UNIVERSITÉ D'ABOMEY-CALAVI
—
FACULTÉ DES SCIENCES ET TECHNIQUES

Introduction à la programmation avec Python

Pour la deuxième année de licence MIA

Clément Adandé
clement.adande@imsp-uac.org

—
Année 2024–2025

Version du 19 mai 2025

Table des matières

Introduction

C'est quoi Python ?

Python est un langage de programmation de haut niveau, interprété, multi-paradigmes, créé par Guido van Rossum en 1991. Il est conçu pour être simple à lire et à écrire, avec une syntaxe claire et expressive. Python met l'accent sur la lisibilité du code et la productivité du programmeur.

Pourquoi le connaître ?

Python est devenu l'un des langages de programmation les plus populaires au monde pour plusieurs raisons :

- **Facilité d'apprentissage** : Sa syntaxe claire et intuitive en fait un excellent premier langage
- **Polyvalence** : Utilisé dans de nombreux domaines (développement web, science des données, intelligence artificielle, automatisation, etc.)
- **Grande communauté** : Vaste écosystème de bibliothèques et frameworks
- **Forte demande** : Compétence très recherchée sur le marché du travail

Comment l'utiliser ?

Python peut être utilisé de deux manières principales :

- **Mode interactif** : Idéal pour tester rapidement des commandes et explorer le langage
- **Mode script** : Pour écrire des programmes plus complexes et réutilisables

Une façon pratique d'utiliser Python est via des **notebooks interactifs** comme **Jupyter Notebook** (inclus avec Anaconda) ou **Google Colab** (colab.research.google.com), qui permettent d'écrire du code, d'exécuter des cellules et de visualiser les résultats dans un même environnement.

Google Colab fonctionne entièrement en ligne et ne nécessite aucune installation. C'est une excellente solution pour les étudiants qui n'ont pas encore d'ordinateur : ils peuvent y accéder via leur **smartphone** en utilisant un navigateur mobile (de préférence Google Chrome) ~~et un clavier Bluetooth ou un clavier virtuel~~.

Comment l'installer ?

Python est préinstallé sur la plupart des systèmes Linux et macOS. Pour Windows, vous pouvez télécharger l'installateur depuis le site officiel (www.python.org). Assurez-vous d'installer Python 3 (au moins la version 3.6 ou supérieure).

Une alternative simple consiste à installer la distribution **Anaconda**, qui inclut Python ainsi que de nombreux outils et bibliothèques scientifiques préinstallés (anaconda.com).

Pour vérifier si Python est installé, ouvrez un terminal et tapez :

```
python --version
# ou
python3 --version
```

Code 1 – Vérification de la version de Python

Sous Windows, ceux qui ont installé Anaconda peuvent accéder au terminal Anaconda (*Anaconda Prompt*) en le recherchant dans le menu Démarrer ; ils pourront ensuite y exécuter des commandes Python ou lancer des outils comme Jupyter Notebook.

Notions abordées dans ce TP

Ce TP couvrira les notions fondamentales de la programmation avec Python :

— Variables et types de données
— Opérations et expressions
— Fonctions prédéfinies et personnalisées
— Structures de contrôle (conditions et boucles)
— Structures de données (listes, dictionnaires, etc.)

Évaluation des progrès

Votre progression sera évaluée à travers :
— Des exercices pratiques à réaliser pendant les séances de TP
— Des mini-projets à développer en autonomie à la fin de chaque séance
— Un projet final qui mettra en application l'ensemble des concepts abordés
— Des quiz réguliers pour tester vos connaissances théoriques

Chapitre 1

Premier pas

Ce chapitre vous guidera à travers les concepts fondamentaux de la programmation avec Python. Nous commencerons par des opérations simples, découvrirons les variables et explorerons les différents types de données. Ces bases constituent le socle sur lequel vous construirez votre expertise en programmation.

1 Calculer avec Python

Ouvrez le terminal et entrez la commande : `python3`. Vous obtiendrez quelque chose similaire à ceci :

```
Python 3.11.5 (main, Sep 11 2023, 13:54:46) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Code 2 – Accès à Python en ligne de commande

Vous êtes maintenant dans l'interpréteur Python. Les symboles >>> indiquent que Python est prêt à recevoir des instructions. À la suite de chaque instruction saisie, vous devez appuyer sur la touche **Entrer** de votre clavier pour permettre à Python d'exécuter votre instruction et produire un résultat. Essayons quelques calculs simples :

```
>>> 2+3
5
>>> 10 - 7 # Les espaces rendent le code plus lisible
3
>>> 4 * 5
20
>>> 15 / 3
5.0
>>> 17 / 3
5.666666666666667
>>> 17 // 3  # Division entière
5
>>> 17 % 3   # Modulo (reste de la division)
2
>>> 2 ** 3   # Puissance
8
```

Code 3 – Opérations arithmétiques de base en Python

Vous devez remarquer deux choses ici :

1. Chaque opération est écrite sur une ligne et constitue une **instruction**.
2. Tout ce qui suit le symbole # est ignoré par Python : il s'agit de détails ajoutés pour faciliter la compréhension de vos codes, appelés **commentaires**.

Python respecte les règles de priorité des opérations mathématiques :

```
>>> 2 + 3 * 4
14
>>> (2 + 3) * 4
20
```

Code 4 – Priorité des opérations mathématiques

2 Variables

Une variable est un espace mémoire nommé qui permet de stocker une valeur. Cette valeur peut être modifiée ou utilisée (plus tard) au cours de l'exécution du programme.

2.1 Nommer les variables

En Python, les noms de variables doivent :

1. Commencer par une **lettre** ou un **underscore** (_)
2. Contenir uniquement des lettres, des chiffres et des underscores
3. Être **sensibles à la casse** (majuscules et minuscules sont différentes)

Les mots réservés suivants ne peuvent pas être utilisés comme noms de variables :

```
and, as, assert, break, class, continue, def, del, elif, else, except, False,
finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not,
or, pass, raise, return, True, try, while, with, yield
```

Code 5 – Mots réservés en Python

2.2 Affectation de variables

En Python, la déclaration de variables se fait par affectation (ou assignation) et le type d'une variable n'est pas précisé à sa déclaration, contrairement aux langages de bas niveau comme le C. L'affectation se fait avec l'opérateur = :

```
>>> x = 10       # Un entier
>>> y = 3.14     # Un flottant
>>> nom = "Alice" # Une chaîne de caractères
>>> est_vrai = True # Un booléen
```

Code 6 – Affectation de variables avec différents types

Ces instructions ont déclenché plusieurs opérations dans la mémoire de l'ordinateur :

— créer et mémoriser un **nom de variable** ;
— lui assigner automatiquement un **type** ;
— créer et mémoriser une **valeur** particulière ;
— **relier** le nom de la variable à l'emplacement mémoire de la valeur associée.

Pour afficher la valeur d'une variable dans l'interpréteur, il suffit de taper son nom :

```
>>> X # Ceci produit une erreur, rappelez-vous Python est sensible à la casse
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'X' is not defined
>>> x
10
>>> nom
'Alice'
```

Code 7 – Accès aux valeurs des variables

Mais dans vos scripts Python, vous devez utiliser la fonction **print**() pour afficher la valeur d'une variable. Notez qu'on peut aussi l'utiliser dans l'interpréteur.

```
>>> print(x)
10
>>> print(nom)
Alice
```

Code 8 – Utilisation de la fonction print()

Les variables peuvent être réutilisées dans des expressions :

```
>>> z = x + y
>>> z
13.14
>>> x = x + 1  # Incrémentation
>>> x
11
```

Code 9 – Réutilisation de variables dans des expressions

En Python, il est possible de déclarer plusieurs variables simultanément ou parallèlement :

```
>>> L = l = h = 4.0         # Déclaration simultanée
>>> L
4.0
>>> # Quelle est la différence entre ces deux nombres 4.0 et 4 en Python ?
>>> # 4.0 est un flottant (float) tandis que 4 est un entier (int)
>>> r, pi = 1.5, 3.14    # Déclaration en parallèle
>>> r
1.5
>>> pi
3.14
```

Code 10 – Déclaration simultanée et parallèle de variables

2.3 Types de données de base

Python possède plusieurs types de données natifs. Jusqu'ici nous avons manipulé des entiers **int**, des flottants **float**, des chaînes de caractères **str** et des booléens **bool**. Plus tard, nous allons aborder quelques autres types de données qui rendent Python puissant.

```
>>> type(10)
<class 'int'>          # Entier
>>> type(3.14)
<class 'float'>        # Flottant
>>> type("Bonjour")
<class 'str'>          # Chaîne de caractères
>>> type(True)
<class 'bool'>         # Booléen
```

Code 11 – Types de données de base en Python

3 Opérations et expressions

En combinant des *opérateurs*, on peut utiliser les valeurs et les variables pour former des *expressions*.

```
>>> L, l, h = 4.0, 4.0, 4.0    # Correction ici
>>> volumeCube = L * l * h
>>> volumeCube
64.0
>>> r, pi = 4, 3.14
>>> aireCercle = pi * r ** 2
>>> aireCercle
50.24
>>> volumeCylindre = aireCercle * h
>>> volumeCylindre
200.96
>>> volumeCube + volumeCylindre
264.96
```

Code 12 – Combinaison de variables et opérateurs dans des expressions

Nous avions vu qu'à l'instar de nombreux langages de programmation, Python permet d'effectuer des opérations arithmétiques avec des nombres. Il est possible de manipuler des expressions bien plus complexes et utilisant d'autres types de variables que **int** ou **float**.

3.1 Opérations avec les chaînes

L'une des opérations courantes avec les chaînes de caractères est la concaténation.

```
>>> prenom = "Alice"
>>> nom = "Kandji"
>>> prenom + " " + nom          # Concaténation
'Alice Kandji'
>>> prenom * 3                  # Répétition
'AliceAliceAlice'
>>> len(prenom)                 # Longueur de la chaîne
5
>>> "J'ai 15 ans."
"J'ai 15 ans."
>>> 'J\'ai 15 ans.'                 # Notez la différence entre " et '
"J'ai 15 ans."
>>> prenom + " a 15 ans."       # Notez la différence
'Alice a 15 ans.'
>>> age = 15
>>> prenom + " a " + age  + " ans"  # Qu'est-ce qui ne va pas ici?
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> prenom + " a " + str(age) + " ans"
'Alice a 15 ans'
```

Code 13 – Opérations avec les chaînes de caractères

Il y a deux choses à retenir ici :

1. Les chaînes de caractères s'écrivent entre les symboles d'apostrophes (') ou bien guillemets ("). Pour utiliser l'un de ces symboles au sein d'une chaîne de caractères comme dans la phrase suivante : J'ai 15 ans. ou Entrez votre "nom" :, il suffit de les précéder du caractère antislash (\) à l'intérieur de la chaîne pour que Python les considère comme un caractère spécial.
2. La concaténation se fait uniquement avec les variables de type **str**. Pour concaténer une chaîne de caractères et un nombre, il faut convertir le nombre en **str** en utilisant **str**() comme on a vu avec la dernière instruction.

3.2 Opérations de comparaison

Python permet d'effectuer des comparaisons :

```
>>> a = 10
>>> b = 3
>>> a > b    # Supérieur à
>>> a < b    # Inférieur à
>>> a == b   # Égal à
>>> a != b   # Différent de
>>> a >= b   # Supérieur ou égal à
>>> a <= b   # Inférieur ou égal à
```

Code 14 – Opérations de comparaison en Python

Chacune des opérations de comparaison précédentes a renvoyé l'une des deux valeurs **True** ou **False**. La valeur True est renvoyée lorsque l'expression est vérifiée et False dans le cas contraire. En fait, True et False sont appelés **valeurs logiques**, et sont le résultat de l'évaluation d'une *expression logique*.

3.3 Opérations logiques

Python offre également la possibilité de construire des expressions logiques plus complexes au moyen des connecteurs logiques.

```
>>> x = True
>>> y = False
>>> x and y  # ET logique
False
>>> x or y   # OU logique
True
>>> not x    # NON logique
False
```

Code 15 – Opérations logiques en Python

3.4 Priorité des opérations

De la plus haute à la plus basse priorité :

1. Parenthèses `()`
2. Puissance `**`
3. Multiplication `*`, division `/`, division entière `//`, modulo, `%`
4. Addition `+`, soustraction `-`
5. Comparaisons `==`, `!=`, `<`, `>`, `<=`, `>=`
6. NOT logique `not`
7. AND logique `and`
8. OR logique `or`

3.5 Écrire des scripts Python

Dans ce chapitre jusqu'à présent, nous avions utilisé Python de manière interactive à l'aide de l'interpréteur. Cette façon d'utiliser Python est convenable pour des essais rapides des instructions. Mais elle ne permet pas de réutiliser ultérieurement toutes les instructions que vous aviez saisies. En fait, lorsque vous quittez l'interpréteur vous perdez tout (*ou presque*).

Vous pouvez sauvegarder les instructions dans des fichiers textes avec l'extension `.py`. Il existe plusieurs logiciels qui vous permettent de créer, lire et modifier des fichiers textes. De tels logiciels sont appelés **éditeurs de texte**. Par défaut, vous avez **Gedit** sous Linux et **Bloc-notes** sous Windows.

Ouvrez Gedit ou Bloc-notes, et copiez ce code :

```
nom = input("Tapez votre nom: ")
age = input("Tapez votre âge: ")

print("Bienvenu.e " + nom + ". Vous avez " + age + " ans.")
```

Code 16 – Premier script Python avec interaction utilisateur

Enregistrez ce fichier sous le nom `script1.py`. Pour exécuter votre programme, ouvrez le terminal et tapez la ligne de commande :

```
$ python3 script1.py
```

Code 17 – Exécution d'un script Python en ligne de commande

Assurez-vous de spécifier le chemin de python juste après avoir écrit `python3`.

4 Exercices

1. Un rectangle a pour dimensions 5 et 3. Définir les variables `longueur` et `largeur`, puis calculer l'aire de la surface du rectangle.
2. Assigner les valeurs 3, 5, 7 à trois variables `a`, `b` et `c`. Effectuer l'opération `a-b//c`, et interpréter le résultat obtenu.
3. Considérons les instructions suivantes.

