RÉPUBLIQUE TUNISIENNE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ DE CARTHAGE
Institut Préparatoire aux Études d'Ingénieur de Nabeul

INFORMATIQUE

Programmation Python

1ère ANNÉE MP – PT – PC

Manuel de cours et de Travaux Dirigés pour la 1ère année des classes préparatoires

AUTEUR
Anis SAIED
anis.saied@hotmail.com

ANNÉE UNIVERSITAIRE
2022 – 2023

Introduction

Ce manuel s'adresse aux étudiants de première année des classes préparatoires et a été conçu dans une volonté simple : offrir un support pédagogique unique, cohérent et progressif pour maîtriser les bases de l'algorithmique et de la programmation en Python. Il vise à aider les étudiants à réussir leurs examens tout au long de l'année et à acquérir les connaissances nécessaires pour une préparation efficace au concours de deuxième année. Il contient des résumés de cours, des TD ainsi que des remarques et notes importantes pour guider l'apprentissage.

L'objectif principal de ce manuel est :

- de maîtriser progressivement les concepts fondamentaux de la programmation en Python ;
- de savoir modéliser un problème scientifique et traduire une solution algorithmique en un programme Python clair et efficace.

Organisation du fascicule

Le fascicule est structuré en deux parties complémentaires, permettant une progression graduelle des notions de base vers des applications plus avancées.

Partie 1 — Programmation Python : notions de base

Cette première partie est consacrée à l'apprentissage des fondements de la programmation en Python. Elle aborde l'environnement de développement, les structures de contrôle, les types de données, les fonctions, la gestion des erreurs, la récursivité, ainsi que les algorithmes classiques de tri et de recherche et le calcul de leurs complexités et enfin la manipulation des fichiers. Les exercices proposés permettent d'acquérir de bonnes pratiques de programmation et de passer progressivement de l'algorithmique à l'implémentation en Python.

Partie 2 — Programmation Python : simulation numérique

La seconde partie est dédiée aux applications numériques et scientifiques. Elle introduit l'utilisation des bibliothèques Python adaptées au calcul scientifique, notamment pour l'algèbre linéaire, la résolution numérique des équations, l'intégration numérique et la résolution des équations différentielles. Cette partie vise à renforcer le lien entre les outils de programmation et les problèmes mathématiques rencontrés en classes préparatoires.

Démarche pédagogique

Les Travaux Dirigés sont organisés de manière progressive et cohérente. Chaque TD commence par une présentation des objectifs pédagogiques et des notions abordées, suivie d'exercices classés par difficulté croissante.

Avant-propos

Le manuel est organisé pour permettre une progression graduelle, allant des notions fondamentales aux applications plus avancées, et pour développer chez l'étudiant une démarche rigoureuse de résolution de problèmes. conçus pour encourager le travail autonome et stimuler la réflexion méthodique, sans se limiter à la simple consultation des réponses.

Ainsi, cet ouvrage constitue un outil complet pour acquérir les compétences nécessaires en algorithmique et en programmation Python, tout en développant l'autonomie et la rigueur indispensables à la réussite en classes préparatoires.

Ce travail a été réalisé avec plaisir, et je reste entièrement ouvert à toute critique ou suggestion, que vous pouvez m'adresser à l'adresse `anis.saied@ipein.ucar.tn`. Le manuel ainsi que les corrigés sont également mis en libre accès sur mon site personnel :
`https://anis-saied.com`

Remerciements

Je tiens à remercier mes collègues ainsi que les nombreux enseignants et contributeurs de la communauté *Open Source*, dont les ressources en ligne enrichissent quotidiennement la pédagogie de l'informatique. Je remercie également la direction de l'IPEIN pour avoir accepté ce manuel dans sa bibliothèque et l'avoir rendu accessible à tous les étudiants. Enfin, je souhaite exprimer ma gratitude aux étudiants les plus brillants rencontrés au cours de mes années d'enseignement, qui ont critiqué de manière constructive mes supports de cours et de TD et partagé des idées innovantes lors des séances d'informatique. Ce partage de connaissances et cette collaboration sont le moteur de ce manuel.

Légende

★ Niveau Débutant : Maîtrise du cours immédiate.

★★ Niveau Intermédiaire : Nécessite une réflexion algorithmique.

★★★ Niveau Avancé : Problèmes de concours.

Table des matières

Introduction ii

Avant-propos iv

Liste des algorithmes xi

I Programmation Python 1

Chp. 1 Découverte de Python 2

1.1 L'interpréteur Python (Le Shell) . . . 2
1.2 Variables et affectation . . . 3
1.3 Modules et fonctions mathématiques . . . 3
1.4 Écriture de scripts et Entrées/Sorties . . . 4

Chp. 2 Les structures conditionnelles 5

2.1 Principe et syntaxe en Python . . . 5
2.2 Exercices d'application . . . 6

Chp. 3 Les structures itératives 7

3.1 La boucle `for ... in range(...)` . . . 7
3.2 La boucle `while` . . . 7
3.3 Contrôle de boucle : `break` et `continue` . . . 8
3.4 Exercices d'application . . . 8

Chp. 4 Structures de Données mutables : Listes, Ensembles et Dictionnaires 10

4.1 Les listes . . . 10

4.1.1 Définition et Manipulation de base 10
4.1.2 Fonctions et méthodes associées 11
4.1.3 Itération et Slicing 12
4.1.4 Le problème de la copie 13
4.2 Ensembles et Dictionnaires 13
4.2.1 Les Ensembles : Définition et Opérations 13
4.2.2 Les Dictionnaires : Concepts fondamentaux 14

Chp. 5 Structures de Données non mutables : Tuples et Chaînes de Caractères 16

5.1 Les Tuples 16
5.1.1 Définition et Propriétés 16
5.2 Les chaînes de caractères 18
5.2.1 Création et Codage 18

Chp. 6 Les fonctions en Python 20

6.1 Déclaration et Syntaxe 20
6.2 Passage de paramètres et Mutabilité 21
6.3 Analyse de nombres et logique 21
6.4 La fonction comme objet de première classe 22
6.5 Les expressions Lambda 22
6.6 Traitement de séquences : Map et Filter 23
6.6.1 La fonction map() 23
6.6.2 La fonction filter() 23

Chp. 7 La gestion des exceptions 25

7.1 Comprendre et lever des exceptions 25
7.2 Intercepter les erreurs : le bloc `try-except` 26
7.3 Gestion avancée et clauses optionnelles 26
7.4 Application de synthèse 27

Chp. 8 Les fonctions récursives 29

8.1 Concepts fondamentaux 29
8.2 Implémentations de suites mathématiques 30
8.3 Récursivité sur les listes 31

Chp. 9 Les algorithmes de tri **32**
9.1 Le Tri à bulles (Bubble Sort) . . . 32
9.2 Tri par Sélection . . . 33
9.3 Tri par Insertion . . . 33
9.4 Tri Rapide (Quicksort) . . . 34

Chp. 10 Les algorithmes de recherche **36**
10.1 Recherche séquentielle (ou linéaire) . . . 36
10.2 Recherche dichotomique . . . 37

Chp. 11 Manipulation des fichiers texte **38**
11.1 Lecture de fichiers . . . 38
11.2 Écriture et Modification . . . 39
11.3 Gestion des chemins d'accès . . . 39

II Programmation Python **41**

Chp. 12 Calcul scientifique : NumPy & Matplotlib **42**
12.1 Présentation du module NumPy . . . 42
12.2 Création des tableaux (Vecteurs et Matrices) . . . 43
12.2.1 Création à partir de listes . . . 43
12.2.2 Fonctions de création spécialisées . . . 43
12.2.3 Construction par formule (`fromfunction`) . . . 44
12.3 Lecture et Slicing (Vecteurs et Matrices) . . . 45
12.4 Écriture, Modification et Réduction . . . 45
12.4.1 Mutabilité et modification en masse . . . 45
12.4.2 Opérations de réduction selon les axes . . . 46
12.5 Fonctions Universelles et Vectorisation . . . 46
12.6 Les Fonctions Universelles (ufunc) . . . 46
12.7 Opérations Arithmétiques Terme à Terme . . . 47
12.8 Fonctions Mathématiques Usuelles . . . 47
12.9 Vectorisation de fonctions personnalisées . . . 48
12.10 Série d'exercices . . . 49
12.11 Calcul Matriciel et Algèbre Linéaire . . . 50

12.12 Produits matriciels et vectoriels 50
12.12.1 Produit de matrices 50
12.12.2 Produit scalaire et vectoriel 51
12.13 Inversion et résolution de systèmes 51
12.14 Normes matricielles et vectorielles 52
12.15 Exercices d'application 52
12.16 Visualisation Graphique : Matplotlib 53
12.17 Présentation et fonctions de base 54
12.17.1 La fonction `plot()` 54
12.17.2 Styles et Multi-courbes 55
12.17.3 Manipuler plusieurs graphiques 55
12.17.4 Graphiques à 3 dimensions : Les courbes 56
12.17.5 Options de personnalisation 56
12.17.6 Série d'exercices 57

Chp. 13 Résolution numérique d'équations $f(x) = 0$ 60

13.1 Introduction 60
13.2 La méthode de dichotomie 61
13.2.1 Principe et Théorème 61
13.2.2 Algorithme et Implémentation 61
13.3 Méthode de Newton 62
13.3.1 Principe de la tangente 62
13.3.2 Implémentation en Python 62
13.4 Outils Scipy 63

Chp. 14 Intégration numérique : Rectangles et Trapèzes 64

14.1 Introduction 64
14.2 La méthode des rectangles 65
14.2.1 Principe 65
14.2.2 Algorithme et Implémentation 65
14.3 La méthode des trapèzes 66
14.3.1 Principe 66
14.3.2 Algorithme Python 66
14.4 Déjà disponible en Python 67

Chp. 15 Résolution des Équations Différentielles : Méthode d'Euler **68**

15.1 Définitions Fondamentales 68
15.2 Le Problème de Cauchy 69
15.3 La Méthode d'Euler 69
15.3.1 Algorithme de la méthode d'Euler 69
15.4 Utilisation de `scipy.integrate.odeint` 70
15.5 Applications et Approfondissements 70

Liste des algorithmes

Algo 1 Algorithme du tri à bulles 32
Algo 2 Algorithme du tri par sélection 33
Algo 3 Algorithme du tri par insertion 34
Algo 4 Algorithme de dichotomie 61
Algo 5 Méthode des rectangles à gauche 65
Algo 6 Méthode d'Euler 69

PARTIE I

Programmation Python

Notions de base

Chp. 1. Découverte de Python . . . . . p. 2
Chp. 2. Les structures conditionnelles . . . . . p. 5
Chp. 3. Les structures itératives . . . . . p. 7
Chp. 4. Structures de données mutables . . . . . p. 10
Chp. 5. Structures de données non mutables . . . . . p. 16
Chp. 6. Les fonctions . . . . . p. 20
Chp. 7. La gestion des exceptions . . . . . p. 25
Chp. 8. La récursivité . . . . . p. 29
Chp. 9. Les algorithmes de tri . . . . . p. 32
Chp. 10. Les algorithmes de recherche . . . . . p. 36
Chp. 11. Manipulation des fichiers texte . . . . . p. 38

Chp. 1

Découverte de Python

❒ Objectifs

Ce premier chapitre a pour objectif de faire connaissance avec le langage **Python 3** et son environnement de travail. À l'issue de ce TD, vous devrez savoir :

- ✦ Utiliser la console comme une calculatrice perfectionnée.
- ✦ Manipuler des variables et comprendre l'affectation.
- ✦ Importer des fonctions depuis des modules standards (`math`).
- ✦ Écrire, sauvegarder et exécuter votre premier script.

1.1 L'interpréteur Python (Le Shell)

L'interpréteur Python permet d'exécuter immédiatement des instructions. Après le symbole `>>>`, l'utilisateur saisit des commandes évaluées instantanément.

