



# FORMATIO N

— Python — Par la pratique



m2iinformation.fr



# Bienvenue sur cette formation

*Bienvenue à cette formation Python chez M2i ! Je m'appelle Jules Galian, et je serai votre formateur tout au long de ce parcours. Avec un solide bagage en développement Python, j'ai eu l'occasion de travailler sur divers projets, notamment en développement web, mobile, et dans d'autres domaines comme l'automatisation et l'analyse de données.*

*Mon objectif est de vous fournir des compétences pratiques et applicables immédiatement, en vous accompagnant avec des exemples concrets issus de mon expérience. Je suis ici pour vous guider vers une maîtrise de Python, et ensemble, nous allons explorer tout le potentiel de ce langage. Nous ferons de cette formation une expérience enrichissante, en développant une application concrète : un petit CRM pour rendre les concepts plus tangibles et motivants tout en intégrant les éléments essentiels pour la certification TOSA.*



# Tour de table

Prérequis:

Connaître un langage de programmation.

Sur quels langages avez-vous l'habitude de travailler?

Avez-vous déjà fait de la POO (Programmation orienté objet)?

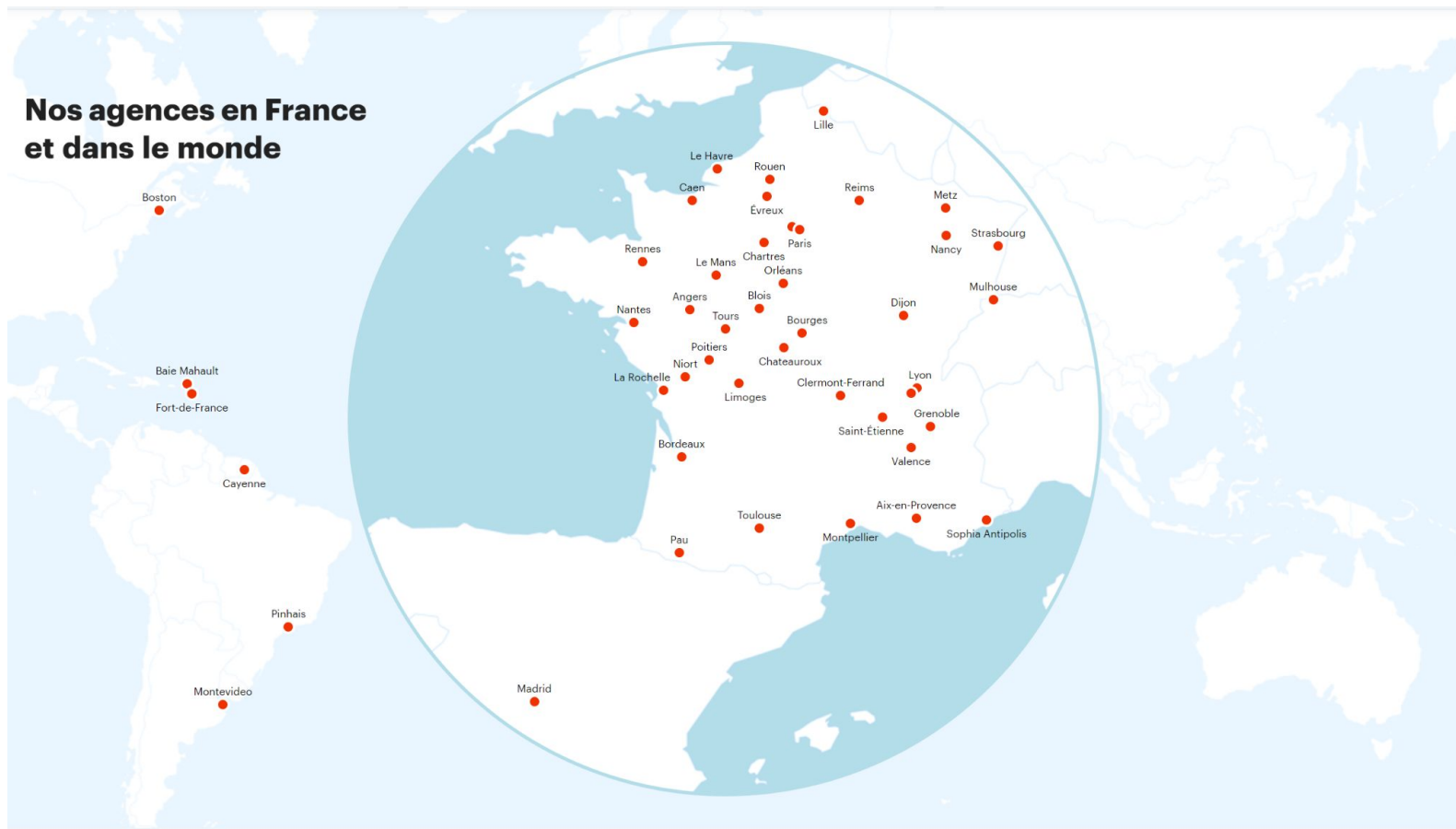
Avez-vous déjà programmé en Python?

Quelles sont vos attentes à l'issu de cette formation?

Passez-vous la certification Python TOSA en fin de stage ? "Indication TOSA sur slides"

# Le réseau M2I Formation

## Nos agences en France et dans le monde



# Le groupe M2I

- Le groupe M2i est leader de la formation IT, Digital et Management en France depuis plus de 35 ans.
- L'engagement pour la qualité en étant certifié Qualiopi et Datadock.
- Plus de 300 collaborateurs dédiés à la montée en compétences de votre capital humain.
- Le catalogue M2I : <https://www.m2information.fr/catalogues/>
- Engagement vers la féminisation du numérique : <https://www.m2information.fr/numerique-au-feminin/>
- La démarche qualité : <https://www.m2information.fr/demarche-qualite/>

# Horaire et convocations

- 9h - 17h (alarm)
- 15 mn de pause le matin (alarm)
- 1h de pause déjeuner (alarm)
- 15 mn de pause l'après midi (alarm)
- Dernier jour à 16h30 (si le plan de cours est terminé uniquement)

# Déroulé et structure de la formation et formalités

- Première chose à faire : signer les feuilles d'émargement et mentionner le caractère obligatoire avec de vrais signatures (ni croix ni initiales) (**M2I sign**)
- Dernier jour de la formation : le centre de formation à l'obligation contractuelle de fournir vos évaluations à votre entreprise avant 15h, donc au retour de la pause déjeuner, 2 ou 3 h avant la fin de la formation, je vous ferai remplir les évaluations formateur.
- La structure d'une journée : présenter une notion théorique, suivie de la pratique (écriture du code), suivi d'un exercice . une fois que j'ai abordé avec vous 3 ou 4 notions, 1 tp de validation des acquis qui porte sur ces notions. Ce TP sera à faire en pair programming (en groupe) ou seul si vous le souhaitez. Lors de ceux-ci vous développerez un petit CRM.
- Si il me reste du temps, je reviens sur toutes les questions hors plan de cours que les stagiaires m'ont posées durant la formation. Obligation contractuelle de respecter le plan de cours.

# Structure et versionning Github

- <https://github.com/DonJul34/pyt-dec/>

Un commit par TP de validation des acquis. (dossier CRM)

Un commit par exercice (dossier Exercises)

Fichier du support de cours en source du github.



# Validation des acquis au quotidien

- Google Forms de demi-journée pour la validation des acquis et l'adaptabilité.
- [https://docs.google.com/forms/d/e/1FAIpQLSdhHK\\_sP6lnWyLZRyYqTAvNvMOtKBWeaLCy-1wjDg\\_i0ZbByA/viewform?usp=sf\\_link](https://docs.google.com/forms/d/e/1FAIpQLSdhHK_sP6lnWyLZRyYqTAvNvMOtKBWeaLCy-1wjDg_i0ZbByA/viewform?usp=sf_link)



# L'application et les TP de la formation



Gestionnaire CRM



Nom

Email

Téléphone

Catégorie (SEO/Web/Autre)

Nombre d'employés

Ajouter Client

Nom	Email	Téléphone	Catégorie	Nombre d'employés	Coût HT (€)	Coût TTC (€)
Galian	jules.galian@gmail.com	0672948722	Seo	569	5.00 €	2865.00 €
Marian	mariandayanna@gmail.com	0672948722	Web	780	7.00 €	5480.00 €
Galian	jules.galian@gmail.com	0672948722	Seo	670	5.00 €	3370.00 €
Marian	mariandayanna@mail.com	06728433239	Seo	8000	5.00 €	40020.00 €
test	test@gmail.com	872323211	Web	9000000000	7.00 €	63000000020.00 €
Galian	jules.galian@gmail.com	0672849822	Web	877	7.00 €	6159.00 €
erzaezeaz	raezaz@mail.com	0672948722	Seo	86	5.00 €	450.00 €
Inconnu			Autre	1	5.00 €	25.00 €
Inconnu			Autre	1	5.00 €	25.00 €
Inconnu			Autre	1	5.00 €	25.00 €

Total Clients : 12

**Chiffre d'Affaires : 63000059484.00 €**

Actualiser les Statistiques

Actualiser les Clients

# Table des matières

## Chapitre 1 : Introduction à Python

- Historique de Python
- Utilisation et Environnement de travail
- Installation de Python et choix de l'IDE
- Normes de codage : PEP8

## Chapitre 2 : Scripts et Programmes

- Gestion des fichiers Python (.py, .pyc)
- Structure minimale d'un programme
- Définition du point d'entrée avec `__main__`
- Encodage des fichiers : UTF-8

## Chapitre 3 : Types de Données

- Types primitifs : Entiers, Flottants, Booléens, Chaînes
- Conversion et casting des types
- Collections : Listes, Tuples, Sets, Dictionnaires

## Chapitre 4 : Opérations de Base

- Opérateurs d'affectation, arithmétiques, logiques, relationnels
- Gestion des entrées/sorties : `input()`, `print()`, formatage avec f-strings
- Commentaires et documentation

## Chapitre 5 : Contrôle de Flux

- Structures conditionnelles : `if`, `else`, `elif`, `match`
- Boucles : `for`, `while`, `range()`
- Gestion des exceptions : `try`, `except`

## Chapitre 6 : Fonctions

- Définition des fonctions avec `def`, paramètres, retour de valeurs
- Fonctions anonymes : lambdas
- Utilisation des variables globales avec `global`
- Paramètres par défaut, `*args`, `**kwargs`

## Chapitre 7 : Gestion des Fichiers

- Lecture et écriture des fichiers avec `open()`
- Modes d'ouverture : lecture, écriture, ajout
- Utilisation des modules `os`, `shutil`, `zlib`

## Chapitre 8 : Modules et Packages

- Création et organisation des modules
- Installation et gestion des packages avec `pip`

## Chapitre 9 : Programmation Orientée Objet (POO)

- Création de classes et instanciation d'objets
- Constructeurs `__init__`
- Héritage simple et multiple
- Méthodes et attributs de classe

## Chapitre 10 : Développement Avancé

- Connexion à une base de données SQL
- Programmation d'interfaces graphiques avec Tkinter
- Techniques de débogage et gestion des erreurs



# Bilan de début de formation

- Retour de début de formation à M2I
- [contact@2aiconcept.com](mailto:contact@2aiconcept.com) en copie
- Au plus tard à 10h30 le premier jour de la formation.

# Chapitre 1 : Introduction à Python

---

# Introduction à Python

## Historique de python

### Création :

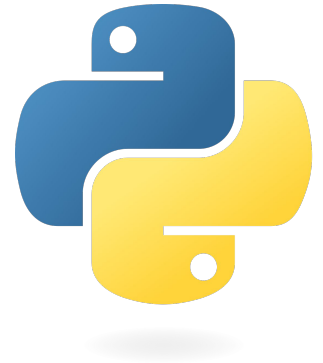
- **Année** : 1989
- **Créateur** : Guido van Rossum (Créateur du langage ABC)

### Origine du nom :

- Inspiré par la troupe comique britannique "**Monty Python**"

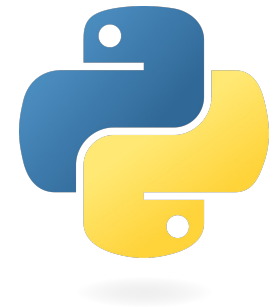
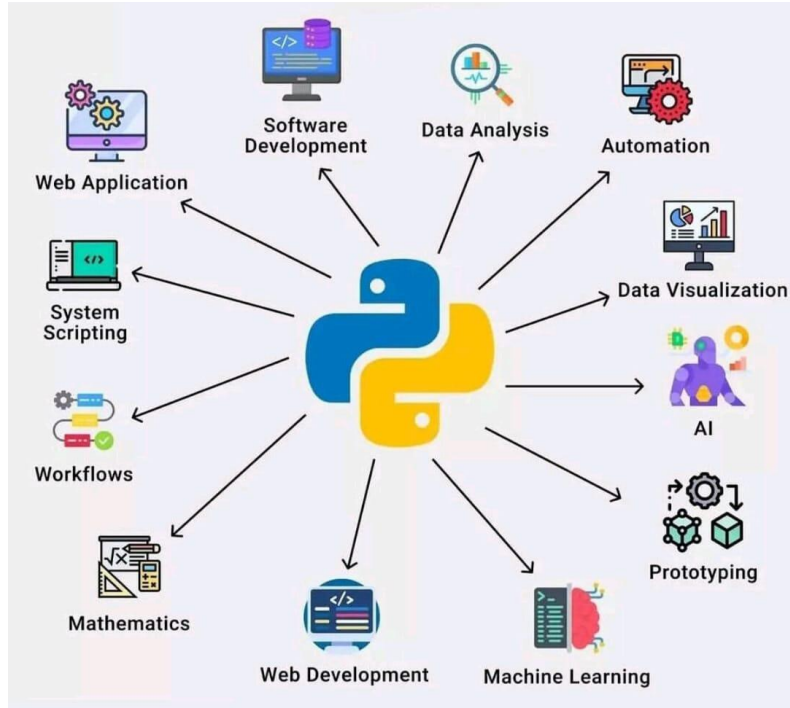
### Philosophie du langage :

- Simplicité et lisibilité du code
- Productivité accrue pour les développeurs
- Syntaxe claire et expressive



# Introduction à Python

Domaines d'application de Python :



# Introduction à Python

## Versions de Python

### Python 2.x

- **Sortie initiale** : 2000
- **Fin de support officiel** : 1er janvier 2020
- **Caractéristiques** :
  - Large base de code existante
  - Incompatibilités avec Python 3.x

### Python 3.x

- **Sortie initiale** : 2008
- **Version actuelle et recommandée**
- **Caractéristiques** :
  - Améliorations majeures du langage
  - Gestion native de l'Unicode
  - Syntaxe et bibliothèques modernisées

## Différences majeures entre Python 2 et Python 3

### Fonction `print`

- Python 2 : `print "Bonjour"`
- Python 3 : `print("Bonjour")`

### Division des entiers

- Python 2 : `5 / 2` donne 2
- Python 3 : `5 / 2` donne 2.5

## Versions mineures de Python 3

### Versions récentes :

**Python 3.7** : Introduit les annotations de type de données

**Python 3.8** : Opérateur d'assignation walrus (`:``=`)

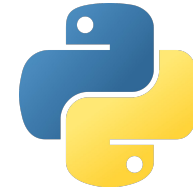
**Python 3.9** : Types génériques, améliorations de dictionnaires

**Python 3.10** : Structure de contrôle `match` (pattern matching)

**Python 3.11** : Améliorations de performance, nouvelles fonctionnalités

### Recommandation :

Utiliser la dernière version stable pour bénéficier des dernières fonctionnalités et améliorations de sécurité





# Introduction à Python

## Choix de l'IDE :

**IDLE** : Éditeur simple fourni avec Python

## Visual Studio Code (VS Code) :

- Léger et extensible
- Extensions Python disponibles

## PyCharm :

- Version **Community** gratuite
- Fonctionnalités avancées pour le développement Python

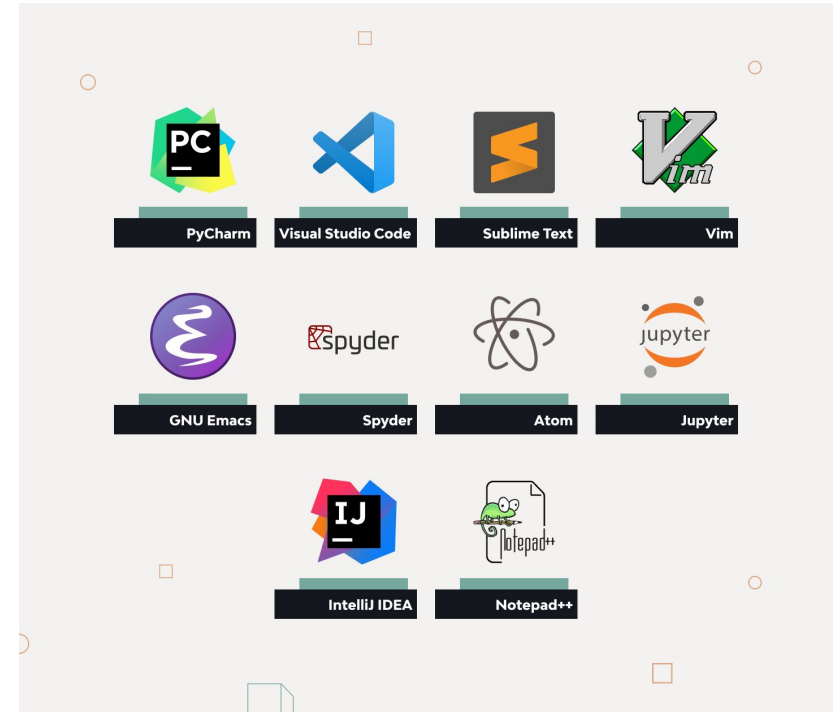
## Anaconda :

- Distribution Python avec de nombreux packages scientifiques
- Inclut Jupyter Notebook pour le code interactif

## Spyder :

- Data science spécialement conçue pour les scientifiques de données, les ingénieurs, et les chercheurs

**Notebooks interactifs** : Jupyter Notebook



## Syntaxe Simple et Lisible

```
print(x) #Erreur
x = 10
print(x) #affiche 10
name = "Python"
```

**Indentation** : Python utilise l'indentation pour structurer le code.

**Variables** : Déclaration dynamique, typage implicite.

**Lecture de code linéaire** : lecture de code ligne par ligne

## Structures de Contrôle

### Conditions

```
if x > 5:
    print("x est grand")
else:
    print("x est petit")
```

### Boucles

```
for i in range(5):
    print(i)

while x > 0:
    print(x)
    x -= 1
```

## Listes, Tuples, Sets, Dictionnaires

```
fruits = ["apple", "banana", "cherry"]

coordinates = (10, 20)

unique_numbers = {1, 2, 3, 3, 2}

student = {"name": "John", "age": 21}
```

### Importation et Utilisation

```
import math
print(math.sqrt(16))
```

# Installation de python ( Mise en pratique )

## Objectifs de l'Exercice

**Tâche :** Installation de Python



## Étapes de l'Exercice

**Étape 1 :** Aller sur le site officiel [python.org/downloads](https://python.org/downloads)

**Étape 2 :** Télécharger la dernière version stable de Python 3.x (par exemple, Python 3.11)

**Étape 3 :** Suivre les instructions d'installation spécifiques à votre système d'exploitation :

### Windows

Cochez "Add Python to PATH" lors de l'installation

**Étape 4 :** Vérifier l'installation:

`python -V`

# Choix et Installation d'un IDE ( **Exercise** )

## Objectifs de l'Exercice

**Tâche** : Choix et Installation d'un IDE



## Étapes de l'Exercice

**Choisir un IDE :**

- **Débutants** : IDLE ou Visual Studio Code
- **Avancés** : PyCharm Community Edition

**Installer l'IDE :**

- **Visual Studio Code** :
  - Téléchargez depuis [code.visualstudio.com](https://code.visualstudio.com)
  - Installez l'extension Python dans VS Code
- **PyCharm** :
  - Téléchargez depuis [jetbrains.com/pycharm](https://jetbrains.com/pycharm)

**Configurer l'IDE :**

- Assurez-vous que l'IDE reconnaît l'interpréteur Python installé
- Créez un nouvel environnement de travail ou projet

# Normes de codage : PEP8

## Qu'est-ce que PEP8 ?

**PEP** : Python Enhancement Proposal

**PEP8** : Guide de style officiel pour le code Python

**Objectif** : Améliorer la lisibilité et la cohérence du cod

```
def addition(a, b):  
    return a + b  
  
resultat = addition(5, 7)  
print("Le résultat est :", resultat)
```

## Principales recommandations :

- **Indentation** : 4 espaces par niveau (pas de tabulations)
- **Longueur des lignes** : Maximum 79 caractères
- **Nommage** :
  - **Variables et fonctions** : `snake_case`
  - **Classes** : `CamelCase`
- **Espaces autour des opérateurs** : `a = b + c`
- **Importations** :
  - Une importation par ligne
  - Les importations standards, puis les imports tiers, puis les imports locaux



# Normes de codage : PEP8

## flake8 : Outil pour vérifier le respect de PEP8

```
pip install flake8
```

## Utilisation

```
flake8 mon_programme.py
```

Cet outil permet de repérer les erreurs de style pour améliorer la lisibilité et la qualité du code.