```
>>> r, pi = 11, 3.14
>>> s = pi * r**2
```

Code 18 – Analyse des types de variables

Afficher le type de chacune des variables `r`, `pi` et `s`.

4. Considérons les instructions suivantes.

```
>>> a, b = 4, 7    # Correction de la syntaxe
>>> print(a, b)
4 7
>>> c = b
>>> b = a
>>> a = c
>>> print(a, b)
7 4
```

Code 19 – Échange de valeurs entre variables

Décrire chacune de ces instructions. Que peut-on dire des variables `a` et `b`?

5. Définir une variable `pub` dont la valeur est la suivante :

```
******************************

**** Commandez maintenant ****
********** En promo **********

******************************
```

Code 20 – Chaîne multiligne avec caractères spéciaux

(La première ligne comporte 30 astérisques et le caractère spécial `\n` permet d'aller à la ligne.)

6. Assigner les valeurs 9, 5 à deux variables `a`, `b`. Quelle est la valeur de cette expression logique : `a % b == 0`?

En résumé, ce chapitre vous a présenté les fondements essentiels de Python : l'utilisation de l'interpréteur, les opérations de base, la manipulation des variables et les différents types de données. Vous avez également découvert comment créer vos premiers scripts Python. Ces concepts constituent la base sur laquelle vous bâtirez votre compétence en programmation. Dans les chapitres suivants, nous explorerons des structures de contrôle plus avancées qui vous permettront de créer des programmes plus dynamiques et interactifs.

Lien vers le quiz : https://forms.gle/yJyezXMcC8q3CUV67
Deadline : ~~Samedi 03 mai 2025 à 22 : 00~~ Dimanche 11 mai 2025 à 20 : 00

Chapitre 2

Contrôle de flux d'exécution

Dans ce chapitre, nous allons explorer les structures qui permettent de contrôler l'exécution de votre programme Python. Jusqu'à présent, nous avons vu comment écrire des instructions qui s'exécutent séquentiellement. Mais pour créer des programmes dynamiques et interactifs, nous devons apprendre à prendre des décisions et à exécuter différentes parties du code selon certaines conditions. Les instructions conditionnelles sont au cœur de cette logique de programmation et vous permettront de créer des programmes capables de s'adapter à diverses situations.

Python exécute les instructions les unes après les autres suivant l'ordre dans lequel elles ont été saisies dans un script. Cet ordre peut changer lorsque Python rencontre une instruction conditionnelle. Les instructions conditionnelles permettent de définir des actions précises et d'adapter le comportement de vos programmes à des cas variés.

1 Instructions conditionnelles

Les instructions conditionnelles permettent d'exécuter différentes parties de code selon qu'une condition est vraie ou fausse.

1.1 Instruction if-elif-else

Copiez et enregistrez ce code dans un script nommé `instruction1.py`. Ensuite, exécutez le code.

```
1 age = 13
2 if age < 18:
3     print("Vous êtes mineur.")  # Rappelez-vous toujours de l'indentation
```

Code 21 – Instruction conditionnelle simple avec if

Vous verrez afficher **Vous êtes mineur.**. Modifiez le script en affectant la valeur 21 à la variable `age`.

Notez bien la syntaxe d'une boucle **`if`** dans le code 21.

— L'instruction conditionnelle commence par le mot-clé **`if`**, suivi d'une **condition**, puis de deux-points `:`.

— Le bloc à exécuter si la condition est vraie est indenté (décalé vers la droite). En fait, en Python, l'**indentation** (souvent 4 espaces) remplace les accolades utilisées dans d'autres langages comme le C. Si l'indentation est incorrecte (trop ou pas assez), Python générera une erreur.

— Si la condition est fausse, le bloc indenté est simplement ignoré. Il n'y a pas d'erreur.

On peut aussi définir une instruction alternative dans le cas où la condition n'est pas vérifiée. Exécutez ce code.

```
1 age = 21
2 if age < 18:
3     print("Vous êtes mineur.")
4 else:
5     print("Vous êtes adulte en âge de travailler.")
```

Code 22 – Structure conditionnelle if-else

Notez ici que l'instruction `else` est facultative. Elle s'utilise conjointement avec l'instruction `if`, et lorsqu'elle est utilisée, elle permet de définir un bloc alternatif dans le cas où la condition n'est pas vérifiée.

On peut également gérer plusieurs conditions avec **`elif`** (contraction de "else if"). Cela permet de tester plusieurs cas différents les uns après les autres.

```
1 age = int(input("Entrez votre âge : "))
2
3 if age < 18:
4     print("Vous êtes mineur.")
5 elif age < 65:
6     print("Vous êtes adulte en âge de travailler.")
7 else:
8     print("Vous êtes senior.")
```

Code 23 – Structure conditionnelle if-elif-else

1.2 Opérateurs de comparaison

Voici différentes façons d'utiliser les opérateurs de comparaison dans les conditions :

```
1  x = 10
2  y = 20
3
4  if x == y:
5      print("x et y sont égaux")
6  if x != y:
7      print("x et y sont différents")
8  if x < y:
9      print("x est inférieur à y")
10 if x > y:
11     print("x est supérieur à y")
12 if x <= y:
13     print("x est inférieur ou égal à y")
14 if x >= y:
15     print("x est supérieur ou égal à y")
```

Code 24 – Utilisation des opérateurs de comparaison

1.3 Conditions composées

Il est possible de combiner plusieurs conditions grâce aux opérateurs logiques :

```python
age = 25
revenu = 30000

if age > 18 and revenu > 25000:
    print("Vous êtes éligible pour un prêt.")

if age < 18 or revenu < 15000:
    print("Vous n'êtes pas éligible pour un prêt standard.")

if not (age < 18):
    print("Vous êtes majeur.")
```

Code 25 – Utilisation des opérateurs logiques dans les conditions

1.4 Conditions imbriquées

Les conditions peuvent être imbriquées les unes dans les autres :

```python
note = 85

if note >= 60:
    print("Vous avez réussi.")
    if note >= 80:
        print("Vous avez obtenu une mention.")
        if note >= 90:
            print("Vous avez obtenu une mention très bien!")
else:
    print("Vous n'avez pas réussi.")
```

Code 26 – Conditions imbriquées

Evitez d'enchaîner trop de conditions imbriquées – cela nuit à la lisibilité. Préférez les instructions **`elif`**

2 Expressions conditionnelles

Python permet d'écrire des expressions conditionnelles compactes :

```python
age = 20
statut = "mineur" if age < 18 else "majeur"
print(statut)  # Affiche "majeur"
```

Code 27 – Expression conditionnelle compacte

3 Exercices

1. **Vérification de l'accès à une ressource**
   Écrire un programme qui demande à l'utilisateur son âge et son statut (« étudiant », « salarié », ou « autre »). L'accès est autorisé uniquement aux étudiants de plus de 18 ans ou aux salariés de plus de 25 ans.
2. **Détection de mots interdits**
   Demandez une phrase à l'utilisateur et vérifiez si elle contient un mot interdit (ex. : « spam », « arnaque », « pub »). Affichez un message d'alerte si nécessaire.

3. **Calcul d'une remise**
   Demandez à l'utilisateur le montant de ses achats. Appliquez une remise de :
   — 5% si le montant est entre 100 et 200 euros,
   — 10% si le montant dépasse 200 euros.
   Affichez le montant final à payer.
4. **Conversion de note en lettre**
   Demandez une note sur 20 et affichez son équivalent sur le système américain :
   — A si note ≥ 16,
   — B si note entre 14 et 16,
   — C si note entre 12 et 14,
   — D si note entre 10 et 12,
   — F sinon.
5. **Validation de mot de passe**
   Demandez à l'utilisateur un mot de passe et vérifiez :
   — qu'il contient au moins 8 caractères,
   — qu'il contient au moins un chiffre.
   Affichez « mot de passe valide » ou « mot de passe invalide ».
6. **Comparaison de deux nombres**
   Demandez deux nombres à l'utilisateur et affichez :
   — s'ils sont égaux,
   — lequel est plus grand,
   — s'ils sont à moins de 1 unité l'un de l'autre.
7. **Classification selon l'heure**
   Demandez l'heure (entre 0 et 23) et affichez :
   — « Bonne nuit » entre 0 et 6,
   — « Bonjour » entre 7 et 11,
   — « Bon après-midi » entre 12 et 18,
   — « Bonsoir » entre 19 et 23.
8. **Vérification d'un identifiant**
   Demandez un identifiant et vérifiez qu'il :
   — commence par une lettre,
   — contient au moins 6 caractères,
   — ne contient pas d'espaces.
9. **Mini formulaire d'inscription**
   Simulez l'inscription à une plateforme : demandez l'âge et le pays. Autorisez l'inscription uniquement si l'utilisateur a plus de 16 ans et habite en Bénin ou Togo.
10. **Égalité entre trois entiers**
    Demandez trois entiers à l'utilisateur. Affichez :
    — s'ils sont tous égaux,
    — s'il y en a au moins deux égaux,
    — sinon qu'ils sont tous différents.

Dans ce chapitre, nous avons appris comment contrôler le flux d'exécution de nos programmes Python grâce aux instructions conditionnelles. Ces structures nous permettent de créer des programmes qui prennent des décisions et s'adaptent en fonction des circonstances. Nous avons exploré les différentes formes de l'instruction if-elif-else, les opérateurs de comparaison, les conditions composées et les expressions conditionnelles compactes. Ces outils sont essentiels pour développer des programmes intelligents et réactifs. Dans les chapitres suivants, nous découvrirons

comment répéter des séquences d'instructions avec les boucles et comment organiser notre code en fonctions réutilisables.

Chapitre 3

Instructions répétitives

Dans ce chapitre, nous allons découvrir comment *répéter automatiquement l'exécution d'instructions* en Python grâce aux boucles. Les instructions répétitives permettent de parcourir des séquences, de répéter des calculs ou d'automatiser des tâches sans avoir à copier plusieurs fois le même code.

Python propose principalement deux typess de boucles :

— `while` : qui répète un bloc tant qu'une condition est vraie ;
— `for` : qui permet d'itérer directement sur les éléments d'une séquence (liste, chaîne, etc.).

1 Boucle while

La boucle `while` exécute un bloc de code tant qu'une condition est vraie. C'est une boucle à test initial.

```
1 i = 1                  # Initialisation du compteur
2 while i <= 5:          # Condition : tant que i est inférieur ou égal à 5
3     print(i)           # Affiche la valeur actuelle de i
4     i += 1             # Incrémente i (i = i + 1)
```

Code 28 – Instruction répétitive avec while

Notez bien la syntaxe d'une boucle **`while`**.

— La boucle commence par le mot-clé `while`, suivie d'une **condition**, puis d'un `:` (deux-points).
— Le bloc de code à répéter doit être **indenté**. Ce bloc est exécuté autant de quoi que la condition est vérifiée.

Exécutez ce code.

```
1 i = 1
2 while i <= 5:
3     print(i)
4     # Notez la différence avec le code précédent
```

Code 29 – Boucle infinie

Stoppez l'exécution de ce code avec `Ctrl+C` si vous êtes dans un terminal, ou utilisez le bouton d'arrêt si vous êtes dans un environnement comme Google Colab ou Jupyter Notebook.

Ce second exemple illustre ce qu'on appelle une **boucle infinie**. Étant donné que la variable `i` n'est jamais modifiée à l'intérieur de la boucle, la condition `i <= 5` reste toujours vraie, et le bloc `print(i)` est exécuté indéfiniment.

Cela montre l'importance de mettre à jour correctement les variables de contrôle à l'intérieur de la boucle, sous peine de voir le programme "bloqué" dans une exécution sans fin.

Que fait le code 29 ?

```
somme = 0
n = 1
while n <= 10:
    somme += n
    n += 1
print(somme)
```

Code 30 – Accumulation contrôlée avec une boucle while

2 Boucle for

La boucle `for` est utilisée pour itérer sur une séquence (liste, tuple, chaîne, etc.) ou sur une série de valeurs générées automatiquement, comme avec la fonction **`range()`**.