Exercice 1.1 — Utilisation de la console ★

1. Ouvrez l'IDLE Python et calculez les expressions suivantes :

   Somme : $54 + 23$

   Produit : 123×87

   Division : $34/5$

2. Testez les opérateurs suivants dans la console et déduisez-en leur rôle :

```
>>> 17 // 3
>>> 17 % 3
>>> 2 ** 10
```

3. Est-il possible d'effectuer plusieurs calculs sur une même ligne en utilisant le séparateur `;` ?

1.2 Variables et affectation

Une variable est une étiquette collée sur une valeur stockée en mémoire. L'attribution d'une valeur se fait avec le symbole `=`

Exercice 1.2 — Manipulation des variables ★★

1. Créez une variable `x` contenant la valeur 3.
2. Créez une variable `y` égale à $7 \times x$ et affichez-la.
3. Modifiez la valeur de `x`. La valeur de `y` est-elle mise à jour automatiquement ? Expliquez.
4. Utilisez la fonction `type(x)` pour vérifier la nature de votre variable.
5. **Astuce Pythonique :** Testez l'instruction `a, b = 1, 2`.
   Que contient les variables `a` et `b` ?

☞ Remarque Importante

En Python, le signe = n'est pas une égalité mathématique mais une **affectation**.
L'instruction `x = x + 1` est donc parfaitement valide.

❷ Astuce

Utilisez `x += 1` comme raccourci pour l'incrémentation.

1.3 Modules et fonctions mathématiques

Exercice 1.3 — Bibliothèque standard ★★

1. Tapez `import math` dans la console.
2. Calculez $\sqrt{16}$ en utilisant `math.sqrt(16)`.

3. Calculez la valeur de $\sin(\pi/4)$. *Note : π est accessible via* `math.pi`.
4. Utilisez `help(math)` pour voir la liste des fonctions disponibles.

1.4 Écriture de scripts et Entrées/Sorties

Exercice 1.4 — Mon premier script ★

Écrivez un script nommé `bonjour.py` qui demande le prénom de l'utilisateur et lui répond "Bonjour [Prénom]".
Indice : utilisez la fonction `input()`.

Exercice 1.5 — Calcul de prix ★★

Réalisez un programme qui demande à l'utilisateur :

- Le prix hors taxes (HT) d'un article.
- Le taux de TVA (en pourcentage).

Le programme doit afficher le prix toutes taxes comprises (TTC) avec un message clair.

Chp. 2

Les structures conditionnelles

❒ Objectifs

Ce TD aborde les mécanismes de prise de décision en Python. À la fin de cette séance, vous devrez maîtriser :

- La syntaxe des blocs `if`, `elif` et `else`.
- Le concept d'indentation pour délimiter les blocs de code.
- L'utilisation des opérateurs logiques et des booléens.

2.1 Principe et syntaxe en Python

Une **instruction conditionnelle** permet d'exécuter des instructions seulement si une condition est remplie. En Python, on utilise les mots clés `if`, `else` et `elif`.

```
if C:
    P1  # Bloc exécuté si la condition C est vraie
else:
    P2  # Bloc exécuté si la condition C est faux
```

☞ Remarque sur l'indentation

Indenter signifie décaler vers la droite. En Python, cela définit la structure du code : les instructions ayant le même décalage appartiennent au même **bloc**.

2.2 Exercices d'application

Exercice 2.1 — Indice de Masse Corporelle ★★

L'indice de masse corporelle est calculé par la formule : $IMC = \frac{\text{poids}}{\text{taille}^2}$.

1. Créez un fichier intitulé `test_poids.py`. Définissez deux variables `taille` (en m) et `poids` (en kg) affectées avec vos propres valeurs.
2. Écrire un script qui calcule l'IMC, le stocke dans une variable `imc`, et affiche :
   - "Surpoids" si $IMC > 25$.
   - "Sous-poids" si $IMC < 18$.
   - "Normal" sinon.
3. Testez votre programme pour un individu de $1,80$ m et 50 kg, puis $1,50$ m et 100 kg.

Exercice 2.2 — Équation du second degré ★★★

On considère l'équation $ax^2 + bx + c = 0$, où a, b, c sont des entiers.

1. Écrire un script qui demande les coefficients à l'utilisateur et calcule le discriminant $\Delta = b^2 - 4ac$.
2. En fonction de Δ, affichez le nombre de solutions et leurs valeurs numériques.
3. **Cas complexe :** Si $\Delta < 0$, le programme doit afficher les deux racines complexes. *Indice : vous pouvez utiliser* `import cmath` *pour gérer les racines de nombres négatifs.*
4. Testez le programme avec les triplets (a, b, c) suivants : $(1, 2, -3)$, $(1, -2, 1)$ et $(2, -2, 1)$.

Chp. 3

Les structures itératives

❒ Objectifs

Ce TD explore les répétitions d'instructions (boucles). À la fin de cette séance, vous devrez maîtriser :

- ✦ La boucle `for` pour les répétitions déterminées (bornées).
- ✦ La fonction `range()` et ses paramètres.
- ✦ La boucle `while` pour les répétitions conditionnelles.
- ✦ Les instructions de contrôle de flux : `break` et `continue`.

3.1 La boucle `for ... in range(...)`

La boucle `for` est une boucle **inconditionnelle** (ou bornée). Elle sert à exécuter un bloc d'instructions un nombre connu de fois.

[Image of for loop flowchart in Python]

```
for compteur in range(debut, fin, pas):
    # bloc d'instructions
```

- `range(d, f, p)` génère une séquence de `d` à `f` (exclu) avec un pas `p`.
- Par défaut, `debut = 0` et `pas = 1`.

3.2 La boucle `while`

La boucle `while` est une boucle **conditionnelle**. Elle répète des instructions tant qu'une expression logique est vraie (`True`).

```python
while condition:
    # bloc d'instructions
```

☞ Attention à la boucle infinie

Avec `while`, assurez-vous que la condition devienne fausse à un moment donné, sinon le programme ne s'arrêtera jamais !

3.3 Contrôle de boucle : `break` et `continue`

- `break` : Sort immédiatement de la boucle.
- `continue` : Arrête l'itération actuelle et passe directement à la suivante.

3.4 Exercices d'application

Exercice 3.1 — Pratique de la boucle `for` ★

1. Afficher tous les multiples de 7 inférieurs à 100.
2. Calculer la somme des carrés de tous les entiers de 1 à 1000.
3. Écrire un programme qui affiche sur 10 lignes la table des carrés des 10 premiers entiers non-nuls.
4. Calculer et afficher $\sum_{k=p}^{n^3} \frac{1}{k}$, où p et n sont saisis par l'utilisateur.

Exercice 3.2 — Suites de Mycielski ★★

Les suites de Mycielski sont définies par :

$$\begin{cases} m_1 = 2, & m_k = 2m_{k-1} + 1, & k \geq 2, \\ c_1 = 1, & c_k = 3c_{k-1} + m_{k-1}, & k \geq 2. \end{cases}$$

Écrire un script `mycielski.py` qui saisit $n > 0$, puis calcule et affiche c_n.

Exercice 3.3 — Diviseurs avec `while` ★★

1. Écrire un programme demandant un entier N, qui affiche tous les diviseurs de N. (Tester avec $N = 540$).
2. Écrire un programme demandant un entier $N \geq 2$ et qui affiche son plus petit diviseur strictement supérieur à 1.

Exercice 3.4 — La Persistance ★★★

La *persistance* d'un nombre est le nombre de fois où l'on doit multiplier ses chiffres entre eux avant d'obtenir un seul chiffre.
Exemple : $6788 \longrightarrow 2688 \longrightarrow 768 \longrightarrow 336 \longrightarrow 54 \longrightarrow 20 \longrightarrow 0$ (Persistance = 6).
Écrire un script `persistance.py` qui calcule la persistance d'un entier $n > 0$.

Chp. 4

Structures de Données mutables : Listes, Ensembles et Dictionnaires

4.1 Les listes

❒ Objectifs

Une *liste* est une séquence ordonnée et modifiable d'objets. À la fin de cette section, vous devrez savoir :

- ✦ Créer, indexer et modifier une liste.
- ✦ Utiliser les méthodes principales (`append`, `pop`, `sort`).
- ✦ Maîtriser le *slicing* et la définition par compréhension.
- ✦ Différencier copie superficielle et copie profonde (`deepcopy`).

4.1.1 Définition et Manipulation de base

Une liste appartient au type `list`. Elle est délimitée par des crochets `[ ]`.

- **Indexation :** Les éléments sont numérotés de 0 à $n-1$.
- **Indexation négative :** $L[-1]$ désigne le dernier élément.
- **Mutabilité :** Contrairement aux chaînes de caractères, les listes sont modifiables (`L[i] = x`).

Création de listes

```python
L1 = []                          # Liste vide
L2 = [1, 2, 3]                   # Liste d'entiers
L3 = ["a", 2.5, [1, 2]]          # Types hétérogènes et listes imbriquées
# Création par compréhension
L4 = [i**2 for i in range(11) if i % 2 == 0]
```

Exercice 4.1 — Premiers pas avec les listes ★

1. Créer la liste `L = [2, 0, 5, [7, 6, 17], 4]`.
   Remplacez le 0 par 8, puis le 17 par 22.
2. Construire par compréhension la liste des multiples de 7 entre 1 et 100.

Exercice 4.2 — Inversion et Rotation ★

1. Écrire un script qui inverse une liste sans utiliser la méthode `reverse()` ou le slicing.
2. Réaliser une rotation à gauche des éléments d'une liste (le premier devient le dernier).

4.1.2 Fonctions et méthodes associées

- `L.append(x)` : Ajoute x à la fin.
- `L.insert(i, x)` : Insère x à l'indice i.
- `L.pop()` : Supprime et renvoie le dernier élément.
- `len(L)` : Renvoie la taille de la liste.
- `L.sort()` : Trie la liste sur place.

Exercice 4.3 — Manipulation de listes ★★

1. À partir d'un entier $n = 1234$, construire la liste `L = [1, 2, 3, 4]` sans utiliser de conversion en chaîne de caractères.

2. Écrire un programme qui saisit 5 entiers positifs et les affiche triés par ordre décroissant.

4.1.3 Itération et Slicing

La syntaxe `L[début:fin:pas]` permet d'extraire une sous-partie d'une liste.

Exemples de Slicing :

```
L = [10, 20, 30, 40, 50, 60]

sub1 = L[1:4]    # De l'indice 1 à 3 : [20, 30, 40]
sub2 = L[:3]     # Les 3 premiers éléments : [10, 20, 30]
sub3 = L[::2]    # Un élément sur deux : [10, 30, 50]
inv  = L[::-1]   # Inversion de la liste : [60, 50, 40, 30, 20, 10]
```

Exemples d'Itérations :

Il existe deux manières classiques de parcourir une liste :

```
# Parcours direct (par valeur)
for x in L:
    print(x)

# Parcours par indice (utile pour modifier la liste)
for i in range(len(L)):
    print(f"Indice {i} : valeur {L[i]}")
```

Exercice 4.4 — Opérations fondamentales sur les listes ★★

1. Écrire un script qui calcule le produit de tous les éléments d'une liste de nombres.
2. Écrire un programme qui, à partir d'une liste L, renvoie une liste contenant deux sous-listes : celle des nombres pairs et celle des nombres impairs.

Exercice 4.5 — Crible d'Érathostène ★★★

En utilisant une liste de booléens initialisée à `True` par compréhension, implémentez l'algorithme du crible pour trouver tous les nombres premiers inférieurs à 100.

4.1.4 Le problème de la copie

☞ Attention aux références

L'instruction `L2 = L1` ne copie pas la liste, elle donne un deuxième nom au **même objet**.

Exercice 4.6 — Remplacement ★★★

Écrire un programme qui prend une liste L et deux valeurs x_1, x_2, et renvoie une **copie** de L où toutes les occurrences de x_1 ont été remplacées par x_2.

4.2 Ensembles et Dictionnaires

❒ Objectifs

Cette section traite des structures de données non-linéaires : les `set` et les `dict`. Vous devrez savoir :

- ✦ Gérer l'unicité des données avec les ensembles.
- ✦ Réaliser des opérations mathématiques (Union, Intersection).
- ✦ Manipuler des couples clés-valeurs avec les dictionnaires.

