Objectif. Dans ces exercices, vous apprendrez à utiliser NumPy pour créer, inspecter et manipuler des tableaux. Les mêmes opérations apparaissent directement dans le Mini-projet A, où les tableaux représentent des images. Les comprendre aujourd'hui facilitera grandement l'approche du mini-projet.

Ces exercices sont autonomes. Vous n'avez besoin que de NumPy et de Pillow (pour sauvegarder les images). Vous n'avez pas besoin des fichiers du mini-projet.

Installation. Ouvrez Visual Studio Code, ouvrez un terminal et créez un environnement virtuel :

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

Activez l'environnement virtuel selon votre système d'exploitation :

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

Installez ensuite les bibliothèques nécessaires :

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

Créez un nouveau fichier **`exercises.py`** avec les deux lignes suivantes en tête, ainsi que la fonction **`save_image`**. Chaque extrait de code ci-dessous suppose que ces imports sont présents.

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

Pour sauvegarder un tableau en tant que fichier image, utilisez cette fonction utilitaire tout au long de la séance :

```python
def save_image(arr, filename):
    """Sauvegarde un tableau numpy en tant que fichier image."""
    Image.fromarray(arr.astype("uint8")).save(filename)
```

Pour exécuter le code, assurez-vous que votre environnement virtuel est activé, puis lancez :

```
python3 exercises.py
```

Exercice 1. Pourquoi NumPy ?

Les listes Python sont pratiques, mais elles ne supportent pas les opérations mathématiques dont nous avons besoin pour le traitement d'images.

(a) Exécutez le code suivant et observez le résultat :

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

Est-ce ce à quoi vous vous attendiez ? Les valeurs sont-elles multipliées ?

(b) Essayez maintenant :

```python
l + 5
```

Quelle erreur obtenez-vous, et pourquoi ?

(c) Essayez les mêmes opérations avec NumPy :

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

Chaque élément est traité individuellement. C'est ce qu'on appelle une **opération élément par élément** et c'est le fondement du traitement d'images avec NumPy.

(d) **Question.** En une phrase, expliquez pourquoi les tableaux NumPy sont plus utiles que les listes Python pour les calculs numériques sur des images.

Exercice 2. Forme et dimensions d'un tableau

Chaque tableau NumPy possède une forme — un tuple décrivant le nombre d'éléments dans chaque dimension. Dans le Mini-projet A, une image couleur a la forme $(hauteur, largeur, 3)$ et une image en niveaux de gris a la forme $(hauteur, largeur)$.

(a) Créez un tableau 1D et inspectez-le :

```
a = np.array([10, 20, 30, 40, 50])
print(a.shape)   # combien d'éléments ?
print(a.dtype)   # quel est le type des valeurs ?
```

(b) Créez un tableau 2D (une matrice) et vérifiez sa forme :

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

Combien de lignes ? Combien de colonnes ?

(c) Créez des tableaux pré-remplis avec des valeurs :

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

Quel **`dtype`** **`np.zeros`** utilise-t-il par défaut ?

(d) Dans le Mini-projet A, toutes les images en niveaux de gris utilisent **`dtype=np.int16`**. Créez-en une et vérifiez :

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

Nous utilisons **`int16`** (et non **`uint8`**) car les calculs intermédiaires du flou peuvent temporairement dépasser la plage 0–255, et **`int16`** peut également stocker des valeurs négatives.

Exercice 3. Indexation et découpage

NumPy permet de lire et d'écrire des éléments individuels ou des sous-régions rectangulaires entières avec une syntaxe compacte. C'est l'outil central de la manipulation d'images : lire un pixel, extraire une région ou colorier une zone.

(a) Accédez à des valeurs individuelles dans un tableau 2D :

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

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

L'indexation commence à 0. **`m[ligne, colonne]`** — jamais **`m[colonne, ligne]`**.

(b) Accédez à des lignes et colonnes entières avec **`:`** (signifiant « tout ») :

```
print(m[0, :])   # première ligne     => [10, 20, 30]
print(m[:, 1])   # deuxième colonne   => [20, 50, 80]
print(m[-1, :])  # dernière ligne     => [70, 80, 90]
```

(c) Extrayez une sous-région rectangulaire avec **`m[a:b, c:d]`** (lignes **`a`** à **`b-1`**, colonnes **`c`** à **`d-1`**) :

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

C'est ainsi que vous extrairez un bord de tuile ou un motif d'une image.

(d) **Écrire dans une région.** Les tranches peuvent apparaître à gauche d'une affectation pour écraser une zone. C'est ainsi que vous dessinerez des bordures sur des images dans le Mini-projet B :

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

Quelles valeurs ont changé ?