```
for i in range(1, 6):       # Itère de 1 à 5 inclus
    print(i)                # Affiche la valeur actuelle de i
```

Code 31 – Boucle for avec la fonction range()

Notez bien la syntaxe d'une boucle **`for`** :

— Elle commence par le mot-clé **`for`**, suivi d'une variable qui prend successivement les valeurs de la séquence.
— Le mot-clé **`in`** est utilisé pour parcourir une séquence (ici, `range(1, 6)`).
— Le bloc d'instructions à exécuter est indenté, comme pour la boucle `while`.

Le code 31 est une version équivalente du code 28 mais écrite avec une boucle **`for`**.

Génération de séquences numériques avec la fonction `range()`

La fonction **`range()`** permet de générer des séquences d'entiers à utiliser notamment dans les boucles **`for`**. Elle peut être appelée de trois façons différentes :

— **`range(stop)`** : Génère les entiers de **`0`** jusqu'à **`stop-1`**. C'est le cas le plus simple.
Exemple : `range(5)` produit `0, 1, 2, 3, 4`.
— **`range(start, stop)`** : Génère les entiers de **`start`** jusqu'à **`stop-1`**.
Exemple : `range(3, 7)` produit `3, 4, 5, 6`.
— **`range(start, stop, step)`** : Génère les entiers de **`start`** à **`stop-1`** en incrémentant de **`step`**. Le paramètre **`step`** peut être positif ou négatif.
Exemple : `range(2, 10, 2)` produit `2, 4, 6, 8`; et `range(10, 0, -2)` produit `10, 8, 6, 4, 2`.

La boucle `for` est souvent préférée lorsque le nombre d'itérations est connu à l'avance. Contrairement à `while`, il n'est pas nécessaire de gérer manuellement une variable de contrôle (comme **`i += 1`**).

Que fait le code 32 ?

```
somme = 0
for n in range(1, 11, 2):
    somme += n
print(somme)
```

Code 32 – Accumulation contrôlée avec une boucle for

3 Instructions break et continue

Dans une boucle, on peut vouloir modifier le déroulement normal de l'itération. Les instructions `break` et `continue` permettent cela.

L'instruction break

`break` sert à interrompre immédiatement l'exécution d'une boucle `for` ou `while`, même si la condition de fin n'est pas encore atteinte.

```
1 for i in range(10):
2     if i == 5:
3         break
4     print(i)
```

Code 33 – Arrêt prématuré d'une boucle avec `break`

Ce code 33 affiche les nombres de 0 à 4. Dès que `i` vaut 5, l'instruction `break` est exécutée et la boucle est interrompue.

L'instruction continue

`continue` permet de passer immédiatement à l'itération suivante de la boucle, en sautant le reste du bloc en cours.

```
1 for i in range(1, 6):
2     if i == 3:
3         continue
4     print(i)
```

Code 34 – Ignorer certaines valeurs avec `continue`

Ce code affiche les nombres 1, 2, 4, 5. La valeur 3 est "ignorée" car `continue` saute l'instruction `print(i)` quand `i == 3`.

4 Boucles imbriquées

```
1 for i in range(1, 6):
2     for j in range(1, 6):
3         print(f"{i} x {j} = {i*j}")    # Notez l'utilisation de f-string
4     print("-" * 10)
```

Code 35 – Table de multiplication des deux boucles for

5 Exercices

1. Écrivez un programme qui affiche tous les nombres de 1 à 20 en utilisant une boucle `while`.
2. Écrivez un programme qui calcule la somme des nombres de 1 à 100 en utilisant une boucle `for` et la fonction `range()`.
3. Créez un programme qui affiche les 10 premiers nombres pairs positifs en utilisant une boucle. Proposez une solution avec `while` et une autre avec `for`.

4. Écrivez un programme qui demande à l'utilisateur de saisir un nombre positif, puis affiche le compte à rebours de ce nombre jusqu'à 0.

```
nombre = int(input("Entrez un nombre positif : "))
# Votre code ici
```

Code 36 – Modèle pour l'exercice 4

5. Écrivez un programme qui calcule la factorielle d'un nombre entier positif n, entré par l'utilisateur. La factorielle, notée n!, est le produit des entiers de 1 à n.

```
nombre = int(input("Entrez un nombre entier positif : "))
factorielle = 1
# Votre code ici
```

Code 37 – Modèle pour l'exercice 5

6. La suite de Fibonacci est définie par la formule mathématique suivante :

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \text{pour tout } n \geq 2. \end{cases}$$

Écris un programme en Python qui affiche les 10 premiers termes de la suite de Fibonacci.

7. Écrivez un programme qui génère la table de multiplication complète (de 1 à 10) sous forme de tableau en utilisant des boucles imbriquées. Formatez la sortie de manière à obtenir un tableau aligné.
8. Créez un programme qui trouve tous les nombres premiers entre 2 et 50 en utilisant l'algorithme du crible d'Ératosthène simplifié :
   — Parcourez tous les nombres de 2 à 50
   — Pour chaque nombre, vérifiez s'il est divisible par un des nombres premiers déjà trouvés
   — S'il n'est divisible par aucun, c'est un nombre premier
9. Écrivez un programme qui demande à l'utilisateur de deviner un nombre entre 1 et 100. Le programme doit indiquer si le nombre entré est trop grand ou trop petit, jusqu'à ce que l'utilisateur trouve le bon nombre. Utilisez une boucle `while` et l'instruction `break` pour terminer le jeu lorsque la réponse est correcte.

```
from random import randint

nombre_secret = randint(1, 100)
print("J'ai choisi un nombre entre 1 et 100.  A vous de le deviner !")

# Votre code ici
```

Code 38 – Modèle pour l'exercice 8

10. Créez un programme qui dessine un motif pyramidal d'étoiles comme celui-ci :

```
    *
   ***
  *****
 *******
*********
```

Utilisez des boucles imbriquées et la concaténation de chaînes pour construire chaque ligne du motif.

11. Écrivez un programme qui simule un système bancaire simple avec les fonctionnalités suivantes :
    — Le solde initial est de 1000 FCFA
    — Un menu interactif permet à l'utilisateur de choisir entre :
        (a) Consulter le solde
        (b) Effectuer un dépôt
        (c) Effectuer un retrait
        (d) Quitter le programme
    — Le programme continue de fonctionner jusqu'à ce que l'utilisateur choisisse de quitter
    — Les transactions sont enregistrées dans un historique qui peut être affiché
    — Utilisez des boucles `while`, des conditions, et les instructions `break` et `continue` de manière appropriée

```python
solde = 1000
historique = []    # On parlera des listes bientôt

print("Bienvenue dans votre système bancaire")

while True:
    print("\nMenu:")
    print("1. Consulter le solde")
    print("2. Effectuer un dépôt")
    print("3. Effectuer un retrait")
    print("4. Afficher l'historique des transactions")
    print("5. Quitter")

    choix = input("\nVotre choix (1-5) : ")

    # Votre code ici
```

Code 39 – Modèle pour l'exercice 10

Dans ce chapitre, nous avons découvert comment automatiser des tâches répétitives à l'aide des boucles `while` et `for`. Ces structures de contrôle sont fondamentales pour écrire des programmes efficaces, compacts et dynamiques. La boucle `while` est idéale lorsque l'on ne connaît pas à l'avance le nombre d'itérations, tandis que la boucle `for` est particulièrement adaptée à l'itération sur des séquences ou des plages numériques.

Nous avons également vu comment contrôler finement le déroulement d'une boucle grâce aux instructions `break` (pour interrompre prématurément une boucle) et `continue` (pour ignorer une itération).

Dans les prochains chapitres, nous les combinerons avec les fonctions pour résoudre des problèmes plus avancés.

Lien vers le quiz : https://quizy-uzl0.onrender.com/
Deadline : ~~Dimanche 11 mai 2025 à 20 : 00~~ Mercredi 14 mai 2025 à 20 : 00

Chapitre 4

Fonctions

Les fonctions sont des blocs de code réutilisables qui permettent de structurer, simplifier et clarifier un programme. Python propose un grand nombre de fonctions prédéfinies (ou « built-in »), utiles pour accomplir des tâches courantes telles que les calculs numériques, la manipulation de chaînes de caractères, ou encore le tri de collections. En parallèle, Python donne également la possibilité de regrouper du code dans des fonctions personnalisées pour le rendre plus lisible et modulaire. Ce chapitre présente d'abord les fonctions intégrées, puis les techniques pour importer et utiliser des fonctions depuis des modules de la bibliothèque standard.

1 Fonctions prédéfinies

Python dispose de nombreuses fonctions intégrées (appelées fonctions "built-in"). Nous présentons une liste non exhaustive des plus courantes fonctions.

1.1 Quelques fonctions prédéfinies

`abs(x)`

La fonction `abs()` retourne la valeur absolue d'un nombre, utile pour ignorer le signe d'un résultat.

```
1 print(abs(-15))
2 print(abs(3.14))
```

Code 40 – Exemple d'utilisation de abs()

`max(iterable)`

La fonction `max()` permet d'obtenir la plus grande valeur parmi des éléments fournis.

```
1 print(max(5, 8, 2, 10, 3))
2 print(max([1, 4, 9, 2, 5]))
```

Code 41 – Exemple d'utilisation de max()

`min(iterable)`

La fonction `min()` retourne la plus petite valeur d'un ensemble ou d'une séquence.

```
1 print(min(5, 8, 2, 10, 3))
2 print(min([1, 4, 9, 2, 5]))
```

Code 42 – Exemple d'utilisation de min()

`sum(iterable)`

La fonction `sum()` calcule la somme de tous les éléments d'un itérable numérique.

```
1 print(sum([1, 2, 3, 4, 5]))
2 print(sum((10, 20, 30)))
```

Code 43 – Exemple d'utilisation de sum()

`round(number[, ndigits])`

La fonction `round()` arrondit un nombre à un certain nombre de décimales, si précisé.

```
1 print(round(3.14159))
2 print(round(2.71828, 2))
3 print(round(9.9999, 1))
```

Code 44 – Exemple d'utilisation de round()

`sorted(iterable)`

La fonction `sorted()` retourne une nouvelle liste triée, sans modifier l'original.

```
1 print(sorted([3, 1, 4, 1, 5, 9, 2]))
2 print(sorted("python"))
3
4 mots = ["Python", "est", "un", "langage", "incroyable"]
5 print(sorted(mots, key=len))  # Trie par longueur
```

Code 45 – Exemple d'utilisation de sorted()

`int(x)`

La fonction `int()` convertit une valeur en entier, ce qui est utile pour manipuler des données numériques.

```
1 print(int(3.14))
2 print(int("42"))
3 print(int("0b1010", 2))  # Convertir un nombre binaire
```

Code 46 – Exemple d'utilisation de int()

`float(x)`

La fonction `float()` permet d'obtenir un nombre à virgule flottante à partir d'un entier ou d'une chaîne.

```
1 print(float(3))
2 print(float("3.14"))
3 print(float("1.23e-4")) # Convertir une puissance de 10
```

Code 47 – Exemple d'utilisation de float()

`str(x)`

La fonction `str()` convertit n'importe quelle valeur en chaîne de caractères.

```
print(str(42))
print(str(3.14159))
print(str([1, 2, 3]))    # Notez bien la valeur affichée
```

Code 48 – Exemple d'utilisation de str()

`bool(x)`

La fonction `bool()` sert à évaluer la "vérité" d'une expression ou d'une valeur.

```
print(bool(0))
print(bool(42))
print(bool(""))
print(bool("Python"))
print(bool([]))
```

Code 49 – Exemple d'utilisation de bool()

1.2 Importer des modules

Python est livré avec de nombreuses bibliothèques (modules) qui étendent ses fonctionnalités. Pour les utiliser, il faut les importer.

Import complet d'un module

Cette méthode permet d'importer l'intégralité d'un module, dont on accède ensuite aux fonctions via la notation pointée (`module.nom_de_fonction`).

```
import math

rayon = 5
aire = math.pi * math.pow(rayon, 2)
print(f"L'aire d'un cercle de rayon {rayon} est {aire}")
print(f"La racine carrée de 16 est {math.sqrt(16)}")
```

Code 50 – Import d'un module complet

Import avec un alias

On peut attribuer un alias à un module pour rendre son nom plus court et simplifier son utilisation, surtout lorsqu'il est long ou souvent répété.

```
import numpy as np

vecteur = np.array([1, 2, 3, 4, 5])
print(f"Tableau : {vecteur}")
print(f"Moyenne : {np.mean(vecteur)}")
```

Code 51 – Import d'un module avec un alias

Import sélectif de fonctions spécifiques

Il est aussi possible d'importer seulement certaines fonctions d'un module, ce qui rend leur appel plus direct et allège le code.

```
1 from math import sqrt, sin, cos, pi # Notez la façon d'importer
2
3 angle = pi / 4 # Notez l'utilisation de pi (sans préfixe)
4 print(f"Racine carrée de 25 : {sqrt(25)}")
5 print(f"sin({angle}) = {sin(angle)}")
6 print(f"cos({angle}) = {cos(angle)}")
```

Code 52 – Import sélectif de fonctions

Import de toutes les fonctions d'un module

Cette forme d'import permet de récupérer *toutes les fonctions* d'un module sans préfixe, mais elle doit être utilisée avec précaution.