# Chapitre 2 : Scripts et —— Programmes

# Gestion des fichiers Python (.py, .pyc)

## Fichiers `.py`

- Contiennent le code source écrit en Python.
- Utilisés pour écrire et éditer des scripts et modules.

## Fichiers `.pyc`

- Fichiers compilés générés automatiquement par Python.
- Contiennent le bytecode optimisé pour une exécution plus rapide.
- Situés dans le dossier `__pycache__`.

## Processus d'exécution

- Lorsqu'un script `.py` est exécuté, Python le compile en `.pyc` si nécessaire.
- Les fichiers `.pyc` sont utilisés pour accélérer les exécutions futures.



# Exemple de cas Réel: Application Web en Production

Imaginons une application web développée avec Django, un framework Python pour le développement web. Ce type d'application peut inclure des centaines de fichiers `.py` pour gérer les routes, les contrôleurs, les modèles de données, et bien d'autres éléments.

1. **Premier Démarrage** : Lors du premier lancement du serveur Django, chaque fichier `.py` est compilé en bytecode et des fichiers `.pyc` sont générés. Cela peut être un peu lent, car la compilation de chaque fichier en bytecode prend du temps.
2. **Optimisation des Exécutions Suivantes** : Au prochain redémarrage du serveur, Django n'a plus besoin de recompiler les fichiers inchangés, car il peut utiliser directement les fichiers `.pyc` dans `__pycache__`. Le démarrage du serveur devient alors plus rapide, car il charge le bytecode directement.
3. **Mises à Jour du Code** : Lorsqu'un développeur modifie un fichier `.py`, par exemple pour corriger un bug ou ajouter une nouvelle fonctionnalité, Python détecte que le fichier source est plus récent que le fichier `.pyc` existant. Il recompilera donc automatiquement le fichier `.py` en un nouveau `.pyc`, puis utilisera ce dernier pour les prochaines exécutions.
4. **Avantages en Production** : Dans un environnement de production, il est essentiel d'optimiser les temps de réponse et la consommation des ressources. En utilisant les fichiers `.pyc`, une application web peut redémarrer plus rapidement et répondre aux requêtes de manière plus efficace, car elle évite la surcharge de compilation à chaque démarrage.

# Structure minimal d'un programme ( Mise en pratique )

## Objectifs de la mise en pratique

### Objectif :

Comprendre l'utilisation de `print()` et les opérations arithmétiques de base en Python.

### Instructions :

1. **Imprimez des phrases** : Utilisez la fonction `print()` pour afficher quelques phrases.
2. **Effectuez des opérations** : Utilisez les opérations arithmétiques de base (addition, soustraction, multiplication, division) et affichez les résultats avec `print()`.

## Étape 1 : Imprimez une phrase de bienvenue

```
# Étape 1 : Imprimez une phrase de bienvenue
print("Bienvenue dans le monde de Python !")
print("5 + 3 =", 5 + 3)
```

# Définition du point d'entrée avec `__main__`

## Mise en pratique

### Pourquoi utiliser `__main__` ?

- Permet de définir le comportement du script lorsqu'il est exécuté directement vs importé comme module.

### Script exécuté directement :

```
def main():  
    # Code principal  
    print("Exécution du script principal.")  
  
if __name__ == "__main__":  
    main()
```

### Qu'est-ce que `__name__` ?

- La variable spéciale `__name__` est automatiquement définie par Python lorsqu'un script est exécuté.
- Elle permet de déterminer si un script est exécuté directement ou s'il est importé en tant que module dans un autre script.
- **Valeurs possibles de `__name__` :**
  - Si le script est exécuté directement, `__name__` est défini à `"__main__"`.
  - Si le script est importé, `__name__` prend le nom du fichier sans son extension

# Encodage des fichiers : UTF-8

## Importance de l'encodage :

- Détermine comment les caractères sont représentés en bytes.
- Crucial pour le support des caractères spéciaux et internationaux.

## Définir l'encodage dans Python :

- Par défaut, Python 3 utilise UTF-8.
- Optionnellement, spécifier l'encodage en en-tête :

```
# -*- coding: utf-8 -*-
```

**Décodage et encodage manuel** : Python permet de décoder et d'encoder manuellement une chaîne. Cela est utile pour convertir entre différents formats d'encodage si nécessaire :

```
# Convertir une chaîne UTF-8 en bytes
texte = "Bonjour"
bytes_utf8 = texte.encode('utf-8')

# Décoder des bytes en chaîne
texte_decode = bytes_utf8.decode('utf-8')
```

## Manipulation des encodages :

- **Lecture avec encodage spécifié** :  
python

```
with open('fichier.txt', 'r', encoding='utf-8') as f:
    contenu = f.read()
```

## Écriture avec encodage spécifié :

```
with open('fichier.txt', 'w', encoding='utf-8',
errors='replace') as f:
    f.write("Bonjour, monde!")
```

**Gestion des erreurs d'encodage** : Lors de la lecture ou de l'écriture, utiliser l'option `errors='ignore'` ou `errors='replace'` peut aider à gérer les erreurs :

# À Retenir: Chapitre 2

## Fichiers `.py` vs `.pyc` :

- `.py` contient le code source.
- `.pyc` est le bytecode compilé, situé dans le dossier `__pycache__`.

## Structure d'un script Python :

- Inclut généralement des imports, des définitions de fonctions/classes, et des instructions exécutables.

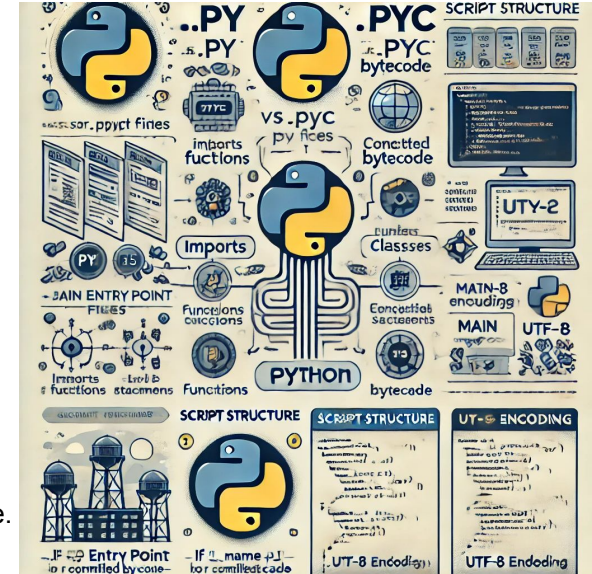
## Point d'entrée avec `__name__ == "__main__"` :

- Permet de contrôler le comportement du script lorsqu'il est exécuté directement ou importé.
- Pratique pour tester des scripts tout en les rendant réutilisables comme modules.

## Encodage UTF-8 :

- Recommandé pour gérer correctement les caractères spéciaux.
- Utiliser `# -*- coding: utf-8 -*-` pour spécifier l'encodage en en-tête, si nécessaire.

**Bonne pratique :** Toujours inclure un point d'entrée avec `if __name__ == "__main__":` dans les scripts pour une meilleure modularité du code.



# Chapitre 3 : Type de Données

---

# Types Primitifs : Entiers et Flottants

## Mise en pratique

**Entiers (`int`)** : Nombres sans décimale. Exemple : `42`, `-10`, `0`

- **Opérations** : Addition (+), soustraction (-), multiplication (\*), division entière (//), modulo (%), puissance (\*\*).

```
a = 5
b = 3
print(a + b)  # Affiche 8
print(a ** 2) # Affiche 25
```

**Flottants (`float`)** : Nombres à virgule flottante (décimaux). Exemple : `3.14`, `-0.001`

- **Précision** : Peut avoir une certaine limitation en termes de précision lors des calculs.

```
pi = 3.14159
print(pi * 2)  # Affiche 6.28318
```

# Types Primitifs : Booléens et Chaînes de Caractères

**Booléens (bool)** : Vrai (True) ou Faux (False)

- **Opérations logiques** : `and`, `or`, `not`

```
is_raining = True
has_umbrella = False
print(is_raining and has_umbrella) # Affiche
False
```

**Chaînes de caractères (str)** : Suite de caractères, entourée de guillemets simples ('...') ou doubles ("...").

- **Concaténation** : `+` pour concaténer deux chaînes.

```
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name
print(full_name) # Affiche "John Doe"
```



# Conversion et Casting des Types

**Conversion implicite :** Python convertit automatiquement certains types si nécessaire.

- Exemple : Addition d'un entier et d'un flottant

```
result = 5 + 3.0 # Python convertit 5 en 5.0
print(result)   # Affiche 8.0
```

**Conversion explicite (casting) :** Utiliser des fonctions de conversion pour changer le type d'une valeur.

- **Fonctions courantes :**
  - `int()` : Convertir en entier.
  - `float()` : Convertir en flottant.
  - `str()` : Convertir en chaîne de caractères.
  - `bool()` : Convertir en booléen.

```
age_str = "25"
age_int = int(age_str)
print(age_int) # Affiche 25
print(float(age_int)) # Affiche 25.0
print(bool(0)) # Affiche False
```

# Collections : Sets et Dictionnaires

**Sets (set)** : Collections non ordonnées et sans doublons.

- **Définition** : Utiliser des accolades {}. Exemple :  
`unique_numbers = {1, 2, 3, 3}`
- **Opérations** : Ajout (`add()`), suppression (`remove()`), union (`|`), intersection (`&`).

```
colors = {"red", "green", "blue"}
colors.add("yellow")
print(colors)  # Affiche {'red', 'green', 'blue', 'yellow'}
```

**Dictionnaires (dict)** : Collections non ordonnées de paires clé-valeur.

- **Définition** : Utiliser des accolades {}. Exemple :  
`person = {"name": "John", "age": 30}`
- **Accès** : Par clé (`person["name"]`).
- **Opérations** : Ajout, modification, suppression d'éléments.

```
student = {"name": "Alice", "age": 22}
student["major"] = "Physics"
print(student)  # Affiche {'name': 'Alice', 'age': 22, 'major': 'Physics'}
```

# Conversion avancée de types en Python

## Conversion entre types de collections

En Python, il est également possible de convertir entre différents types de collections (listes, ensembles, tuples, dictionnaires). Ces conversions permettent de changer la structure de données pour bénéficier des propriétés uniques de chaque type.

### 1. Conversion de listes en ensembles et en tuples

- **list()** : Crée une liste à partir d'une autre collection (ensemble, tuple, chaîne de caractères, etc.).
- **set()** : Convertit une collection (liste, tuple, chaîne de caractères) en un ensemble, éliminant ainsi les doublons car les ensembles ne contiennent que des éléments uniques.
- **tuple()** : Transforme une collection (liste, ensemble, chaîne de caractères) en un tuple. Les tuples étant immuables, cette conversion est utile pour des données constantes.

### Conversion de chaînes de caractères en listes, ensembles et tuples

- Il est possible de convertir une chaîne de caractères en une collection de caractères, que ce soit sous forme de liste, d'ensemble ou de tuple.

```
# Exemple : Conversion d'une liste en un ensemble pour éliminer
les doublons
nombres = [1, 2, 3, 4, 3, 2]
nombres_uniques = set(nombres)
print(nombres_uniques) # Affiche : {1, 2, 3, 4}

# Exemple : Conversion d'un ensemble en tuple pour une
structure immuable
nombres_tuple = tuple(nombres_uniques)
print(nombres_tuple) # Affiche : (1, 2, 3, 4)

# Exemple : Conversion d'une chaîne en liste
texte = "hello"
texte_liste = list(texte)
print(texte_liste) # Affiche : ['h', 'e', 'l', 'l', 'o']

# Conversion en ensemble pour éliminer les doublons
texte_ensemble = set(texte)
print(texte_ensemble) # Affiche : {'h', 'e', 'o', 'l'}

# Conversion en tuple pour une structure immuable
texte_tuple = tuple(texte)
print(texte_tuple) # Affiche : ('h', 'e', 'l', 'l', 'o')
```



# Dictionnaire : Exemple

```
# Parcourir chaque étudiant et afficher les informations
for student in students:
    print("\nInformations de l'étudiant :")
    print("Nom :", student["name"])
    print("Âge :", student["age"])
    print("Filière :", student["major"])
    print("Notes :", student["grades"])
    print("Email :", student["contact_info"]["email"])
    print("Téléphone :", student["contact_info"]["phone"])

    # Ajouter une nouvelle note
    student["grades"].append(95)
    print("Notes après ajout :", student["grades"])

    # Calculer et afficher la moyenne des notes
    average_grade = sum(student["grades"]) / len(student["grades"])
    print("Moyenne des notes :", average_grade)

    # Mettre à jour l'adresse e-mail
    student["contact_info"]["email"] =
f"{student['name'].lower().updated@example.com"}
    print("Email mis à jour :", student["contact_info"]["email"])

    # Supprimer la spécialité (major)
    del student["major"]
    print("Informations après suppression de la spécialité :", student)

# Affichage de tous les étudiants mis à jour
print("\nDétails complets de tous les étudiants :")
for student in students:
    print(student)
```

```
# Liste de dictionnaires représentant plusieurs étudiants
students = [
    {
        "name": "Alice",
        "age": 22,
        "major": "Physics",
        "grades": [88, 92, 79, 85],
        "contact_info": {
            "email": "alice@example.com",
            "phone": "123-456-7890"
        }
    },
    {
        "name": "Bob",
        "age": 24,
        "major": "Mathematics",
        "grades": [90, 85, 88, 92],
        "contact_info": {
            "email": "bob@example.com",
            "phone": "987-654-3210"
        }
    },
    {
        "name": "Carol",
        "age": 23,
        "major": "Computer Science",
        "grades": [75, 80, 85, 90],
        "contact_info": {
            "email": "carol@example.com",
            "phone": "555-123-4567"
        }
    }
]
```

# Utiliser OpenAI pour la Programmation Python

**Pourquoi chatgpt pour Python ?** OpenAI peut vous assister dans des tâches de programmation, de la génération de code à la résolution de bugs et à l'optimisation.

## Exemples de Prompts :

- **Création de Fonctions :**
  - *Prompt : "Écris une fonction Python pour calculer la distance entre deux points en 3D, avec des paramètres x, y, z."*
  - **Modules utiles :** `math` pour `sqrt()`.
- **Gestion de Fichiers :**
  - *Prompt : "Montre-moi un script Python pour lire un fichier CSV et afficher la première colonne."*
  - **Modules utiles :** `csv`, `pandas`.
- **Automatisation de Tâches :**
  - *Prompt : "Programme un script pour automatiser le téléchargement d'images à partir d'une URL."*
  - **Modules utiles :** `requests`, `os`, `shutil`.

## Conseils pour un Prompt Efficace :

1. Soyez précis : décrivez exactement ce que vous voulez.
2. Mentionnez les modules nécessaires.
3. Demandez un code commenté pour mieux comprendre chaque étape.

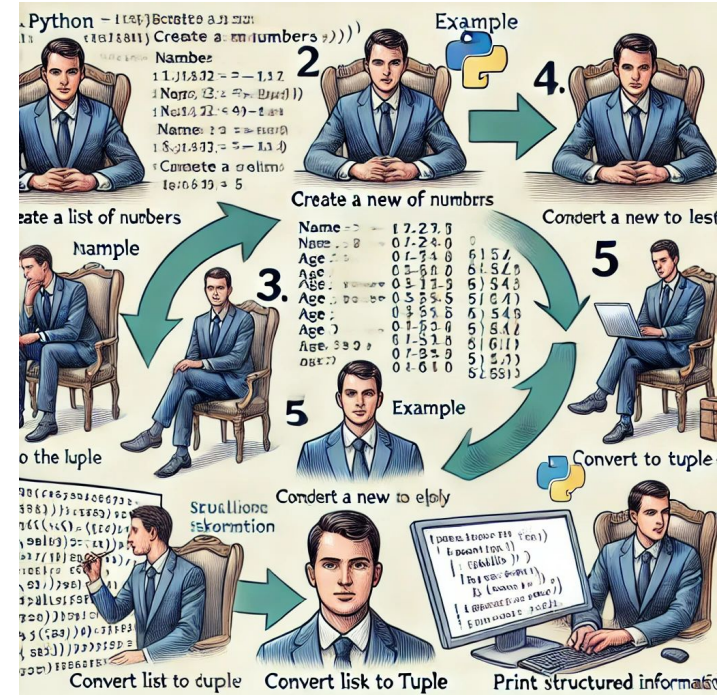
## Astuces:

- Dans ce cours je vous recommande d'utiliser openai car python est tellement partagé et communautaire que le code créer par ChatGPT est généralement très qualitatif
- Ajouter dans le prompt "Commente chaque ligne de code que tu fais, l'utilisation d'une méthode plutôt qu'une autre et les alternatives possible afin de mieux comprendre les possibilités qu'offre Python."
- Vérifier relire, et corriger selon vos propres compréhension et intentions finale.

# Exercise

## Exercice : Gestion des Collections et Conversion

- Créer une liste de nombres :  
`numbers = [1, 2, 3, 4, 5]`
- 1. Ajouter un nouvel élément à la liste.
- 2. Convertir la liste en un tuple.
- 3. Créer un dictionnaire contenant des informations sur une personne : nom, âge, profession.
- 4. Ajouter un set pour les compétences de cette personne.
- 5. Imprimer toutes les informations de manière structurée.



# À Retenir

## Types Primitifs :

- **Entiers (`int`)** : Nombres sans décimale.
- **Flottants (`float`)** : Nombres avec décimales.
- **Booléens (`bool`)** : Vrai ou Faux.
- **Chaînes (`str`)** : Texte.