4.2.1 Les Ensembles : Définition et Opérations

Les ensembles sont entourés d'accolades `{ }`. Chaque élément est unique et non indexé.

Exercice 4.7 — Analyse de caractères ★

Écrire un programme qui, à partir d'une chaîne de caractères `s`, crée un ensemble `lettres` contenant toutes les lettres uniques utilisées.

Exercice 4.8 — Manipulation de doublons ★★

1. Soit une liste `L`. Supprimez tous les doublons de `L` en une seule ligne de code.
2. Soit une liste de chaînes. On veut supprimer les doublons sans tenir compte de la casse. Proposez une solution utilisant les ensembles.

Exercice 4.9 — Logique ensembliste ★★

1. Vérifier rapidement si un mot saisi par l'utilisateur contient des voyelles.
2. **Produit Cartésien :** Soient deux ensembles E et F. Écrire un programme qui génère l'ensemble $E \times F$.

Exercice 4.10 — Ensembles de caractères communs ★★

Écrire un programme qui demande deux mots à l'utilisateur et affiche :

- Les lettres présentes dans les deux mots.
- Les lettres présentes dans le premier mot mais pas dans le second.
- L'ensemble total des lettres utilisées par les deux mots.

Exercice 4.11 — Synthèse Ensembles ★★★

À partir d'une liste d'entiers, créer une liste contenant deux ensembles : celui des nombres pairs et celui des nombres impairs présents.

4.2.2 Les Dictionnaires : Concepts fondamentaux

Le dictionnaire remplace les indices numériques par des **clés**.

Exercice 4.12 — Gestion de base de données scolaire ★★

1. Créer un dictionnaire `matieres` contenant des sous-dictionnaires précisant `coeff` et `nb_heures`.
2. Ajouter un étudiant nommé "Sami", âgé de 19 ans, à une liste d'étudiants imbriquée.
3. Afficher uniquement les matières dont le coefficient est supérieur ou égal à 10.

Exercice 4.13 — Inversion de dictionnaire ★★

Écrire un programme qui échange les clés et les valeurs d'un dictionnaire (ex : transformer Anglais-Français en Français-Anglais).

Exercice 4.14 — Calcul de panier ★★

Soit un dictionnaire `stock = "pommes": 10, "bananes": 5, "oranges": 8` et un dictionnaire `prix = "pommes": 2.5, "bananes": 1.8, "oranges": 3.0`. Calculez la valeur totale du stock.

Exercice 4.15 — Mini Base de Données ★★

Créer un script qui demande des noms, âges et tailles, puis permet de consulter les informations formatées à partir d'un nom saisi.

Exercice 4.16 — Analyse de texte ★★★

1. Écrire un script qui compte le nombre d'occurrences de chaque lettre dans un texte.
2. **Indexation :** Stocker, pour chaque mot du texte, la liste de ses positions (indices de début).

Exercice 4.17 — Gestion d'un répertoire ★★★

Écrire un programme de gestion de contacts (nom et téléphone) avec un menu permettant : 1) d'ajouter un contact, 2) de rechercher un numéro, 3) de supprimer un contact, 4) d'afficher tout le répertoire trié par nom.

Chp. 5

Structures de Données non mutables : Tuples et Chaînes de Caractères

❒ Objectifs

Ce TD explore les types de données **immuables**. À la différence des listes, ces structures ne peuvent plus être modifiées après leur création, garantissant ainsi l'intégrité des données.

- ✦ Maîtriser les **tuples** pour le stockage de données hétérogènes.
- ✦ Manipuler les **chaînes de caractères** (Unicode).
- ✦ Utiliser le *slicing* et l'affectation multiple (*unpacking*).
- ✦ Comprendre les conversions de types et le codage ASCII.

5.1 Les Tuples

Un *tuple* (ou n-uplet) est une séquence ordonnée et **immuable**. Une fois créé, on ne peut ni modifier, ni ajouter, ni supprimer d'éléments.

5.1.1 Définition et Propriétés

- **Syntaxe :** Les parenthèses sont recommandées : `t = (1, 2)`.
- **Cas particulier :** Un tuple à un seul élément nécessite une virgule : `t = (5,)`.
- **Affectation multiple :** `x, y = (10, 20)` permet d'extraire les valeurs instantanément.

Exercice 5.1 — Manipulation d'indices et slicing ★

Soit le tuple : `tup = (1, 3, 5, 'Hilbert', (12, 'ipt', 'génial'), 4.6)`. Écrire les instructions Python pour afficher :

- La chaîne `'Hilbert'`.
- Le dernier élément du tuple.
- Le sous-tuple `(3, 5)`.
- Les éléments `'ipt'` et `'génial'` (extraits du sous-tuple).

Exercice 5.2 — Calculs sur les nombres ★★

1. Écrire un script qui lit un nombre réel (ex : 65.21), sépare la partie entière de la partie décimale et les affiche sous forme de tuple : `(65, 21)`.
2. Écrire un programme qui, à partir d'un entier n, construit le tuple de ses chiffres.

Exercice 5.3 — Logique et Itérations ★★

1. Écrire un script qui saisit un tuple de n nombres et affiche le produit de ses éléments.
2. **Parité :** À partir d'un tuple d'entiers, créer un couple de deux tuples : le premier contenant les éléments pairs, le second les impairs.
3. **Recherche :** Écrire un script qui teste si deux tuples ont au moins un élément en commun.

Exercice 5.4 — Analyse de séquences ★★★

1. **Maximum consécutif :** Étant donné un tuple `t`, déterminer le nombre maximum de zéros consécutifs (ex : pour `(1,0,2,0,0,4)`, le résultat est 2).
2. **Sous-ensembles :** Écrire un script qui, pour un tuple donné, génère un tuple contenant tous les sous-tuples possibles (ensemble des parties).

Exercice 5.5 — Nouvel exercice : Fusion et Tri ★★

Soient deux tuples d'entiers déjà triés. Écrire un programme qui fusionne ces deux tuples en un troisième tuple, lui aussi trié, sans utiliser la méthode `sort()`.

5.2 Les chaînes de caractères

Une *chaîne de caractères* (`str`) est une séquence **ordonnée** et **immuable** de symboles Unicode.

5.2.1 Création et Codage

- `ord(char)` : Retourne le code ASCII (ex : `ord('A')` → 65).
- `chr(code)` : Retourne le caractère (ex : `chr(97)` → 'a').

Exercice 5.6 — Formatage et Conversion ★

1. Convertissez la chaîne `"3.14"` en un nombre flottant. Que se passe-t-il pour `"abc"` ?
2. En utilisant les *f-strings*, affichez le message : `"Bonjour M. Ali"` à partir des variables `titre="M."` et `nom="Ali"`.

Exercice 5.7 — Exploration des méthodes ★★

Testez `lower()`, `upper()`, `find("gram")`, `replace("P", "p")` et `split("r")` sur `s = "Programmation"`. Déduisez-en leur rôle.

Exercice 5.8 — Algorithmes de base ★★

1. Compter le nombre d'occurrences du caractère « e » dans une phrase, puis afficher la phrase sans aucun « e ».
2. **Analyse de données :** Convertir la chaîne `'1.0 3.14 7 8.4'` en une liste de réels : `[1.0, 3.14, 7.0, 8.4]`.

Exercice 5.9 — Traitement de texte et Slicing ★★

1. **Inversion :** Utilisez le slicing pour inverser une chaîne (`"python"` → `"nohtyp"`).
2. **Palindrome :** Tester si une chaîne est un palindrome.
3. **Insertion :** Insérer des astérisques entre chaque lettre d'un mot.

Exercice 5.10 — Analyse de séquences biologiques ★★★

Soit `seq = "ACCTAGCCATGTAGAATCGCCTAGGCTTTAGCTAGCTCTAGC"`.

1. Déterminer quelle base (A, C, G ou T) est la plus fréquente.
2. **Bigrammes :** Générer la liste de tous les couples de lettres possibles (`"AA", "AC", ...`).
3. Compter le nombre d'occurrences du bigramme `"TA"`.

Exercice 5.11 — Distance de Hamming ★★★

Calculer le nombre de positions où deux chaînes de même longueur diffèrent. *Exemple :* `"aaba"` *et* `"aaha"` $\longrightarrow$ *Distance = 1.*

Exercice 5.12 — Chiffrement de César ★★★

Le chiffrement de César consiste à décaler chaque lettre d'un message d'un certain rang k dans l'alphabet. Écrire une fonction qui prend une chaîne et un entier k et renvoie le message chiffré (utilisez `ord()` et `chr()`).

Chp. 6

Les fonctions en Python

❒ Objectifs

L'usage des fonctions permet de décomposer les tâches complexes en tâches simples, d'éviter la répétition et de rendre le code réutilisable. À l'issue de ce TD, vous devrez savoir :

- Déclarer une fonction avec la syntaxe `def` et utiliser des expressions `lambda`.
- Maîtriser le passage de paramètres (mutables vs non-mutables) et la portée des variables.
- Manipuler des fonctions comme des objets (fonctions d'ordre supérieur).
- Utiliser les outils de traitement de séquences (`map` et `filter`).

6.1 Déclaration et Syntaxe

Une fonction est un bloc d'instructions possédant un nom, recevant des paramètres et renvoyant un résultat via l'instruction `return`.

Rappel de syntaxe

```
def nom_fonction(param1, param2):
    ''' Documentation (Docstring) '''
    # Corps de la fonction (indenté)
    return resultat
```

Exercice 6.1 — Bases des fonctions ★

1. Écrire une fonction `fact(n)` qui calcule et retourne le factoriel d'un entier n.

2. Écrire une fonction `somme_liste(L)` qui retourne la somme des éléments d'une liste L.
3. Testez l'appel de vos fonctions dans le script principal. Que se passe-t-il si une fonction ne contient pas de `return` ?

6.2 Passage de paramètres et Mutabilité

Python utilise deux modes de passage selon le type de l'objet :

- **Par valeur :** Pour les types non-mutables (`int, float, str, tuple`).
- **Par référence (partage) :** Pour les types mutables (`list, dict, set`).

Exercice 6.2 — Analyse de comportement ★★

Exécutez le code suivant et expliquez les résultats affichés :

```
def modif(n, L):
    n = n + 1
    L.append(100)

a = 10
ma_liste = [1, 2]
modif(a, ma_liste)
print(a, ma_liste)
```

6.3 Analyse de nombres et logique

Exercice 6.3 — Nombres amis ★★

1. Créez une fonction `diviseurs(n)` qui retourne la liste des diviseurs propres de n.
2. Deux nombres sont **amis** si chacun est égal à la somme des diviseurs propres de l'autre. Écrire `sont_amis(n, m)` qui renvoie un booléen.

Exercice 6.4 — Nombres Premiers et Jumeaux ★★★

1. Écrire `est_premier(n)` renvoyant `True` si n est premier.
2. Deux nombres premiers sont **jumeaux** si leur différence est égale à 2 (ex : 5 et 7). Écrire une fonction `jumeaux(n)` qui affiche tous les couples de jumeaux inférieurs à n.

Exercice 6.5 — Conjecture de Goldbach ★★★

La conjecture énonce que tout nombre pair supérieur à 3 est la somme de deux nombres premiers. Écrire une fonction `verif_Goldbach(n)` qui vérifie cette propriété pour tous les nombres pairs jusqu'à $n = 1000$.

6.4 La fonction comme objet de première classe

En Python, une fonction est une valeur comme une autre. Elle peut être passée en argument, renvoyée par une autre fonction ou stockée dans une variable.

Exercice 6.6 — Fonctions d'ordre supérieur ★

1. Écrire une fonction `carre(n)` qui retourne le carré d'un entier.
2. Écrire une fonction `somme_fonction(f, n)` qui calcule la somme $\sum_{i=0}^{n} f(i)$.
3. Testez `somme_fonction(carre, 10)`. Expliquez pourquoi on ne met pas de parenthèses à `carre` lors de l'appel.