```
1 from random import *
2
3 # Utilisation directe des fonctions du module random
4 print(f"Nombre aléatoire entre 0 et 1 : {random()}")
5 print(f"Entier aléatoire entre 1 et 10 : {randint(1, 10)}")
6 print(f"Élément aléatoire d'une liste : {choice(['pomme', 'orange'])}")
```

Code 53 – Import de toutes les fonctions d'un module

Remarque importante

L'import de toutes les fonctions avec `from module import *` n'est généralement pas recommandé dans un code professionnel car :

- Il peut créer des conflits de noms inattendus
- Il rend difficile de savoir d'où viennent les fonctions utilisées
- Il peut alourdir inutilement l'espace de noms

Préférez les imports spécifiques ou l'utilisation d'alias.

Modules courants de la bibliothèque standard

Voici quelques modules fréquemment utilisés :

- **math** : Fonctions mathématiques (sin, cos, log, exp, etc.)
- **random** : Génération de nombres aléatoires
- **datetime** : Manipulation de dates et heures
- **os** : Interactions avec le système d'exploitation
- **sys** : Accès à des variables et fonctions spécifiques au système
- **json** : Encodage et décodage de données JSON
- **re** : Expressions régulières pour le traitement de texte
- **time** : Fonctions liées au temps
- **collections** : Structures de données spécialisées
- **itertools** : Fonctions créant des itérateurs efficaces

L'import de modules est un concept fondamental en Python qui vous permet d'étendre considérablement les capacités de vos programmes en tirant parti du vaste écosystème de bibliothèques disponibles. La bibliothèque standard de Python offre déjà une multitude d'outils pour répondre à des besoins variés, et des milliers d'autres bibliothèques tierces sont disponibles via le gestionnaire de paquets `pip`.

2 Définition de fonctions

Les fonctions permettent de regrouper des instructions qui peuvent être appelées plusieurs fois.

2.1 Définition de fonctions

Exécutez ce code.

```
1 a = 10
2 b = 12
3 print(f'La somme de {a} + {b} est {a + b}')
4 c = 1.3
5 d = -1
6 print(f'La somme de {c} + {d} est {c + d}')
7 e = 1e7        # $10^7$
8 f = 2e-6       # $2 \times 10^{-6}$
9 print(f'La somme de {e} + {f} est {e + f}')
```

Code 54 – Somme de deux nombres avec des variables simples

Ce code 54 calcule la somme de deux nombres dans trois cas différents. Nous pouvons simplifier ce calcul en utilisant une fonction.

Exécutez ce code.

```
1 def affiche_somme(num_1, num_2):
2   print(f'La somme de {num_1} + {num_2} est {num_1 + num_2}')
3
4 affiche_somme(10, 12)
5 affiche_somme(1.3, -1)
6 affiche_somme(1e7, 2e-6)
```

Code 55 – Définition d'une fonction pour calculer la somme

Notez la syntaxe de la définition de fonction.

— Elle commence par le mot-clé **`def`**, suivi du **nom de la fonction**, d'une paire de parenthèses **`()`** contenant éventuellement des **paramètres**, et du symbole **`:`** (deux-points).

— Le **corps** de la fonction est écrit dans un bloc indenté. Encore une fois, l'indentation est essentielle en Python, elle permet de délimiter le bloc des instructions que la fonction exécutera.

— La fonction peut éventuellement avoir une **valeur de retour**. Si c'est le cas, cette valeur est spécifiée à l'aide du mot-clé **`return`**, qui marque également la **fin de l'exécution** de la fonction.

Par exemple, dans le code 55, la fonction `affiche_somme(num_1, num_2)` prend deux paramètres `num_1` et `num_2`, puis affiche leur somme à l'aide de la fonction `print()`.

2.2 Paramètres et arguments

La définition et l'utilisation des paramètres dans les fonctions Python offrent une grande flexibilité, permettant de créer des interfaces claires et personnalisables pour vos fonctions.

```
1 def salutation():
2     print("Bonjour, bienvenue dans le cours de Python !")
3
4 salutation()
```

Code 56 – Fonction sans paramètre

```
1 from datetime import datetime
2
3 def afficher_heure():
4     maintenant = datetime.now()
5     print("Heure actuelle :", maintenant.strftime("%H:%M:%S"))
6
7 afficher_heure()
```

Code 57 – Fonction sans paramètre

```
1 def puissance(base, exposant):
2     return base ** exposant
3
4 print(puissance(2, 3))
```

Code 58 – Paramètres positionnels

```
1 def saluer(nom, message="Bonjour"):
2     print(f"{message}, {nom}!")
3
4 saluer("Alice")
5 saluer("Bob", "Salut")
```

Code 59 – Paramètres par défaut

```
1 def decrire_personne(nom, age, ville):
2     print(f"{nom}, {age} ans, habite à {ville}")
3
4 decrire_personne(nom="Alice", age=30, ville="Paris")
5 decrire_personne(ville="Lyon", nom="Bob", age=25)
```

Code 60 – Arguments nommés

2.3 Retour de valeurs

Les fonctions Python peuvent retourner des résultats de différentes manières, ce qui permet d'optimiser la façon dont vous organisez et exploitez vos algorithmes.

```
1 def carre(x):
2     return x ** 2
3
4 resultat = carre(4)
5 print(resultat)
```

Code 61 – Retour d'une valeur

```
1 def operation(a, b):
2     somme = a + b
3     difference = a - b
4     produit = a * b
5     quotient = a / b
6     return somme, difference, produit, quotient
7
8 s, d, p, q = operation(10, 2)
9 print(s, d, p, q)  # 12 8 20 5.0
```

Code 62 – Retour de plusieurs valeurs

2.4 Portée des variables

La compréhension de la portée des variables est essentielle pour éviter les comportements inattendus et gérer efficacement l'accès aux données dans différentes parties de votre programme.

```
1  x = 10  # Variable globale
2
3  def fonction():
4      y = 5  # Variable locale
5      print('Affichage depuis la fonction...')
6      print('x=', x)  # Accès à la variable globale
7      print('y=',  y) # Accès à la variable locale
8
9  fonction()
10
11 print('Affichage hors de la fonction...')
12 print('x=', x)
13 print('y=',  y)
```

Code 63 – Portées locales et globales des variables

2.5 Fonctions anonymes (lambda)

Les fonctions lambda offrent une syntaxe concise pour créer des fonctions simples à la volée, particulièrement utiles pour des opérations brèves et dans le contexte des fonctions d'ordre supérieur.

Les lambdas sont idéales pour les opérations simples pouvant être exprimées en une seule expression.

```python
# Fonction lambda qui retourne le carré d'un nombre
carre = lambda x: x**2
print(carre(5))  # 25

# Utilisation avec des fonctions comme filter() et map()
nombres = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
nombres_pairs = list(filter(lambda x: x % 2 == 0, nombres))
print(nombres_pairs)  # [2, 4, 6, 8, 10]

carres = list(map(lambda x: x**2, nombres))
print(carres)  # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Code 64 – Fonctions lambda - syntaxe et utilisations de base

2.6 Fonctions récursives

Les fonctions récursives, qui s'appellent elles-mêmes, constituent un paradigme puissant pour résoudre des problèmes qui peuvent être décomposés en sous-problèmes similaires mais plus simples.

La récursion repose sur une condition de base qui arrête la récursion et une étape récursive qui décompose le problème.

```python
def factorielle(n):
    if n == 0 or n == 1:  # Condition de base
        return 1
    else:  # Étape récursive
        return n * factorielle(n-1)

print(factorielle(5))  # 120
```

Code 65 – Implémentation récursive de la fonction factorielle

```python
def fibonacci(n):
    if n <= 0:
        return "L'entrée doit être un entier positif"
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

for i in range(1, 11):
    print(f"fibonacci({i}) = {fibonacci(i)}")
```

Code 66 – Implémentation récursive de la suite de Fibonacci

3 Exercices

1. **Conversion de température :**
   Écrivez une fonction `celsius_vers_fahrenheit(celsius)` qui convertit une température de Celsius en Fahrenheit en utilisant la formule : $F = C \times \frac{9}{5} + 32$. Testez votre fonction avec plusieurs valeurs.
2. **Calculateur d'IMC :**
   Créez une fonction `calculer_imc(poids, taille)` qui calcule l'Indice de Masse Corporelle (poids en kg divisé par le carré de la taille en mètres). Ajoutez une seconde fonction `interpreter_imc(imc)` qui retourne une interprétation textuelle selon les valeurs suivantes :
   — $IMC < 18.5$: "Insuffisance pondérale"
   — $18.5 \leq IMC < 25$: "Poids normal"
   — $25 \leq IMC < 30$: "Surpoids"
   — $IMC \leq 30$: "Obésité"
3. **Générateur de tables de multiplication :**
   Écrivez une fonction `afficher_table(n, max=10)` qui affiche la table de multiplication de `n` jusqu'à `max`. Utilisez des paramètres par défaut et incluez une mise en forme lisible.
4. **Vérificateur de nombre premier :**
   Créez une fonction `est_premier(n)` qui retourne `True` si `n` est un nombre premier et `False` sinon. Testez votre fonction sur différentes valeurs.
5. **Calculateur de statistiques :**
   Développez une fonction `statistiques(liste)` qui prend une liste de nombres et retourne un tuple contenant la moyenne, le minimum, le maximum et l'écart-type de ces nombres. Utilisez les fonctions prédéfinies appropriées et importez le module nécessaire pour l'écart-type.
6. **Générateur de mots de passe :**
   Écrivez une fonction
   `generer_mot_de_passe(longueur, inclure_chiffres=True, inclure_symboles=False)`
   qui génère un mot de passe aléatoire avec les caractéristiques spécifiées. Utilisez le module `random` et des paramètres par défaut. Les symboles peuvent inclure `! @ # $ % ^ & * ()`.
7. **Analyseur de chaîne de caractères :**
   Créez une fonction `analyser_texte(texte)` qui retourne :
   — Le nombre total de caractères
   — Le nombre de lettres
   — Le nombre de chiffres
   — Le nombre d'espaces
   — Le nombre de voyelles
   Utilisez des boucles et des conditions pour analyser chaque caractère.
8. **Convertisseur décimal-binaire récursif :**
   Implémentez une fonction récursive `decimal_vers_binaire(n)` qui convertit un nombre décimal en sa représentation binaire (sous forme de chaîne). N'utilisez pas les fonctions de conversion intégrées comme `bin()`. Indice : la conversion récursive peut être réalisée en divisant successivement par 2 et en concaténant les restes.
9. **Calculateur d'expressions mathématiques :**
   Développez une fonction `calculer_expression(expression)` qui évalue une expression mathématique simple fournie sous forme de chaîne (par exemple, "3 + 4 * 2"). Votre fonction doit supporter l'addition, la soustraction, la multiplication et la division. Utilisez la fonction `eval()` avec précaution ou implémentez votre propre évaluateur.

10. **Jeu de devinette avec fonction récursive :**
Créez un jeu où l'ordinateur choisit un nombre aléatoire et l'utilisateur doit le deviner. Implémentez ce jeu en utilisant une fonction récursive `deviner(nombre_secret, min=1, max=100, essais=0)` qui guide l'utilisateur avec des indices "plus haut" ou "plus bas" et compte le nombre d'essais. La fonction doit se rappeler elle-même jusqu'à ce que l'utilisateur trouve la bonne réponse.

11. **Résolution numérique d'une EDO :**
On souhaite résoudre numériquement le problème de Cauchy suivant :

$$\begin{cases} y'(t) = y(t) + t - 5.23498, \quad \text{pour } t \in]0,1] \\ y(0) = 1 \end{cases} \tag{4.1}$$

On discrétise un intervalle $[a,b]$ en points $t_n = t_0 + nh$, pour $n = 0,1,...,N$, où $h = \frac{b-a}{N}$ est le pas de discrétisation et $t_0 = a$.
On souhaite approcher la solution de (4.1) à l'instant t_n par y_n, (i.e $y_n \approx y(t_n)$) en utilisant la méthode de Runge-Kutta d'ordre 3 suivante :

$$\begin{cases} K_1 = h \cdot f(t_n, y_n) \\ K_2 = h \cdot f\left(t_n + \dfrac{h}{2}, y_n + \dfrac{K_1}{2}\right) \\ K_3 = h \cdot f(t_n + h, y_n - K_1 + 2K_2) \\ y_{n+1} = y_n + \dfrac{1}{6}(K_1 + 4K_2 + K_3) \end{cases}, \tag{4.2}$$

f étant le second membre de l'équation différentielle donné par :

$$f(t,y) = y + t - 5.23498.$$

(a) Écrivez une fonction Python `f(t, y)` qui implémente le second membre de l'équation différentielle.

(b) Implémentez une fonction `rk3_step(f, t_n, y_n, h)` qui calcule les coefficients K_1, K_2, K_3, puis retourne la valeur de y_{n+1}.

(c) Complétez la fonction `resoudre_edo(f, y0, a, b, h)` pour approcher la solution de l'équation différentielle sur l'intervalle $[a,b]$ à l'aide du schéma RK3 (voir code 67). Vous devez calculer successivement les valeurs approchées y_n aux points $t_n = a + nh$, puis retourner les listes contenant les t_n et les y_n correspondants.

(d) Testez votre fonction avec les paramètres suivants :

$$y_0 = 1, \quad t_0 = 0, \quad t_f = 1.0, \quad h = 0.2$$

Affichez les couples (t_n, y_n) calculés.