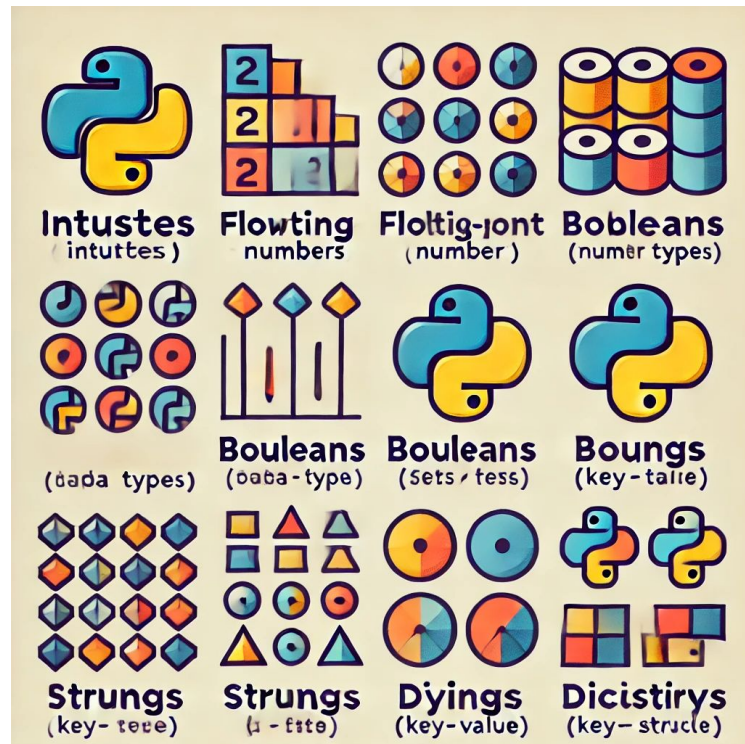
## Conversions :

- Implicites lors d'opérations mixtes (entier + flottant).
- Explicites via les fonctions `int()`, `float()`, `str()`, etc.
- Conversions complexes possibles

## Collections :

- **Listes** : Ordonnées, modifiables, utilisent des crochets `[]`.
- **Tuples** : Ordonnés, immuables, utilisent des parenthèses `()`.
- **Sets** : Non ordonnés, sans doublons, utilisent des accolades `{}`.
- **Dictionnaires** : Non ordonnés, stockent des paires clé-valeur, utilisent des accolades `{}`.

**Utilisation des Collections** : Permet de regrouper et manipuler plusieurs valeurs de manière structurée et efficace.





# Correction

```
Liste des nombres : [1, 2, 3, 4, 5, 6]
```

```
Tuple des nombres : (1, 2, 3, 4, 5, 6)
```

```
Informations sur la personne :
```

```
Nom : Alice
```

```
Âge : 30
```

```
Profession : Développeur
```

```
Compétences : Python, JavaScript, SQL
```

**Ajout d'un élément à la liste** : La méthode `.append()` permet de modifier la liste en ajoutant de nouveaux éléments.

**Conversion en tuple** : La conversion avec `tuple()` rend la collection immuable.

**Ajout du set au dictionnaire** : Les sets garantissent l'unicité des éléments. Ici, les compétences sont stockées sans doublons.

**Accès aux éléments du dictionnaire** : Utilisation des clés (`person_info['nom']`) pour accéder aux valeurs.

```
# Étape 1 : Créer une liste de nombres
numbers = [1, 2, 3, 4, 5]

# Étape 2 : Ajouter un nouvel élément à la liste
numbers.append(6)

# Étape 3 : Convertir la liste en un tuple
numbers_tuple = tuple(numbers)

# Étape 4 : Créer un dictionnaire contenant des informations sur une
personne
person_info = {
    "nom": "Alice",
    "âge": 30,
    "profession": "Développeur"
}

# Étape 5 : Ajouter un set pour les compétences de cette personne
skills_set = {"Python", "JavaScript", "SQL"}
person_info["compétences"] = skills_set

# Étape 6 : Imprimer toutes les informations de manière structurée
print("Liste des nombres :", numbers)
print("Tuple des nombres :", numbers_tuple)
print("\nInformations sur la personne :")
print(f"Nom : {person_info['nom']}")
print(f"Âge : {person_info['âge']}")
print(f"Profession : {person_info['profession']}")
print(f"Compétences : {' '.join(person_info['compétences'])}")
```



# Chapitre 4 : Opérations de

---

# Base

# Opérateurs d'Affectation

## Opérateurs d'affectation simples :

- `=` : Assigner une valeur à une variable

```
x = 10
```

## Opérateurs d'affectation combinés :

- `+=` : Additionner et assigner.

```
x += 5 # Équivaut à x = x + 5
```

`-=` : Soustraire et assigner.

`*=` : Multiplier et assigner.

`/=` : Diviser et assigner.

`//=` : Division entière et assignation.

`%=` : Modulo et assignation.

`**=` : Puissance et assignation.

## Astuce

Les opérateurs combinés permettent d'écrire du code plus concis et lisible.

Attention à l'ordre d'évaluation dans les expressions plus complexes.

# Opérateur d'Assignment Walrus (:=)

## Éviter des Calculs Répétitifs

```
# Sans walrus operator
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
if len(my_list) > 10:
    length = len(my_list)
    print(f"La liste a {length} éléments")
```

```
# Avec walrus operator
if (length := len(my_list)) > 10:
    print(f"La liste a {length} éléments")
```

L'intérêt principal du walrus operator est donc d'**éviter la redondance** et de **gagner en concision** et en **efficacité**

Le retour de la valeur permet de **combinaison assignation et évaluation**, économisant du code et des calculs.

# Opérateurs Arithmétiques

## Liste des opérateurs :

- `+` : Addition.
- `-` : Soustraction.
- `*` : Multiplication.
- `/` : Division.
- `//` : Division entière.
- `%` : Modulo (reste de la division).
- `**` : Exponentiation (puissance).

## Astuce

Utilisez `//` pour obtenir le quotient entier, utile pour des opérations qui nécessitent des valeurs entières sans les décimales.

```
a = 15
b = 4
c = a + b
d = a % b
e = a ** 2
e += a
print(a // b)
print(a % b)  # Affiche 3 (reste de 15 / 4)
print(a ** 2)  # Affiche 225
```

# Opérateurs Logiques

## Opérateurs :

- **and** : Retourne **True** si les deux opérandes sont vrais.
- **or** : Retourne **True** si au moins un des opérandes est vrai.
- **not** : Inverse la valeur logique de l'opérande.

## Astuce

Les opérateurs logiques sont très utiles dans les structures conditionnelles (**if**, **while**) pour contrôler le flux du programme.

```
x = 10
y = 20
print(x > 5 and y < 25)  # Affiche True
print(not (x != y))      # Affiche True
```

# Opérateurs Relationnels

## Liste des opérateurs :

- `==` : Égal à.
- `!=` : Différent de.
- `>` : Supérieur à.
- `<` : Inférieur à.
- `>=` : Supérieur ou égal à.
- `<=` : Inférieur ou égal à.

## Astuce

Toujours utiliser `==` pour comparer les valeurs et non `=` qui est utilisé pour l'affectation.

```
a = 5
b = 10
c = a == b
c = False
if c:

print(a != b)  # Affiche True
print(a <= b)  # Affiche True
```

# Gestion des Entrées/Sorties

## Entrée utilisateur avec `input()` :

- Permet de récupérer des informations saisies par l'utilisateur.

```
name = input("Entrez votre nom : ")  
print(f"Bonjour, {name} !")
```

## Affichage avec `print()` :

- Peut afficher des variables, des chaînes, et d'autres types.

```
age = 25  
print("J'ai", age, "ans.")
```

## Astuce

`input()` retourne toujours une chaîne. Utilisez le casting (`int()`, `float()`) pour convertir les entrées utilisateur au besoin.

# Formatage de Chaînes avec f-strings

**Utilisation des f-strings** : Introduites dans Python 3.6+, elles permettent d'incorporer des variables directement dans des chaînes.

## Syntaxe:

```
name = "Alice"  
age = 30  
print(f"Nom : {name}, Âge : {age}")
```

## Avantages :

- Plus lisible que les anciennes méthodes de formatage (%) et `.format()`.
- Permet d'incorporer des expressions directement dans les accolades

```
print(f"L'année prochaine, j'aurai {age + 1}  
ans.")
```

## Astuce

Les **f-strings** sont une manière puissante et élégante de formater les chaînes de caractères.

Supporte également la mise en forme avancée, comme la précision des nombres :

```
pi = 3.14159  
print(f"Pi avec deux décimales : {pi:.2f}")
```



# Opérateurs de Mise en Forme Avancée dans les f-strings

Les **f-strings** permettent une variété d'options de formatage avancé pour les chaînes, les nombres et d'autres types de données. Voici quelques opérateurs de mise en forme couramment utilisés :

## 1. Alignement du texte

- **Gauche (<), Centre (^), Droite (>) :**
  - Utilisez ces opérateurs pour aligner du texte dans un espace réservé de taille fixe.
  - **Syntaxe :** `{variable:<width},`  
`{variable:^width},`  
`{variable:>width}`

```
text = "Python"
print(f"|{text:<10}|") # Aligné à gauche :
'Python      '
print(f"|{text:^10}|") # Centre : '  Python  '
print(f"|{text:>10}|") # Aligné à droite : '
Python'
```

## Remplissage avec des caractères

- Vous pouvez également spécifier un caractère de remplissage avec l'alignement.
  - **Syntaxe :** `{variable:fill<width}`

```
text = "42"
print(f"{text:0>5}") # Rempli avec des zéros à
gauche : '00042'
print(f"{text:*^7}") # Centre avec des étoiles :
'**42***'
```

## Formatage des nombres à virgule flottante

- **Précision (.nf) :** Contrôler le nombre de chiffres après la virgule.
  - **Syntaxe :** `{variable:.nf}`

```
pi = 3.14159
print(f"{pi:.2f}") # Affiche '3.14'
print(f"{pi:.4f}") # Affiche '3.1416'
```

# Opérateurs de Mise en Forme Avancée dans les f-strings

Les **f-strings** permettent une variété d'options de formatage avancé pour les chaînes, les nombres et d'autres types de données. Voici quelques opérateurs de mise en forme couramment utilisés :

## Affichage en pourcentage

- **Pourcentage (%) :**
  - Multiplie le nombre par 100 et ajoute le signe %.
  - **Syntaxe :** `{variable:.nf%}`

```
ratio = 0.25
print(f"{ratio:.2%}") # Affiche '25.00%'
```

## Remplissage avec des zéros

- **Remplir avec des zéros (0) :**
  - Ajoute des zéros à gauche pour obtenir une largeur fixe.

```
number = 42
print(f"{number:05}") # Affiche '00042'
```

## Affichage en notation scientifique

- **Notation scientifique (e) :**
  - Formate le nombre en notation scientifique.
  - **Syntaxe :** `{variable:.ne}`

```
large_number = 123456789
print(f"{large_number:.2e}") # Affiche '1.23e+08'
```

## Formatage en binaire, octal, hexadécimal

- **Binaire (b), Octal (o), Hexadécimal (x ou X) :**
  - Affiche les nombres entiers dans différentes bases.

```
number = 255
print(f"{number:b}") # Affiche '11111111'
(binaire)
print(f"{number:o}") # Affiche '377' (octal)
print(f"{number:x}") # Affiche 'ff' (hexadécimal,
minuscule)
print(f"{number:X}") # Affiche 'FF' (hexadécimal,
majuscule)
```

# Commentaires et Documentation

## Commentaires :

Ligne simple : Utiliser #.

```
# Ceci est un commentaire
```

Multi-lignes : Utiliser des guillemets triples ("""...""")  
ou '''...''').

```
"""  
Ceci est un commentaire  
sur plusieurs lignes.  
"""
```

## Documentation des fonctions :

- Les docstrings sont des commentaires qui décrivent une fonction, situés immédiatement après la définition de la fonction.

```
def add(a, b):  
    """Retourne la somme de a et b."""  
    return a + b
```

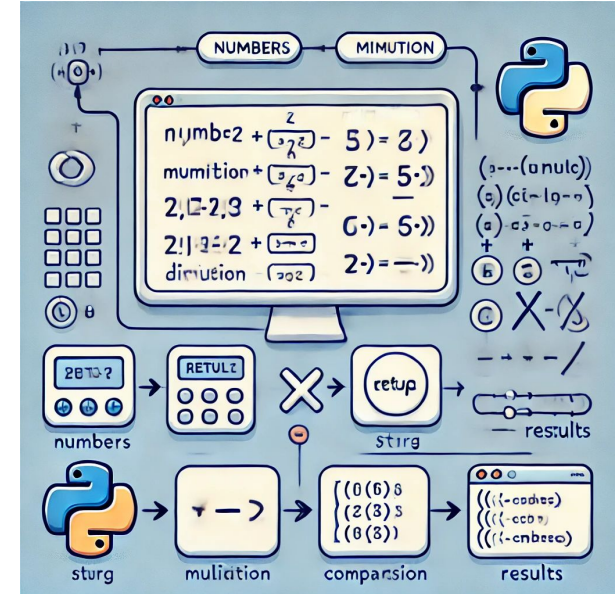
## Astuce

Les commentaires et la documentation rendent le code plus facile à comprendre et à maintenir. Utilisez-les régulièrement pour expliquer les parties complexes de votre code.

# Exercice

## Opérations de Base

1. **Créer un programme** qui demande à l'utilisateur de saisir deux nombres.
2. **Effectuer les opérations** arithmétiques de base (+, -, \*, /) sur ces deux nombres et afficher les résultats.
3. **Demander à l'utilisateur** de saisir une chaîne, puis l'afficher en utilisant un **f-string**.
4. **Utiliser les opérateurs logiques** pour vérifier si le premier nombre est supérieur ou égal au second.
5. **Documenter les étapes** du code avec des commentaires.





# Correction

**Entrées utilisateur :** Les deux nombres sont demandés et convertis en flottants pour prendre en compte les nombres décimaux.

**Opérations arithmétiques :** Les opérations de base sont effectuées et les résultats sont stockés dans des variables.

**Affichage des résultats :** Les résultats des opérations sont affichés en utilisant des **f-strings** pour un formatage clair et lisible.

**Gestion de la division par zéro :** Un contrôle est ajouté pour éviter une erreur lors de la division par zéro.

**Saisie d'une chaîne :** L'utilisateur entre un texte qui est ensuite affiché à l'aide d'un **f-string**.

**Opérateurs logiques :** Le programme vérifie si le premier nombre est supérieur ou égal au second et affiche le résultat en conséquence.

**Commentaires :** Chaque étape est documentée pour clarifier le rôle de chaque section du code.

```
# Demander à l'utilisateur de saisir deux nombres
num1 = float(input("Entrez le premier nombre : ")) # Conversion en
flottant pour prendre en charge les décimales
num2 = float(input("Entrez le second nombre : "))

# Effectuer les opérations arithmétiques de base et afficher les
résultats
addition = num1 + num2
soustraction = num1 - num2
multiplication = num1 * num2
division = num1 / num2 if num2 != 0 else "Division par zéro impossible"
# Gestion de la division par zéro

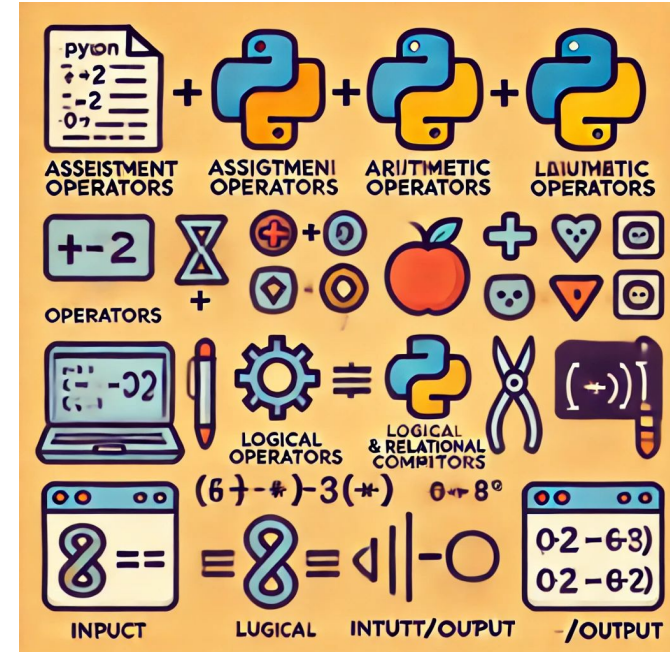
print(f"Addition : {num1} + {num2} = {addition}")
print(f"Soustraction : {num1} - {num2} = {soustraction}")
print(f"Multiplication : {num1} * {num2} = {multiplication}")
print(f"Division : {num1} / {num2} = {division}")

# Demander à l'utilisateur de saisir une chaîne et l'afficher en
utilisant un f-string
text = input("Saisissez un texte : ")
print(f"Le texte que vous avez saisi est : {text}")

# Utiliser les opérateurs logiques pour vérifier si le premier nombre
est supérieur ou égal au second
if num1 >= num2:
    print(f"Le premier nombre ({num1}) est supérieur ou égal au second
nombre ({num2}).")
else:
    print(f"Le premier nombre ({num1}) est inférieur au second nombre
({num2}).")
```

## A retenir

- **Opérateurs d'affectation** permettent de manipuler les variables efficacement (`+=`, `-=`, etc.).
- **Opérateurs arithmétiques, logiques et relationnels** sont essentiels pour les calculs et les comparaisons.
- **Entrées/sorties** :
  - `input()` pour les saisies utilisateur.
  - `print()` pour afficher les résultats.
  - Les `f-strings` facilitent le formatage et l'insertion de variables dans les chaînes.
- **Commentaires** :
  - Utilisez `#` pour les commentaires en ligne.
  - Les docstrings (`""" ... """`) servent à documenter les fonctions et classes



# Chapitre 5 : Contrôle de Flux

---

# Structures Conditionnelles : **if**, **else**, **elif**

**Condition **if**** : Permet d'exécuter un bloc de code uniquement si une condition est vraie.

```
age = 18
if age >= 18:
    print("Vous êtes majeur.")
```

**Condition **else**** : S'exécute si toutes les conditions **if** et **elif** précédentes sont fausses.

```
age = 16
if age >= 18:
    print("Vous êtes majeur.")
else:
    print("Vous êtes mineur.")
```

**Condition **elif**** : Permet de tester des conditions supplémentaires si la condition **if** est fausse.

```
age = 20
if age < 13:
    print("Vous êtes un enfant.")
elif 13 <= age < 18:
    print("Vous êtes un adolescent.")
else:
    print("Vous êtes un adulte.")
```

## Astuce

**Ordre des conditions** : Les conditions sont évaluées dans l'ordre. Le premier bloc dont la condition est vraie s'exécute, puis le reste est ignoré.

**Indentation** : Indentez correctement les blocs **if**, **else**, et **elif** pour assurer le bon fonctionnement du code.



