Compléments

I. Tracé de courbes et représentations graphiques

Le langage Python possède une bibliothèque très puissante pour créer des représentations graphiques : matplotlib. Cette bibliothèque permet de tracer des courbes, des nuages de points, des histogrammes, et bien d'autres types de graphiques. Dans ce cours, nous allons nous concentrer sur le module pyplot de cette bibliothèque.

1) Importation et premiers tracés

Importation de pyplot

Pour utiliser les fonctions de tracé graphique, on importe le module `pyplot` de la bibliothèque `matplotlib` en lui donnant généralement l'alias `plt` :

```
import matplotlib.pyplot as plt
```

Définition 1 : *Tracé d'une courbe*

Pour tracer une courbe représentant une fonction $y = f(x)$, on utilise la fonction `plt.plot(x, y)` où :

- `x` est une liste contenant les abscisses des points
- `y` est une liste contenant les ordonnées correspondantes

Pour afficher le graphique, on utilise ensuite la fonction `plt.show()`.

Exemple : Traçons la courbe représentative de la fonction $f(x) = x^2$ sur l'intervalle $[-3, 3]$:

```
import matplotlib.pyplot as plt

# Création des listes de coordonnées
x = [i / 10 for i in range(-30, 31)]    # De -3.0 à 3.0 par pas de 0.1
y = [i ** 2 for i in x]

# Tracé de la courbe
plt.plot(x, y)
plt.show()
```

Remarque : Pour obtenir un tracé plus précis, il faut augmenter le nombre de points calculés. Plus les listes `x` et `y` contiennent de valeurs, plus la courbe sera lisse.

2) Personnalisation des graphiques

Fonctions de personnalisation

Voici les principales fonctions permettant de personnaliser un graphique :

- ⋆ `plt.title("Titre")` : ajoute un titre au graphique
- ⋆ `plt.xlabel("Abscisses")` : légende de l'axe des abscisses
- ⋆ `plt.ylabel("Ordonnées")` : légende de l'axe des ordonnées
- ⋆ `plt.grid()` : affiche une grille
- ⋆ `plt.legend()` : affiche la légende (si les courbes ont un label)

Exemple : Reprenons l'exemple précédent en ajoutant des éléments de personnalisation :

```
x = [i / 10 for i in range(-30, 31)]
y = [i ** 2 for i in x]

plt.plot(x, y, label="f(x) = x^2")
plt.title("Courbe représentative de f(x) = x^2")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid()
plt.legend()
plt.show()
```

3) Tracer plusieurs courbes

Définition 2 : *Superposition de courbes*

Pour tracer plusieurs courbes sur le même graphique, il suffit d'appeler plusieurs fois la fonction `plt.plot()` avant d'utiliser `plt.show()`.

Exemple : Traçons les courbes de $f(x) = x^2$ et $g(x) = x^3$ sur le même graphique :

```
x = [i / 10 for i in range(-30, 31)]
y1 = [i ** 2 for i in x]
y2 = [i ** 3 for i in x]

plt.plot(x, y1, label="f(x) = x^2", color="blue")
plt.plot(x, y2, label="g(x) = x^3", color="red")
plt.title("Comparaison de deux fonctions")
plt.xlabel("x")
plt.ylabel("y")
plt.grid()
plt.legend()
plt.show()
```

Paramètres de style

La fonction `plt.plot()` accepte plusieurs paramètres optionnels :

- ⋆ `color` ou `c` : couleur de la courbe (`"blue"`, `"red"`, `"green"`, etc.)
- ⋆ `linestyle` ou `ls` : style de ligne (`"-"` continu, `"--"` pointillés, `":"` points)
- ⋆ `linewidth` ou `lw` : épaisseur de la ligne
- ⋆ `marker` : type de marqueur pour les points (`"o"` cercle, `"s"` carré, `"^"` triangle)
- ⋆ `label` : étiquette pour la légende

Il est possible de combiner les paramètres de style en un seul paramètre écrit dans une chaîne de caractères. Par exemple les deux instructions suivantes sont équivalentes :

```
plt.plot(x, y, color="blue", linestyle="--", marker="o")
plt.plot(x, y, "b--o")
```

4) Autres types de graphiques

Définition 3 : *Nuage de points*

La fonction `plt.scatter(x, y)` permet de tracer un nuage de points au lieu d'une courbe continue.