(e) **Écrivez vous-même les expressions de découpage.** Étant donné `a = np.arange(25).reshape(5, 5)`, écrivez l'expression qui extrait chacun des éléments suivants et vérifiez en l'exécutant :

- la deuxième ligne entière
- les deux dernières colonnes
- un bloc 3×3 commençant à la ligne 1, colonne 1

Exercice 4. Opérations élément par élément et comparaison de tableaux

Les opérations NumPy s'appliquent à chaque élément simultanément, sans boucles. C'est essentiel pour le traitement d'images : on ne peut pas se permettre une boucle Python sur chaque pixel d'une grande image.

(a) Arithmétique entre deux tableaux de même forme :

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

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

(b) **Soustraction de deux régions d'image.**

```python
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)      # les valeurs peuvent être négatives
```

Pourquoi cela poserait-il problème si on utilisait `dtype=np.uint8` au lieu de `int16` ? (Indice : que se passe-t-il lorsqu'on soustrait 160 de 150 dans un entier non signé sur 8 bits ?)

(c) **Valeur absolue.**

```python
abs_diff = np.abs(img1 - img2)
print(abs_diff)   # toutes les valeurs >= 0
```

(d) **Moyenne.**

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

m = np.array([[1, 2, 3],
              [4, 5, 6]])
print(np.mean(m))         # moyenne de toutes les 6 valeurs
print(np.mean(m[0, :]))   # moyenne de la première ligne seulement
```

Quelles valeurs attendez-vous ?

(e) **Score de similarité — écrivez-le vous-même.** En utilisant `np.abs`, `np.mean` et les tableaux ci-dessous, calculez un score de similarité dans $[0, 1]$ où 0 signifie identique et 1 signifie maximalement différent. Vous avez vu toutes les pièces dans les parties (b)–(d) ; combinez-les maintenant.

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

score = ...   # votre code ici
print(score)

# Vérifiez aussi votre formule sur ces deux cas :
# cas 1 : zone1 == zone2                => score attendu : 0.0
# cas 2 : tout-zéros vs tout-255 (1x2) => score attendu : 1.0
```

Exercice 5. Créer et dessiner sur des images en niveaux de gris

Une image en niveaux de gris est un tableau 2D de forme (*hauteur*, *largeur*) avec des valeurs entières de 0 (noir) à 255 (blanc).

(a) Créez une image noire 10×10 et sauvegardez-la :

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

Ouvrez **`black.png`**. Que voyez-vous ?

(b) Créez un dégradé vertical — chaque ligne reçoit une teinte de gris plus claire :

```python
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")
```

Ouvrez **`gradient.png`**. Dans quelle direction va le dégradé ? Quelle modification unique permettrait de le faire aller de gauche à droite ?

(c) **Dessinez le contour d'un rectangle — écrivez-le vous-même.** En partant d'une image noire 10×10, dessinez le contour d'un rectangle blanc avec les coins intérieurs aux lignes 2–7, colonnes 2–7. Utilisez des boucles (une boucle par bord). Sauvegardez le résultat sous **`box_loops.png`**.

(d) **Dessinez le même rectangle avec des tranches.** Répétez maintenant (c) en utilisant uniquement quatre affectations de tranches NumPy — sans boucles. Sauvegardez le résultat sous **`box_slices.png`** et vérifiez qu'il est identique.

Indice : **`img[2, 2:8] = 255`** dessine le bord supérieur. Écrivez vous-même les trois lignes restantes.

Cette approche par tranches est celle que vous utiliserez pour implémenter **`highlight_rectangle`** dans le Mini-projet B.

Exercice 6. Images couleur

Une image couleur est un tableau 3D de forme (*hauteur*, *largeur*, 3). Les trois valeurs de chaque pixel sont les intensités rouge, verte et bleue, chacune comprise entre 0 et 255.

(a) Créez une image couleur (tout en noir) et inspectez un pixel :

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

(b) Peignez un dégradé : le rouge augmente vers le bas, le bleu vers la droite :

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

Ouvrez **`gradient_rgb.png`**. Quelle couleur a le coin supérieur gauche ? Le coin supérieur droit ? Le coin inférieur gauche ?

(c) **`img[:, :, 0]`** extrait toutes les valeurs rouges sous forme d'un tableau 2D de forme **`(250, 250)`**. Écrivez l'expression pour extraire le canal bleu et affichez sa forme pour vérifier.

(d) Mettez un canal entier à zéro avec une seule affectation de tranche :

```
img[:, :, 1] = 0    # supprimer tout le vert
save_image(img, "no_green.png")
```

(e) **Écrivez vous-même ce qui suit.** En partant d'une nouvelle image couleur 100×100 entièrement noire :

- peignez la moitié supérieure (**`lignes 0--49`**) en rouge pur
- peignez la moitié inférieure (**`lignes 50--99`**) en bleu pur
- mettez le pixel en **`(50, 50)`** en blanc **`[255, 255, 255]`**

Sauvegardez le résultat sous **`halves.png`**.