☞ Concepts clés

- **Passage :** `f(g)` → g est passée comme argument.
- **Stockage :** `afficher = print` → `afficher("Hello")` fonctionne.

6.5 Les expressions Lambda

Une expression lambda est une fonction anonyme définie sur une seule ligne. Elle est idéale pour des opérations simples et éphémères.

Syntaxe

```
lambda arguments : expression
```

Exercice 6.7 — Pratiquer la fonction Lambda ★★

1. Réécrire l'appel de `somme_fonction` de l'exercice précédent en utilisant une fonction lambda pour calculer le carré directement dans l'argument.
2. Créer une fonction lambda nommée `somme` qui prend deux arguments x et y et retourne leur somme.

6.6 Traitement de séquences : Map et Filter

Les fonctions `map` et `filter` permettent de traiter des listes sans utiliser de boucles `for` explicites.

6.6.1 La fonction map()

Elle applique une fonction à chaque élément d'une séquence.

Syntaxe : map

```
map(fonction, itérable)
```

Exercice 6.8 — Transformation de listes ★★

1. Soit `L1 = [1, 2, 3]` et `L2 = [4, 5, 6]`. Utilisez `map` et une fonction lambda pour créer une liste `L3` contenant la somme des éléments correspondants de `L1` et `L2`.
2. Pourquoi doit-on souvent utiliser le constructeur `list()` autour de `map()` ?

6.6.2 La fonction filter()

Elle extrait les éléments d'une séquence pour lesquels une fonction renvoie `True`.

</> Syntaxe : filter

```
filter(condition, itérable)
```

Exercice 6.9 — Filtrage de données ★★

1. À l'aide de `filter`, créez une liste ne contenant que les nombres impairs d'une liste `L = range(20)`.
2. **Application :** Écrire un script qui génère les n premiers nombres de la suite de Fibonacci, puis filtre séparément les nombres pairs et impairs de cette suite.

✎ Résumé technique

map(f, L) : Transforme chaque élément x de L en $f(x)$.

filter(p, L) : Garde les éléments x de L si $p(x)$ est vrai.

Chp. 7

La gestion des exceptions

❒ Objectifs

Même un code syntaxiquement correct peut provoquer une erreur lors de son exécution. À l'issue de ce TD, vous devrez savoir :

- ✦ Identifier les principaux types d'exceptions (`ValueError`, `IndexError`, etc.).
- ✦ Intercepter une erreur avec le bloc `try ... except` pour éviter l'arrêt du programme.
- ✦ Déclencher volontairement une exception avec `raise`.
- ✦ Utiliser les clauses `else` et `finally` pour sécuriser votre code.

7.1 Comprendre et lever des exceptions

Lorsqu'une erreur survient, Python "lève" une exception. Si elle n'est pas gérée, le programme s'arrête brutalement.

Exercice 7.1 — Identifier les erreurs ★

Prédisez le type d'exception levé par les instructions suivantes, puis vérifiez dans la console :

```
>>> 10 * (1/0)
>>> int("bonjour")
>>> L = [1, 2]; print(L[5])
>>> d = {"a": 1}; print(d["b"])
```

Forcer une erreur : `raise`

Vous pouvez lever manuellement une exception si une condition métier n'est pas remplie :

```
age = int(input("Votre age : "))
if age < 0:
    raise ValueError("L'âge ne peut pas être négatif !")
```

7.2 Intercepter les erreurs : le bloc `try-except`

La gestion d'exception permet d'anticiper une erreur et de proposer une solution.

Syntaxe : Gestion des erreurs

```
try:
   # Bloc de code à tester
except TypeErreur:
   # Bloc exécuté en cas d'erreur
```

Exercice 7.2 — Saisie sécurisée ★★

Écrire une fonction `saisie()` qui demande un entier strictement positif.

1. Utilisez une boucle `while True` et un bloc `try-except` pour redemander la saisie tant que l'utilisateur n'entre pas un nombre valide.
2. Gérez l'exception `ValueError` avec un message explicite.

7.3 Gestion avancée et clauses optionnelles

Il est possible de gérer plusieurs types d'erreurs spécifiquement et d'ajouter des étapes de finalisation.

</> Syntaxe : Gestion exhaustive des exceptions

```
try:
    # Code susceptible de générer des erreurs
except ErreurType1:
    # Traitement spécifique pour ErreurType1
except (ErreurType2, ErreurType3):
    # Traitement groupé pour plusieurs erreurs
else:
    # Exécuté uniquement si AUCUNE exception n'est levée
finally:
    # Toujours exécuté (nettoyage, libération de mémoire)
```

Exercice 7.3 — Manipulation de structures ★★

Écrire une fonction `creer_liste(n)` qui génère une liste des n premiers nombres premiers.

1. Demandez à l'utilisateur un indice i pour afficher le i-ème élément.
2. Ajoutez un bloc `try-except` gérant spécifiquement `IndexError` et `ValueError`.

✎ A retenir

- **`except` :** Intercepte l'erreur.
- **`else` :** S'exécute uniquement si **aucune** exception n'a été levée.
- **`finally` :** S'exécute **toujours** (utile pour fermer un fichier ou une connexion).

7.4 Application de synthèse

Exercice 7.4 — Dictionnaire et listes ★★★

Écrire une fonction `ajouter_elt(d, L_nom, x)` qui ajoute un élément x à la liste nommée `L_nom` présente dans le dictionnaire d.

- Utilisez un bloc `try-except` pour gérer la `KeyError` (si la liste n'existe pas encore).

- Intégrez une clause `else` pour confirmer l'ajout et une clause `finally` pour afficher l'état final du dictionnaire.

Chp. 8

Les fonctions récursives

❐ Objectifs

La récursivité est une méthode de programmation où une fonction s'appelle elle-même pour résoudre un problème. À l'issue de ce TD, vous devrez savoir :

- ✦ Identifier et définir le **cas de base** (condition d'arrêt).
- ✦ Définir le **cas de propagation** (appel récursif).
- ✦ Comprendre le mécanisme de l'empilement des appels (contexte).
- ✦ Transposer une récurrence mathématique en code Python.

8.1 Concepts fondamentaux

Une fonction est dite récursive si elle est utilisée dans sa propre définition. Elle repose sur deux piliers essentiels :

- **Le cas de base :** C'est l'arrêt de la récursion. Sans lui, le programme s'exécute indéfiniment.
- **Le cas de propagation :** L'appel récursif dont les arguments doivent converger vers le cas de base.

</> Syntaxe : Fonction Récursive

```
def fonction_recursive(arguments):
    if condition_d_arret:
        return valeur_finale # Cas de base
    else:
        return fonction_recursive(nouveaux_arguments)
```

Exercice 8.1 — Analyse de code ★

Soit la fonction suivante :

```python
def f(a, b):
    if b == 1:
        return a
    else:
        return a + f(a, b - 1)
```

1. Quel est le résultat affiché par `print(f(3, 5))` ?
2. Dessiner le schéma d'exécution (la pile d'appels) pour `f(3, 5)`.
3. Quel est le cas de base et qu'est-ce qui garantit l'arrêt du programme ?
4. Que calcule mathématiquement cette fonction `f(a, b)` pour des entiers naturels non nuls ?

☞ Rappel Important

À chaque nouvel appel récursif, Python crée un nouveau **contexte** (espace mémoire local). Les variables portent les mêmes noms mais stockent des valeurs différentes selon la profondeur de l'appel.

8.2 Implémentations de suites mathématiques

La récursivité est la transposition directe de la récurrence mathématique.

Exercice 8.2 — Suites classiques ★★

Définir les fonctions récursives suivantes :

1. `fact(n)` : calcule la factorielle de n ($n!$).
2. `somme(n)` : calcule la somme des n premiers entiers.
3. `syracuse(n)` : affiche les termes de la suite de Syracuse jusqu'à atteindre 1.

Exercice 8.3 — Suite triple ★★

Écrire une fonction récursive pour calculer les termes de la suite (U_n) définie par :

$$\begin{cases} U_0 = U_1 = U_2 = 1 \\ U_{n+3} = U_{n+2} + U_{n+1} + U_n, \quad \forall n \geq 0 \end{cases}$$

8.3 Récursivité sur les listes

Les listes se prêtent parfaitement à la récursivité en traitant le premier élément, puis le reste de la liste.

Exercice 8.4 — Calculs sur listes ★★★

1. Écrire `somme_rec(L)` qui renvoie la somme des éléments d'une liste L par récursivité.
2. Écrire `somme_termes(L)` qui renvoie une liste des sommes cumulées. *Exemple : `[1, 4, 3]` devient `[1, 5, 8]`.*

Chp. 9

Les algorithmes de tri

❒ Objectifs

Le tri consiste à ordonner les éléments d'une liste. C'est une opération fondamentale pour optimiser d'autres algorithmes. À l'issue de ce TD, vous devrez savoir :

- ✦ Implémenter les tris classiques dits "lents" ($O(n^2)$) : Bulle, Sélection, Insertion.
- ✦ Comprendre et coder le **Tri Rapide** (Quicksort) basé sur la récursivité.
- ✦ Analyser l'efficacité des algorithmes par la mesure du temps d'exécution.

9.1 Le Tri à bulles (Bubble Sort)

Le principe est de faire « remonter » le plus grand élément vers la droite par échanges successifs d'éléments adjacents.

Algorithme 1 . Algorithme du tri à bulles

Données : Une liste L de taille n

Résultat : La liste L triée en place

```
1 pour i de 0 à n-2 faire
2     pour j de 0 à n-i-2 faire
3         si L[j] > L[j+1] alors
4             Échanger L[j] et L[j+1]
5         sinon
              // L'élément est à sa place
```

Exercice 9.1 — Implémentation et Analyse ★★

1. Implémenter le tri à bulles en Python.

2. **Analyse temporelle :** Mesurez le temps d'exécution pour des listes de tailles $N \in [100, 500, 1000, 2000, 5000]$.
3. **Affichage :** Affichez les résultats sur la console sous la forme d'un tableau indiquant la taille N et le temps correspondant en secondes.

9.2 Tri par Sélection

On cherche l'élément minimum de la partie non triée et on l'échange avec l'élément à la position actuelle.

Algorithme 2 . Algorithme du tri par sélection

Données : Une liste L de taille n

Résultat : La liste L triée

```
1 pour i de 0 à n-2 faire
2     min ← i
3     pour j de i+1 à n-1 faire
4         si L[j] < L[min] alors
5             min ← j
6     Échanger L[i] et L[min]
```

Exercice 9.2 — Variantes du tri par sélection ★★

1. Créer la fonction `tri_selection(L)` en Python.
2. **Défi :** Proposer une version **récursive** de cet algorithme.

9.3 Tri par Insertion

On insère chaque nouvel élément à sa place exacte dans la partie gauche de la liste, qui est maintenue triée.

Algorithme 3 . Algorithme du tri par insertion

Données : Une liste L de taille n

Résultat : La liste L triée

```
1 pour i de 1 à n-1 faire
2     v ← L[i]
3     j ← i
4     j > 0 et L[j-1] > v L[j] ← L[j − 1]
5     j ← j − 1
6     L[j] ← v
```

Exercice 9.3 — Simulation et implémentation du Tri par Insertion ★★

1. **Trace d'exécution :** Soit la liste $L = [12, 11, 13, 5, 6]$. Dessiner l'état de la liste à la fin de chaque itération de la boucle principale (`Pour i de 1 à n-1`).
2. **Implémentation :** Écrire la fonction `tri_insertion(L)` en Python.
3. Quelle est la complexité de ce type de tri ?
4. **Comparaison console :** Comparer le temps d'exécution du tri par insertion et du tri à bulles sur une liste inversée de taille $N = 2000$. Affichez le résultat sur la console.