Exemple : Créons un nuage de points représentant des mesures expérimentales :

```
# Données expérimentales
temps = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
temperature = [20, 22, 25, 29, 34, 40, 47, 55, 64, 74, 85]

plt.scatter(temps, temperature, color="red", marker="o")
plt.title("Evolution de la température")
plt.xlabel("Temps (min)")
plt.ylabel("Température (deg C)")
plt.grid()
plt.show()
```

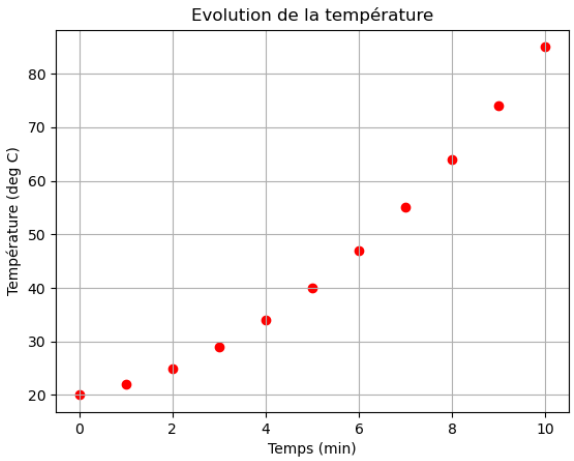

FIGURE 1 – Evolution de la température

Autres fonctions de tracé

- ⋆ `plt.bar(x, y)` : diagramme en bâtons (histogramme)
- ⋆ `plt.pie(valeurs, labels=etiquettes)` : diagramme circulaire (camembert)
- ⋆ `plt.hist(donnees, bins=n)` : histogramme de distribution

Exemple : Créons un diagramme en bâtons représentant des résultats d'élections :

```
import matplotlib.pyplot as plt

candidats = ["Candidat A", "Candidat B", "Candidat C", "Candidat D"]
voix = [234, 567, 432, 189]

plt.bar(candidats, voix, color=["blue", "red", "green", "orange"])
plt.title("Résultats des élections")
plt.ylabel("Nombre de voix")
plt.show()
```

Remarque : Pour enregistrer un graphique dans un fichier au lieu de l'afficher, on utilise `plt.savefig("nom_fichier.png")` à la place de `plt.show()`.

II. Lecture et écriture dans un fichier texte

La manipulation de fichiers est une opération fondamentale en informatique. Elle permet de sauvegarder des données de manière persistante et de les récupérer lors d'une prochaine exécution du programme.

1) Ouverture et fermeture de fichiers

Définition 4 : *Ouverture d'un fichier*

Pour ouvrir un fichier, on utilise la fonction `open()` qui prend deux paramètres principaux :

- ⋆ le nom du fichier (chemin relatif ou absolu)
- ⋆ le mode d'ouverture qui indique ce qu'on souhaite faire avec le fichier

La fonction renvoie un objet fichier qu'on peut manipuler.

Modes d'ouverture

Les principaux modes d'ouverture sont :

Mode	Description
`"r"`	Lecture seule (*read*) – mode par défaut
`"w"`	Écriture (*write*) – efface le contenu existant
`"a"`	Ajout (*append*) – ajoute à la fin sans effacer
`"r+"`	Lecture et écriture

Remarque : Après avoir terminé de manipuler un fichier, il est **impératif** de le fermer avec la méthode `close()` pour libérer les ressources.

Exemple : Ouverture et fermeture d'un fichier en lecture :

```
>>> fichier = open("donnees.txt", "r")
>>> # ... manipulation du fichier ...
>>> fichier.close()
```

2) Lecture de fichiers

Méthodes de lecture

Il existe trois méthodes principales pour lire le contenu d'un fichier :

- ⋆ `read()` : lit l'intégralité du fichier et renvoie une chaîne de caractères
- ⋆ `readline()` : lit une seule ligne et renvoie une chaîne de caractères
- ⋆ `readlines()` : lit toutes les lignes et renvoie une liste de chaînes

Exemple : Supposons que nous avons un fichier `notes.txt` contenant :

```
Alice: 15
Bob: 12
Charlie: 18
```

Lisons ce fichier avec `read()` :

```
>>> fichier = open("notes.txt", "r")
>>> contenu = fichier.read()
>>> print(contenu)
Alice: 15
Bob: 12
Charlie: 18
>>> fichier.close()
```

Exemple : Lisons le même fichier avec `readlines()` :

```
>>> fichier = open("notes.txt", "r")
>>> lignes = fichier.readlines()
>>> print(lignes)
['Alice: 15\n', 'Bob: 12\n', 'Charlie: 18\n']
>>> fichier.close()
```

Remarque : Les méthodes `readline()` et `readlines()` conservent le caractère de retour à la ligne `\n` à la fin de chaque ligne. Pour le supprimer, on peut utiliser la méthode `strip()` sur chaque chaîne.