# Structure Conditionnelle : **match** (Python 3.10+)

**match** : Nouvelle structure introduite dans Python 3.10, similaire au **switch** des autres langages.

```
status_code = 404
match status_code:
    case 200:
        print("Succès")
    case 404:
        print("Page non trouvée")
    case 500:
        print("Erreur du serveur")
    case _:
        print("Code inconnu")
```

**Cas par défaut** : Utilisez **case \_** pour capturer les valeurs qui ne correspondent à aucun autre cas.

**Utilisation** : Utile pour gérer plusieurs cas possibles d'une même variable, rendant le code plus lisible.

# Boucles : **for**

**Boucle **for**** : Parcourt une séquence (liste, tuple, chaîne, etc.) élément par élément.

```
fruits = ["pomme", "banane", "cerise"]
for fruit in fruits:
    print(fruit)
```

**Utiliser **range()**** : Crée une séquence de nombres, souvent utilisée pour itérer un nombre fixe de fois.

```
for i in range(5):
    print(f"Compteur : {i}")
```

**À Retenir :**

**Fonctionnement** : La boucle s'arrête automatiquement une fois tous les éléments de la séquence parcourus.

**Utilisation de **range(start, stop, step)**** : Vous pouvez spécifier des valeurs de début, de fin, et d'incrément.

# Boucles : **while**

**Boucle **while**** : S'exécute tant qu'une condition est vraie.

```
count = 0
while count < 5:
    print(f"Compteur : {count}")
    count += 1
```

**Boucles infinies** : Faites attention aux boucles infinies. Elles surviennent si la condition ne devient jamais fausse.

```
count = 1
while count > 0:
    print(f"Compteur : {count}")
    count += 1
```

**À Retenir :**

**Contrôle de la condition** : Assurez-vous que la condition finira par devenir fausse pour éviter des boucles infinies.

**Utilisation** : Les boucles **while** sont idéales quand le nombre d'itérations n'est pas défini à l'avance.

# Gestion des Exceptions : **try, except**

**Pourquoi gérer les exceptions ?** : Permet de contrôler les erreurs potentielles pendant l'exécution et d'éviter que le programme ne se termine brutalement.

**Structure de base :**

```
try:
    number = int(input("Entrez un nombre : "))
    result = 10 / number
    print(f"Résultat : {result}")
except ValueError:
    print("Veuillez entrer un nombre valide.")
except ZeroDivisionError:
    print("Impossible de diviser par zéro.")
```

**Multiples **except**** : Vous pouvez gérer différentes exceptions en utilisant plusieurs blocs **except**.

**À Retenir :**

**Bloc **try**** : Code susceptible de provoquer une erreur.

**Blocs **except**** : Code qui s'exécute si une exception survient.

**Préciser le type d'exception** : Permet de réagir différemment selon le type d'erreur.

# Combinaison de Boucles et Contrôles de Flux

**Objectif :** Gérer des données avec des conditions multiples et combiner des boucles pour des opérations avancées.

**Exemple :** Filtrer et agréger des données en fonction de plusieurs critères.

## Explication :

- **Conditions multiples :** Vérifie si la moyenne est  $> 80$  avant d'ajouter au total.
- **Boucle :** La boucle `for` parcourt chaque étudiant pour appliquer le filtre et agréger les données.

Seul 2 étudiants seront print()

```
# Liste d'étudiants avec des noms et des moyennes de notes
students = [
    {"name": "Alice", "average": 88},
    {"name": "Bob", "average": 76},
    {"name": "Carol", "average": 92},
    {"name": "Dave", "average": 67}
]

# Filtrer les étudiants ayant une moyenne > 80 et calculer la somme de leurs moyennes
sum_high_scores = 0
for student in students:
    if student["average"] > 80:
        print(f"{student['name']} a une moyenne de {student['average']}")
        sum_high_scores += student["average"]

print("Somme des moyennes supérieures à 80 :", sum_high_scores)
```

# Utilisation de ZIP et Enumerate

**Explication :** `zip()` associe les matières et les notes,  
`enumerate()` ajoute un index à chaque note.

```
subjects = ["Math", "Science", "History"]
alice_grades = [85, 90, 88]

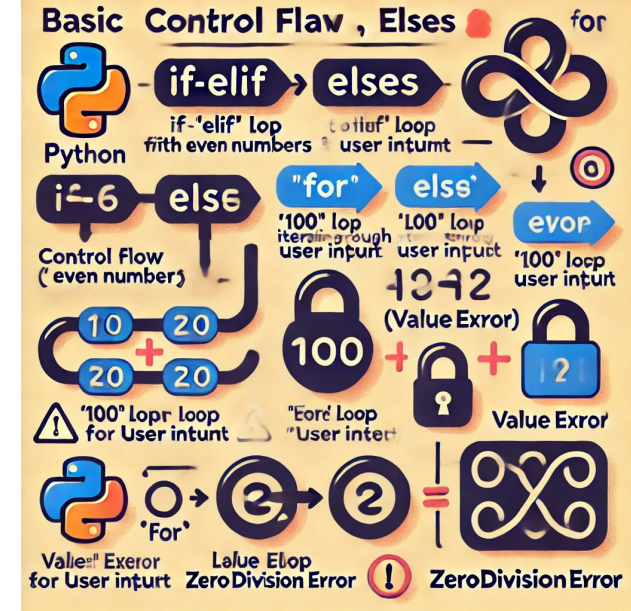
# Utiliser zip() pour associer les matières et les notes
for subject, grade in zip(subjects, alice_grades):
    print(f"{subject}: {grade}")

# Utiliser enumerate() pour afficher l'index et la valeur
for index, grade in enumerate(alice_grades):
    print(f"Index {index}: Note {grade}")
```

# Mise en Pratique

## Exercice : Contrôle de Flux

1. **Conditions :**
  - Demander à l'utilisateur de saisir un nombre.
  - Utiliser `if`, `elif`, et `else` pour déterminer si le nombre est négatif, zéro, ou positif.
2. **Boucles :**
  - Créer une boucle `for` pour afficher tous les nombres pairs entre 1 et 20.
  - Créer une boucle `while` qui demande à l'utilisateur un mot de passe jusqu'à ce qu'il saisisse le bon.
3. **Exceptions :**
  - Demander à l'utilisateur de saisir un nombre pour diviser 100.
  - Gérer les erreurs `ValueError` et `ZeroDivisionError`.



# Correction

## Conditions :

- Demande à l'utilisateur de saisir un nombre, puis utilise des structures `if`, `elif`, et `else` pour vérifier si le nombre est négatif, zéro ou positif.

## Boucles :

- La boucle `for` utilise `range(1, 21)` pour parcourir les nombres de 1 à 20 et affiche les nombres pairs à l'aide de l'opérateur modulo (%).
- La boucle `while` demande à l'utilisateur de saisir le mot de passe jusqu'à ce qu'il entre le bon mot de passe. Si le mot de passe est incorrect, l'utilisateur est invité à essayer de nouveau.

## Exceptions :

- Le bloc `try` demande à l'utilisateur de saisir un nombre pour diviser 100.
- Les blocs `except` gèrent les erreurs `ValueError` (si l'utilisateur n'entre pas un nombre valide) et `ZeroDivisionError` (si l'utilisateur tente de diviser par zéro).

```
# Demander à l'utilisateur de saisir un nombre
number = float(input("Saisissez un nombre : "))

# Utiliser if, elif, et else pour déterminer si le nombre est négatif, zéro, ou positif
if number < 0:
    print("Le nombre est négatif.")
elif number == 0:
    print("Le nombre est zéro.")
else:
    print("Le nombre est positif.")

# Boucles
# Boucle for pour afficher tous les nombres pairs entre 1 et 20
print("Nombres pairs entre 1 et 20 :")
for i in range(1, 21):
    print(i)
    if i % 2 == 0:
        print(i)

# Boucle while qui demande à l'utilisateur un mot de passe jusqu'à ce qu'il saisisse le bon
correct_password = "python123"
password = ""
while password != correct_password:
    password = input("Saisissez le mot de passe : ")
    if password != correct_password:
        print("Mot de passe incorrect, essayez de nouveau.")
print("Mot de passe correct, accès autorisé.")

# Exceptions
# Demander à l'utilisateur de saisir un nombre pour diviser 100
try:
    divisor = float(input("Saisissez un nombre pour diviser 100 : "))
    result = 100 / divisor
    print(f"100 divisé par {divisor} est égal à {result}.")
except ValueError:
    print("Erreur : Veuillez entrer un nombre valide.")
except ZeroDivisionError:
    print("Erreur : Division par zéro impossible.")
```



# À Retenir

**Structures conditionnelles** : `if`, `elif`, `else` et `match` permettent de contrôler le flux d'exécution selon les conditions.

**Boucles** :

- `for` : Parcourt des séquences (listes, tuples, chaînes, etc.).
- `while` : S'exécute tant qu'une condition reste vraie.

**Gestion des exceptions** : `try`, `except` permettent de gérer les erreurs et d'éviter les plantages du programme.

# À Retenir

**Structures conditionnelles** : `if`, `elif`, `else` et `match` permettent de contrôler le flux d'exécution selon les conditions.

**Boucles** :

- `for` : Parcourt des séquences (listes, tuples, chaînes, etc.).
- `while` : S'exécute tant qu'une condition reste vraie.

**Gestion des exceptions** : `try`, `except` permettent de gérer les erreurs et d'éviter les plantages du programme.

# Chapitre 6 : Fonctions

---

# Définition des Fonctions avec **def**

## Qu'est-ce qu'une fonction ?

- Un bloc de code réutilisable conçu pour effectuer une tâche spécifique.
- Utilise le mot-clé **def** pour être définie.

## Syntaxe de base :

```
def nom_de_la_fonction(param1, param2):  
    # Instructions  
    return resultat
```

## Exemple Simple:

```
def addition(a, b):  
    result = a + b  
    return result  
  
resultat = addition(5, 3)  
print(resultat) # Affiche 8
```

## Astuce

La valeur de retour (**return**) peut être utilisée dans d'autres parties du code.

# Retourner des Valeurs

Le mot-clé `return` termine l'exécution de la fonction et renvoie une valeur.

```
def carre(nombre):  
    return nombre ** 2  
print(carre(4))  # Affiche 16
```

## Plusieurs valeurs de retour :

- Une fonction peut retourner plusieurs valeurs sous forme de tuple.

```
def operations(a, b):  
    somme = a + b  
    produit = a * b  
    return somme, produit  
  
s, p = operations(3, 4)  
print(s)  # Affiche 7  
print(p)  # Affiche 12
```

## Astuce

Les fonctions peuvent retourner un ou plusieurs résultats.

Si une fonction ne contient pas de `return`, elle retourne `None` par défaut.

# Fonctions Anonymes : **lambda**

## Qu'est-ce qu'une fonction **lambda** ?

- Une fonction anonyme, définie en une seule ligne.
- Utilisée pour des opérations simples.

```
carre = lambda x: x ** 2
print(carre(5))  # Affiche 25
```

## Utilisation courante dans des fonctions comme **map()**, **filter()**, et **sorted()** :

```
nombres = [1, 2, 3, 4, 5]
carre_nombres = list(map(lambda x: x ** 2,
nombres))
print(carre_nombres)  # Affiche [1, 4, 9, 16, 25]
```

## Astuce

Les **lambdas** sont des fonctions anonymes pour des calculs simples.

Elles ne contiennent pas de **return**, l'expression évaluée est automatiquement retournée.

# Utilisation des Variables Globales avec **global**

## Variables locales vs globales :

- **Locales** : Définies à l'intérieur d'une fonction et ne sont pas accessibles à l'extérieur.
- **Globales** : Définies en dehors des fonctions et accessibles dans tout le programme.

## Accéder aux variables globales :

- Pour modifier une variable globale à l'intérieur d'une fonction, utilisez le mot-clé **global**.

## Astuce

Utilisez **global** avec précaution, car il peut rendre le code plus difficile à comprendre et à déboguer.

Privilégiez les paramètres de fonction pour transmettre les données.

```
compteur = 0 # Variable globale

def incrementer():
    global compteur # Déclare l'utilisation de la
    variable globale
    compteur += 1

incrementer()
print(compteur) # Affiche 1
```

# Paramètres par Défaut

## Définition :

- Vous pouvez spécifier des valeurs par défaut pour les paramètres, permettant ainsi d'appeler la fonction sans fournir tous les arguments.

```
def salutation(nom, message="Bonjour"):  
    print(f"{message}, {nom} !")  
  
salutation("Alice") # Affiche "Bonjour, Alice !"  
salutation("Bob", "Salut") # Affiche "Salut, Bob  
!"
```

## Astuce

- Les paramètres avec des valeurs par défaut doivent être placés après les paramètres sans valeurs par défaut dans la définition de la fonction.



# Paramètres Variables : **\*args** et **\*\*kwargs**

**\*args** : Permet de passer un nombre variable d'arguments positionnels à une fonction.

```
def addition(*nombres):  
    total = 0  
    for nombre in nombres:  
        total += nombre  
    return total  
  
print(addition(1, 2, 3, 4)) # Affiche 10
```

**\*\*kwargs** : Permet de passer un nombre variable d'arguments nommés à une fonction.

```
def afficher_informations(**infos):  
    for cle, valeur in infos.items():  
        print(f"{cle} : {valeur}")  
  
afficher_informations(nom="Alice", age=30,  
ville="Paris")  
# Affiche :  
# nom : Alice  
# age : 30  
# ville : Paris
```

## À Retenir :

- **\*args** est utilisé pour les arguments positionnels multiples.
- **\*\*kwargs** est utilisé pour les arguments nommés multiples.
- Vous pouvez les combiner dans une fonction : `def ma_fonction(arg1, *args, **kwargs):`.

# Exercice

## Exercice : Création de Fonctions

1. **Créer une fonction `calculer_facture`** qui prend les paramètres suivants :
  - Le prix unitaire (`prix`), la quantité (`quantite`), et un taux de TVA (`taux_tva`) avec une valeur par défaut de 0.20.
  - Retourne le montant total TTC ( $\text{prix} \times \text{quantité} \times (1 + \text{taux\_tva})$ ).
2. **Créer une fonction `rechercher_client`** qui prend un nombre variable d'arguments nommés (`**kwargs`), comme `nom`, `email`, `telephone`.
  - Affiche les informations du client de manière structurée.
3. **Créer une fonction anonyme `lambda`** qui prend deux arguments et retourne leur produit.

# Mise en Pratique : Correction

## Fonction `calculer_facture` :

- Cette fonction prend trois paramètres : `prix`, `quantite`, et `taux_tva`, avec une valeur par défaut de 0.20 pour la TVA.
- Elle calcule le montant total TTC et le retourne.

## Fonction `rechercher_client` :

- Cette fonction utilise `**kwargs` pour accepter un nombre variable d'arguments nommés (comme `nom`, `email`, `telephone`).
- Elle affiche chaque information du client dans un format structuré.

## Fonction anonyme `lambda` :

- Une fonction `lambda` qui prend deux arguments et retourne leur produit.

```
def calculer_facture(prix, quantite, taux_tva=0.20):
    """Calcule le montant total TTC."""
    montant_ttc = prix * quantite * (1 + taux_tva)
    return montant_ttc

# Exemple d'utilisation
facture = calculer_facture(100, 5) # taux_tva par défaut à 0.20
print(f"Montant total TTC : {facture} €") # Affiche "Montant total TTC : 600.0 €"
```

```
def rechercher_client(**kwargs):
    """Affiche les informations du client."""
    print("\n--- Informations du client ---")
    for cle, valeur in kwargs.items():
        print(f"{cle.capitalize()} : {valeur}")

# Exemple d'utilisation
rechercher_client(nom="Alice", email="alice@example.com", telephone="123456789")
```

```
produit = lambda x, y: x * y

# Exemple d'utilisation
resultat = produit(4, 5)
print(f"Le produit des deux nombres est : {resultat}") # Affiche "Le produit des deux nombres est : 20"
```

# A retenir

**Définition des fonctions** : Utilisez `def` pour créer des blocs de code réutilisables.

**Retour des valeurs** : Les fonctions peuvent retourner une ou plusieurs valeurs à l'aide de `return`.

**Fonctions anonymes (`lambda`)** : Utile pour les opérations simples et rapides.

**Variables globales** : Utilisez `global` pour accéder et modifier les variables globales dans une fonction, mais préférez utiliser des paramètres pour passer des données.

**Paramètres par défaut** : Simplifient l'appel de fonctions en rendant certains arguments optionnels.

**`*args` et `**kwargs`** : Fournissent une flexibilité dans le nombre d'arguments que les fonctions peuvent accepter.

# Chapitre 7 : Gestion des fichiers

# Lecture et Écriture des Fichiers avec `open()`

**Fonction `open()`** : Utilisée pour ouvrir un fichier et effectuer des opérations (lecture, écriture, etc.).

- **Syntaxe** : `open(nom_du_fichier, mode)`
- Modes courants :
  - `'r'` : Lecture seule.
  - `'w'` : Écriture (écrase le contenu existant).
  - `'a'` : Ajout (ajoute du contenu sans effacer l'existant).
  - `'b'` : Mode binaire (lecture ou écriture de fichiers non-textes comme les images).
  - `'+'` : Lecture et écriture simultanées.

## Astuce

Le bloc `with` assure la fermeture automatique du fichier après l'opération.

```
# Écrire du texte dans un fichier
with open('exemple.txt', 'w', encoding='utf-8') as
fichier:
    fichier.write("Bonjour, monde!\n")
    fichier.write("Python facilite la gestion des
fichiers.")
```

```
# Lire le contenu d'un fichier
with open('exemple.txt', 'r', encoding='utf-8') as
fichier:
    contenu = fichier.read()
    print(contenu)
```

# Utilisation du Bloc **with**

## Pourquoi Utiliser le Bloc **with** ?

- **Gestion automatique des ressources** : Le bloc **with** garantit la fermeture automatique du fichier une fois que les opérations sont terminées, même si une erreur survient.
- **Code plus propre et sûr** : Évite d'oublier de fermer les fichiers manuellement, ce qui peut entraîner des fuites de ressources.

## Exemple Comparatif :

### Sans **with** (ouverture manuelle et fermeture)

```
fichier = open('exemple.txt', 'w')
try:
    fichier.write('Hello, world!')
finally:
    fichier.close()
```

**Inconvénients** : Vous devez explicitement appeler **fichier.close()** dans le bloc **finally** pour vous assurer que le fichier est toujours fermé, même en cas d'erreur.

### Avantages du Bloc **with** :

- **Facilité d'utilisation** : Moins de code et moins de risques d'erreurs.
- **Gestion des exceptions** : Assure la fermeture des fichiers en cas d'exception pendant l'écriture ou la lecture.
- **Bonne pratique** : Toujours utiliser le bloc **with** pour ouvrir et manipuler des fichiers pour une meilleure gestion des ressources.

### Avec **with** (ouverture automatique et fermeture)

```
with open('exemple.txt', 'w') as fichier:
    fichier.write('Hello, world!')
```

**Avantages** : Le fichier est automatiquement fermé dès que le bloc **with** se termine, sans avoir besoin d'appeler **close()** manuellement.