9.4 Tri Rapide (Quicksort)

C'est un algorithme **diviser pour régner**. Le principe repose sur le choix d'un *pivot* pour partitionner la liste en deux sous-problèmes plus simples.
Le *tri rapide* suit une logique récursive :

- **Cas de base :** Si `L` est vide ou contient un seul élément, elle est déjà triée ;
- **Partitionnement :** On choisit un élément `e` ∈ `L` (appelé le pivot), que l'on retire de `L` ;
- **Division :** On construit deux sous-listes :
  - `Li` formée des éléments inférieurs ou égaux à `e` ;
  - `Ls` formée des éléments strictement supérieurs à `e` ;
- **Combinaison :** Le résultat est la concaténation de `tri_rapide(Li)`, du pivot `[e]`, et de `tri_rapide(Ls)`.

Exercice 9.4 — Implémentation du Quicksort ★★★

1. Créer une fonction `tri_rapide(L)` qui trie la liste `L` selon le principe énoncé ci-dessus (utilisez les listes par compréhension pour simplifier l'écriture).
2. **Analyse de complexité :**
   - Quelle est la complexité dans le meilleur des cas (pivot médian) ?
   - Quelle est la complexité dans le pire des cas (liste déjà triée et pivot à l'extrémité) ?

Exercice 9.5 — Puissance de la récursivité ★★★

1. Implémenter `tri_rapide(L)` en utilisant la concaténation de listes.
2. Comparez son temps d'exécution avec celui du tri à bulles pour $N = 1000$.

— Fin du TD—

Chp. 10

Les algorithmes de recherche

❒ Objectifs

La recherche d'un élément dans une structure de données est une opération fondamentale. Ce TD explore deux stratégies principales :

- La **recherche séquentielle** : méthode universelle (tableaux triés ou non).
- La **recherche dichotomique** : optimisation majeure pour les listes triées.
- L'analyse de **complexité** : comparer l'efficacité temporelle ($O(n)$ vs $O(\log n)$).

10.1 Recherche séquentielle (ou linéaire)

Cette méthode consiste à parcourir le tableau un par un jusqu'à trouver l'élément ou atteindre la fin. Elle ne nécessite aucune hypothèse sur l'ordre des données.

Implémentation de référence :

```python
def recherche_seq(L, x):
    n = len(L)
    for i in range(n):
        if L[i] == x:
            return i # Élément trouvé, on renvoie l'indice
    return None # Fin de parcours, non trouvé
```

Exercice 10.1 — Maximum et occurrences ★

1. Écrire une fonction renvoyant l'indice de la première occurrence du **maximum** d'une liste d'entiers. Quelle est sa complexité ?

2. Écrire une fonction qui renvoie les **deux plus grands** éléments d'une liste en un seul parcours (complexité linéaire).

Exercice 10.2 — Recherche de motif (Pattern Matching) ★★

On cherche une séquence de valeurs dans un tableau (ex : un mot dans un texte).

1. Écrire `recherche_motif(m, t)` qui renvoie l'indice du début de la première occurrence du tableau m dans t.
2. Modifier la fonction pour qu'elle retourne **toutes** les positions d'un motif dans un texte.

10.2 Recherche dichotomique

Si la liste est **triée**, on peut appliquer le principe "diviser pour régner". On compare l'élément cherché avec celui du milieu, ce qui permet d'éliminer la moitié des candidats à chaque étape.

✎ Algorithme de la dichotomie

Complexité : $O(\log_2 n)$. C'est un gain considérable : pour un tableau de 1 000 éléments, 10 comparaisons suffisent au lieu de 1 000.

Exercice 10.3 — Dichotomie itérative ★★

1. Proposer une fonction `rech_dicho(x, L)` qui renvoie un booléen.
2. Testez votre code sur `L = [1, 3, 5, 7, 8, 13, 14, 17, 19]` avec les valeurs 13, 2 et 20.
3. Écrire `rechDicho1(t, n, x)` qui cherche x dans les n premiers éléments et renvoie l'indice (ou -1 si absent).

Exercice 10.4 — Version récursive ★★★

Écrire la fonction `rechDicho2(t, n, x)` en version **récursive**.
Indice : Il faudra probablement définir une fonction auxiliaire avec des bornes `debut` et `fin` en paramètres.

Chp. 11

Manipulation des fichiers texte

❒ Objectifs

Les fichiers permettent de sauvegarder des données de manière permanente. À l'issue de ce TD, vous devrez savoir :

- Ouvrir, lire et fermer un fichier texte en mode lecture (`'r'`).
- Créer et modifier des fichiers en mode écriture (`'w'`) et ajout (`'a'`).
- Manipuler les séquences d'échappement (`\n`) et les méthodes de lecture ligne par ligne.
- Gérer les chemins d'accès absolus et relatifs avec le module `os`.

11.1 Lecture de fichiers

La manipulation d'un fichier suit toujours le cycle : **Ouverture → Traitement → Fermeture**.

```
f = open("test.txt", "r") # 'r' pour Read (Lecture)
contenu = f.read()        # Lit tout le fichier
f.close()                 # Libère les ressources
```

Exercice 11.1 — Exploration de la lecture ★

1. Que renvoie la méthode `read()` lorsque la fin du fichier est atteinte ?
2. Testez et décrivez la différence entre `f.readline()` et `f.readlines()`.
3. Écrivez une fonction `lecture(nom)` qui affiche le contenu d'un fichier ligne par ligne en supprimant les caractères de saut de ligne (`\n`).

4. Utilisez un bloc `try-except` pour gérer le cas où le fichier demandé n'existe pas (`FileNotFoundError`).

11.2 Écriture et Modification

Il existe deux modes principaux pour modifier un fichier :

- **Mode 'w' (Write) :** Écrase le contenu existant ou crée le fichier.
- **Mode 'a' (Append) :** Ajoute du texte à la fin du fichier existant.

Exercice 11.2 — Saisie et stockage ★★

1. Écrivez une fonction `ecriture()` qui enregistre dans un fichier tout ce que l'utilisateur saisit au clavier jusqu'à ce qu'il entre une ligne vide.
2. Assurez-vous que chaque saisie commence sur une nouvelle ligne dans le fichier.
3. Créez une fonction `ajout(nom, texte)` qui insère une ligne à la fin d'un fichier existant sans effacer le reste.

11.3 Gestion des chemins d'accès

Pour manipuler des fichiers situés dans d'autres répertoires, on utilise le module `os`.

Méthodes utiles :
`os.getcwd()` : Récupère le répertoire de travail actuel.
`os.chdir(chemin)` : Change le répertoire courant.

Exercice 11.3 — Traitement de fichiers ★★

1. Écrivez une fonction `copier(source, destination)` qui recopie le contenu d'un fichier dans un autre.
2. Écrire une fonction `comparer(f1, f2)` qui signale la première ligne où les deux fichiers diffèrent.

Exercice 11.4 — Structuration de données ★★★

1. **Export :** Créer une fonction `remplir(fichier, listes)` qui enregistre une liste de listes d'entiers dans un fichier, un groupe par ligne, séparés par des virgules.
2. **Recherche :** Créer une fonction `recherche(valeur, fichier)` qui retourne les coordonnées (ligne, colonne) de chaque occurrence d'un entier dans un fichier formaté.
3. **Analyse :** Écrivez un script qui cherche un mot spécifique dans tous les fichiers `.txt` d'un répertoire donné et affiche le nom du fichier ainsi que le numéro de ligne.

PARTIE II

Programmation Python

Simulation numérique

Chp. 11. Algèbre linéaire (NumPy & Matplotlib) . . . . . p. 38

Chp. 12. Résolution d'équations $f(x) = 0$ . . . . . p. 42

Chp. 13. Méthodes d'intégration numérique . . . . . p. 60

Chp. 14. Résolution des Équations Différentielles (Euler) . . . . . p. 64

II

Chp. 12

Calcul scientifique : NumPy & Matplotlib

❒ Objectifs

On présente ici les bases essentielles du module `NumPy`, bibliothèque centrale du calcul scientifique en Python. À l'issue de ce TD, vous serez capables de :

- Créer et manipuler des tableaux `ndarray` (1D et 2D) ;
- Maîtriser le *slicing* (coupes) et l'indexation avancée ;
- Effectuer des opérations vectorielles et matricielles efficaces ;
- Utiliser les fonctions de réduction (somme, moyenne) selon les axes.

12.1 Présentation du module NumPy

Le module `numpy` fournit le type `ndarray` (*N-dimensional array*). Contrairement aux listes Python standards :

- Les tableaux sont **homogènes** (un seul type de données par tableau) ;
- Leur taille est **fixée** à la création ;
- Les opérations sont **vectorisées** (calculs C optimisés en arrière-plan).

```
1 import numpy as np
```

Exercice 12.1 — Installation ★

Vérifiez votre installation en affichant la version de NumPy installée :
`print(np.__version__)`

12.2 Création des tableaux (Vecteurs et Matrices)

L'objectif de cette section est d'apprendre à initialiser des structures `ndarray` soit manuellement à partir de données existantes, soit automatiquement via des générateurs optimisés.

12.2.1 Création à partir de listes

La méthode `np.array()` permet de convertir des structures standards Python en tableaux NumPy pour bénéficier de la puissance du calcul vectoriel.

```
a = np.array([1, 2, 3]) # Crée un vecteur (1D) à partir d'une liste
print(a.dtype)          # Affiche le type de données stockées (ex: int64)

# Crée une matrice (2D) à partir d'une liste de listes
m = np.array([[1, 2, 3], [2, 3, 4], [0, 1, 0]])

liste_python = a.tolist() # Convertit le tableau NumPy en liste Python
↪ standard
```

> **Exercice 12.2 — Types** ★
>
> Créez un tableau à partir de la liste `[1, 2, 3.5]`. Utilisez `a.dtype` pour voir comment NumPy a géré le mélange entier/flottant.

12.2.2 Fonctions de création spécialisées

NumPy propose des fonctions pour générer rapidement des séquences numériques sans utiliser de boucles explicites.

Vecteurs et séquences

```
t0 = np.arange(0, 10, 2)  # Séquence de 0 à 10 (exclu) avec un pas de 2
t1 = np.arange(3.)        # Séquence de 0 à 3 (exclu) convertie en
↪ flottants
t4 = np.linspace(0, 10, 5) # 5 valeurs équiréparties entre 0 et 10 inclus
t3 = np.array([2] * 5)    # Crée un vecteur de taille 5 contenant
↪ uniquement des 2
```

Matrices particulières

Les fonctions suivantes facilitent l'initialisation de structures classiques utilisées en algèbre linéaire.

```
m_ones = np.ones((3, 5))   # Matrice 3x5 remplie de 1.0 (flottants par défaut)
m_zeros = np.zeros(10)     # Vecteur de 10 zéros
m_eye = np.eye(3)          # Matrice identité de dimension 3x3
m_diag = np.diag([1, 2, 3]) # Matrice diagonale avec 1, 2, 3 sur la diagonale
```

Exercice 12.3 — Vecteurs ★★

1. Construire un tableau contenant les multiples de 3 entre 0 et 27 (inclus).
2. Construire un tableau de taille 100, dont les extrémités sont 0 et 27.

12.2.3 Construction par formule (`fromfunction`)

La méthode `np.fromfunction()` permet de générer un tableau en calculant la valeur de chaque élément à partir de ses coordonnées (i, j) grâce à une fonction utilisateur.

```
def ma_formule(i, j):
    return 10 * i + j # i représente l'indice de ligne, j celui de colonne

# Génère une matrice 4x5 où chaque élément est calculé par 'ma_formule'
t_formule = np.fromfunction(ma_formule, (4, 5), dtype=int)
```

Exercice 12.4 — Table de multiplication ★★

Utilisez `fromfunction` pour créer une matrice 10×10 représentant la table de multiplication (où $M_{i,j} = i \times j$).

12.3 Lecture et Slicing (Vecteurs et Matrices)

L'accès aux données se fait par des indices ou des plages d'indices (slicing).

```
m = np.arange(24).reshape(4, 6)
val = m[1, 2]       # Élément ligne 1, col 2
ligne_0 = m[0, :]   # Toute la première ligne
col_1 = m[:, 1]     # Toute la deuxième colonne
bloc = m[:2, :3]    # Sous-matrice (2 lignes, 3 colonnes)
inverse = m[::-1]   # Inverse l'ordre des lignes
```

Exercice 12.5 — Extraction ★★

Soit `M = np.arange(100).reshape(10, 10)`. Extraire le bloc central de 4×4 éléments (lignes et colonnes d'indices 3 à 6 inclus).