Exemple : Traitement ligne par ligne avec suppression des retours à la ligne :

```
1 fichier = open("notes.txt", "r")
2 lignes = fichier.readlines()
3 fichier.close()
4
5 for ligne in lignes:
6     ligne_nettoyee = ligne.strip()  # Enlève \n
7     print(ligne_nettoyee)
```

Lecture ligne par ligne avec une boucle

Une méthode élégante pour parcourir un fichier consiste à itérer directement sur l'objet fichier :

```
1 fichier = open("notes.txt", "r")
2 for ligne in fichier:
3     print(ligne.strip())
4 fichier.close()
```

Cette méthode est plus économe en mémoire car elle ne charge pas tout le fichier en une fois.

3) Écriture dans des fichiers

Définition 5 : *Écriture dans un fichier*

Pour écrire dans un fichier, on utilise la méthode `write()` qui prend en paramètre une chaîne de caractères à écrire.

L'ouverture d'un fichier en mode `"w"` **efface tout son contenu** s'il existe déjà. Pour ajouter du contenu sans effacer, il faut utiliser le mode `"a"`.

Exemple : Créons un fichier et écrivons-y des données :

```
fichier = open("resultats.txt", "w")
fichier.write("Résultats des expériences\n")
fichier.write("=" * 30 + "\n")
fichier.write("Expérience 1: 45.3\n")
fichier.write("Expérience 2: 48.7\n")
fichier.close()
```

Remarque : La méthode `write()` n'ajoute **pas automatiquement** de retour à la ligne. Il faut explicitement écrire `\n` pour passer à la ligne suivante.

Exemple : Ajout de données à la fin d'un fichier existant avec le mode `"a"` :

```
fichier = open("resultats.txt", "a")
fichier.write("Expérience 3: 51.2\n")
fichier.close()
```

4) Gestion sécurisée avec `with`

Définition 6 : *Instruction `with`*

L'instruction `with` permet d'ouvrir un fichier de manière sécurisée. Le fichier est automatiquement fermé à la fin du bloc `with`, même en cas d'erreur.

Syntaxe de `with`

```
with open("fichier.txt", "mode") as variable:
    # Instructions utilisant la variable fichier
    # Le fichier est automatiquement fermé à la sortie du bloc
```

Exemple : Lecture d'un fichier avec `with` :

```
with open("notes.txt", "r") as fichier:
    contenu = fichier.read()
    print(contenu)
# Le fichier est automatiquement fermé ici
```

Exemple : Écriture dans un fichier avec `with` :

```
with open("rapport.txt", "w") as fichier:
    fichier.write("Début du rapport\n")
    for i in range(1, 6):
        fichier.write(f"Ligne {i}\n")
    fichier.write("Fin du rapport\n")
```

Remarque :

- ⋆ L'utilisation de `with` est considérée comme une **bonne pratique** car elle garantit la fermeture du fichier.
- ⋆ On peut imbriquer plusieurs `with` pour manipuler plusieurs fichiers simultanément.

Exemple : Copie du contenu d'un fichier dans un autre :

```
with open("source.txt", "r") as source:
    with open("destination.txt", "w") as dest:
        contenu = source.read()
        dest.write(contenu)
```

III. Manipulation d'images

Une image numérique peut être représentée comme un tableau de pixels. Chaque pixel possède une couleur définie par des valeurs numériques. Python permet de manipuler facilement ces images grâce aux bibliothèques NumPy et Matplotlib. Cette approche présente l'avantage de travailler directement avec des tableaux NumPy, ce qui permet d'effectuer des opérations matricielles très efficaces.