# Modes Avancés d'Ouverture de Fichiers

## Modes avancés :

- **'x'** : Crée un fichier. Génère une erreur (`FileExistsError`) si le fichier existe déjà.
- **'r+'** : Ouvre le fichier pour lecture et écriture. Le fichier doit exister.
- **'w+'** : Ouvre le fichier pour écriture et lecture. Écrase le contenu s'il existe déjà ou crée un nouveau fichier.
- **'a+'** : Ouvre le fichier pour ajout et lecture. Crée un fichier s'il n'existe pas.

```
# Créer un fichier avec 'x' (lève une erreur s'il existe déjà)
try:
    with open('nouveau_fichier.txt', 'x') as f:
        f.write('Ceci est un nouveau fichier.')
except FileExistsError:
    print("Le fichier existe déjà.")

# Lire et écrire dans un fichier existant avec 'r+'
with open('existant.txt', 'r+') as f:
    contenu = f.read()
    f.write('\nAjout de nouveau contenu.')

# Écrire et lire avec 'w+' (écrase le fichier existant)
with open('nouveau_fichier.txt', 'w+') as f:
    f.write('Nouveau contenu.')
    f.seek(0) # Revenir au début du fichier pour lire
    print(f.read())

# Ajouter et lire avec 'a+'
with open('nouveau_fichier.txt', 'a+') as f:
    f.write('\nContenu ajouté.')
    f.seek(0)
    print(f.read())
```



# Lecture Ligne par Ligne

## Méthodes de lecture :

- **readline()** : Lit une seule ligne à la fois. Utile pour traiter les fichiers ligne par ligne.
- **readlines()** : Lit toutes les lignes du fichier et retourne une liste contenant chaque ligne.

```
# Lecture ligne par ligne avec readline()
with open('existant.txt', 'r') as f:
    ligne = f.readline()
    while ligne:
        print(ligne.strip()) # Affiche chaque
                             ligne sans les espaces
        ligne = f.readline()

# Lecture de toutes les lignes avec readlines()
with open('existant.txt', 'r') as f:
    lignes = f.readlines()
    for ligne in lignes:
        print(ligne.strip())
```

# Gestion des Exceptions

Erreurs courantes lors de la manipulation des fichiers :

- **FileNotFoundError** : Le fichier spécifié n'existe pas.
- **PermissionError** : Permissions insuffisantes pour accéder ou modifier le fichier.
- **IsADirectoryError** : Le chemin spécifié est un répertoire, et non un fichier.

```
try:
    with open('fichier_inexistant.txt', 'r') as f:
        contenu = f.read()
except FileNotFoundError:
    print("Erreur : Le fichier n'existe pas.")
except PermissionError:
    print("Erreur : Permissions insuffisantes pour ouvrir ce fichier.")
except IsADirectoryError:
    print("Erreur : Le chemin spécifié est un répertoire, pas un fichier.")
except Exception as e:
    print(f"Erreur inattendue : {e}")
```

# Optimisation et Importation Sélective des Modules

**Objectif** : Optimiser les imports et utiliser différentes syntaxes pour des imports ciblés.

## Types d'Imports :

- **Importation complète** : Importe tout le module, ce qui peut être moins performant si seules quelques fonctions sont nécessaires.
- **Importation ciblée** : Permet d'importer uniquement les fonctions nécessaires, optimisant la mémoire et le temps de chargement.
- **Importation avec alias** : Renommer le module pour simplifier l'accès aux fonctions.

```
import math  
print(math.sqrt(16))  # Affiche 4.0
```

```
from math import sqrt, pi  
print(sqrt(16))  # Affiche 4.0  
print(pi)        # Affiche 3.141592653589793
```

```
import numpy as np  
import tensorflow as tf  
  
array = np.array([1, 2, 3, 4])  
print(array)
```

# Utilisation du Module **os**

Le module **os** permet d'interagir avec le système de fichiers.

## Fonctions utiles :

- **os.path.exists()** : Vérifie si un fichier ou un répertoire existe.
- **os.remove()** : Supprime un fichier.
- **os.rename()** : Renomme un fichier.
- **os.listdir()** : Liste les fichiers et répertoires dans un dossier.
- **os.makedirs()** : Crée des répertoires.

## Astuce

Le module **os** est essentiel pour interagir avec le système de fichiers et effectuer des opérations comme la gestion des dossiers et fichiers.

```
import os

# Vérifier si le fichier existe
if os.path.exists('exemple.txt'):
    # Renommer le fichier
    os.rename('exemple.txt', 'nouveau_nom.txt')
else:
    print("Le fichier n'existe pas.")
```

# Navigation dans le Système de Fichiers avec **os**

## Présentation des Fonctions :

- **os.getcwd()** : Retourne le chemin du répertoire de travail courant.
- **os.chdir(path)** : Change le répertoire courant vers le chemin spécifié (**path**).
- **os.mkdir(path)** : Crée un nouveau dossier avec le chemin spécifié (**path**).

## Utilisation :

- **os.getcwd()** est utile pour vérifier où se situe le script en cours d'exécution.
- **os.chdir()** permet de changer de répertoire pour travailler dans un autre dossier. Pratique pour organiser les fichiers sans avoir à spécifier des chemins absolus.
- **os.mkdir()** crée des dossiers pour organiser vos fichiers, si le répertoire n'existe pas déjà.

## Astuce

```
nouveau_repertoire = '/path/vers/nouveau/repertoire'
try:
    os.chdir(nouveau_repertoire)
    print(f"Répertoire changé à : {os.getcwd()}")
except FileNotFoundError:
    print("Erreur : Le répertoire spécifié n'existe pas.")

# Créer un nouveau dossier
dossier_a_creer = 'nouveau_dossier'
try:
    os.mkdir(dossier_a_creer)
    print(f"Dossier '{dossier_a_creer}' créé avec succès.")
except FileExistsError:
    print(f"Erreur : Le dossier '{dossier_a_creer}' existe déjà.")

# Aller dans le nouveau dossier
os.chdir(dossier_a_creer)
print(f"Répertoire actuel après changement : {os.getcwd()}")
```

# Utilisation du Module **shutil**

Le module **shutil** offre des fonctions pour la manipulation de fichiers et de répertoires.

## Fonctions utiles :

- **shutil.copy()** : Copie un fichier.
- **shutil.move()** : Déplace ou renomme un fichier.
- **shutil.rmtree()** : Supprime un répertoire et tout son contenu.

## Astuce

**shutil** est utilisé pour des opérations plus avancées, comme la copie et le déplacement de fichiers ou de répertoires entiers.

```
import shutil

# Copier un fichier
shutil.copy('nouveau_nom.txt',
'copie_nouveau_nom.txt')
```

# Opérations Avancées avec **shutil**

## Présentation des Fonctions :

- **shutil.copytree(src, dst)** : Copie un répertoire (**src**) et tout son contenu vers un autre emplacement (**dst**). Si **dst** n'existe pas, il sera créé.
- **shutil.move(src, dst)** : Déplace un fichier ou un répertoire (**src**) vers une nouvelle destination (**dst**). Peut aussi être utilisé pour renommer des fichiers.
- **shutil.rmtree(path)** : Supprime un dossier et tout son contenu. Utilisez avec prudence, car cette action est irréversible.

## Utilisation :

- **shutil.copytree()** est idéal pour créer des sauvegardes de dossiers ou copier des structures de répertoires.
- **shutil.move()** permet de réorganiser les fichiers et dossiers en les déplaçant dans des sous-répertoires.

```
# Déplacer un fichier vers un nouveau répertoire
dossier_nouveau = 'destination_dossier'
os.mkdir(dossier_nouveau)
fichier_a_deplacer = 'source_dossier/fichier.txt'
nouvelle_destination = f'{dossier_nouveau}/fichier.txt'
try:
    shutil.move(fichier_a_deplacer, nouvelle_destination)
    print(f"'{fichier_a_deplacer}' déplacé vers '{nouvelle_destination}'.")
except FileNotFoundError:
    print(f"Erreur : Le fichier '{fichier_a_deplacer}' n'existe pas.")

# Supprimer un dossier et son contenu
try:
    shutil.rmtree('source_dossier')
    print("'source_dossier' supprimé avec succès.")
except FileNotFoundError:
    print("Erreur : Le dossier 'source_dossier' n'existe pas.")
```

# Utilisation du Module **zlib**

## Compression des fichiers :

- Le module **zlib** permet de compresser et décompresser des données.
- Utile pour réduire la taille des fichiers texte.

```
import zlib

texte = b"Bonjour, ceci est une longue chaîne de
caractères à compresser."

# Compression
texte_comprime = zlib.compress(texte)
print(f"Texte compressé : {texte_comprime}")

# Décompression
texte_decomprime =
zlib.decompress(texte_comprime)
print(f"Texte décompressé :
{texte_decomprime.decode('utf-8')}")
```

## Astuce

**zlib** travaille avec des données binaires, donc utilisez des chaînes de bytes (**b"texte"**).

La compression peut réduire la taille des fichiers avant de les enregistrer.

.



# Utilisation du Module **csv**

- Le module `csv` est intégré dans la bibliothèque standard de Python.
- Il permet de lire et d'écrire dans des fichiers CSV (Comma-Separated Values).
- Idéal pour manipuler des données sous forme tabulaire (tableaux) avec Python.

## Astuces:

- Toujours utiliser le mode `'with'` pour ouvrir des fichiers (assure une fermeture correcte).
- Utiliser `newline=''` pour éviter des lignes vides sous Windows.
- Bien définir les en-têtes pour une utilisation plus claire avec `DictReader` et `DictWriter`.

```
import csv

#Retourne un objet qui itère sur les lignes du
fichier.

with open('fichier.csv', mode='r') as fichier:
    lecteur = csv.reader(fichier)
    for ligne in lecteur:
        print(ligne)

#Utilisé pour écrire des lignes dans un fichier.

with open('fichier.csv', mode='w', newline='') as
fichier:
    ecrivain = csv.writer(fichier)
    ecrivain.writerow([ 'Nom', 'Âge', 'Profession' ])
    ecrivain.writerow([ 'Alice', '30', 'Ingénieur' ])

with open('fichier.csv', mode='a', newline='') as
fichier:
    ecrivain = csv.writer(fichier)
    ecrivain.writerow([ 'Nom', 'Âge', 'Profession' ])
    ecrivain.writerow([ 'Alice', '30', 'Ingénieur' ])
```

# TP de validation des acquis : Jour 1

## Objectif : Création d'un CRM Basique

- Structurer un programme Python simple.
- Collecter et gérer des données clients.
- Sauvegarder les informations dans un fichier CSV.
- Utiliser des fonctions pour modulariser le code.
- Gérer les exceptions et les entrées utilisateur.

### Créer le programme CRM :

- Afficher un message de bienvenue pour introduire le CRM.
- Collecter les informations d'un client (nom, email, téléphone, nombre d'employés).
- Calculer les coûts du CRM en fonction du nombre d'employés (avec gestion des exceptions ).
- Le coût de base est de 20€ plus 5€ par employé.
- **Ajoute une TVA de 20% pour obtenir le montant total (TTC).**
- Sauvegarder les informations du client dans un fichier CSV.
- Afficher une liste des clients en lisant les données du fichier CSV.

### Décomposer le programme en plusieurs fonctions :

- **Fonction `bienvenue()`**: Afficher un message de bienvenue.
- **Fonction `creer_client()`**: Collecter les données d'un client et calculer le coût du CRM.
- **Fonction `sauvegarder_client_csv(client)`**: Sauvegarder les informations du client dans un fichier CSV.
- **Fonction `afficher_clients_csv()`**: Afficher les clients enregistrés dans le fichier CSV.

### Utiliser un menu interactif :

- Créer une boucle qui affiche un menu et attend le choix de l'utilisateur :
  - **Option 1**: Ajouter un client et sauvegarder ses informations dans un fichier CSV.
  - **Option 2**: Afficher tous les clients enregistrés dans le fichier CSV.
  - **Option 3**: Quitter le programme.

### Gérer les erreurs :

- Utiliser des blocs `try` et `except` pour gérer les erreurs, comme les entrées utilisateur incorrectes ou la division par zéro.

### Point d'entrée du programme :

- Utiliser `if __name__ == "__main__":` pour lancer l'application CRM.

## Résultat Attendu :

Les stagiaires doivent créer un programme qui :

- Collecte des informations sur un client et les sauvegarde dans un fichier CSV.
- Calcule automatiquement les coûts liés au nombre d'employés.
- Affiche tous les clients enregistrés dans le fichier CSV.
- Utilise un menu simple pour ajouter des clients, afficher la liste, ou quitter le programme.

# Correction du TP - Explication du Code

<https://github.com/DonJul34/pyt>

1. **Fonction `bienvenue()` :**
  - Affiche un message de bienvenue pour indiquer à l'utilisateur l'objectif du programme.
2. **Fonction `creer_client()` :**
  - Rassemble les informations du client (nom, email, téléphone).
  - Demande le nombre d'employés, et utilise un `try` et un `except` pour gérer les erreurs d'entrée (l'utilisateur doit entrer un nombre entier).
  - Calcule le coût du CRM :
    - Le coût de base est de 20€ plus 5€ par employé.
    - Ajoute une TVA de 20% pour obtenir le montant total (TTC).
  - Retourne les informations du client sous forme d'un dictionnaire.
3. **Fonction `sauvegarder_client_csv(client, fichier="clients.csv") :`**
  - Ouvre (ou crée) le fichier CSV spécifié en mode append ('a'), pour ajouter les informations du client à la suite du fichier existant.
  - Utilise `csv.writer` pour écrire les informations du client dans le fichier.
  - Affiche un message de confirmation de la sauvegarde réussie.
4. **Fonction `afficher_clients_csv(fichier="clients.csv") :`**
  - Ouvre le fichier CSV en mode lecture ('r').
  - Utilise `csv.reader` pour lire chaque ligne du fichier et affiche les informations du client formatées.
  - Gère l'exception `FileNotFoundError` si le fichier n'existe pas encore, en informant l'utilisateur.
5. **Point d'entrée (`if __name__ == "__main__":`) :**
  - Démarre le programme et affiche le message de bienvenue.
  - Initialise une boucle `while` qui affiche un menu et attend le choix de l'utilisateur :
    - **Option 1 :** Appelle `creer_client()` pour ajouter un client, puis `sauvegarder_client_csv()` pour stocker les données dans le fichier CSV.
    - **Option 2 :** Appelle `afficher_clients_csv()` pour afficher tous les clients enregistrés dans le fichier CSV.
    - **Option 3 :** Quitte le programme.
  - Si l'utilisateur entre une option non valide, il est invité à réessayer.

**Gestion des erreurs :** Utilise `try` et `except` pour gérer les erreurs d'entrée, notamment lors de la saisie du nombre d'employés.

**Sauvegarde des données :** Les informations des clients sont stockées dans un fichier CSV pour permettre une conservation permanente des données.

**Modularité :** Le code est découpé en plusieurs fonctions pour favoriser la lisibilité et la réutilisation.

**Menu interactif :** Offre une interface de ligne de commande pour interagir avec le CRM.

# A retenir

- **Fonction `open()`** : Utilisée pour ouvrir les fichiers dans différents modes (`'r'`, `'w'`, `'a'`).
- **Modules `os` et `shutil`** : Fournissent des outils puissants pour manipuler les fichiers et répertoires.
- **Module `zlib`** : Permet la compression et la décompression des données.
- **Meilleures pratiques** : Utilisez toujours le bloc `with` pour gérer les fichiers et assurez-vous de vérifier l'existence des fichiers avant de les manipuler.

# Chapitre 8 : Modules et Packages

---

# Qu'est-ce qu'un Module ?

Un **module** est un fichier contenant des définitions (fonctions, variables, classes) et des instructions Python.

Il permet de **réutiliser** et **organiser** le code en le regroupant dans des fichiers séparés.

Un module est simplement un fichier `.py`.

**Exemple** : Un fichier nommé `math_utils.py` contenant des fonctions mathématiques.

```
# math_utils.py

def addition(a, b):
    return a + b

def soustraction(a, b):
    return a - b
```

## Astuce

Utiliser des modules permet de séparer les fonctionnalités du programme en morceaux réutilisables.

Les noms de fichiers des modules doivent être uniques et respecter les conventions de nommage Python.

# Importer un Module

Pour utiliser les fonctions d'un module, il faut l'**importer** dans votre script.

## Syntaxe de base :

```
import math_utils

result =
math_utils.addition(5, 3)
print(result)  # Affiche 8
```

## Importer des éléments spécifiques d'un module :

- Utilisez **from** pour importer uniquement certaines fonctions ou variables.

```
from math_utils import addition

print(addition(10, 4))  # Affiche 14
```

## Astuce

Importer des modules permet d'accéder aux fonctions et variables définies dans d'autres fichiers.

Pour éviter les conflits de noms, il est possible d'utiliser des alias lors de l'import :

```
import math_utils as mu

print(mu.addition(2, 3))
```

# Importation Dynamique avec **importlib**

L'importation dynamique permet de charger des modules à l'exécution, facilitant la modularité et la flexibilité du code.

Le module **importlib** est utilisé pour importer un module au moment où le programme s'exécute.

- **Avantage** : Utile lorsque le nom du module à importer dépend d'une entrée utilisateur ou d'une configuration.
- **Inconvénient** : Utiliser `reload()` pour recharger un module. Il est parfois nécessaire de recharger un module déjà importé, surtout après des modifications.

```
import importlib

# Importer un module dynamiquement
nom_du_module = 'math_utils'
module_importe =
importlib.import_module(nom_du_module)

# Utiliser une fonction du module importé
from math_utils import addition    5)

print(addition(10, 4))  # Affiche 14
```

```
importlib.reload(module_importe)
```



# Organisation des Modules dans des Packages

Un **package** est un dossier contenant plusieurs modules et un fichier spécial `__init__.py`.

Le fichier `__init__.py` indique à Python que le dossier doit être traité comme un package.

**Structure d'un package :**

```
mon_package/  
├── __init__.py  
├── module1.py  
└── module2.py
```

**Création d'un Package :**

1. **Créez un dossier** nommé `mon_package`.
2. **Ajoutez un fichier** `__init__.py` (qui peut être vide) dans ce dossier.
3. **Ajoutez des modules** (`module1.py`, `module2.py`) dans ce dossier.

```
# Dossier : mon_package/  
# Fichier : mon_package/module1.py  
  
def saluer(nom):  
    print(f"Bonjour, {nom} !")  
  
# Fichier principal  
from mon_package import module1  
  
module1.saluer("Alice") # Affiche  
"Bonjour, Alice !"
```

# Organisation Avancée des Packages

## Sous-packages

- Un **sous-package** permet d'organiser les modules de manière hiérarchique à l'intérieur d'un package. Cela facilite la gestion des grandes applications en divisant les fonctionnalités en catégories logiques.

## Exemple de Structure de Package :

```
projet/  
├─ __init__.py  
├─ sous_package1/  
│   ├─ __init__.py  
│   ├─ module1.py  
│   └─ module2.py  
└─ sous_package2/  
    ├─ __init__.py  
    ├─ module3.py  
    └─ module4.py
```

Chaque dossier de sous-package (comme `sous_package1`) doit contenir un fichier `__init__.py` pour être reconnu comme un package.

```
from sous_package1.module1 import  
fonction_module1  
  
fonction_module1()
```

# Installation et Gestion des Packages avec **pip**

**pip** est l'outil de gestion des packages Python. Il permet d'installer, de mettre à jour et de désinstaller des packages depuis le Python Package Index (PyPI).

## Vérifier l'installation de **pip** (CMD, GIT BASH)

```
pip --version
```

## Installer un Package :

- Utilisez la commande **pip install** suivie du nom du package.

```
pip install requests
```

```
import requests

response =
requests.get('https://api.example.com/data')
print(response.text)
```

## Mettre à jour un Package :

- Utilisez **pip install --upgrade**

```
pip install --upgrade requests
```

## Désinstaller un Package :

- Utilisez **pip uninstall**.

```
pip uninstall requests
```

## Lister les Packages Installés :

- Pour voir tous les packages installés dans votre environnement, utilisez :

```
pip list
```

# Pourquoi Utiliser un Environnement Virtuel avec **venv** ?

## Isolation de l'Environnement :

- isoler les dépendances d'un projet Python des autres projets ou de l'installation globale de Python sur votre système.
- Problèmes de compatibilité avec des packages existants du système

## Gestion Facile des Dépendances :

- **venv** permet de gérer facilement les dépendances spécifiques à un projet. Vous pouvez installer, mettre à jour et supprimer des paquets sans affecter d'autres projets ou le système global.

## Portabilité du Projet :

- Utiliser un environnement virtuel rend un projet plus portable.