(e) **(Bonus)** : Comparez les valeurs obtenues avec la solution exacte donnée par :

$$y(t) = -3.23498e^t - t + 4.23498.$$

Pour chaque point t_n, calculez l'erreur absolue :

$$\text{erreur}_n = |y_n - y(t_n)|$$

Affichez les erreurs et tracez un graphe des erreurs en fonction de t_n.
Tracez à la fois la solution numérique et la solution exacte sur un même graphique.
Note : Utilisez `matplotlib.pyplot` pour l'affichage.

```python
def resoudre_edo(f, y0, a, b, h):
    # Nombre total de pas à effectuer
    N = int((b - a) / h)
    # Initialisation des variables t_n et y_n
    t_n = a
    y_n = y0
    # Initialisation des listes pour stocker les valeurs de t et y
    t_vals = [t_n]         # Contient t_0
    y_vals = [y_n]        # Contient y_0

    # Boucle de résolution pas à pas
    for n in range(N):
        y_np1 =  ... # Calcul y_{n+1} à l'aide de RK3
        t_n += h    # Mise à jour du temps t_{n+1}
        y_n = y_np1 # Mise à jour de y_n pour l'itération suivante

        # Stockage des résultats
        t_vals.append(t_n)
        y_vals.append(y_n)
    return t_vals, y_vals
```

Code 67 – Résolution d'une EDO avec un schéma RK3

Les fonctions sont au cœur de la programmation efficace. En apprenant à utiliser les fonctions intégrées, à importer des fonctions depuis des modules, puis à organiser votre propre code en fonctions, vous développez des compétences essentielles pour écrire des programmes robustes, lisibles et évolutifs. La maîtrise des fonctions permet aussi d'éviter les répétitions et de tester facilement les différentes parties d'un programme. Dans le prochain chapitre, nous allons explorer des structures de données avancées, telles que les chaînes de caractères, les listes, les tuples et bien d'autres encore, qui permettent de manipuler l'information de manière plus flexible et efficace.

Chapitre 5

Structures de données

Les structures de données sont des moyens organisés de stocker et de manipuler des informations dans un programme. Python offre plusieurs structures de données natives qui permettent de résoudre efficacement différents types de problèmes. Ce chapitre présente les principales structures de données en Python : les listes, les tuples, les dictionnaires, les ensembles et les chaînes de caractères. Pour chacune, nous étudierons leur création, leur manipulation et leurs cas d'utilisation typiques.

1 Listes

Les listes sont l'une des structures de données les plus polyvalentes et utilisées en Python. Une liste est une collection ordonnée et modifiable d'éléments pouvant être de types différents.

1.1 Création de listes

Une liste se définit en plaçant des éléments entre crochets `[ ]`, séparés par des virgules.

```
liste_vide = []  # Liste vide
nombres = [1, 2, 3, 4, 5]  # Liste d'entiers
mixte = [42, "Python", 3.14, True]  # Liste avec différents types de données
# Liste imbriquée (liste de listes)
matrice = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

print(nombres)
print(mixte)
print(matrice[1][2])
```

Code 68 – Création de listes

1.2 Accès aux éléments

Les éléments d'une liste sont accessibles par leur **indice**, en commençant par 0 pour le premier élément.

```
langages = ["Python", "Java", "C++", "JavaScript", "Ruby"]

# Accès par indice positif
print(langages[0])  # Premier élément
print(langages[2])

# Accès par indice négatif
print(langages[-1])  # Dernier élément
print(langages[-3])

# Découpage (slicing)
print(langages[1:4])
print(langages[:3])
print(langages[2:])
print(langages[::2])    # Tous les éléments pairs (pas de 2)
```

Code 69 – Accès aux éléments d'une liste

1.3 Modification des listes

Les listes sont **modifiables** (mutables), ce qui signifie qu'on peut changer leurs éléments après création.

```
fruits = ["pomme", "banane", "orange", "kiwi"]

# Modification d'un élément
fruits[1] = "fraise"
print(fruits)

# Ajout d'éléments
fruits.append("mangue")
print(fruits)

fruits.insert(2, "ananas")     # Notez la différence avec append
print(fruits)

# Suppression d'éléments
fruits.remove("kiwi")
print(fruits)

del fruits[0]
print(fruits)

dernier = fruits.pop()            # Enlève et retourne le dernier élément
print(f"Élément retiré : {dernier}")
print(fruits)

# Autres méthodes utiles
fruits.extend(["cerise", "poire"])  # Ajoute plusieurs éléments
print(fruits)

print(f"Position de 'ananas' : {fruits.index('ananas')}")
print(f"Nombre de 'cerise' : {fruits.count('cerise')}")
```

Code 70 – Modification des éléments d'une liste

1.4 Opérations courantes sur les listes

Python offre de nombreuses opérations pour manipuler efficacement les listes.

```
nombres = [3, 1, 4, 1, 5, 9, 2, 6]

# Tri
nombres.sort()                    # Notez la différence avec sorted(nombres)
print(f"Liste triée : {nombres}")

# Inversion
nombres.reverse()
print(f"Liste inversée : {nombres}")
```

Code 71 – Tri d'une liste

```
liste1 = [1, 2, 3]
liste2 = [4, 5, 6]
concatenee = liste1 + liste2
print(f"Concaténation : {concatenee}")

repetee = [0] * 5
print(f"Répétition : {repetee}")
```

Code 72 – Concaténation et répétition de listes

```
carres = [x**2 for x in range(1, 6)]
print(f"Carrés : {carres}")

pairs = [x for x in range(10) if x % 2 == 0]
print(f"Nombres pairs : {pairs}")
```

Code 73 – Définition de liste en compréhension

1.5 Fonctions utiles avec les listes

Python propose plusieurs fonctions intégrées pour opérer sur les listes.

```
nombres = [15, 4, 8, 23, 16, 42, 7, 9]

print(f"Longueur : {len(nombres)}")
print(f"Maximum : {max(nombres)}")
print(f"Minimum : {min(nombres)}")
print(f"Somme : {sum(nombres)}")
```

Code 74 – Fonctions utiles avec les listes

```
nombres = [15, 4, 8, 23, 16, 42, 7, 9]

for i in range(len(nombres)):
    print(f"À l'indice {i}, on trouve {nombres[i]}")

i = 0
for n in nombres:
    print(f"À l'indice {i}, on trouve {nombres[i]}")
    i += 1

for i, valeur in enumerate(nombres):
    print(f"À l'indice {i}, on trouve {valeur}")
```

Code 75 – Parcours de listes

2 Tuples

Les tuples sont similaires aux listes, mais ils sont **immuables** (ne peuvent pas être modifiés après création). Ils sont souvent utilisés pour représenter des collections de données connexes qui ne doivent pas changer.

2.1 Création de tuples

Un tuple se définit en plaçant des éléments entre parenthèses ( ), séparés par des virgules. Les parenthèses sont facultatives dans de nombreux cas.

```
tuple_vide = ()    # Tuple vide

# Tuple avec un seul élément (notez la virgule obligatoire)
singleton = (42,)    # ou singleton = 42,
coordonnees = (3, 4)
personne = ("Alice", 30, "Paris")    # Tuple avec différents types
structure = ((1, 2), (3, 4), (5, 6))    # Tuple imbriqué

print(personne)
print(f"Nom : {personne[0]}, Âge : {personne[1]}")
```

Code 76 – Création de tuples

2.2 Accès aux éléments

L'accès aux éléments d'un tuple s'effectue de la même manière que pour les listes.

```
jours = ("Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "
    Dimanche")

# Accès par indice
print(jours[0])
print(jours[-1])

# Découpage
print(jours[1:4])
print(jours[:5])
print(jours[5:])

# Déballage de tuple
lundi, mardi, *reste_semaine, dimanche = jours
print(f"Premier jour : {lundi}")
print(f"Jour du milieu : {reste_semaine}")
print(f"Dernier jour : {dimanche}")

lundi, mardi, reste_semaine, dimanche = jours # Pourquoi ça ne marche pas ?
```

Code 77 – Accès aux éléments d'un tuple

2.3 Opérations sur les tuples

Bien qu'immuables, les tuples permettent certaines opérations sans modification.

```python
t1 = (1, 2, 3)
t2 = (4, 5, 6)

t3 = t1 + t2
print(f"Concaténation : {t3}")

t4 = t1 * 3
print(f"Répétition : {t4}")
```

Code 78 – Concaténation et répétition de tuples

```python
sequence = (1, 2, 3, 1, 4, 1, 5)
print(f"Nombre de 1 : {sequence.count(1)}")
print(f"Position du premier 4 : {sequence.index(4)}")
```

Code 79 – Recherche et comptage

```python
liste = list(t1)
liste.append(4)
t5 = tuple(liste)
print(f"Nouveau tuple : {t5}")
```

Code 80 – Conversion liste <-> tuple

```python
def afficher_coordonnees():
    coordonnees = ((1, 2), (3, 4), (5, 6)) # Tuple imbriqué

    for x, y in coordonnees:
        print(f"x = {x}, y = {y}")

afficher_coordonnees()
```

Code 81 – Parcours de tuples avec une boucle for

Pourquoi utiliser des tuples ?

Les tuples présentent plusieurs avantages par rapport aux listes :

- **Performance** : Les tuples sont légèrement plus rapides et utilisent moins de mémoire que les listes
- **Protection des données** : L'immuabilité garantit que les données ne seront pas modifiées accidentellement
- **Clés de dictionnaires** : Les tuples peuvent être utilisés comme clés de dictionnaires (contrairement aux listes)
- **Retours multiples** : Idéaux pour les fonctions qui retournent plusieurs valeurs

3 Dictionnaires

Les dictionnaires sont des collections non ordonnées de paires clé-valeur où chaque clé doit être unique. Ils permettent d'accéder rapidement aux valeurs à partir de leurs clés.

3.1 Création de dictionnaires

Un dictionnaire se définit avec des accolades { } contenant des paires clé-valeur séparées par des virgules.

```
# Dictionnaire vide
dict_vide = {}

# Dictionnaire simple
etudiant = {
    "nom": "Dupont",
    "prenom": "Jean",
    "age": 20,
    "notes": [12, 14, 16]
}

# Création avec la fonction dict()
contact = dict(nom="Dubois", telephone="0123456789", email="dubois@example.
    com")

# Création à partir de séquences de paires
items = [("pomme", 1.2), ("banane", 0.8), ("orange", 1.5)]
prix = dict(items)

print(etudiant)
print(prix)
```

Code 82 – Création de dictionnaires

3.2 Accès et modification

Les valeurs sont accessibles et modifiables à l'aide de leurs clés.

```
personne = {
    "nom": "Martin",
    "prenom": "Sophie",
    "age": 28,
    "ville": "Lyon"
}

# Accès aux valeurs
print(f"Nom : {personne['nom']}")
print(f"Âge : {personne['age']}")

# Accès sécurisé avec get()
print(f"Profession : {personne.get('profession', 'Non spécifiée')}")

# Notez la différence
```

Code 83 – Accès aux dictionnaires

```python
# Modification de valeurs
personne["age"] = 29
print(f"Nouvel âge : {personne['age']}")

# Ajout de nouvelles paires clé-valeur
personne["profession"] = "Ingénieure"
print(personne)

# Suppression d'éléments
del personne["ville"]
profession = personne.pop("profession")
print(f"Profession supprimée : {profession}")
print(personne)

# Vider un dictionnaire
personne.clear()
print(f"Dictionnaire vidé : {personne}")
```

Code 84 – Modification des dictionnaires

3.3 Compréhension de dictionnaire

Similaire à la compréhension de liste, la compréhension de dictionnaire permet de créer des dictionnaires de manière concise.

```python
# Création d'un dictionnaire de carrés
carres = {x: x**2 for x in range(1, 6)}
print(carres)

# Filtrage avec une condition
nombres_pairs = {x: x**2 for x in range(10) if x % 2 == 0}
print(nombres_pairs)

# Transformation de dictionnaire
prix_euros = {"pomme": 1.2, "banane": 0.8, "orange": 1.5}
prix_dollars = {fruit: prix * 1.1 for fruit, prix in prix_euros.items()}
print(prix_dollars)

# Création à partir de deux listes
fruits = ["pomme", "banane", "orange"]
quantites = [10, 20, 15]
inventaire = {fruit: qte for fruit, qte in zip(fruits, quantites)}
print(inventaire)
```

Code 85 – Compréhension de dictionnaire

3.4 Opérations courantes sur les dictionnaires

Python offre diverses méthodes pour manipuler les dictionnaires efficacement.

```python
stock = {
    "pommes": 50,
    "oranges": 25,
    "bananes": 30
}

# Obtention des clés, valeurs et paires
cles = stock.keys()
valeurs = stock.values()
items = stock.items()

print(f"Clés : {list(cles)}")
print(f"Valeurs : {list(valeurs)}")
print(f"Paires : {list(items)}")

# Fusion de dictionnaires
nouveaux_fruits = {"fraises": 20, "kiwis": 15}
stock.update(nouveaux_fruits)
print(f"Stock mis à jour : {stock}")

# Vérification d'existence
if "pommes" in stock:
    print(f"En stock : {stock['pommes']} pommes")

# Copie de dictionnaire
stock_copie = stock.copy()  # ou dict(stock)
stock_copie["pommes"] = 0
print(f"Original : {stock['pommes']} pommes")
print(f"Copie : {stock_copie['pommes']} pommes")

# Dictionnaires imbriqués
menu = {
    "entrees": {"salade": 5, "soupe": 4},
    "plats": {"poulet": 12, "poisson": 15},
    "desserts": {"glace": 6, "tarte": 5}
}

print(f"Prix de la soupe : {menu['entrees']['soupe']} FCFA")
```

Code 86 – Opérations courantes sur les dictionnaires

4 Ensembles

Les ensembles sont des collections non ordonnées d'éléments uniques. Ils implémentent les opérations mathématiques sur les ensembles comme l'union, l'intersection, etc.