12.4 Écriture, Modification et Réduction

12.4.1 Mutabilité et modification en masse

NumPy permet de modifier des sous-ensembles de données par indexation booléenne (masques).

```
a = np.arange(10)
a[7] = 21
a[4:7] = [111, 222, 333]
a[a % 2 == 0] = 0 # Met à zéro tous les nombres pairs
a.put([0, 9], [99, 88]) # Remplace les éléments aux indices 0 et 9
```

Exercice 12.6 — Seuillage ★★

Créez un vecteur de 20 nombres aléatoires entre 0 et 100. Remplacez toutes les valeurs supérieures à 50 par la valeur 50.

12.4.2 Opérations de réduction selon les axes

Les fonctions de réduction calculent une statistique sur tout le tableau ou selon un axe précis.

```
m = np.arange(15).reshape(3, 5)
total = m.sum()        # Somme de tous les éléments
moy_cols = m.mean(axis=0) # Moyenne de chaque colonne (résultat: vecteur de taille 5)
max_lignes = m.max(axis=1) # Max de chaque ligne (résultat: vecteur de taille 3)
```

Exercice 12.7 — Analyse de données ★★★

Soit une matrice de notes `M` (3 étudiants en lignes, 4 matières en colonnes).

1. Calculez la moyenne de chaque étudiant.
2. Calculez la meilleure note obtenue dans chaque matière.

12.5 Fonctions Universelles et Vectorisation

❒ Objectifs

Cette section approfondit l'utilisation de NumPy en explorant la puissance du calcul vectorisé. À l'issue de ce TD, vous serez capables de :

- Utiliser les fonctions universelles (*ufuncs*) pour des calculs ultra-rapides ;
- Maîtriser les opérations arithmétiques terme à terme ;
- Vectoriser vos propres fonctions Python personnalisées ;
- Appliquer des fonctions mathématiques complexes sur des tableaux de données.

12.6 Les Fonctions Universelles (ufunc)

Une *fonction universelle* (ou *ufunc*) est une fonction qui s'applique terme à terme aux éléments d'un tableau.

Si f est une *ufunc* et si $a = [a_0, a_1, \ldots, a_{n-1}]$ est un tableau, alors $f(a)$ renvoie le tableau $[f(a_0), f(a_1), \ldots, f(a_{n-1})]$.

- Les a_k peuvent être des tableaux (ex : lignes d'une matrice), la fonction s'applique alors récursivement.
- Si on effectue une opération entre un tableau a et un scalaire x, l'opération est distribuée sur chaque élément de a.

Note : La documentation officielle est disponible ici : `http://docs.scipy.org/doc/numpy/reference/ufuncs.html`

12.7 Opérations Arithmétiques Terme à Terme

Les opérateurs standards (+, ∗, /, ==, etc.) agissent de manière **élémentaire** dans NumPy. L'expression `a * b` n'est donc **pas** un produit matriciel au sens de l'algèbre linéaire, mais un produit terme à terme.

```
# Création de tableaux aléatoires pour l'exemple
a = np.random.randint(0, 10, (2, 5))
b = np.random.randint(1, 10, (2, 5))

# --- Exemples d'opérations vectorisées ---
neg = -a                # Opposé (équivalent à np.negative(a))
somme = a + b           # Addition terme à terme (np.add)
produit = a * b         # Multiplication terme à terme (np.multiply)
quotient = a / b        # Quotient flottant (np.divide)
puissance = a ** b      # Élévation à la puissance (np.power)
masque = (a == b)       # Tableau de booléens (True si égaux)
```

12.8 Fonctions Mathématiques Usuelles

NumPy propose des versions "universalisées" des fonctions de la bibliothèque `math`. Elles doivent être préfixées par `np`.

- **Logarithmes :** `np.log` (népérien), `np.log10`, `np.log2`.

- **Trigonométrie :** `np.sin, np.cos, np.tan, np.arcsin, np.sinh`, etc.
- **Conversion d'angles :** `np.degrees` (rad → deg) et `np.radians` (deg → rad).

```python
# Exemple : Calcul trigonométrique sur un vecteur
rad = np.array([0, np.pi/6, np.pi/4, np.pi/2])
deg = np.degrees(rad) # Résultat : [0., 30., 45., 90.]
```

12.9 Vectorisation de fonctions personnalisées

Pour qu'une fonction Python classique f (conçue pour un scalaire) puisse s'appliquer terme à terme à un tableau, il faut la *« vectoriser »* via `np.vectorize`.

Exemple 1 : Fonction mathématique complexe Considérons $f(x) = \frac{x+\sin(x)}{x+\cos(x)}$.

```python
def f(x):
    return (x + np.sin(x)) / (x + np.cos(x))

a = np.arange(1, 7)
vf = np.vectorize(f) # Création de la version universelle
print(vf(a))         # Application au tableau
```

Exemple 2 : Fonction avec conditions (*if/else*)

```python
def f(x, y):
    return 0 if x < y else y

vf = np.vectorize(f)
a = np.array([1, 2, 3, 4, 5, 6])
b = np.array([4, 1, 8, 7, 2, 9])

# Comparaison de deux tableaux
print(vf(a, b)) # Résultat : [0, 1, 0, 0, 2, 0]
```

12.10 Série d'exercices

Exercice 12.8 — Loi Normale ★

Soit la fonction de densité de la loi normale centrée réduite : $h(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$.

1. Créer un tableau `xt` contenant 41 valeurs réparties uniformément entre -4 et 4.
2. Créer un tableau `ht` contenant l'évaluation de la fonction h sur chacune de ces valeurs en utilisant les fonctions universelles de NumPy.

Exercice 12.9 — Manipulation de base ★

1. Créer un vecteur V de 10 entiers aléatoires entre 1 et 100.
2. Extraire tous les nombres pairs de V en utilisant un masque booléen.
3. Remplacer tous les multiples de 3 par la valeur -1.

Exercice 12.10 — Normalisation et Statistique ★★

1. Créer une matrice M de taille $(5, 5)$ contenant des nombres réels aléatoires (`np.random.rand`).
2. Calculer la somme de chaque colonne.
3. Diviser chaque élément de la matrice par la somme de sa colonne respective (Normalisation).
4. Vérifier que la somme des éléments de chaque colonne de la nouvelle matrice vaut bien 1.

Exercice 12.11 — Vectorisation conditionnelle ★★

On souhaite définir une fonction "Seuil" : $s(x) = 1$ si $x > 0.5$, sinon $s(x) = 0$.

1. Écrire la fonction Python `seuil(x)`.
2. Vectoriser cette fonction.
3. Appliquer cette fonction à une matrice 4×4 de nombres aléatoires entre 0 et 1.

Exercice 12.12 — Approximation de Pi par la méthode de Viète ★★★

La formule de Viète utilise des racines imbriquées pour approximer π.

1. En utilisant NumPy, calculer les 10 premiers termes de la suite $u_{n+1} = \sqrt{2 + u_n}$ avec $u_0 = \sqrt{2}$.
2. En déduire une approximation de π sachant que $\frac{2}{\pi} = \lim_{n\to\infty} \prod_{i=1}^{n} \frac{u_i}{2}$.

12.11 Calcul Matriciel et Algèbre Linéaire

❒ Objectifs

Cette section traite des capacités de NumPy pour l'algèbre linéaire, outils fondamentaux pour les sciences de l'ingénieur. À l'issue de ce TD, vous serez capables de :

- Maîtriser les produits scalaires, vectoriels et matriciels ;
- Manipuler les transposées et calculer des déterminants ;
- Résoudre des systèmes d'équations linéaires $Ax = B$;
- Calculer des normes vectorielles et matricielles.

12.12 Produits matriciels et vectoriels

Contrairement à l'opérateur $*$, qui multiplie terme à terme, NumPy utilise des fonctions spécifiques pour les produits de l'algèbre linéaire.

12.12.1 Produit de matrices

Pour effectuer le produit matriciel $C = A \times B$, on utilise la fonction `np.dot(a, b)`.

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[1, 1, 1], [2, 2, 2]])

# Produit matriciel (2x2) * (2x3) -> (2x3)
res = np.dot(A, B)
```

```
# On peut aussi enchaîner les produits
final = np.dot(A, B).dot(np.ones((3, 2)))
```

12.12.2 Produit scalaire et vectoriel

- **Produit scalaire :** `np.vdot(u, v)` calcule $\sum u_i v_i$.
- **Produit vectoriel :** `np.cross(u, v)` est utilisé pour les vecteurs de $\mathbb{R}^3$.

```
u = np.array([1, 2, 3])
v = np.array([4, 5, 6])

scal = np.vdot(u, v)  # Résultat: 32
vec = np.cross(u, v)   # Résultat: array([-3,  6, -3])
```

12.13 Inversion et résolution de systèmes

Le sous-module `numpy.linalg` (souvent abrégé `alg`) contient les fonctions essentielles pour l'étude des matrices carrées.

```
import numpy.linalg as alg

A = np.array([[1, -2, 3], [2, 1, 4], [5, -1, 2]])
B = np.array([14, 12, 13])

# 1. Transposée
print(A.T)              # ou np.transpose(A)

# 2. Déterminant
det_A = alg.det(A)      # Calcule le déterminant de la matrice

# 3. Résolution de Ax = B
# Plus stable numériquement que de calculer l'inverse explicitement
x = alg.solve(A, B)     # Résultat: [ 1., -2.,  3.]
```

12.14 Normes matricielles et vectorielles

La fonction `linalg.norm(x)` calcule par défaut la norme de Frobenius pour les matrices : $\| M \| = \sqrt{\sum |m_{i,j}|^2}$

```
m = np.arange(-10, 10).reshape(5, 4)
norme = np.linalg.norm(m) # x ≈ 25.88
```

12.15 Exercices d'application

Exercice 12.13 — Vérification de propriétés matricielles ★

Soient deux matrices $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ et $B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$.

1. Vérifier par le calcul NumPy que $A \times B \neq B \times A$.
2. Vérifier que $(A \times B)^T = B^T \times A^T$.

Exercice 12.14 — Géométrie dans l'espace ★

On considère les vecteurs $\vec{u}(1, 2, 3)$ et $\vec{v}(-1, 0, 2)$.

1. Calculer le produit vectoriel $\vec{w} = \vec{u} \wedge \vec{v}$.
2. Vérifier par un produit scalaire que $\vec{w}$ est orthogonal à $\vec{u}$ et à $\vec{v}$.

Exercice 12.15 — Puissance d'une matrice ★★

Soit la matrice $M = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$.

1. Calculer M^2, M^3 et M^{10} en utilisant `np.linalg.matrix_power(M, n)`.
2. Conjecturer la forme de M^n pour tout $n \in \mathbb{N}$.

Exercice 12.16 — Résolution de circuit électrique ★★

L'application des lois de Kirchhoff à un circuit mène au système suivant pour les intensités i_1, i_2 et i_3 :

$$\begin{cases} i_1 - i_2 - i_3 = 0 \\ 10i_1 + 20i_2 = 12 \\ 20i_2 - 5i_3 = 0 \end{cases}$$

1. Écrire le système sous forme $AX = B$.
2. Résoudre le système avec NumPy pour trouver les intensités.

Exercice 12.17 — Matrices de rotation ★★★

Une matrice de rotation d'angle θ est définie par $R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$.

1. Créer une fonction `rotation(theta)` qui renvoie la matrice NumPy correspondante.
2. Vérifier pour $\theta = \pi/3$ que R_θ est une matrice orthogonale (c'est-à-dire que $R_\theta \times R_\theta^T = I$).
3. Calculer le déterminant de R_θ pour plusieurs valeurs de θ. Que remarquez-vous ?

12.16 Visualisation Graphique : Matplotlib

❒ Objectifs

Cette section introduit les outils de tracé de courbes et de représentation de données. À l'issue de ce TD, vous serez capables de :

- Générer des graphiques 2D simples et complexes ;
- Maîtriser le multi-fenêtrage, les sous-graphiques (*subplots*) et la 3D ;
- Personnaliser l'apparence des courbes (couleurs, styles, marqueurs) ;
- Annoter un graphique (titres, légendes, axes).

12.17 Présentation et fonctions de base

Le module `Matplotlib` est l'outil standard pour tracer des courbes en Python. On l'importe généralement ainsi :

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

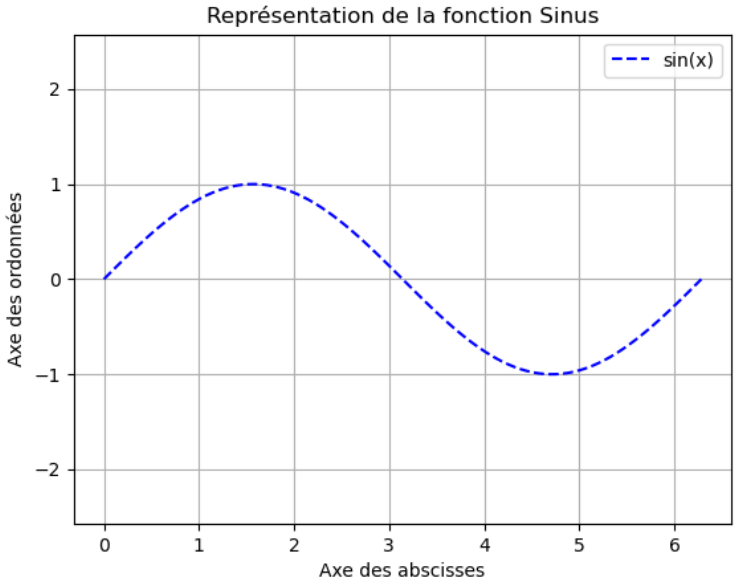
12.17.1 La fonction `plot()`

Les fonctions `plt.plot` attendent des tableaux de points. La fonction **`plot(x, y)`** relie les points dont les **abscisses** sont dans **x** et les **ordonnées** dans **y**.
Essayer l'exemple suivant :