## Comment ?

### Création de l'Environnement Virtuel :

Commande :

```
python -m venv aiappenv
```

Objectif : Isoler l'environnement de développement pour éviter les conflits de paquets avec d'autres projets.

### Activation de l'Environnement Virtuel :

Sous Windows :

```
aiappenv\Scripts\activate
```

Git bash: 

```
source aiappenv/scripts/activate
```

Sous macOS/Linux :

```
source aiappenv/bin/activate
```

Objectif : Activer l'environnement virtuel pour installer les dépendances spécifiques au projet sans affecter le système global.

# Tests Unitaires pour les Modules

## Introduction aux Tests Unitaires

Les tests unitaires permettent de vérifier que chaque module ou fonction de votre code fonctionne correctement. Ils isolent des parties spécifiques du code et les testent individuellement pour s'assurer qu'elles produisent les résultats attendus.

## Pourquoi Tester ?

- **Réduire les Bugs** : Détecter rapidement les erreurs et assurer la fiabilité du code.
- **Assurer la Maintainabilité** : Faciliter l'évolution du code en vérifiant que chaque modification ne casse pas le fonctionnement existant.
- **Garantir les Spécifications** : S'assurer que chaque module répond aux spécifications attendues.

## Utilisation de Mock Data pour les Tests

Lorsqu'une fonction ou méthode dépend d'une ressource externe, comme une API ou une base de données, il est judicieux de "moquer" ces dépendances pour simuler leurs comportements sans réellement les appeler. Pour cela, le module `unittest.mock` propose la fonction `patch`.

```
import unittest

from mon_module import addition

class TestMonModule(unittest.TestCase):

    def test_addition(self):

        self.assertEqual(addition(2, 3),
                           5)

        self.assertEqual(addition(-1, 1),
                           0)

if __name__ == '__main__':
    unittest.main()
```

# Utilisation de Mock Data pour les Tests

## Utilisation de Mock Data pour les Tests

Lorsqu'une fonction ou méthode dépend d'une ressource externe, comme une API ou une base de données, il est judicieux de "moquer" ces dépendances pour simuler leurs comportements sans réellement les appeler. Pour cela, le module `unittest.mock` propose la fonction `patch`.

```
# mon_module.py
import requests

def get_data_from_api(url):
    response = requests.get(url)
    return response.json()
```

```
import unittest
from unittest.mock import patch
from mon_module import get_data_from_api

class TestMonModule(unittest.TestCase):
    @patch('mon_module.requests.get')
    def test_get_data_from_api(self, mock_get):
        # Définition de la réponse mockée
        mock_get.return_value.json.return_value = {"name": "Alice", "age": 30}

        # Appel de la fonction
        url = "http://api.example.com/data"
        result = get_data_from_api(url)

        # Vérification du résultat
        self.assertEqual(result, {"name": "Alice", "age": 30})
        mock_get.assert_called_once_with(url) # Vérifier que la requête a bien été appelée avec l'URL

if __name__ == '__main__':
    unittest.main()
```

# Différent package pour différentes utilisation

## Web Development

**Django:** Un framework web complet pour développer des applications web sécurisées et évolutives.

- **Caractéristiques** : ORM intégré, support de l'authentification, système de templates, et une interface d'administration.
- **Exemple d'utilisation** : Création de sites web complexes, backends pour des applications.

**Flask:** Un micro-framework minimaliste pour des applications web légères et modulaires.

- **Caractéristiques** : Très flexible, idéal pour les applications simples et les prototypes.
- **Exemple d'utilisation** : API REST, petites applications web.

**FastAPI:** Framework moderne pour créer des API web rapides, basées sur Python type hints.

- **Caractéristiques** : Haute performance, validation automatique des données.
- **Exemple d'utilisation** : Création d'API REST performantes.

## Web scraping:

**BeautifulSoup:** Pour analyser et extraire des données de fichiers HTML et XML.

- **Installation** : `pip install beautifulsoup4`
- **Caractéristiques** : Naviguer et rechercher dans le DOM des pages web.
- **Exemple d'utilisation** : Extraction de données d'une page web statique.

**Selenium:** Automatisation du navigateur pour interagir avec des pages web dynamiques (JavaScript).

- **Installation** : `pip install selenium`
- **Caractéristiques** : Contrôle automatisé des navigateurs pour tester ou scraper des sites nécessitant une interaction.
- **Exemple d'utilisation** : Remplissage automatique de formulaires, extraction de données sur des pages web nécessitant du JavaScript.

**Scrapy:** Un framework puissant pour le scraping et le crawling web.

- **Installation** : `pip install scrapy`
- **Caractéristiques** : Scraper des sites de manière rapide et structurée avec une configuration flexible.
- **Exemple d'utilisation** : Extraction à grande échelle des données, crawling de plusieurs pages.

# Différent package pour différentes utilisation

## Data Science et Analyse de Données

**NumPy:** Base de l'écosystème scientifique de Python pour les calculs numériques.

- **Installation :** `pip install numpy`
- **Caractéristiques :** Manipulation efficace des tableaux multidimensionnels et des matrices.
- **Exemple d'utilisation :** Calculs mathématiques, transformations de données.

**Pandas:** Pour la manipulation et l'analyse des données sous forme de DataFrames.

- **Installation :** `pip install pandas`
- **Caractéristiques :** Nettoyage, analyse, et manipulation des ensembles de données.
- **Exemple d'utilisation :** Chargement de données CSV, nettoyage et analyse des données.

**Matplotlib & Seaborn:** Bibliothèques pour la visualisation des données.

- **Installation :** `pip install matplotlib seaborn`
- **Caractéristiques :** Créer des graphiques statiques et des visualisations interactives.
- **Exemple d'utilisation :** Visualisation des distributions, graphiques en barres, et tracés linéaires.

## Développement de Scripts et Automatisation

**Argparse:** Pour créer des interfaces en ligne de commande.

- **Inclus dans la bibliothèque standard :** Pas besoin d'installation.
- **Caractéristiques :** Créez des commandes et des arguments pour vos scripts.
- **Exemple d'utilisation :** Ajouter des options et des arguments à un script Python.

**Click:** Un package pour créer des interfaces en ligne de commande de manière simple et rapide.

- **Caractéristiques :** Crée des commandes à plusieurs niveaux et des menus CLI (Command Line Interface).
- **Exemple d'utilisation :** Scripts CLI interactifs et modulaires.

**Schedule:** Pour automatiser et planifier des tâches.

- **Installation :** `pip install schedule`
- **Caractéristiques :** Permet de définir des tâches à exécuter périodiquement.
- **Exemple d'utilisation :** Automatisation des scripts pour s'exécuter à intervalles réguliers.



# Machine Learning et Intelligence Artificielle

## Machine Learning et Intelligence Artificielle

**TensorFlow:** Une bibliothèque open-source pour le machine learning et l'intelligence artificielle.

- **Caractéristiques** : Supporte la création de modèles complexes d'apprentissage profond (deep learning).
- **Exemple d'utilisation** : Réseaux de neurones pour la reconnaissance d'image, traitement du langage naturel.

**PyTorch:** Un framework pour le deep learning.

- **Caractéristiques** : Modèles de machine learning flexibles, idéal pour la recherche et l'expérimentation.
- **Exemple d'utilisation** : Modèles de traitement d'images, réseaux neuronaux récurrents.

**Keras:** API haut niveau pour créer et entraîner des modèles de deep learning.

- **Caractéristiques** : Facile à apprendre et à utiliser pour les débutants en deep learning.
- **Exemple d'utilisation** : Conception et entraînement de modèles de réseaux de neurones simples.

## Tests et Débogage

**Unittest:** Module intégré pour créer et exécuter des tests unitaires.

- **Inclus dans la bibliothèque standard** : Pas besoin d'installation.
- **Caractéristiques** : Testez les fonctions, méthodes, et modules individuellement.
- **Exemple d'utilisation** : Automatisation des tests de fonctionnalités.

**Pytest:** Un package pour simplifier l'écriture des tests unitaires.

- **Caractéristiques** : Syntaxe simple pour les tests, permet d'écrire des tests plus concis et modulaires.
- **Exemple d'utilisation** : Tests automatisés et rapports détaillés.

# Mise en Pratique

## Exercice : Création et Utilisation de Modules et Packages

1. Créer un module **operations.py** qui contient les fonctions **addition**, **soustraction**, **multiplication**, et **division**.
2. Créer un package **mon\_calculateur** :
  - Inclure **\_\_init\_\_.py** et le module **operations.py**.
3. Créer un script principal :
  - Importer et utiliser les fonctions du module **operations.py**.

# Mise en Pratique : Corrigé

## Étape 1 : Créer le Module `operations.py`

- Créez un fichier nommé `operations.py` et définissez les fonctions de base : `addition`, `soustraction`, `multiplication`, et `division`.

```
# operations.py

def addition(a, b):
    return a + b

def soustraction(a, b):
    return a - b

def multiplication(a, b):
    return a * b

def division(a, b):
    if b != 0:
        return a / b
    else:
        return "Erreur : Division par zéro."
```

# Mise en Pratique : Corrigé

## Étape 2 : Créer un Package `mon_calculateur`

1. Créez un dossier nommé `mon_calculateur`.
2. À l'intérieur de ce dossier, créez un fichier `__init__.py` (il peut être vide pour l'instant).
3. Placez le fichier `operations.py` dans ce dossier.

•

```
mon_calculateur/  
├── __init__.py  
└── operations.py
```

# Mise en Pratique : Corrigé

## Étape 3 : Créer le Script Principal

- Dans le répertoire principal, créez un script nommé `main.py`.
- Importez et utilisez les fonctions du module `operations.py` dans le package `mon_calculateur`.

```
# main.py

from mon_calculateur import operations

# Utilisation des fonctions du module
a, b = 10, 5
print(f"Addition : {operations.addition(a, b)}")          # Affiche 15
print(f"Soustraction : {operations.soustraction(a, b)}")  # Affiche 5
print(f"Multiplication : {operations.multiplication(a, b)}") # Affiche 50
print(f"Division : {operations.division(a, b)}")          # Affiche 2.0
```

# Mise en Pratique : Corrigé

## Étape 4 : Installer un Package Externe avec `pip`

- Installez le package `requests` pour effectuer des requêtes HTTP.

```
pip install requests
```

## Étape 5 : Utiliser `requests` pour Effectuer une Requête GET

- Ajoutez du code dans `main.py` pour utiliser `requests` et effectuer une requête GET sur une API publique.

```
import requests

# Effectuer une requête GET sur une API publique
response = requests.get('https://api.github.com')
if response.status_code == 200:
    print("Requête réussie !")
    print(response.json())
else:
    print(f"Échec de la requête. Statut : {response.status_code}")
```

# A retenir

**Modules** : Fichiers Python (.py) contenant des définitions et du code réutilisable.

**Packages** : Dossiers contenant des modules et un fichier `__init__.py`, permettant une organisation plus complexe.

**Importation** : Utilisez `import` pour accéder aux fonctions et variables d'un module.

**pip** : Outil de gestion des packages, essentiel pour installer et gérer des bibliothèques externes.

# **Chapitre 9 : Programmation Orientée Objet (POO)**



# Introduction à la Programmation Orientée Objet (POO)

## Qu'est-ce que la POO ?

- La POO est un paradigme de programmation basé sur la création d'**objets**, qui sont des instances de **classes**.
- Permet de modéliser des entités du monde réel (comme des voitures, des utilisateurs) en regroupant des **données** (attributs) et des **comportements** (méthodes).

## Concepts clés :

- **Classe** : Modèle ou plan pour créer des objets.
- **Objet** : Instance d'une classe. Chaque objet possède ses propres données.
- **Méthode** : Fonction définie à l'intérieur d'une classe.
- **Attribut** : Variable stockée dans un objet.

# Création de Classes et Instanciation d'Objets

## Définition d'une Classe :

- Utilisez le mot-clé `class` suivi du nom de la classe. Les noms de classes commencent généralement par une majuscule.
- Les méthodes d'une classe ont `self` comme premier paramètre pour accéder aux attributs de l'objet.

```
class Personne:
    # Méthode initiale (constructeur)
    def __init__(self, nom, age):
        self.nom = nom # Attribut d'instance
        self.age = age

    # Méthode de la classe
    def se_presenter(self):
        print(f"Bonjour, je m'appelle {self.nom}
et j'ai {self.age} ans.")
```

## Instanciation d'un Objet :

- Pour créer un objet, utilisez la syntaxe  
`nom_de_l_objet =`  
`NomDeLaClasse(arguments).`

```
personnel = Personne("Alice", 30)
personnel.se_presenter() # Affiche : Bonjour, je
m'appelle Alice et j'ai 30 ans.
```

# Les Constructeurs (`__init__`)

- Le constructeur `__init__` est une méthode spéciale appelée automatiquement lors de l'instanciation d'un objet.
- Permet d'initialiser les attributs d'une classe.

## A retenir:

`__init__` initialise les attributs d'un objet lors de sa création.

Les paramètres passés à `__init__` sont les données nécessaires pour créer un objet de cette classe.

```
class Voiture:
    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele
        self.vitesse = 0  # Initialisation d'un
attribut par défaut

    def accelerer(self, montant):
        self.vitesse += montant
        print(f"La voiture {self.marque}
{self.modele} accélère à {self.vitesse} km/h.")

ma_voiture = Voiture("Tesla", "Model S")
ma_voiture.accelerer(50)  # Affiche : La voiture
Tesla Model S accélère à 50 km/h.
```

# Méthodes Magiques (Dunder Methods)

Les méthodes magiques (ou méthodes dunder) sont des méthodes spéciales permettant de définir des comportements pour des opérations intégrées, comme l'addition, la représentation en chaîne, etc.

Exemples de méthodes magiques :

- `__str__` : Définit le comportement de `print()` pour l'objet.
- `__repr__` : Fournit une représentation officielle de l'objet, utile pour le débogage.
- `__eq__` : Permet de définir la logique de comparaison entre deux objets (`==`).

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other):
        return self.x == other.x and self.y ==
other.y

# Utilisation
p1 = Point(2, 3)
p2 = Point(2, 3)
print(p1)    # Affiche : Point(2, 3)
print(p1 == p2)  # Affiche : True
```

# Héritage Simple

## Qu'est-ce que l'héritage ?

- Permet de créer une **nouvelle classe** à partir d'une classe existante.
- La classe **enfant** hérite des attributs et méthodes de la classe **parent**.

## A retenir:

L'héritage permet de **réutiliser** et **étendre** les fonctionnalités des classes existantes.

Les classes enfants peuvent **surcharger** les méthodes de la classe parent.

```
class Animal:
    def __init__(self, nom):
        self.nom = nom

    def parler(self):
        print("Cet animal fait un bruit.")

class Chien(Animal): # Hérite de la classe Animal
    def parler(self):
        print(f"{self.nom} aboie.")

mon_chien = Chien("Rex")
mon_chien.parler() # Affiche : Rex aboie.
```

# Héritage Multiple

## Qu'est-ce que l'héritage multiple ?

- Permet à une classe d'hériter de **plusieurs classes parent**.
- Syntaxe : `class NouvelleClasse(ClasseParent1, ClasseParent2):`.

## A retenir:

L'héritage multiple permet à une classe de combiner des fonctionnalités provenant de plusieurs classes.

Peut poser des problèmes de complexité, surtout en cas de conflit entre les méthodes héritées (résolu par l'**ordre de résolution des méthodes**).

```
class Volant:
    def voler(self):
        print("Cet objet peut voler.")

class Nageant:
    def nager(self):
        print("Cet objet peut nager.")

class Canard(Volant, Nageant): # Hérite des deux classes
    def parler(self):
        print("Coin! Coin!")

mon_canard = Canard()
mon_canard.voler() # Affiche : Cet objet peut voler.
mon_canard.nager() # Affiche : Cet objet peut nager.
mon_canard.parler() # Affiche : Coin! Coin!
```

# Polymorphisme

## Polymorphisme

- Le polymorphisme permet d'utiliser une interface commune pour des objets de différents types. C'est un concept clé de la POO.
- **Méthodes Polymorphiques** : Elles peuvent être redéfinies dans les classes enfants pour avoir des comportements spécifiques.

```
class Animal:
    def parler(self):
        pass

class Chien(Animal):
    def parler(self):
        return "Aboie"

class Chat(Animal):
    def parler(self):
        return "Miaule"

# Utilisation polymorphique
animaux = [Chien(), Chat()]

for animal in animaux:
    print(animal.parler()) # Appelle la
    méthode spécifique à chaque classe
```

# Méthodes et Attributs de Classe

## Méthodes de Classe :

- Utilisent le décorateur `@classmethod` et le mot-clé `cls` pour accéder aux attributs de la classe.

## Attributs de Classe :

- Sont partagés par toutes les instances de la classe.

Les **méthodes de classe** affectent les attributs de la classe et non des instances spécifiques.

Les **attributs de classe** sont partagés par toutes les instances, contrairement aux attributs d'instance.

```
class CompteBancaire:
    taux_interet = 0.05 # Attribut de classe

    def __init__(self, titulaire, solde):
        self.titulaire = titulaire
        self.solde = solde

    @classmethod
    def changer_taux_interet(cls, nouveau_taux):
        cls.taux_interet = nouveau_taux

    def afficher_solde(self):
        print(f"Le solde de {self.titulaire} est de {self.solde} €
avec un taux d'intérêt de {self.taux_interet * 100}%.")

compte1 = CompteBancaire("Alice", 1000)
compte2 = CompteBancaire("Bob", 2000)

compte1.afficher_solde() # Affiche : Le solde de Alice est de 1000
€ avec un taux d'intérêt de 5.0%.
CompteBancaire.changer_taux_interet(0.06)
compte2.afficher_solde() # Affiche : Le solde de Bob est de 2000 €
avec un taux d'intérêt de 6.0%.
```



# Les Propriétés (Getters et Setters)

## Les Propriétés en Python

- Les propriétés permettent de contrôler l'accès aux attributs d'un objet tout en encapsulant la logique pour les obtenir et les modifier.

### Utiliser les Getters et Setters avec `@property`

- `@property` : Transforme une méthode en un attribut, permettant de lire sa valeur comme s'il s'agissait d'un attribut.
- `@nom_attribut.setter` : Permet de définir une méthode pour modifier l'attribut, ajoutant ainsi des vérifications ou des transformations.

```
class Personne:
    def __init__(self, nom):
        self._nom = nom

    @property
    def nom(self):
        return self._nom

    @nom.setter
    def nom(self, valeur):
        if isinstance(valeur, str) and valeur:
            self._nom = valeur
        else:
            raise ValueError("Le nom doit être une chaîne de caractères non vide.")

# Utilisation
p = Personne("Alice")
print(p.nom)  # Accès au nom
p.nom = "Bob"  # Modification du nom
```

# Méthodes spéciales `__getattr__` et `__getattribute__`

## `__getattr__`

pour gérer l'accès aux attributs qui **n'existent pas** (c'est-à-dire qui ne sont pas définis dans l'instance ou la classe).

Elle est appelée uniquement si l'attribut spécifié est introuvable.

```
class Personne:
    def __init__(self, nom):
        self.nom = nom

    def __getattr__(self, nom_attribut):
        return f"L'attribut '{nom_attribut}' n'existe pas."

# Exemple
personne = Personne("Alice")
print(personne.nom) # Affiche "Alice"
print(personne.age) # Affiche "L'attribut 'age' n'existe pas."
```

## `__getattribute__`

Appelée pour **tous les accès** aux attributs de l'instance, même ceux qui existent.