1) Représentation des couleurs

Définition 7 : *Pixel*

Un pixel est le plus petit élément d'une image numérique. Chaque pixel possède une couleur qui peut être représentée de différentes manières :

- ⋆ En niveaux de gris : une seule valeur entre 0 (noir) et 255 (blanc), ou entre 0.0 et 1.0
- ⋆ En RGB (Red, Green, Blue) : un triplet (r, g, b) où chaque composante varie entre 0 et 255, ou entre 0.0 et 1.0

Les valeurs sont des flottants entre 0.0 et 1.0 pour les images au format PNG et sont des entiers entre 0 et 255 pour tous les autres formats.

Exemple :

- ⋆ Le noir en RGB : $(0, 0, 0)$ ou $(0.0, 0.0, 0.0)$
- ⋆ Le blanc en RGB : $(255, 255, 255)$ ou $(1.0, 1.0, 1.0)$
- ⋆ Le rouge pur : $(255, 0, 0)$ ou $(1.0, 0.0, 0.0)$
- ⋆ Le vert pur : $(0, 255, 0)$ ou $(0.0, 1.0, 0.0)$
- ⋆ Le bleu pur : $(0, 0, 255)$ ou $(0.0, 0.0, 1.0)$
- ⋆ Le jaune : $(255, 255, 0)$ ou $(1.0, 1.0, 0.0)$
- ⋆ Le gris moyen : $(128, 128, 128)$ ou $(0.5, 0.5, 0.5)$

2) Représentation d'une image comme tableau

Définition 8 : *Image comme tableau NumPy*

Avec NumPy, une image est représentée comme un tableau multidimensionnel :

- ⋆ Une image en niveaux de gris est un tableau 2D de dimensions $(hauteur, largeur)$
- ⋆ Une image RGB est un tableau 3D de dimensions $(hauteur, largeur, 3)$

Chaque valeur est un nombre flottant entre 0.0 et 1.0, ou un entier entre 0 et 255 selon le format.

Remarque : Dans les tableaux NumPy, l'indexation suit l'ordre `[ligne, colonne]`, c'est-à-dire `[y, x]` où `y` représente la ligne (hauteur) et `x` la colonne (largeur). L'origine $(0, 0)$ correspond au coin supérieur gauche de l'image.

3) Importation et chargement d'images

Importation des bibliothèques

Pour manipuler des images avec NumPy et Matplotlib :

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

- ⋆ `numpy` permet de manipuler les tableaux représentant les images
- ⋆ `matplotlib.pyplot` permet de charger, afficher et sauvegarder les images

Définition 9 : *Chargement d'une image*

La fonction `plt.imread(nom_fichier)` charge une image et la renvoie sous forme de tableau NumPy.

Exemple : Chargement et affichage d'une image :

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # Chargement de l'image
5  image = plt.imread("photo.jpg")
6
7  # Affichage
8  plt.imshow(image)
9  plt.axis('off')  # Cache les axes
10 plt.show()
```

4) Propriétés d'une image

Attributs d'un tableau NumPy image

Un tableau NumPy représentant une image possède plusieurs attributs utiles :

- ⋆ `shape` : un tuple donnant les dimensions `(hauteur, largeur, canaux)`
- ⋆ `dtype` : le type des données (`uint8` pour entiers 0-255, `float64` pour flottants 0.0-1.0)
- ⋆ `size` : le nombre total d'éléments dans le tableau

Exemple : Obtention des informations d'une image :

```
>>> img = plt.imread("photo.jpg")
>>> img.shape
(600, 800, 3)
>>> img.dtype
dtype('float32')
>>> hauteur, largeur, canaux = img.shape
>>> print(f"Dimensions : {largeur} x {hauteur}, {canaux} canaux")
Dimensions : 800 x 600, 3 canaux
```

Remarque : Pour une image en niveaux de gris, `shape` ne renvoie que deux valeurs : `(hauteur, largeur)`.

5) Création d'une image simple

Définition 10 : *Création d'un tableau image*

Pour créer une nouvelle image, on utilise les fonctions NumPy :

- ⋆ `np.zeros((hauteur, largeur, 3))` : crée une image noire
- ⋆ `np.ones((hauteur, largeur, 3))` : crée une image blanche (si on multiplie par 255 ou 1.0)
- ⋆ `np.full((hauteur, largeur, 3), couleur)` : crée une image d'une couleur uniforme

Exemple : Créons une image rouge de 150×200 pixels :

