 1] = 0    # remove all green
save_image(img, "no_green.png")
```

(e) **Write the following yourself.** Starting from a fresh all-black 100×100 colour image:

- paint the top half (**`rows 0--49`**) pure red
- paint the bottom half (**`rows 50--99`**) pure blue
- set the pixel at **`(50, 50)`** to white **`[255, 255, 255]`**

Save the result as **`halves.png`**.