```
class Personne:
    def __init__(self, nom):
        self.nom = nom

    def __getattribute__(self, nom_attribut):
        print(f"Accès à l'attribut '{nom_attribut}'")
        return super().__getattribute__(nom_attribut)

# Exemple
personne = Personne("Alice")
print(personne.nom) # Affiche "Accès à l'attribut 'nom'" puis "Alice"
```

```
class Personne:
    def __init__(self, nom):
        self.nom = nom # Cela appelle
        __setattr__('nom', nom)

    def __setattr__(self, nom_attribut, valeur):
        print(f"Modification de '{nom_attribut}' à
        '{valeur}'")

        if nom_attribut == "nom" and not valeur:
            raise ValueError("Le nom ne peut pas
            être vide.")

        super().__setattr__(nom_attribut, valeur)

# Exemple
personne = Personne("Alice") # Affiche
"Modification de 'nom' à 'Alice'"
personne.nom = "Bob" # Affiche
"Modification de 'nom' à 'Bob'"

```

# Méthode `__setattr__` pour les setters personnalisés

`setattr()` et `getattr()` sont des fonctions Python intégrées pour accéder et modifier dynamiquement les attributs d'un objet sans appeler explicitement les méthodes `getter` ou `setter`.

- `getattr(objet, nom_attribut)` : Renvoie la valeur de l'attribut `nom_attribut` de `objet`, ou une valeur par défaut si l'attribut n'existe pas.
- `setattr(objet, nom_attribut, valeur)` : Affecte `valeur` à l'attribut `nom_attribut` de `objet`.

```
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

personne = Personne("Alice", 30)
print(getattr(personne, "nom")) # Affiche "Alice"
setattr(personne, "age", 31)    # Modifie l'âge
print(personne.age)             # Affiche 31
```

# Composition d'Objets

## Composition d'Objets

- La composition consiste à intégrer des objets d'une classe à l'intérieur d'une autre pour former des structures complexes.
- Permet de réutiliser des classes existantes plutôt que d'hériter d'une classe de base.

```
class Moteur:
    def demarrer(self):
        print("Moteur démarré")

class Voiture:
    def __init__(self, marque):
        self.marque = marque
        self.moteur = Moteur() # Composition de
                                # l'objet Moteur

    def demarrer_voiture(self):
        self.moteur.demarrer()
        print(f"La voiture {self.marque} démarre.")

# Utilisation
voiture = Voiture("Toyota")
voiture.demarrer_voiture()
```

# Exercice : Gestion d'une Bibliothèque

Vous allez créer un programme pour gérer une bibliothèque. Ce programme permettra d'ajouter, de supprimer et de lister des livres avec des détails spécifiques. Voici les étapes :

## 1. Définir les classes principales :

- **Livre (Book)** : Cette classe représentera un livre avec des attributs comme le titre, l'auteur, l'année de publication et le genre.
- **Bibliothèque (Library)** : Cette classe représentera la bibliothèque contenant une collection de livres et des méthodes pour gérer cette collection.

## 2. Classe Livre (Book) :

- Créez un constructeur `__init__` pour initialiser les attributs `titre`, `auteur`, `annee`, et `genre`.
- Ajoutez une méthode `__str__` pour afficher les détails du livre sous forme de chaîne de caractères.

## 3. Classe Bibliothèque (Library) :

- Utilisez une liste pour stocker les livres dans la bibliothèque.
- Ajoutez des méthodes pour :
  - **Ajouter un livre** (`ajouter_livre`) : Prend un objet `Livre` et l'ajoute à la collection.
  - **Supprimer un livre** (`supprimer_livre`) : Supprime un livre en fonction de son titre.
  - **Lister les livres** (`lister_livres`) : Affiche tous les livres présents dans la bibliothèque.
  - **Rechercher par auteur** (`rechercher_par_auteur`) : Retourne une liste de livres écrits par un auteur donné.

## 4. Tester le Programme :

- Créez une instance de `Bibliothèque`.
- Ajoutez quelques livres, supprimez-en un et listez les livres restants.

# Exercice : Gestion d'une Bibliothèque

```

    if resultats:
        for livre in resultats:
            print(livre)
    else:
        print(f"Aucun livre trouvé pour l'auteur :{auteur}")

# Utilisation des classes
# Création de quelques livres
livre1 = Livre("Le Petit Prince", "Antoine de Saint-Exupéry", 1943, "Conte")
livre2 = Livre("1984", "George Orwell", 1949, "Dystopie")
livre3 = Livre("Les Misérables", "Victor Hugo", 1862, "Roman")

# Ajout des livres à la bibliothèque
bibliotheque.ajouter_livre(livre1)
bibliotheque.ajouter_livre(livre2)
bibliotheque.ajouter_livre(livre3)

# Lister les livres
print("\nListe des livres dans la bibliothèque :)")
bibliotheque.lister_livres()

# Recherche par auteur
print("\nRecherche des livres par George Orwell :)")
bibliotheque.rechercher_par_auteur("George Orwell")

# Suppression d'un livre
print("\nSuppression d'un livre (1984) :)")
bibliotheque.supprimer_livre(1984)

# Lister les livres après suppression
print("\nListe des livres après suppression :)")
bibliotheque.lister_livres()

```

```

# Classe Livre
class Livre:
    def __init__(self, titre, auteur, annee, genre):
        self.titre = titre
        self.auteur = auteur
        self.annee = annee
        self.genre = genre

    def __str__(self):
        return f"(self.titre), par {self.auteur} ({self.annee}) - Genre : {self.genre}"

# Classe Bibliothèque
class Bibliotheque:
    def __init__(self):
        self.livres = []

    def ajouter_livre(self, livre):
        self.livres.append(livre)
        print(f"Livre ajouté : {livre}")

    def supprimer_livre(self, titre):
        for livre in self.livres:
            if livre.titre == titre:
                self.livres.remove(livre)
                print(f"Livre supprimé : {livre}")
                return
        print("Livre non trouvé.")

    def lister_livres(self):
        if not self.livres:
            print("Aucun livre dans la bibliothèque.")
        else:
            for livre in self.livres:
                print(livre)

    def rechercher_par_auteur(self, auteur):
        resultats = [livre for livre in self.livres if livre.auteur == auteur]
        bibliotheque = Bibliotheque()

```

# À Retenir

- **Classes et Objets** : Les classes définissent la structure des objets. Les objets sont des instances de ces classes.
- **Constructeur (`__init__`)** : Initialise les attributs d'un objet lors de sa création.
- **Héritage** : Permet de réutiliser et d'étendre des classes existantes.
- **Méthodes et Attributs de Classe** : Les méthodes de classe utilisent le décorateur `@classmethod`. Les attributs de classe sont partagés par toutes les instances.



# TP de validation des acquis

## Étape 1 : Structurer le Projet

Créez les fichiers suivants pour organiser votre projet CRM :

1. `main.py` : Le script principal qui exécutera le programme.
2. `client.py` : Contiendra la classe `Client` pour gérer les clients.
3. `admin.py` : Contiendra les fonctions pour la gestion des clients (ajout, modification, suppression).
4. `statistiques.py` : Pour les fonctions d'analyse et de statistiques des clients.

## Étape 2 : Créer la Classe `Client` dans `client.py`

Créez une classe `Client` qui inclut les attributs et méthodes suivants :

- **Attributs** : `nom`, `email`, `telephone`, `categorie`, `nombre_employes`, `prix_par_employe`, `cout_ht`, et `cout_ttc`.
- **Méthodes** :
  - `__init__()` : Le constructeur initialise tous les attributs.
  - `afficher_infos()` : Retourne les informations du client de manière structurée.
  - `calculer_chiffre_affaire()` : Calcule et retourne le chiffre d'affaires TTC généré par ce client.

## Étape 3 : Gestion des Clients dans `admin.py`

Créez des fonctions pour gérer les clients :

- `creer_client()` : Demande à l'utilisateur de saisir les informations d'un client et renvoie une instance de la classe `Client`.
- `modifier_client(client)` : Permet de modifier les informations d'un client existant.
- `supprimer_client(clients, nom_client)` : Supprime un client de la liste des clients.

## Étape 4 : Ajouter le Menu Principal dans `main.py`

Créez le menu principal dans `main.py` pour naviguer entre les différentes actions :

- **Options** :
  - Ajouter un client
  - Afficher les clients
  - Modifier un client
  - Supprimer un client
  - Afficher les statistiques (sera implémenté à l'étape suivante)
  - Quitter le programme
- **Point d'entrée** : Utilisez `if __name__ == "__main__":` pour exécuter le programme.

# TP de validation des acquis

## Étape 5 : Statistiques dans `statistiques.py`

- Implémentez les fonctions suivantes dans `statistiques.py` :
  - `calculer_chiffre_affaire_total(clients)` : Calcule et retourne le chiffre d'affaires total généré par tous les clients.
  - `clients_par_categorie(clients)` : Compte et retourne le nombre de clients dans chaque catégorie.
  - `afficher_statistiques(clients)` : Affiche le nombre total de clients, le chiffre d'affaires total, et le nombre de clients par catégorie.

## Étape 6 : Gestion des Clients avec Fichier CSV

1. Ajoutez les fonctions suivantes dans `admin.py` :
  - `sauvegarder_client_csv(client, fichier="clients.csv")` : Sauvegarde un client dans un fichier CSV.
  - `afficher_clients_csv(fichier="clients.csv")` : Affiche les clients enregistrés dans un fichier CSV.
2. Intégrez ces fonctionnalités dans `main.py` :
  - Lorsque l'utilisateur ajoute un client, sauvegardez-le dans le fichier CSV.
  - L'option 2 du menu doit afficher les clients depuis le fichier CSV.

## Étape 7 : Mise en Œuvre Finale et Tests

- Validez toutes les fonctionnalités :
  - Ajouter, modifier, supprimer, afficher les clients.
  - Afficher les statistiques.
  - Sauvegarder et charger les clients depuis le fichier CSV.
- Réalisez une série de tests pour s'assurer que le programme fonctionne comme prévu et qu'il gère correctement les erreurs (entrées invalides, divisions par zéro, etc.).

# Correction du TP - Explication du Code

<https://github.com/DonJul34/pyt>

## Organisation du Projet et Modularisation

Le projet CRM est bien structuré en plusieurs modules :

- **client.py** : Gère la classe `Client` qui encapsule les attributs (nom, email, téléphone, etc.) et méthodes (affichage des informations, calcul du chiffre d'affaires) liés à chaque client.
- **admin.py** : Contient les fonctions pour créer, modifier, et supprimer des clients, centralisant ainsi la gestion des clients.
- **statistiques.py** : Calcule les statistiques globales, comme le chiffre d'affaires total et le nombre de clients par catégorie.
- **main.py** : Point d'entrée du programme, avec un menu utilisateur pour accéder aux différentes fonctionnalités.

Cette structure modulaire respecte le principe de **séparation des responsabilités**, ce qui facilite la maintenance et l'extensibilité du code.

## Gestion des Clients avec la POO

- La classe `Client` (dans `client.py`) est utilisée pour modéliser les clients. Elle inclut un constructeur (`__init__`) pour initialiser les attributs et des méthodes (`afficher_infos`, `calculer_chiffre_affaire`) pour manipuler les données des clients.
- Les fonctions dans `admin.py` utilisent cette classe pour créer, modifier, et supprimer des clients, offrant une encapsulation des données et des méthodes relatives aux clients.

**Avantages** : L'utilisation de la POO permet de regrouper les données et les comportements des clients, rendant le code plus intuitif et facile à gérer.

# Correction du TP - Explication du Code

<https://github.com/DonJul34/pyt>

## Fonctionnalités Principales dans `main.py`

- **Menu Interactif** : Propose un menu utilisateur basé sur une boucle `while` et des conditions pour naviguer entre les différentes fonctionnalités (ajout, affichage, modification, suppression, statistiques).
- **Appel des Fonctions** : En fonction du choix de l'utilisateur, les fonctions appropriées sont appelées pour manipuler la liste des clients ou afficher les statistiques.

**Avantages** : Cette approche rend l'application intuitive et permet une interaction en temps réel avec l'utilisateur.

## Gestion des Statistiques

- Le module `statistiques.py` fournit des statistiques essentielles telles que le chiffre d'affaires total (`calculer_chiffre_affaire_total`) et le nombre de clients par catégorie (`clients_par_categorie`).
- La fonction `afficher_statistiques` centralise ces données et les affiche à l'utilisateur.

**Avantages** : Offrir un aperçu global des clients via des statistiques renforce l'utilité de l'application en donnant des informations clés à l'utilisateur.

## Stockage des Données dans un Fichier CSV

- Les clients sont sauvegardés dans un fichier CSV pour assurer la persistance des données.
- Les fonctions de gestion des clients (`sauvegarder_client_csv`, `afficher_clients_csv`) facilitent l'écriture et la lecture des données à partir du fichier CSV.

**Avantages** : La persistance des données dans un fichier CSV rend le système utilisable au-delà de la session en cours, permettant de stocker et de récupérer les informations des clients.

# Chapitre 10 : Développement

---

## Avancé

# Connexion à une Base de Données SQL

**Introduction** : Les bases de données SQL (comme SQLite, MySQL, PostgreSQL) sont largement utilisées pour stocker et gérer des données dans les applications.

## Pourquoi se connecter à une base de données ?

- Permet la **persistance** des données, c'est-à-dire de stocker les données de manière permanente.
- Facilite les **opérations complexes** (recherches, tris, filtres) sur de grandes quantités d'informations.

## Bibliothèque Utilisée : **sqlite3**

- **sqlite3** est une bibliothèque intégrée dans Python pour manipuler une base de données SQLite.

## Créer une connexion :

```
import sqlite3

# Connexion à une base de données (ou création si
elle n'existe pas)
connexion = sqlite3.connect('clients.db')
```

# Création d'une Table dans SQLite

Une fois connecté, vous pouvez créer des tables pour organiser vos données.

```
curseur = connexion.cursor()
curseur.execute('''
    CREATE TABLE IF NOT EXISTS clients (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        nom TEXT NOT NULL,
        email TEXT NOT NULL,
        telephone TEXT,
        categorie TEXT,
        nombre_employes INTEGER
    )
''')
connexion.commit()
```

## Insertion et Requêtes SQL

- Insertion

```
curseur.execute('''
    INSERT INTO clients (nom, email, telephone, categorie, nombre_employes)
    VALUES (?, ?, ?, ?, ?)
''', ("Alice", "alice@example.com", "1234567890", "Web", 10))
connexion.commit()
```

# Création d'une Table dans SQLite

## Insertion et Requêtes SQL

- Insertion

```
curseur.execute('''
    INSERT INTO clients (nom, email, telephone, categorie, nombre_employes)
    VALUES (?, ?, ?, ?, ?)
''', ("Alice", "alice@example.com", "1234567890", "Web", 10))
connexion.commit()
```

## Requêtes :

```
curseur.execute('SELECT * FROM clients')
resultats = curseur.fetchall()
for row in resultats:
    print(row)
```



# Amélioration des Opérations CRUD avec des Requêtes Paramétrées

## Pourquoi Utiliser des Requêtes Paramétrées ?

- Éviter les attaques par injection SQL.
- Séparer les données des commandes SQL, ce qui rend les requêtes plus sûres et flexibles.

```
import sqlite3

def ajouter_client(nom, email):
    connection = sqlite3.connect('crm.db')
    cursor = connection.cursor()
    # Utilisation de paramètres pour sécuriser la requête
    cursor.execute("INSERT INTO clients (nom, email) VALUES (?, ?)", (nom,
email))
    connection.commit()
    connection.close()

ajouter_client("Alice", "alice@example.com")
```

# Utilisation des Context Managers avec les Bases de Données

## Utilisation des Context Managers pour la Gestion de Connexion

- Garantir la fermeture de la connexion même en cas d'erreur.

```
import sqlite3

def obtenir_clients():
    with sqlite3.connect('crm.db') as connection:
        cursor = connection.cursor()
        cursor.execute("SELECT * FROM clients")
        clients = cursor.fetchall()
    return clients
```

**Avantage** : Le bloc `with` assure que `connection.close()` est appelé automatiquement, ce qui évite les fuites de connexions.

# Programmation d'Interfaces Graphiques avec Tkinter

## Introduction à Tkinter :

- **Tkinter** est la bibliothèque intégrée à Python pour créer des **interfaces graphiques (GUI)**.
- Permet de créer des fenêtres, des boutons, des formulaires, et bien plus.

## Création d'une Fenêtre de Base

```
import tkinter as tk

# Créer une fenêtre
fenetre = tk.Tk()
fenetre.title("Application CRM")
fenetre.geometry("400x300")

# Lancer la boucle principale
fenetre.mainloop()
```

# Programmation d'Interfaces Graphiques avec Tkinter

## Ajout de Widgets (Éléments d'Interface)

**Label** : Affiche du texte.

```
label = tk.Label(fenetre, text="Bienvenue dans le  
CRM")  
label.pack()
```

**Entrée de texte:**

```
entree_nom = tk.Entry(fenetre)  
entree_nom.pack()
```

**Bouton :**

```
def bouton_action():  
    print(f"Nom entré : {entree_nom.get()}")  
  
bouton = tk.Button(fenetre, text="Valider",  
command=bouton_action)  
bouton.pack()
```

## Structure des Programmes Tkinter

- **Boucle principale (`mainloop()`)** : Gère l'affichage et les événements de l'interface.
- **Pack, Grid, Place** : Méthodes pour organiser les widgets dans la fenêtre.

# Améliorer l'Esthétique de l'Interface avec **ttk**

La bibliothèque **ttk** (Themed Tkinter Widgets) propose des widgets améliorés avec des thèmes modernes et plus esthétiques.

Utiliser **ttk** pour remplacer les widgets Tkinter de base tels que **Button**, **Label**, et **Entry**.

## Personnalisation des Thèmes avec **ttk**

- Vous pouvez personnaliser le thème global avec **style**.

```
style = ttk.Style()
style.theme_use('clam') # Choisir parmi les
thèmes disponibles (clam, default, etc.)
```

```
import tkinter as tk
from tkinter import ttk

# Création de la fenêtre principale
fenetre = tk.Tk()
fenetre.title("Application CRM")

# Création d'un bouton `ttk`
bouton = ttk.Button(fenetre, text="Ajouter
Client")
bouton.pack()

# Exécution de la boucle principale
fenetre.mainloop()
```

# Disposition et Gestion des Grilles dans Tkinter

## Disposition des Éléments avec `grid()` et `pack()`

- **`pack()`** : Place les éléments dans la fenêtre de manière séquentielle.
  - Utilisation : `pack(side="left", fill="x", expand=True)`
- **`grid(row, column)`** : Place les éléments dans une grille (plus flexible pour des interfaces complexes).

```
label_nom = tk.Label(fenetre, text="Nom:")
label_nom.grid(row=0, column=0, padx=5, pady=5)

entry_nom = tk.Entry(fenetre)
entry_nom.grid(row=0, column=1, padx=5, pady=5)
```

**Astuce** : Privilégiez `grid()` pour des interfaces complexes nécessitant une disposition précise, et `pack()` pour une structure simple.

# Débogage d'Interfaces Tkinter

- Utiliser `print()` et `messagebox` : Affichez les valeurs des variables et les erreurs dans des fenêtres contextuelles pour déboguer l'interface graphique.
- Module `traceback` : Capturez les erreurs pour un débogage efficace.

**Astuce** : Utilisez `traceback.print_exc()` pour voir l'erreur complète dans la console, et `messagebox` pour alerter l'utilisateur.