4.1 Création d'ensembles

Un ensemble se définit avec des accolades `{ }` contenant des éléments séparés par des virgules, ou avec la fonction `set()`.

```python
# Ensemble vide
ensemble_vide = set() # Notez la différence avec {}

# Ensemble de nombres
nombres = {1, 2, 3, 4, 5}

# Création à partir d'une liste (élimine les doublons)
liste = [1, 2, 2, 3, 3, 3, 4, 5, 5]
ensemble = set(liste)
print(f"Liste originale : {liste}")
print(f"Ensemble : {ensemble}")

# Ensemble de chaînes
couleurs = {"rouge", "vert", "bleu"}
print(couleurs)
```

Code 87 – Création d'ensembles

4.2 Opérations sur les ensembles

Les ensembles supportent diverses opérations mathématiques d'ensembles.

```python
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# Union (éléments dans A OU B)
union = A | B  # ou A.union(B)
print(f"A union B = {union}")

# Intersection (éléments dans A ET B)
intersection = A & B  # ou A.intersection(B)
print(f"A inter B = {intersection}")

# Différence (éléments dans A mais pas dans B)
difference = A - B  # ou A.difference(B)
print(f"A - B = {difference}")

# Différence symétrique (éléments dans A ou B mais pas les deux)
diff_sym = A ^ B  # ou A.symmetric_difference(B)
print(f"Diff sum de A et B = {diff_sym}")

# Vérification d'inclusion
C = {1, 2}
print(f"C sous-ensemble de A ? {C <= A}")  # ou C.issubset(A)
print(f"A sur-ensemble de C ? {A >= C}")  # ou A.issuperset(C)

# Test de disjonction
D = {9, 10}
print(f"A et D disjoints ? {A.isdisjoint(D)}")
```

Code 88 – Opérations sur les ensembles

4.3 Modification d'ensembles

Contrairement aux tuples, les ensembles sont mutables et peuvent être modifiés.

```
couleurs = {"rouge", "vert", "bleu"}

# Ajout d'éléments
couleurs.add("jaune")
print(couleurs)

# Ajout de plusieurs éléments
couleurs.update(["orange", "violet"])
print(couleurs)

# Suppression d'éléments
couleurs.remove("vert")  # Lève une erreur si l'élément n'existe pas
print(couleurs)

couleurs.discard("marron")  # Ne fait rien si l'élément n'existe pas
print(couleurs)

# Extraction aléatoire
element = couleurs.pop()
print(f"Élément extrait : {element}")
print(couleurs)

# Ensembles "gelés" (immuables)
couleurs_fixes = frozenset(["cyan", "magenta", "jaune", "noir"])
print(couleurs_fixes)
# couleurs_fixes.add("blanc")  # Erreur: frozenset ne supporte pas l'ajout
```

Code 89 – Modification d'ensembles

Applications courantes des ensembles

Les ensembles sont particulièrement utiles pour :

- Éliminer les doublons d'une collection
- Tester rapidement l'appartenance d'un élément (plus rapide que les listes)
- Calculer des différences entre collections
- Trouver des éléments communs entre collections
- Vérifier des relations entre ensembles de données

5 Manipulation de chaînes

Les chaînes de caractères, bien que considérées comme un type de base, peuvent être vues comme des structures de données : des séquences immuables de caractères avec de nombreuses méthodes de manipulation.

5.1 Création et accès aux caractères

Une chaîne se crée avec des guillemets simples ou doubles, et on accède aux caractères par indice comme pour les listes.

```python
# Création de chaînes
message = "Bonjour Python !"
citation = 'La connaissance est pouvoir.'

# Chaînes multi-lignes
texte = """Ceci est un texte
qui s'étend sur plusieurs
lignes."""

# Accès aux caractères
print(message[0])        # Premier caractère
print(message[-1])

# Découpage
print(message[8:14])
print(message[:7])
print(message[8:])

# Les chaînes sont immuables
message[0] = "b"  # Erreur: les chaînes sont immuables
```

Code 90 – Création et accès aux chaînes

5.2 Méthodes de chaînes

Python offre un grand nombre de méthodes pour manipuler les chaînes.

```python
texte = "Python est un langage de programmation puissant"

# Recherche et comptage
print(f"'Python' est présent ? {'Python' in texte}")
print(f"Première occurrence de 'a' : {texte.find('a')}")
print(f"Nombre de 'a' : {texte.count('a')}")

# Modification de casse
print(texte.upper())             # Tout en majuscules
print(texte.lower())             # Tout en minuscules
print(texte.capitalize())        # Première lettre en majuscule
print(texte.title())             # Première lettre de chaque mot en majuscule

# Vérification de contenu
print(f"Commence par 'Python' ? {texte.startswith('Python')}")
print(f"Se termine par '!' ? {texte.endswith('!')}")
print(f"Alphanumérique ? {'Python3'.isalnum()}")
print(f"Alphabétique ? {'Python'.isalpha()}")
print(f"Numérique ? {'123'.isdigit()}")
```

Code 91 – Méthodes de chaînes

```python
texte = "Python est un langage de programmation puissant"
# Remplacement
nouveau_texte = texte.replace("Python", "Ruby")
print(nouveau_texte)

# Suppression d'espaces
texte_espaces = "   texte avec espaces   "
print(f"'{texte_espaces.strip()}'")        # Des deux côtés
print(f"'{texte_espaces.lstrip()}'")       # À gauche
print(f"'{texte_espaces.rstrip()}'")       # À droite
```

Code 92 – Méthodes de chaînes

5.3 Division et jointure

La division et la jointure sont des opérations fondamentales sur les chaînes.

```python
# Division de chaîne
phrase = "Python est un langage facile à apprendre"
mots = phrase.split()
print(mots)

csv = "Python,Java,C++,JavaScript,Ruby"
langages = csv.split(",")
print(langages)

# Division avec limite
parties = phrase.split(" ", 2)  # Divise en 3 parties max
print(parties)

# Jointure de chaînes
separateur = ", "
liste = ["pommes", "oranges", "bananes"]
texte = separateur.join(liste)
print(texte)

# Création de chaîne à partir de liste
chemin = "/".join(["dossier", "sous-dossier", "fichier.txt"])
print(chemin)
```

Code 93 – Division et jointure de chaînes

5.4 Formatage de chaînes

Python propose plusieurs méthodes pour formater des chaînes.

```python
nom = "Alice"
age = 30

# Concaténation simple
message1 = "Bonjour, " + nom + ". Vous avez " + str(age) + " ans."
print(message1)

# Formatage avec %
message2 = "Bonjour, %s. Vous avez %d ans." % (nom, age)
print(message2)
```

Code 94 – Formatage de chaînes avec + et %

```python
nom = "Alice"
age = 30

message3 = "Bonjour, {}. Vous avez {} ans.".format(nom, age)
print(message3)

message4 = "Bonjour, {0}. Vous avez {1} ans. {0} est un joli prénom.".format(
    nom, age)
print(message4)

message5 = "Bonjour, {nom}. Vous avez {age} ans.".format(nom=nom, age=age)
print(message5)
```

Code 95 – Formatage de chaînes avec la méthode format()

```python
nom = "Alice"
age = 30

message6 = f"Bonjour, {nom}. Vous avez {age} ans."
print(message6)

# Formatage de nombres
pi = 3.14159265359
print(f"Pi avec 2 décimales : {pi:.2f}")
print(f"Pi avec précision scientifique : {pi:.2e}")
print(f"Nombre avec séparateur de milliers : {1000000:,}")
print(f"Pourcentage : {0.25:.1%}")
```

Code 96 – Formatage de chaînes avec f-strings

6 Exercices

1. **Gestionnaire de contacts :**
   Créez un programme qui gère un carnet d'adresses. Utilisez un dictionnaire où les clés sont les noms des contacts et les valeurs sont des dictionnaires contenant les informations (téléphone, email, adresse). Implémentez les fonctions pour ajouter, supprimer, chercher et mettre à jour des contacts.

2. **Compteur de fréquence :**
   Écrivez une fonction qui prend une chaîne de caractères en entrée et retourne un dictionnaire contenant la fréquence de chaque caractère. Ignorez la casse (majuscules/minuscules) et les espaces.
3. **Gestion de panier d'achats :**
   Créez un système simple de panier d'achats utilisant des listes et des dictionnaires. Le panier doit permettre d'ajouter des articles avec leur prix et leur quantité, de supprimer des articles, et de calculer le total.
4. **Intersection de listes :**
   Écrivez une fonction qui prend plusieurs listes en entrée et retourne une nouvelle liste contenant uniquement les éléments présents dans toutes les listes d'entrée. N'utilisez pas directement la méthode 'intersection' des ensembles.
5. **Manipulation de phrases :**
   Créez une fonction qui prend une phrase en entrée et retourne une nouvelle chaîne où chaque mot est inversé, mais l'ordre des mots dans la phrase reste inchangé. Par exemple, "Python est génial" devient "nohtyP tse lainég".
6. **Analyseur de texte :**
   Développez un programme qui analyse un texte et fournit des statistiques : nombre total de mots, nombre de mots uniques, mots les plus fréquents, longueur moyenne des mots, etc. Utilisez des dictionnaires et des ensembles pour optimiser l'analyse.
7. **Vérificateur d'anagrammes :**
   Écrivez une fonction qui détermine si deux chaînes sont des anagrammes l'une de l'autre (contiennent exactement les mêmes lettres, potentiellement dans un ordre différent). Ignorez les espaces et la casse.
8. **Système de notation d'étudiants :**
   Créez un système qui gère les notes des étudiants pour différentes matières. Utilisez des dictionnaires imbriqués pour organiser les données par étudiant, matière et note. Implémentez des fonctions pour calculer la moyenne par étudiant et par matière.
9. **Trieur de dictionnaires :**
   Écrivez une fonction qui prend une liste de dictionnaires et une clé en entrée, puis retourne la liste triée selon les valeurs de cette clé dans chaque dictionnaire. Par exemple, pour trier une liste de dictionnaires de personnes selon leur âge.
10. **Analyseur de fichier CSV :**
    Développez un programme qui lit un fichier CSV (simulé par une liste de listes), le convertit en une liste de dictionnaires où les clés sont les en-têtes de colonnes, puis permet d'effectuer des opérations comme le filtrage, le tri et l'agrégation sur ces données.

Dans ce chapitre, nous avons découvert les principales structures de données de Python : listes, tuples, dictionnaires, ensembles et chaînes de caractères. Chacune offre des fonctionnalités spécifiques : des séquences modifiables ou immuables, des associations clé-valeur, des opérations ensemblistes ou des manipulations textuelles permettant d'organiser et de traiter les données de manière efficace.

Maîtriser ces structures est indispensable pour développer des programmes robustes. Leur combinaison permet de modéliser des situations variées et complexes : une liste de dictionnaires, un tuple de listes ou un dictionnaire de tuples ne sont que quelques exemples de cette richesse. Le bon choix de structure dépend toujours du contexte et des besoins du problème à résoudre.

Chapitre 6

Visualiser avec Matplotlib

La visualisation de données est une étape essentielle dans l'analyse et la compréhension des phénomènes. En Python, la bibliothèque `matplotlib` permet de créer des graphiques variés : courbes, histogrammes, nuages de points, etc. Ce chapitre présente quelques exemples simples de visualisations à l'aide de listes Python. L'objectif est de permettre au lecteur de produire des graphiques clairs et informatifs.

1 Tracer une courbe simple

Le module `pyplot` de `matplotlib` fournit des fonctions pour visualiser des données.

```
1 import matplotlib.pyplot as plt # Important pour pouvoir visualiser
2
3 x = [0, 1, 2, 3, 4, 5]
4 y = [0, 1, 4, 9, 16, 25]
5
6 plt.plot(x, y)
7 plt.show()
```

Code 97 – Tracé simple d'une courbe

Ce code 97 trace la courbe reliant les paires de points formées à partir des listes `x` et `y`. La fonction `plt.plot()` nécessite au moins un argument obligatoire. On peut rendre notre visualisation plus compréhensible en ajoutant plus de détails.

```
1  import matplotlib.pyplot as plt
2
3  x = [0, 1, 2, 3, 4, 5]
4  y = [0, 1, 4, 9, 16, 25]
5
6  plt.plot(x, y)
7  plt.title("Carrés des entiers")
8  plt.xlabel("x")
9  plt.ylabel("x^2") # ou "$x^2$"
10 plt.grid(True) # Essayer False aussi
11 plt.show()
```

Code 98 – Tracé de la courbe d'une fonction et ajout d'informations

Ce code 98 trace la courbe reliant les paires de points formées à partir des listes `x` et `y`, avec des axes et une grille pour améliorer la lisibilité.

2 Superposer plusieurs courbes

On peut afficher plusieurs courbes dans un même graphique en utilisant autant de quoi que c'est nécessaire `plt.plot()`.

```python
import matplotlib.pyplot as plt

x = [0, 1, 2, 3, 4, 5]
y1 = [xi**2 for xi in x] # Calcul des carrés des éléments de la liste x
y2 = [xi**3 for xi in x]

plt.plot(x, y1, label='x^2') # label pour nommer cette courbe
plt.plot(x, y2, label='x^3')
plt.title("Carrés et cubes")
plt.xlabel("x")
plt.ylabel("Valeurs")
plt.legend() # Enlever cette ligne et observer
plt.show()
```

Code 99 – Courbes superposées : carrés et cubes

3 Tracer un histogramme

Les histogrammes servent à visualiser la répartition des valeurs.

```python
import matplotlib.pyplot as plt

notes = [12, 15, 14, 10, 8, 17, 19, 13, 12, 14]

plt.hist(notes, bins=5, color='blue', edgecolor='black')
plt.title("Répartition des notes")
plt.xlabel("Note")
plt.ylabel("Effectif")
plt.savefig("histogramme.pdf", dpi=300) # Enregistrer au format pdf
plt.savefig("histogramme.png", dpi=300) # Enregistrer au format png
plt.show()
```

Code 100 – Histogramme de la répartition des notes

Ici, `bins=5` divise l'intervalle des notes en cinq classes.
Changer la couleur de l'histogramme.