```
import matplotlib.pyplot as plt
import numpy as np
from math import pi

x = np.linspace(0, 2*pi, 100)
y = np.sin(x)

plt.plot(x, y, label="sin(x)", color="blue", linestyle="--")
plt.axis("equal")    # Échelle identique sur les deux axes
plt.xlabel("Axe des abscisses")
plt.ylabel("Axe des ordonnées")
plt.title("Représentation de la fonction Sinus")
plt.legend()
plt.grid(True)
plt.show()
```



12.17.2 Styles et Multi-courbes

Il est possible de superposer deux courbes sur le même graphique :

```
x = np.linspace(-pi, pi, 100)
plt.plot(x, np.sin(x), 'r-', label="Sinus")
plt.plot(x, np.cos(x), 'g--', label="Cosinus")
plt.scatter(0, 0, color='black', marker='o')
plt.legend()
plt.show()
```

12.17.3 Manipuler plusieurs graphiques

Fenêtres multiples

On peut utiliser plusieurs fenêtres graphiques simultanément en stockant les figures dans des variables :

```
t = np.linspace(0, 2*pi, 50)
fig1 = plt.figure()
```

```
plt.plot(t, np.cos(t), label="cos")
fig2 = plt.figure()
plt.plot(t, np.sin(t), label="sin")
plt.show()
```

Sous-graphiques (`subplot`)

La commande `subplot(n, m, k)` subdivise la fenêtre en $n \times m$ cases. k spécifie la case active (de gauche à droite, haut en bas).

```
plt.subplot(2, 1, 1) # 2 lignes, 1 colonne, 1ère case
plt.plot(t, np.cos(t))
plt.subplot(2, 1, 2) # 2ème case
plt.plot(t, np.sin(t))
plt.show()
```

12.17.4 Graphiques à 3 dimensions : Les courbes

Il est possible de tracer des courbes dans l'espace avec la librairie `Axes3D`. Exemple d'une hélice :

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.gca(projection='3d')
t = np.linspace(0, 4*np.pi, 100)
plt.plot(np.cos(t), np.sin(t), t, label="helice")
plt.show()
```

12.17.5 Options de personnalisation

Pour connaître toutes les options, le mieux est de se référer à la documentation de `Matplotlib`. Voici les principales :

- **Titres et légendes :** `plt.title("Mon titre")`, `plt.legend()`.

- **Lignes :** `linewidth=2.0` (épaisseur), `linestyle='-','--','-.',':'`.
- **Couleurs :** `'g'` (vert), `'r'` (rouge), `'k'` (noir), `'b'` (bleu), `'c'` (cyan), `'m'` (magenta), `'y'` (jaune).
- **Bornes :** `plt.axis([xmin, xmax, ymin, ymax])`.
- **Symboles (markers) :** Permet de marquer les points de données : `['o', 's', '+', '*', 'x', 'd', 'v', '^']`.

12.17.6 Série d'exercices

Exercice 12.18 — Géométrie et trajectoire ★

1. Écrire un script qui demande les coordonnées de trois points $A(x_1, y_1), B(x_2, y_2), C(x_3, y_3)$ et trace le triangle ABC.
2. Réaliser le graphe de la fonction hauteur $y(t) = v_0 t - \frac{1}{2} g t^2$ pour $v_0 = 10, g = 9.81$ et $t \in [0, 2\frac{v_0}{g}]$.

Exercice 12.19 — Ondes et échantillonnage ★★

1. Tracer $x \longmapsto \cos(x)$ sur $[-5, 5]$ avec `np.arange(-5, 5, 0.1)`.
2. La fonction $f(x, t) = e^{-(x-3t)^2} \sin(3\pi(x - t))$ décrit une onde. Visualisez cette fonction pour $x \in [-4, 4]$ à $t = 0$.

Exercice 12.20 — Courbes paramétriques et statistiques ★★★

1. **Lissajous :** Tracer $x(t) = \sin(3t + \pi/2)$ et $y(t) = \sin(2t)$ pour $t \in [0, 2\pi]$.
2. **Cœur :** Tracez la cardioïde définie par $x = 16\sin^3(t)$ et $y = 13\cos(t) - 5\cos(2t) - 2\cos(3t) - \cos(4t)$. Utilisez `plt.fill(x, y, 'r')`.

Exercice 12.21 — Comparaison de signaux et Subplots ★★

On souhaite comparer deux signaux amortis.

1. Définir deux fonctions $f_1(t) = e^{-t}\cos(2\pi t)$ et $f_2(t) = e^{-0.5t}\cos(2\pi t)$ sur l'intervalle $t \in [0, 10]$.

2. Créer une figure contenant deux sous-graphiques superposés (`subplot`) :
   - En haut : tracer f_1 en ligne continue bleue avec une grille.
   - En bas : tracer f_2 en ligne pointillée rouge.
3. Ajouter un titre global à la figure et nommer les axes de chaque graphique.

Exercice 12.22 — Polygones et personnalisation ★★

L'objectif est de tracer des polygones réguliers.

1. Rappeler les équations paramétriques (x, y) d'un cercle de rayon R.
2. En utilisant `np.linspace`, tracer un hexagone régulier inscrit dans un cercle de rayon $R = 5$ (utiliser 7 points pour fermer la boucle).
3. Colorier l'intérieur de l'hexagone en jaune et marquer chaque sommet par un carré noir (`'ks'`).
4. Régler les axes avec `plt.axis("equal")` pour que l'hexagone ne paraisse pas aplati.

Exercice 12.23 — Exploration de la 3D : L'Hélice Conique ★★★

En vous inspirant de l'exemple de l'hélice cylindrique :

1. Créer une hélice conique définie par :

$$x(t) = t \cdot \cos(t), \quad y(t) = t \cdot \sin(t), \quad z(t) = t$$

   pour $t \in [0, 10\pi]$.
2. Tracer cette courbe en 3D en utilisant une ligne d'épaisseur 2 et de couleur `magenta`.
3. Ajouter des étiquettes aux axes : "Axe X", "Axe Y" et "Altitude Z".

Exercice 12.24 — Analyse de données réelles (Simulation) ★★★

On simule le relevé de température d'une salle sur 24h.

1. Créer un vecteur `heures` de 0 à 23.
2. Créer un vecteur `temp` contenant des valeurs aléatoires fluctuant autour de 20°C (utiliser `20 + np.random.randn(24)`).

3. Tracer les points avec `plt.scatter` (nuage de points) en utilisant des étoiles comme marqueurs.
4. Calculer la moyenne des températures et tracer une ligne horizontale (utiliser `plt.axhline`) représentant cette moyenne.

Note : *Pour les versions récentes de Matplotlib, la syntaxe 3D recommandée est :*
`ax = fig.add_subplot(111, projection='3d')`

Chp. 13

Résolution numérique d'équations

$f(x) = 0$

❒ Objectifs

L'objectif de ce TD est d'implémenter et de comparer des méthodes algorithmiques permettant de trouver une solution approchée d'une équation que l'on ne sait pas résoudre analytiquement.

- ✦ Comprendre et implémenter la méthode par **dichotomie**.
- ✦ Comprendre et implémenter la **méthode de Newton** (tangentes).
- ✦ Analyser la vitesse de convergence et la précision des résultats.
- ✦ Utiliser les fonctions de résolution de la bibliothèque `scipy.optimize`.

13.1 Introduction

De nombreux problèmes en chimie, mathématiques, physique et sciences de l'ingénieur nécessitent la résolution d'équations de type $h(x) = g(x)$ où h et g sont des fonctions et où x est l'inconnue.

Lorsque la résolution formelle de l'équation n'est pas simple, ou même impossible, on utilise une *résolution numérique*. Pour cela, on se ramène à une équation avec second membre nul : $f(x) = g(x) - h(x) = 0$.

13.2 La méthode de dichotomie

13.2.1 Principe et Théorème

Le théorème des valeurs intermédiaires (TVI) nous assure que si f est continue sur $[a, b]$ et que $f(a) \times f(b) < 0$, alors il existe au moins une solution $c \in [a, b]$ telle que $f(c) = 0$. La méthode de dichotomie consiste à diviser l'intervalle par deux à chaque étape pour "emprisonner" cette solution.

13.2.2 Algorithme et Implémentation

L'amplitude de l'intervalle après n itérations est donnée par : $L_n = \frac{b-a}{2^n}$

Algorithme 4 . Algorithme de dichotomie

Entrée : Une fonction f, bornes a et b, précision ϵ

Sortie : Un intervalle $[u, v]$ de largeur $\leq \epsilon$ contenant la racine

```
1 u ← a
2 v ← b
3 tant que (v − u) > ϵ faire
4     m ← (u + v)/2
5     si f(u) × f(m) < 0 alors
6         v ← m
7     sinon
8         u ← m
9 retourner u, v
```

Exercice 13.1 — Vitesse de convergence ★★

Soit $p \in \mathbb{N}$. Peut-on évaluer le nombre n d'itérations qui permettront d'obtenir $b_n - a_n \leq 10^{-p}$? Justifiez votre réponse en isolant n dans l'inéquation $\frac{b-a}{2^n} \leq 10^{-p}$.

Exercice 13.2 — Mise en pratique : racines de polynômes ★★

On choisit la fonction $f : x \longmapsto x^2 + 6x + 1$.

1. Calculez ses racines formellement (méthode du discriminant).

2. Déterminez des valeurs approchées à 10^{-12} près de chacune des racines avec la fonction `dicho(f, a, b, eps)`.
3. **Analyse :** Ajoutez un compteur dans votre fonction pour vérifier si le nombre d'itérations réel correspond à votre calcul théorique de l'exercice précédent.

13.3 Méthode de Newton

13.3.1 Principe de la tangente

La méthode de Newton (ou méthode de la tangente) est souvent beaucoup plus rapide que la dichotomie. Au lieu de couper l'intervalle en deux, on utilise la dérivée de la fonction pour "projeter" la solution suivante.