```
# Création d'une image rouge (valeurs entre 0 et 1)
img = np.zeros((150, 200, 3))
img[:, :, 0] = 1.0  # Canal rouge à 1.0

# Affichage
plt.imshow(img)
plt.axis('off')
plt.show()
```

Exemple : Autre méthode pour créer une image d'une couleur spécifique :

```
# Création d'une image jaune
hauteur, largeur = 150, 200
img = np.ones((hauteur, largeur, 3))
img[:, :, 0] = 1.0  # Rouge
img[:, :, 1] = 1.0  # Vert
img[:, :, 2] = 0.0  # Bleu

plt.imshow(img)
plt.axis('off')
plt.show()
```

6) Manipulation des pixels

Accès aux pixels avec NumPy

Pour accéder ou modifier la couleur d'un pixel, on utilise l'indexation des tableaux :

- ⋆ `img[y, x]` : renvoie la couleur du pixel à la ligne `y` et colonne `x`
- ⋆ `img[y, x] = [r, g, b]` : modifie la couleur du pixel
- ⋆ `img[y, x, canal]` : accède à un canal spécifique (0=rouge, 1=vert, 2=bleu)

Attention : l'origine $(0, 0)$ est en haut à gauche, l'indexation est `[ligne, colonne]`.

Exemple : Lecture de la couleur d'un pixel :

```
>>> img = plt.imread("photo.jpg")
>>> couleur = img[50, 100]  # Ligne 50, colonne 100
>>> print(couleur)
[0.918 0.569 0.263]
```

Exemple : Création d'une image avec un dégradé horizontal :

```
# Création d'une image vide
largeur = 256
hauteur = 100
img = np.zeros((hauteur, largeur, 3))

# Création du dégradé (méthode efficace avec NumPy)
for x in range(largeur):
    img[:, x, 0] = x / 255.0  # Canal rouge varie de 0 à 1

plt.imshow(img)
plt.axis('off')
plt.show()
```

Remarque : On peut créer le même dégradé de manière encore plus efficace avec le broadcasting NumPy :

```
img[:, :, 0] = np.linspace(0, 1, largeur)
```

7) Conversions d'images

Définition 11 : *Conversion en niveaux de gris*

Pour convertir une image RGB en niveaux de gris, on utilise la formule standard :

$$\text{gris} = 0.2989 \times R + 0.5870 \times G + 0.1140 \times B$$

Ces coefficients correspondent à la perception de luminosité par l'œil humain.

Exemple : Conversion d'une image couleur en niveaux de gris :

```
# Chargement de l'image couleur
img_couleur = plt.imread("photo.jpg")

# Conversion en niveaux de gris
img_gris = 0.2989 * img_couleur[:, :, 0] + \
           0.5870 * img_couleur[:, :, 1] + \
           0.1140 * img_couleur[:, :, 2]

# Affichage côte à côte
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(img_couleur)
plt.title("Image couleur")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(img_gris, cmap='gray')
plt.title("Image en niveaux de gris")
plt.axis('off')

plt.show()
```

Remarque : On peut aussi utiliser la méthode `np.dot()` pour une conversion plus concise :

```
coeffs = np.array([0.2989, 0.5870, 0.1140])
img_gris = np.dot(img_couleur, coeffs)
```

8) Enregistrement d'images

Définition 12 : *Sauvegarde d'une image*

La fonction `plt.imsave(nom_fichier, tableau)` permet d'enregistrer un tableau NumPy comme image. Le format est déterminé automatiquement par l'extension du fichier.

Exemple : Création et sauvegarde d'une image avec un motif géométrique :

```
# Création d'une image carrée blanche
taille = 200
img = np.ones((taille, taille, 3))

# Dessin d'un damier
taille_case = 20
for i in range(0, taille, taille_case):
    for j in range(0, taille, taille_case):
        if ((i // taille_case) + (j // taille_case)) % 2 == 0:
            img[i:i+taille_case, j:j+taille_case] = [0, 0, 0]

# Sauvegarde
plt.imsave("damier.png", img)
```

Remarque : L'avantage de NumPy est qu'on peut utiliser le slicing pour modifier des blocs de pixels en une seule opération, ce qui est beaucoup plus rapide que la modification pixel par pixel.