4 Tracer un nuage de points

Un nuage de points permet de visualiser des données dispersées.

```python
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2.1, 4.2, 6.1, 8.0, 10.1]

plt.scatter(x, y, marker='o') # Tester x ou * ou + comme marker
plt.title("Nuage de points")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

Code 101 – Nuage de points

Ce code 101 affiche les paires de points formées à partir des listes `x` et `y`.

5 Exercices

1. Représenter graphiquement la fonction $f(x) = 3x + 2$ pour $x \in [0, 10]$. Ajouter un titre, les noms des axes et une grille.
2. Tracer sur le même graphique les courbes de $y = x$, $y = x^2$ et $y = x^3$ pour x allant de -5 à 5. Utiliser une légende pour les distinguer.
3. Un élève a reçu les notes suivantes durant l'année : 12, 14, 15, 13, 16, 18, 14, 15, 17, 16. Représenter la répartition de ces notes avec un histogramme en 5 classes.
4. Tracer un nuage de points représentant les distances (en km) et durées (en minutes) de 6 trajets : `distances = [2, 5, 7, 3, 10, 4]` et `durees = [5, 12, 15, 7, 22, 9]`.
5. Simuler la croissance d'une plante sur 7 jours : `jours = [1, 2, 3, 4, 5, 6, 7]`, `hauteurs = [3, 4.5, 6, 7, 9, 10, 12]`. Tracer la courbe de croissance et ajouter des éléments de style (couleur, marqueur).
6. Tracer la courbe représentant $f(x) = |x|$ pour $x \in [-5, 5]$. Utiliser une boucle pour construire la liste y.
7. Créer un histogramme représentant le nombre de livres lus par mois par un élève sur 12 mois : `[2, 1, 3, 4, 2, 0, 5, 3, 2, 1, 4, 3]`.
8. Tracer une fonction définie par morceaux :

$$f(x) = \begin{cases} x^2 & \text{si } x < 0 \\ x + 1 & \text{si } x \geq 0 \end{cases}$$

   pour $x \in [-5, 5]$.
9. Observer ce code et corriger l'erreur s'il y en a :

```
x = [0, 1, 2, 3, 4]
y = [1, 2, 4, 8]

plt.plot(x, y)
plt.title("Exemple")
plt.show()
```

Code 102 – Code à corriger

Ce chapitre a montré comment créer des graphiques simples à l'aide de `matplotlib`, en se basant uniquement sur des listes Python. Que ce soit pour représenter une fonction, comparer des courbes ou explorer la distribution de données, ces outils permettent une meilleure compréhension visuelle des informations.

Chapitre 7

Du calcul scientifique avec NumPy

NumPy (Numerical Python) est l'une des bibliothèques les plus fondamentales pour le calcul scientifique en Python. Elle fournit un support pour les tableaux multidimensionnels et les matrices, ainsi qu'une vaste collection de fonctions mathématiques de haut niveau pour opérer sur ces structures de données. Dans ce chapitre, nous allons explorer les bases de NumPy.

1 Création des tableaux

Le type de données fondamental dans NumPy est le `ndarray` (N-dimensional array), qui représente un tableau multidimensionnel d'éléments de même type.

Exécutez ce code.

```
1  import numpy as np
2  tab1 = [0, 4, -1]
3  u = np.array(tab1)
4  print('Affiche u \n', u)
5
6  tab2 = [[0, 4, -1]]
7  v = np.array(tab2)
8  print('Affiche v \n', v)     # Notez la différence entre u et v
9
10 tab3 = [[10, -1], [2, 3], [0, 1]]
11 w = np.array(tab3)
12 print('Affiche w \n', w)
```

Code 103 – Création de tableaux

Ce code 103 illustre la création de tableaux NumPy à partir de listes Python. Le tableau `u` est un vecteur à une dimension, tandis que `v` est une matrice à une ligne et `w` est une matrice à trois lignes deux colonnes.

Créer et afficher les matrices suivantes :

$$A = \begin{pmatrix} 10 & 0 & 0 & 1 \\ -1 & -1 & 2 & 3 \\ 0 & 9 & 0 & 0 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & 2 & 0 \\ 0 & 2 & 2 \\ 2 & 0 & 2 \end{pmatrix} \qquad C = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} \qquad v_1 = \begin{pmatrix} 1 & -1 & 1 & 2 \end{pmatrix} \qquad v_2 = \begin{pmatrix} 1 \\ -1 \\ 1 \\ 2 \end{pmatrix}$$

On peut aussi créer des tableaux à partir des tableaux prédéfinis par NumPy.

Que fait ce code ?

```
1  import numpy as np
2
3  A = np.zeros((3, 4))
4  B = np.ones((2, 3))
5  C = np.arange(0, 10, 2)
6  D = np.linspace(0, 1, 5)
7  E = np.eye(5)
8  F = np.random.random((2, 2)) # Exécutez plusieurs fois et affichez à chaque
       fois
```

Code 104 – Création de tableaux

2 Opérations sur les tableaux

On peut aussi créer un tableau NumPy en faisant des opérations arithmétiques ou en évaluant l'image d'un tableau par une fonction mathématique.

```
1 import numpy as np
2 a = np.array([1, 2, 3, 4])
3 b = np.array([5, 6, 7, 8])
4
5 op1 = a + b
6 op2 = 2 * a - 3 * b
7 op3 = a * b
8 op4 = (a / b) ** a
```

Code 105 – Opération arithmétique sur les tableaux

Notez que le code 105 effectue des opérations élément par élément (ou vectorisées) entre les tableaux NumPy. Chaque opération est appliquée indépendamment à chaque paire d'éléments situés à la même position dans les tableaux `a` et `b`.

Que fait ce code?

```
1  import numpy as np
2
3  A = np.array([[1, 2], [3, 4]])
4  B = np.array([[0, 6, 3], [2, 0, 1]])
5  u = np.array([1, -1])
6  v = np.array([2, 1])
7
8  w = u.dot(v)
9  C = A.dot(u)
10 D = A @ B                  # ou np.matmul(A, B)
11 E = B @ A                  # Qu'est-ce que va pas ici ?
12 At = A.T                   # ou A.tranpose()
13
14 A_inv = np.linalg.inv(A)   # Que fait ceci? Comparez A_inv @ A et A @ A_inv
15
16 det_A = np.linalg.det(A)   # Déterminant
17
18 b = np.array([1, 2])
19 x = np.linalg.solve(A, b)  # Résolution du système Ax = b
```

Code 106 – Opérations d'algèbre linéaire

3 Indexation et découpage

L'accès aux éléments d'un tableau NumPy est similaire à celui des listes Python, mais avec des fonctionnalités avancées :

```python
import numpy as np

A = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

print(A[0, 1])    # Accès à l'élément situé à la ligne 1 et colonne 2

print(A[0:2, 1:3])  # Découpage (slicing)

print(A[[0, 2], [1, 3]])  # Qu'affiche ceci ?
```

Code 107 – Indexation et découpage

4 Fonctions utiles

NumPy offre une panoplie de fonctions permettant de résumer ou transformer rapidement des tableaux.

4.1 Statistiques de base

Que fait ce code ?

```python
import numpy as np

A = np.array([[1, 2, 3], [4, 5, 6]])

print(np.mean(A))
print(np.median(A))
print(np.std(A))
print(np.var(A))
print(np.min(A))
print(np.max(A))
print(np.sum(A))
```

Code 108 – Statistiques de base

Chaque fonction renvoie une statistique appliquée à l'ensemble du tableau. Quelles sont les plus sensibles aux valeurs extrêmes ?

Que fait ce code ?

```python
import numpy as np

A = np.array([[1, 2, 3], [4, 5, 6]])

print(np.mean(A, axis=0))
print(np.sum(A, axis=1))
print(np.max(A, axis=0))
print(np.min(A, axis=1))
```

Code 109 – Agrégation selon les axes

Le paramètre `axis` contrôle la direction de l'agrégation : `0` pour les colonnes, `1` pour les lignes. Pouvez-vous visualiser cela sur un papier ?

4.2 Fonctions mathématiques (ufuncs)

Voici quelques opérations élément par élément. Essayez de déduire la transformation appliquée à chaque valeur.

```
1 import numpy as np
2
3 A = np.array([[1, 0, 1], [3, 1, -4]])
4
5 print(np.sqrt(A))  # Que vaut élément situé à [1, -1] ? Pourquoi ?
6 print(np.exp(A))
7 print(np.log(A))   # Que vaut élément situé à [0, 1] ? Pourquoi ?
8 print(np.sin(A))
9 print(np.abs(A - 3))
```

Code 110 – Fonctions mathématiques (ufuncs)

Toutes ces fonctions s'appliquent élément par élément, sans qu'il soit nécessaire d'écrire une boucle explicite. Pour ce type d'opérations, il est fortement recommandé d'utiliser les tableaux NumPy, car ils sont à la fois plus concis et beaucoup plus rapides que les boucles classiques en Python.

4.3 Tri et valeurs uniques

Ces fonctions permettent de réorganiser ou filtrer les valeurs. Pouvez-vous deviner l'effet de chacune avant d'exécuter?