L'équation de la tangente au point x_n est : $y = f'(x_n)(x - x_n) + f(x_n)$. Le point d'intersection de cette tangente avec l'axe des abscisses donne x_{n+1} :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

13.3.2 Implémentation en Python

Exercice 13.3 — Étude d'une équation transcendante ★★★

On considère l'équation $(E) : x - e\sin(x) = 0$ avec $e = 0.5$. On cherche la solution $\alpha \in [1.5, 5]$.

1. Définissez en Python la fonction $f(x)$ et sa dérivée $f'(x)$.
2. Implémentez la fonction `newton(f, fprime, u0, eps)` vue en cours.
3. Comparez le nombre d'itérations nécessaire pour atteindre une précision de 10^{-8} avec Newton versus la dichotomie.

☞ Attention

La méthode de Newton peut diverger ou ne pas converger vers la racine attendue si le point de départ x_0 est mal choisi ou si $f'(x_n)$ devient proche de zéro.

13.4 Outils Scipy

En pratique, on utilise souvent les fonctions optimisées du module `scipy.optimize`.

Exercice 13.4 — Utilisation des fonctions standards ★

Testez le code suivant et comparez les résultats avec vos propres fonctions :

```python
from scipy.optimize import bisect, newton

def f(x): return x**2 - 2
def df(x): return 2*x

res_dicho = bisect(f, 1, 2)
res_newton = newton(f, 1, fprime=df)

print(f"Dichotomie : {res_dicho}")
print(f"Newton : {res_newton}")
```

Chp. 14

Intégration numérique : Rectangles et Trapèzes

❒ Objectifs

L'objectif de ce TD est d'étudier les méthodes permettant d'estimer numériquement l'aire sous une courbe lorsque le calcul de primitive est complexe ou impossible.
L'objectif de ce TD est de :

- ✦ Comprendre le principe de subdivision d'un intervalle.
- ✦ Implémenter la **méthode des rectangles** (approche constante).
- ✦ Implémenter la **méthode des trapèzes** (approche affine).
- ✦ Comparer la précision des méthodes sur des fonctions tests.

14.1 Introduction

De nombreux problèmes en chimie, mathématiques, physique et sciences de l'ingénieur nécessitent le calcul numérique d'intégrales. Le principe général consiste à découper l'intervalle d'intégration en petits morceaux, et approcher sur chacun de ces intervalles la courbe représentative de la fonction f par une forme géométrique simple.

Étant donnés deux réels $a < b$ et une fonction f continue sur $[a, b]$, on cherche une valeur approchée de :

$$I(f) = \int_a^b f(t)\,dt$$

On partage le segment $[a, b]$ à l'aide d'une subdivision $(a_i)_{0\leq i\leq n}$ à pas constant $h = \frac{b-a}{n}$ en posant :

$$\forall i \in [0, n], \quad a_i = a + i\frac{b-a}{n}$$

14.2 La méthode des rectangles

14.2.1 Principe

Cette méthode consiste à approcher f sur chaque sous-intervalle par une fonction constante prenant la valeur de f à gauche de l'intervalle. L'intégrale est approchée par la somme $R_n(f)$ des aires des rectangles ainsi formés.

Chaque rectangle a une largeur $h = \frac{b-a}{n}$ et une hauteur $f(a_i)$, d'où :

$$R_n(f) = \frac{b-a}{n} \sum_{i=0}^{n-1} f(a_i)$$

14.2.2 Algorithme et Implémentation

Algorithme 5 . Méthode des rectangles à gauche

Entrée : Fonction f, bornes a et b, nombre de subdivisions n

Sortie : Valeur approchée de l'intégrale

```
S ← 0
h ← (b − a)/n
pour k allant de 0 à n − 1 faire
    S ← S + f(a + k × h)
retourner S × h
```

Exercice 14.1 — Calcul manuel et script ★

On souhaite calculer $I = \int_1^5 x \ln(x)\, dx$.

1. Écrire une fonction Python `f(x)` utilisant `numpy.log`.
2. Implémenter la fonction `rectangle(f, a, b, n)`.
3. Pour $n = 1000$, vérifiez que vous obtenez $I \approx 14.1018$.

14.3 La méthode des trapèzes

14.3.1 Principe

On approche ici la courbe sur le segment $[a_i, a_{i+1}]$ par un segment de droite reliant les points $(a_i, f(a_i))$ et $(a_{i+1}, f(a_{i+1}))$. On calcule alors une somme d'aires de trapèzes.

L'aire d'un trapèze élémentaire est $h \times \frac{f(a_i)+f(a_{i+1})}{2}$. La somme totale est :

$$T_n(f) = \frac{b-a}{2n} \sum_{i=0}^{n-1} (f(a_i) + f(a_{i+1}))$$

14.3.2 Algorithme Python

```python
def trapezes(f, a, b, n):
    h = (b - a) / n
    somme = 0.0
    for k in range(n):
        somme += (f(a + k*h) + f(a + (k+1)*h)) / 2
    return somme * h
```

Exercice 14.2 — Comparaison des méthodes ★★

En reprenant $I = \int_1^5 x \ln(x)\, dx$:

1. Calculez la valeur exacte de I par une intégration par parties (Rappel : $(\frac{1}{2}x^2 \ln x - \frac{1}{4}x^2)' = x \ln x$).
2. Calculez l'erreur absolue $|I_{exacte} - I_{approche}|$ pour les rectangles et les trapèzes avec $n = 100$. Quelle méthode est la plus précise ?

Exercice 14.3 — Loi normale ★★★

En probabilités, on utilise souvent la fonction $f(t) = e^{-t^2}$. L'intégrale $J = \int_0^1 e^{-t^2}\, dt$ ne possède pas de primitive exprimable avec les fonctions usuelles.

1. Déterminez une valeur approchée de J à 10^{-4} près par la méthode de votre choix.

2. Testez l'influence de n (prenez $n = 10, 100, 1000$) sur la stabilité du résultat.

14.4 Déjà disponible en Python

La bibliothèque `scipy.integrate` propose des fonctions très performantes :

- **`trapz(y, x)`** : Méthode des trapèzes à partir de tableaux de données.
- **`quad(f, a, b)`** : La méthode de référence, utilisant des algorithmes de quadrature adaptative.

Exercice 14.4 — Utilisation de Scipy ★

Exécutez le script suivant et comparez avec vos fonctions :

```python
from scipy.integrate import quad
import numpy as np

f = lambda x : x * np.log(x)
valeur, erreur_estimee = quad(f, 1, 5)
print(f"Résultat Quad : {valeur}")
```

Chp. 15

Résolution des Équations Différentielles : Méthode d'Euler

❒ Objectifs

Après les équations numériques et l'intégration, nous abordons la résolution numérique d'équations différentielles linéaires du premier ordre (EDL1).
L'objectif de ce TD est de :

- Comprendre le formalisme du **problème de Cauchy**.
- Implémenter la **méthode d'Euler** pour approcher une solution.
- Analyser l'impact du pas de subdivision h sur la précision.
- Utiliser la fonction `odeint` de `scipy.integrate` pour une résolution professionnelle.

15.1 Définitions Fondamentales

Exercice 15.1 — Compréhension des définitions ★

Parmi les équations suivantes, identifiez l'ordre et déterminez si elles sont linéaires :

1. $y' + 3x^2 y = e^x$
2. $y'' + \sin(y) = 0$
3. $a(t)y' + b(t)y = 0$

Formalisme Python

Les équations différentielles doivent être écrites sous la forme : $y' = F(y, t)$.
Exemple : $y' + 2ty = 1$ devient $F(y, t) = 1 - 2ty$.

15.2 Le Problème de Cauchy

Un problème de Cauchy est la donnée d'une équation différentielle et d'une **condition initiale** :

$$\begin{cases} y' = F(y, t) \\ y(t_0) = y_0 \end{cases}$$

15.3 La Méthode d'Euler

Le principe est d'approcher la courbe de la solution par une suite de segments de tangentes. En partant de (t_k, y_k), on calcule le point suivant par :

$$y_{k+1} = y_k + h \times F(y_k, t_k)$$

où h est le pas de temps.

15.3.1 Algorithme de la méthode d'Euler

Algorithme 6 . Méthode d'Euler

Entrée : Fonction F, intervalle $[a, b]$, condition initiale y_0, pas h
Sortie : Listes des temps T et des valeurs approchées Y

$y \leftarrow y_0, t \leftarrow a$
$T \leftarrow [a], Y \leftarrow [y_0]$
tant que $t + h \leq b$ **faire**
    $y \leftarrow y + h \times F(y, t)$
    $t \leftarrow t + h$
    Ajouter t à T et y à Y
retourner T, Y

Exercice 15.2 — Mise en œuvre manuelle ★★

On considère $y' = y$ avec $y(0) = 1$ sur $[0, 1]$ avec un pas $h = 0.5$.

1. Calculez manuellement y_1 et y_2.
2. Comparez avec la solution exacte $y(t) = e^t$ au point $t = 1$.

15.4 Utilisation de `scipy.integrate.odeint`

Pour des résolutions plus robustes, on utilise `odeint(F, y0, t)`. Contrairement à notre fonction Euler, `odeint` utilise des algorithmes à pas variable pour garantir une précision optimale.

Exercice 15.3 — Application : Problème de Cauchy ★★

Résoudre sur $[0, 5]$: $\frac{dy}{dt} = -ty + \cos(t)$ avec $y(0) = 1$.

1. Définissez la fonction $F(y, t)$.
2. Utilisez `odeint` pour obtenir la solution.
3. Tracez la courbe avec `matplotlib`.

15.5 Applications et Approfondissements

Exercice 15.4 — Cinétique chimique ★★★

La disparition d'un réactif A suit une loi d'ordre 1 : $\frac{d[A]}{dt} = -\alpha[A]$. On donne $\alpha = 1\, s^{-1}$ et $[A]_0 = 1\, mol/L$.

1. Définissez le problème de Cauchy associé.
2. Comparez graphiquement la solution obtenue par Euler ($h = 0.5$) et celle de `odeint` sur l'intervalle $[0, 6]$.

Exercice 15.5 — Analyse de l'erreur d'Euler ★★

On considère l'équation $y' = -y$ avec $y(0) = 1$ sur l'intervalle $[0, 2]$. La solution exacte est $y(t) = e^{-t}$.

1. Implémentez la méthode d'Euler pour trois valeurs du pas : $h_1 = 0.5$, $h_2 = 0.1$ et $h_3 = 0.01$.
2. Calculez l'erreur finale $E_h = |y_{exact}(2) - y_{estim}(2)|$ pour chaque pas.
3. **Observation :** Comment évolue l'erreur lorsque le pas h est divisé par 10 ?

Exercice 15.6 — Circuit RC (Physique) ★★

La charge $q(t)$ d'un condensateur vérifie : $R\frac{dq}{dt} + \frac{1}{C}q = E$. Données : $R = 1000\,\Omega$, $C = 10^{-6}\,F$, $E = 5\,V$ et $q(0) = 0$.

1. Réécrire l'équation sous la forme $q' = F(q, t)$.
2. Utiliser `odeint` pour simuler la charge sur $t \in [0, 0.01]\,s$.
3. Déterminez graphiquement le temps pour atteindre 95% de la charge maximale.

Exercice 15.7 — Modèle de population (Verhulst) ★★★

L'évolution d'une population P est modélisée par : $\frac{dP}{dt} = rP\left(1 - \frac{P}{K}\right)$ avec $r = 0.5$, $K = 1000$ et $P(0) = 100$.

1. Définissez la fonction $F(P, t)$ en Python.
2. Résolvez le problème avec la méthode d'Euler ($h = 0.1$) sur 20 unités de temps.
3. **Analyse :** Que se passe-t-il si vous commencez avec $P(0) = 1500$?

Exercice 15.8 — Systèmes d'équations ★★★

Le mouvement d'un projectile sans frottement est régi par le système :

$$\begin{cases} v_x' = 0 \\ v_y' = -g \end{cases}$$

En utilisant un vecteur d'état $Y = [x, y, v_x, v_y]$, implémentez la résolution du tir d'un projectile avec $v_0 = 50\,m/s$ et $\theta = 45°$.