```
1 import numpy as np
2
3 A = np.array([[3, 1, 2, 1, 5, 3], [10, 10, -1, 3, 7, 0]])
4
5 print(np.sort(A))
6 print(np.argsort(A)) # Notez la différence avec np.sort(A)
7 print(np.argmax(A))  # Notez la différence avec np.argsort(A)
8 print(np.unique(A))
```

Code 111 – Tri et valeurs uniques

5 Exercices

1. Créez un tableau NumPy contenant les entiers pairs de 0 à 20. Calculez la somme, la moyenne, l'écart-type et la variance de ce tableau.
2. Écrivez une fonction `carre_tableau(tab)` qui prend un tableau NumPy en argument et retourne un nouveau tableau contenant le carré de chaque élément. Testez-la sur un tableau de votre choix.
3. Créez une matrice 4×4 remplie aléatoirement, puis affichez :
   — la somme de chaque ligne,
   — la moyenne de chaque colonne.
4. Implémentez une fonction `produit_scalaire(u, v)` qui calcule le produit scalaire de deux vecteurs NumPy. Vérifiez son fonctionnement avec les vecteurs `u = [1, 2, 3]` et `v = [-1, 0, 1]`.
5. Créez une matrice identité de taille 5×5 et ajoutez-y une matrice aléatoire de même taille. Calculez la matrice inverse du résultat (si elle existe).

6. Créez une matrice 3×3 contenant des entiers au hasard entre 0 et 9. Triez les valeurs de chaque ligne dans l'ordre croissant.
7. Créez une fonction `filtre_positifs(A)` qui retourne un tableau contenant uniquement les éléments strictement positifs d'un tableau NumPy `A` (utilisez les masques booléens).
8. Générer un vecteur `x` de 100 valeurs uniformément espacées entre -2π et 2π. Calculez et tracez les valeurs de $\sin(x)$ et $\cos(x)$. (Optionnel : utiliser `matplotlib`)
9. Créez une fonction `normalize(X)` qui prend un tableau NumPy 2D et retourne le tableau dont chaque colonne a une moyenne nulle et un écart-type égal à 1.
10. Écrivez une fonction `solve_system(A, b)` qui résout un système linéaire $Ax = b$ pour une matrice A donnée et un vecteur b. Générez un exemple test et affichez la solution.

Dans ce chapitre, nous avons découvert les principales fonctionnalités de la bibliothèque `NumPy`, outil incontournable pour toute personne s'intéressant au calcul scientifique avec Python. Nous avons appris à :

— Créer et manipuler des tableaux multidimensionnels (`ndarray`),
— Appliquer des opérations arithmétiques et vectorisées sur les tableaux,
— Utiliser des fonctions mathématiques et statistiques intégrées,
— Effectuer des opérations d'algèbre linéaire,
— Accéder et filtrer efficacement les données dans un tableau.

Ces compétences constituent la base de tout travail en science des données, en apprentissage automatique ou en modélisation numérique. La compréhension et la maîtrise de NumPy vous permettront de tirer parti des performances de calcul offertes par Python, tout en écrivant un code concis, lisible et efficace.

Chapitre 8

Projets

Consignes Générales

— Chaque projet doit être réalisé en groupe de 5 à 6 étudiants.
— Le rendu comprendra un code Python fonctionnel et un rapport de 2 à 3 pages au format PDF.
— Le rapport doit expliquer le problème résolu, l'approche algorithmique, les choix de conception, et les résultats obtenus.
— Le code doit être clair, commenté et bien structuré.
— L'évaluation prendra en compte la qualité du code, la pertinence du rapport, et la rigueur de l'analyse.

Deadline : Mercredi 21 mai 2025

Lien de soumission : https://forms.gle/hq1PGsod3rNTkRza8

1 Sujets de Projets

1.1 Système de gestions de notes

Dans ce projet, il s'agit de concevoir un programme Python capable de gérer les notes d'une classe d'étudiants. Une classe est constituée d'un ensemble d'élèves inscrits dans quatre matières obligatoires : Mathématiques, Physique, Chimie et Informatique. Chaque matière comporte deux évaluations ponctuelles (notées E1 et E2) et deux devoirs (notés D1 et D2). Les notes sont attribuées sur 20.

Le système devra permettre d'enregistrer les élèves, de saisir leurs notes pour chaque matière, et de calculer différentes moyennes. Il sera notamment nécessaire de calculer la moyenne par matière pour chaque élève, la moyenne générale d'un élève sur les quatre matières, ainsi que les moyennes par matière à l'échelle de la classe. Le programme devra également afficher le bulletin d'un élève donné et établir un classement des étudiants en fonction de leur moyenne générale (voir le menu ci-dessous).

— MENU —
1. Ajouter un étudiant
2. Saisir les notes d'un étudiant
3. Afficher le bulletin d'un étudiant
4. Afficher le classement général
5. Moyennes par matière
6. Quitter

L'implémentation se fera à l'aide de structures de données de base comme les listes, les dictionnaires,... On ne fera pas usage de la Programmation Orientée Objet (POO), mais une réflexion sur l'organisation des données et du code est encouragée. ~~En perspective, une version améliorée pourra tirer profit de la POO pour rendre la gestion plus structurée et évolutive.~~

1.2 Interpolation polynomiale

Ce projet porte sur l'approximation de fonctions par interpolation, à partir d'un ensemble de points. L'objectif est d'implémenter la méthode d'interpolation de Lagrange pour approcher une fonction réelle f d'une variable réelle définie sur un intervalle $[a, b]$ dont on ignore l'expression

analytique et dont on connaît la valeur en un nombre fini de points distincts x_i tels que $a \leq x_0 < x_1 < \cdots < x_n \leq b$.

Interpolation de Lagrange

L'interpolation de Lagrange de f consiste à construire un polynôme L de degré au plus n qui coïncide avec f aux points x_i, avec $i \in [\![0, n]\!]$. Ce polynôme est donné par :

$$L(x) = \sum_{i=0}^{n} f(x_i)\ell_i(x), \quad \text{où } \ell_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

Le programme devra permettre de générer un ensemble de points $(x_i)_{0\leq i\leq n}$ de l'intervalle $[a, b]$, et de calculer le polynôme d'interpolation de Lagrange d'une fonction donnée à partir de ces points. Il faudra ensuite tester votre implémentation en considérant trois différentes fonctions, et les représenter graphiquement ainsi les polynômes d'interpolations (voir figure 8.1 par exemple). On mesurera l'erreur d'interpolation avec la norme euclidienne discrète sur l'intervalle $[a, b]$. On finira, en étudiant numériquement l'erreur en fonction du nombre de points utilisés ($n = 4, 8, 16, 32$).

Discrétisation de l'intervalle $[a, b]$

On considérera les deux différents ensembles de points suivants.

1. L'ensemble des points $(x_i)_{0\leq i\leq n}$ est une discrétisation uniforme de l'intervalle $[a, b]$, i.e :

$$x_i = a + \left(\frac{b-a}{n}\right) \times i, \ \forall i \in [\![0, n]\!].$$

2. L'ensemble des points $(x_i)_{0\leq i\leq n}$ est une discrétisation non uniforme dans l'intervalle $[a, b]$ donnée par :

$$x_i = \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{\pi \times i}{n}\right), \ \forall i \in [\![0, n]\!].$$

Erreur d'interpolation

Pour mesurer l'écart entre la fonction f et son polynôme d'interpolation L, on utilise la *norme euclidienne discrète* sur un ensemble de points $(x^{(j)})_{0\leq j\leq 100}$ uniformément répartis sur $[a, b]$. L'erreur d'interpolation est alors donnée par :

$$E_n = \left(\sum_{j=0}^{100} \left(f(x^{(j)}) - L(x^{(j)})\right)^2\right)^{1/2}.$$

Plus E_n est petit, plus le polynôme L est une bonne approximation de la fonction f. Cette erreur sera calculée pour différentes valeurs de n, et tracer l'évolution de l'erreur en fonction des différentes valeurs de n.

Applications :

On utilisera les fonctions suivantes pour évaluer et comparer l'efficacité des méthodes d'interpolation de Lagrange.

— $f_1(x) = \frac{1}{1+25x^2}$ sur $[-1, 1]$;
— $f_2(x) = \sin(2\pi x)$ sur $[0, 1]$;
— $f_3(x) = \ln(x + 2)$ sur $[-1, 1]$.

L'implémentation se fera en Python avec les outils de bases de Python et la librairie Matplotlib.

1.3 Matrices creuses

Ce projet vise à implémenter une structure de données efficace pour manipuler des matrices creuses en Python. Une matrice creuse est une matrice dont la majorité des éléments sont nuls.

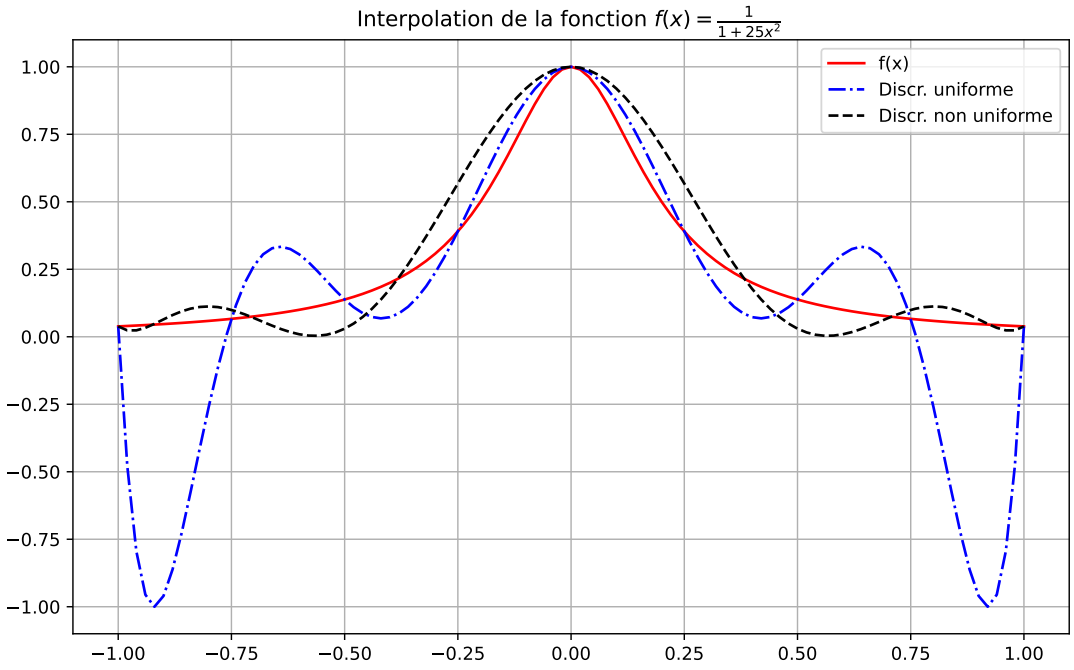

FIGURE 8.1 – Comparaison entre la fonction exacte et son interpolation de Lagrange en utilisant 8 points. En bleu, l'interpolation de Lagrange avec les points discrétisation uniforme. En noir, l'interpolation de Lagrange avec les points discrétisation non uniforme dit de Tchebychev.

Au lieu de stocker tous les éléments (y compris les zéros), nous allons uniquement stocker les éléments non nuls avec leurs positions (les numéros de la ligne et de la colonne).

Une matrice sera représentée à l'aide d'une liste de coordonnées où chaque élément non-nul est stocké sous la forme d'un triplet `(ligne, colonne, valeur)`.

Le projet permettra d'effectuer des opérations de base sur les matrices creuses telles que l'addition, la multiplication matrice-vecteur, et la transposition. Nous implémenterons également des fonctions pour convertir entre le format creux et le format dense, ainsi que pour afficher la matrice.

Toutes les fonctionnalités seront implémentées en utilisant uniquement les structures de données de base de Python (listes, dictionnaires, tuples) sans recourir à des bibliothèques externes ni à la POO.

Représentation de matrices denses

Dans ce projet, les matrices denses seront représentées à l'aide de *listes imbriquées* en Python. Plus précisément, une matrice sera modélisée par une liste de lignes, chaque ligne étant elle-même une liste contenant les éléments de cette ligne.

Par exemple, la matrice

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

sera représentée par la structure suivante :

```
1 [[1, 2, 3],
2  [4, 5, 6]]
```

Cette représentation est permet un accès direct aux éléments par double indexation : `matrice[i][j]` donne l'élément situé à la i-ième ligne et à la j-ième colonne. Elle sera utilisée notamment pour convertir des matrices creuses en format dense et pour valider les résultats des opérations implémentées.

Représentation en liste de coordonnées

Pour manipuler efficacement une matrice creuse, on ne conserve que les éléments non nuls, accompagnés de leur position. On stocke donc :

— le *shape* de la matrice (`(nombre_de_lignes, nombre_de_colonnes)`),

— une liste de triplets `(ligne, colonne, valeur)` pour chaque coefficient non nul.

```
# Représentation en liste de coordonnées de la matrice suivante :
# [[0, 0, 5, 0, 0],
#  [0, 0, 0, 0, 0],
#  [0, 0, 0, 0, 0],
#  [0, -1, 0, 0, 0]]

matrice_creuse = {
    "shape": (4, 5),
    "data": [(0, 2, 5), (3, 1, -1)]
}
```

Affichage de matrices

On donne ici un exemple de la représentation en liste de coordonnées.

```
def afficher_matrice(matrice):
    if not matrice:
        print("Matrice vide")
        return

    col_max = max(len(str(x)) for ligne in matrice for x in ligne)

    for ligne in matrice:
        print("|", end=" ")
        for x in ligne:
            print(str(x).center(col_max), end=" ")
        print("|")
```

Code 112 – Fonction pour afficher les matrices sous forme denses

Applications :

On utilisera la représentation sous forme de liste de coordonnées des matrices A, B et C, pour effectuer les opérations $A+B$, A^T, C^T, Av_1, Bv_1, Cv_2.

$$A=\begin{bmatrix}2&-1&0&0\\-1&2&-1&0\\0&-1&2&-1\\0&0&-1&2\end{bmatrix},\quad B=\begin{bmatrix}1&0&0&0\\0&0&3&0\\0&0&0&0\\0&2&0&0\end{bmatrix},\quad C=\begin{bmatrix}0&0&4&0\\5&0&0&0\\0&0&0&-2\\0&0&0&0\end{bmatrix},\quad v_1=\begin{bmatrix}1\\2\\0\\-1\end{bmatrix},\quad v_2=\begin{bmatrix}0\\3\\-1\\2\end{bmatrix}$$

Comparaison pour la matrice tridiagonale A_n

Soit pour $n\geq 2$ la matrice $A_n\in\mathbb{R}^{n\times n}$ définie par :

$$(A_n)_{i,j}=\begin{cases}2, & \text{si } i=j,\\ -1, & \text{si } |i-j|=1,\\ 0, & \text{sinon.}\end{cases}$$

Autrement dit,

$$A_n = \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}.$$

On souhaite mener une étude empirique pour déterminer à partir de quelle taille n la version creuse surpasse la version dense :

- Générer la matrice tridiagonale A_n pour $n \in \{4, 8, 16, 32, 64, 128\}$.
- Mesurer, à l'aide de `time.perf_counter()` ou du module `timeit`, le temps nécessaire pour effectuer un produit matrice–vecteur dans chacun des deux formats.
- Tracer ensuite à l'aide de Matplotlib :
  - le temps d'exécution de la version dense en fonction de n^2 sur l'axe horizontal ;
  - le temps d'exécution de la version creuse en fonction de n sur le même graphique.

Conclusion

Au terme de ce document, vous avez exploré les fondements de la programmation avec Python — un langage à la fois puissant, lisible et polyvalent. Des opérations arithmétiques aux structures de données, en passant par les fonctions, les boucles, la visualisation avec `matplotlib`, et l'introduction au calcul scientifique avec `NumPy`, ce cours vous a permis de poser des bases solides pour continuer votre parcours en science des données, intelligence artificielle, ou développement d'applications.

Python ne se limite pas à un outil pour débutants ; il est aujourd'hui au cœur des plus grands projets technologiques et scientifiques. Sa communauté active et son écosystème riche (avec des bibliothèques comme `Pandas`, `Scikit-learn`, `TensorFlow`, etc.) permettent de construire des solutions puissantes et modernes dans presque tous les domaines.

L'apprentissage de la programmation est un voyage progressif. Chaque concept introduit ici est une porte ouverte vers des sujets plus avancés. Je vous encourage à pratiquer, à expérimenter, et à résoudre des problèmes concrets. Car c'est en codant régulièrement que vous consoliderez vos connaissances.

Références recommandées

- Gérard Swinnen. *Apprendre à programmer avec Python 3*, Eyrolles, 4^{e} édition. Un classique pour progresser de manière pédagogique, avec de nombreux exercices.
- Moussa Kéita. *Data Science with Python : Algorithm, Statistics, DataViz, DataMining and Machine-Learning*. Un excellent ouvrage pour faire la transition entre programmation Python et applications en science des données.
- Mark Lutz. *Learning Python*, O'Reilly. Une référence plus complète et approfondie.
- Documentation officielle Python : `https://docs.python.org/fr/3/`
- Documentation NumPy : `https://numpy.org/doc/`

Mot de la fin : La programmation est un art autant qu'une science. Soyez curieux, persévérez, et amusez-vous à créer !