```
import tkinter as tk
from tkinter import messagebox
import traceback

def ajouter_client():
    try:
        # Code pour ajouter un client
        raise ValueError("Erreur simulée pour démonstration")
    except Exception as e:
        print(f"Erreur : {e}")
        traceback.print_exc()
        messagebox.showerror("Erreur", "Une erreur est survenue lors de l'ajout du client.")

root = tk.Tk()
button = tk.Button(root, text="Ajouter Client",
                    command=ajouter_client)
button.pack()

root.mainloop()
```

# Méthodes pour Charger les Données avec **Treeview** dans Tkinter

## Affichage des Données dans un **Treeview**

- Récupérer les données de la base de données et les insérer dans le **Treeview**.

```
def afficher_clients(treeview):  
    clients = obtenir_clients()  
    # Effacer toutes les entrées actuelles  
    for item in treeview.get_children():  
        treeview.delete(item)  
    # Insérer de nouvelles données  
    for client in clients:  
        treeview.insert('', 'end',  
            values=client)  
    bouton_actualiser = ttk.Button(fenetre,  
        text="Actualiser", command=lambda:  
            afficher_clients(treeview))  
    bouton_actualiser.pack()
```



# Organisation de l'Interface avec les **Frame**

- Un **Frame** agit comme un conteneur pour les widgets, permettant de mieux organiser l'interface graphique.
- Utilisez plusieurs **Frames** pour diviser l'interface en sections distinctes.

```
import tkinter as tk

root = tk.Tk()

# Création des frames
top_frame = tk.Frame(root)
top_frame.pack(side=tk.TOP, fill=tk.X)

bottom_frame = tk.Frame(root)
bottom_frame.pack(side=tk.BOTTOM, fill=tk.X)

# Ajout de widgets dans les frames
label = tk.Label(top_frame, text="Bienvenue dans l'application CRM")
label.pack()

button = tk.Button(bottom_frame, text="Ajouter Client")
button.pack()

root.mainloop()
```

# Gestion des Événements dans Tkinter

Les widgets Tkinter peuvent répondre à des événements tels que les clics de souris, les pressions de touches et autres interactions utilisateur.

**Méthode `bind()`** : Permet d'attacher un événement spécifique à un widget.

```
import tkinter as tk

def on_click(event):
    print(f"Clic détecté à la position : {event.x}, {event.y}")

root = tk.Tk()
canvas = tk.Canvas(root, width=200, height=200)
canvas.bind("<Button-1>", on_click) # Lier l'événement clic gauche de la souris
canvas.pack()

root.mainloop()
```

# Interaction avec la Base de Données via l'Interface

**Connexion Tkinter + SQLite** : Créez une interface pour ajouter et afficher des clients dans la base de données.

Ajouter un Client via Tkinter

```
def ajouter_client():  
    nom = entree_nom.get()  
    email = entree_email.get()  
    telephone = entree_telephone.get()  
  
    curseur.execute('''  
        INSERT INTO clients (nom, email, telephone, categorie, nombre_employes)  
        VALUES (?, ?, ?, ?, ?)  
    ''', (nom, email, telephone, "Web", 5))  
    connexion.commit()  
  
    label_message.config(text="Client ajouté avec succès !")  
  
# Widgets Tkinter pour ajouter un client  
entree_nom = tk.Entry(fenetre)  
entree_nom.pack()  
entree_email = tk.Entry(fenetre)  
entree_email.pack()  
entree_telephone = tk.Entry(fenetre)  
entree_telephone.pack()  
  
bouton_ajouter = tk.Button(fenetre, text="Ajouter Client", command=ajouter_client)  
bouton_ajouter.pack()  
label_message = tk.Label(fenetre, text="")  
label_message.pack()
```

# Gestion des Erreurs

## Pourquoi gérer les erreurs ?

- Prévenir les plantages inattendus.
- Améliorer l'expérience utilisateur en fournissant des messages d'erreur clairs.

```
try:
    fichier = open('data.txt', 'r')
except FileNotFoundError:
    print("Fichier non trouvé.")
finally:
    fichier.close()
```

## Utiliser **finally** pour Nettoyer

- Bloc **finally** s'exécute toujours, même si une exception a été levée



# Gestion des Erreurs et Sécurité - Introduction au Module **logging**

## Introduction au Module **logging**

Le module **logging** de Python permet de suivre les événements qui se produisent lors de l'exécution du programme. Il offre un moyen de garder une trace des erreurs, avertissements, et autres messages d'information qui peuvent aider au débogage et à la maintenance du code.

- **Pourquoi utiliser **logging** ?**
  - **Suivi des erreurs** : Enregistre les erreurs de manière organisée.
  - **Analyse** : Permet d'examiner le comportement du programme à posteriori.
  - **Sécurité** : Cache des informations sensibles tout en signalant les erreurs.
  - **Flexibilité** : Permet de définir plusieurs niveaux de messages (ex. info, warning, error) et d'enregistrer les logs dans un fichier ou sur la console.

Le module **logging** inclut plusieurs niveaux de gravité pour classer les messages :

- **DEBUG** : Messages détaillés pour le débogage.
- **INFO** : Informations générales sur l'état du programme.
- **WARNING** : Avertissements indiquant des problèmes potentiels.
- **ERROR** : Messages pour les erreurs sérieuses.
- **CRITICAL** : Messages pour des erreurs très graves qui peuvent nécessiter l'arrêt du programme.

```
import logging

# Configuration de base du logging
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("app.log"),
        logging.StreamHandler()
    ]
)

# Exemples de messages de log
logging.debug("Ce message est pour le débogage.")
logging.info("Ce message informe sur le statut général.")
logging.warning("Attention : une action pourrait poser problème.")
logging.error("Erreur : un problème sérieux est survenu.")
logging.critical("Critique : le programme pourrait s'arrêter.")
```

# Gestion des Erreurs - Typing en Python

## Annotations de Type avec `typing`

Les annotations de type permettent de rendre le code plus lisible, maintenable, et facilitent le travail en équipe en spécifiant les types de données attendus pour les variables, les arguments, et les valeurs de retour des fonctions.

## Pourquoi utiliser `typing` ?

- **Documentation claire** : Facilite la compréhension du code.
- **Détection d'erreurs** : Aide les outils d'analyse statique (comme `mypy`) à repérer les erreurs potentielles de type.
- **Productivité accrue** : Favorise la collaboration en rendant le code plus explicite.

**`mypy`** : Utilisé pour vérifier les annotations de type et détecter les erreurs de type dans le code.

**Exécution** : Installez `mypy` et lancez-le avec votre fichier :

```
pip install mypy
mypy mon_programme.py
```

```
from typing import List, Dict, Optional

class GradeNotFoundError (Exception):
    def __init__(self, student_id: int, message:
str = "Notes introuvables" ):
        self.student_id = student_id
        super().__init__(f"{message} pour
l'étudiant ID: {student_id}")

def get_student_grades (student_id: int) ->
Dict[str, float]:
    grades = {
        1: {"math": 90.0, "science": 85.5},
        2: {"math": 78.0, "science": 82.0}
    }
    if student_id not in grades:
        raise GradeNotFoundError(student_id)
    return grades[student_id]

try:
    get_student_grades( 3)
except GradeNotFoundError as e:
    print(e)
```

# Techniques de Débogage

- **Pourquoi le Débogage ?**

- Le débogage permet de détecter et corriger les erreurs de logique, de syntaxe et de runtime dans le code.

## Outils de Débogage en Python

1. **print()** : Utilisé pour afficher les valeurs des variables à différentes étapes de l'exécution.
  - **Exemple :**  
`print(f"Valeur de la variable x : {x}")`
2. **assert** : Permet de vérifier une condition, et lève une exception si la condition est fausse.
  - **Exemple :** `assert x > 0, "x doit être supérieur à 0"`
3. **Module pdb** : Débogueur interactif intégré à Python.

```
import pdb  
pdb.set_trace()
```

Placez `pdb.set_trace()` à l'endroit où vous souhaitez interrompre l'exécution et examiner le code.

**Exceptions (try-except)** : Gérer les erreurs de manière appropriée et fournir des messages d'erreur clairs.

## Utilisation des Outils Externes

- **IDE** : Des environnements de développement intégrés comme PyCharm ou Visual Studio Code offrent des outils avancés de débogage (points d'arrêt, inspection des variables, etc.).

## À Retenir :

- Utilisez `print()` pour un débogage rapide et les outils comme `pdb` pour des cas complexes.
- Gérez les erreurs avec des blocs `try-except` pour éviter les plantages de l'application.

# Introduction à CGI (Common Gateway Interface)

## Qu'est-ce que CGI ?

- Le **Common Gateway Interface** (CGI) est une norme permettant aux serveurs web d'exécuter des programmes (scripts) et de générer du contenu dynamique.
- Les scripts CGI sont souvent écrits en Python, Perl, ou PHP, et permettent de créer des pages web interactives.

## Fonctionnement :

- Un utilisateur envoie une requête (par exemple, en soumettant un formulaire).
- Le serveur web exécute un script CGI (écrit en Python, par exemple) pour traiter la requête.
- Le script génère dynamiquement une page web, renvoyée à l'utilisateur via le serveur.



# Lancement d'un Serveur Web Local en Mode CGI

Vous pouvez utiliser le module intégré `http.server` de Python pour démarrer un serveur CGI sur votre machine locale.

- Ouvrez un terminal, placez-vous dans le dossier principal de votre projet (`projet_cgi`) et lancez le serveur avec cette commande :

```
python -m http.server --cgi 8000
```

## Structure d'un Projet CGI en Python

Pour démarrer un projet CGI en Python, il est essentiel de structurer correctement les fichiers afin que le serveur puisse exécuter les scripts et servir les pages HTML correctement. Voici une structure simple de projet CGI, incluant les fichiers Python, HTML, et autres nécessaires :

```
projet_cgi/  
├─ ajouter_client.html  
├─ cgi-bin/                # Dossier pour les scripts CGI  
│   └─ ajouter_client.py    # Script Python CGI pour ajouter
```

# Mise en Place d'un Script CGI en Python

## Préparation de l'Environnement :

- Placez votre script Python dans le répertoire `cgi-bin` du serveur web (généralement `/usr/lib/cgi-bin/` sur un serveur Apache).
- Assurez-vous que le script a les permissions d'exécution (`chmod +x nom_du_script.py`).

## Structure de Base d'un Script CGI :

- Commencez le script avec la ligne `shebang` pour spécifier l'interpréteur Python.
- Utilisez le module `cgi` pour manipuler les données des requêtes HTTP.

```
#!/usr/bin/env python3

# Importation des modules nécessaires
import cgi

# Entête HTTP pour indiquer le type de contenu
print("Content-Type: text/html\n")

# Récupération des données du formulaire
form = cgi.FieldStorage()
nom = form.getvalue('nom', 'Inconnu')

# Génération de la réponse HTML
print(f"<html><body><h1>Bonjour,
{nom}</h1></body></html>")
```

# Gestion des Formulaires HTML avec CGI

Les scripts CGI interagissent souvent avec des formulaires HTML pour collecter des informations utilisateur.

## Exemple de Formulaire HTML :

```
<form action="/cgi-bin/traitement.py" method="post">
  Nom: <input type="text" name="nom">
  <input type="submit" value="Envoyer">
</form>
```

## Interaction avec le Script CGI :

- Lorsque le formulaire est soumis, les données sont envoyées au script `ajout_client.py` dans le dossier `cgi-bin`.
- Le script CGI utilise le module `cgi` pour récupérer les valeurs saisies par l'utilisateur.

# Utilisation des Méthodes HTTP dans CGI

Les scripts CGI peuvent traiter des requêtes **GET** et **POST** :

- **GET** : Les données sont passées dans l'URL (limitées en taille).
- **POST** : Les données sont envoyées dans le corps de la requête HTTP (plus sécurisé et sans limite de taille).

**Traitement des Données GET et POST :**

- Le module `cgi` permet de manipuler les deux méthodes.

```
# Récupérer les données du formulaire
form = cgi.FieldStorage()
nom = form.getvalue('nom', 'Inconnu')
```

# Mise en Pratique

## Objectif :

Dans cet exercice, vous allez développer un script CGI en Python pour gérer l'ajout de clients dans une base de données CRM via un formulaire HTML. Vous intégrerez le module `logging` pour tracer les informations et faciliter le débogage.

## Contexte :

Votre script `ajouter_client.py` va récupérer les données saisies par l'utilisateur dans un formulaire HTML (nom, email, téléphone, catégorie, et nombre d'employés) et les ajouter à la base de données CRM. Le script doit également gérer les erreurs et enregistrer des logs à chaque étape pour assurer un suivi complet.

## Étapes à Suivre :

1. **Création du Formulaire HTML :**
  - Créez un fichier `ajouter_client.html` pour collecter les informations du client.
  - Assurez-vous que le formulaire envoie une requête POST au script CGI
2. **Création du Script CGI :**
  - Créez un script Python nommé `ajouter_client.py` dans le dossier `cgi-bin`.
  - Utilisez le module `cgi` pour récupérer les données du formulaire, et `logging` pour enregistrer chaque étape.
  - Assurez-vous que les erreurs de saisie (comme les champs vides ou un nombre d'employés non valide) sont gérées avec des messages appropriés.
3. **Tester et Déboguier :**
  - Ouvrez `ajouter_client.html` dans votre navigateur en utilisant `http://localhost:8000/ajouter_client.html`.
  - Remplissez le formulaire et soumettez-le.
  - Consultez `debug.log` pour vérifier les étapes de l'exécution et les valeurs reçues du formulaire.

1. Ouvrez l'invite de commande (`cmd`) et naviguez vers le répertoire où se trouvent vos fichiers :

```
cd C:\mon_serveur_cgi
```

3. Lancez le serveur CGI intégré de Python :

```
python -m http.server --cgi 8000
```

4. Ouvrez votre navigateur web et accédez à l'adresse suivante :

```
http://localhost:8000/formulaire.html
```

Lorsque vous soumettez le formulaire, le script CGI `inscription.py` sera exécuté. Le serveur intégré de Python prendra en charge l'exécution du script sans aucune configuration supplémentaire.

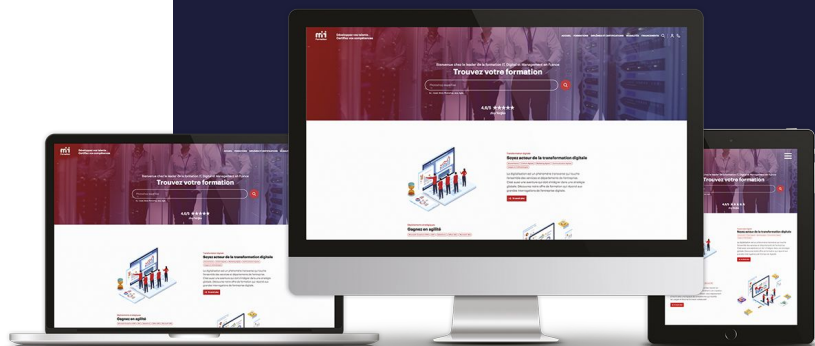
# À Retenir

1. **Connexion à une Base de Données SQL :**
  - Utiliser des bibliothèques comme `sqlite3` pour intégrer et interagir avec une base de données dans vos applications Python.
  - Structurer les requêtes pour des opérations CRUD (Créer, Lire, Mettre à jour, Supprimer).
2. **Interfaces Graphiques avec Tkinter :**
  - Tkinter permet de créer des interfaces utilisateur basiques en Python.
  - Structurer des éléments comme les boutons, champs de texte, et labels pour construire une interface intuitive.
3. **Techniques de Débogage et Gestion des Erreurs :**
  - Utiliser `try`, `except` pour capturer et gérer les erreurs sans interrompre l'exécution du programme.
  - Implémenter le module `logging` pour un suivi détaillé et permanent des erreurs et de l'état de l'application.



# TP de validation des acquis Variables, Tableaux et Fonctions

---



m2information.fr



# TP de validation des acquis Chapitre 10

## Partie 1 : Connexion à une Base de Données SQL

1. Créez un module `db.py` pour gérer la base de données SQLite :
  - **Fonction `creer_table_clients()`** : Créez une table `clients` dans la base de données `crm.db` pour stocker les informations des clients.
  - **Fonction `ajouter_client_bd(client)`** : Insérez un nouveau client dans la table `clients` en passant un objet `Client` en paramètre.
  - **Fonction `recuperer_clients_bd()`** : Récupérez tous les clients depuis la base de données et renvoyez-les sous forme de liste.
2. **Instructions :**
  - Chaque client doit contenir les informations suivantes : `nom`, `email`, `telephone`, `categorie`, `nombre_employes`, `prix_par_employe`, `cout_ht`, et `cout_ttc`.
  - Assurez-vous d'utiliser les fonctionnalités de `sqlite3` pour effectuer les opérations CRUD (Create, Read, Update, Delete) sur les clients.

# TP de validation des acquis Chapitre 10

## Partie 2 : Programmation d'Interfaces Graphiques avec Tkinter

1. Créez un module `crm_ui.py` pour l'interface graphique :
  - **Initialisation de l'interface** : Utilisez la classe `CRMApp` pour créer l'interface graphique de l'application avec les widgets nécessaires (entrées de texte, boutons, labels) pour ajouter et afficher les clients.
  - **Tableau de bord (Treeview)** : Ajoutez un `Treeview` pour afficher la liste des clients. Configurez les colonnes pour afficher les informations telles que `nom`, `email`, `telephone`, `categorie`, `nombre d'employés`, `cout_ht`, et `cout_ttc`.
  - **Boutons** : Implémentez les boutons pour les fonctionnalités suivantes :
    - **Ajouter un client** : Utilisez la méthode `ajouter_client()` pour ajouter un client dans la base de données et mettre à jour l'affichage.
    - **Actualiser les clients** : Implémentez la méthode `afficher_clients()` pour charger les clients depuis la base de données et les afficher dans le `Treeview`.
    - **Actualiser les statistiques** : Créez une méthode `afficher_statistiques()` pour calculer et afficher le nombre total de clients et le chiffre d'affaires total.
2. **Instructions** :
  - Utilisez les éléments graphiques de Tkinter (`Label`, `Entry`, `Button`, `Treeview`) pour créer une interface utilisateur conviviale.
  - La méthode `ajouter_client()` doit créer un objet `Client`, l'ajouter à la base de données via `ajouter_client_bd()` et mettre à jour l'interface avec les nouvelles données.
  - Ajoutez des messages de confirmation (`messagebox.showinfo`) pour informer l'utilisateur des actions effectuées (par exemple, client ajouté avec succès).

# TP de validation des acquis Chapitre 10

## Partie 4 : Techniques de Débogage et Gestion des Erreurs

### 1. Gestion des erreurs :

- Dans le module `admin.py`, utilisez des blocs `try-except` pour valider les entrées utilisateur lors de la création d'un client (par exemple, vérifier que le nombre d'employés est un nombre entier).
- Dans `crm_ui.py`, gérez les erreurs potentielles lors de l'interaction avec la base de données et l'interface graphique (par exemple, gestion des champs vides).
- Affichez des messages d'erreur clairs et adaptés en utilisant `messagebox.showerror()` pour informer l'utilisateur.

### 2. Débogage :

- Utilisez des instructions `print()` ou des outils de débogage intégrés dans votre IDE pour suivre l'état de vos variables et vérifier le flux d'exécution du programme.
- Ajoutez des points de contrôle (`assert`) si nécessaire pour valider certaines conditions dans vos méthodes.

# TP de validation des acquis Chapitre 10

## Partie 5 : Statistiques et Mise à Jour de l'Interface

### 1. Statistiques :

- Dans le module `statistiques.py`, ajoutez les fonctions nécessaires pour calculer les statistiques générales, comme le chiffre d'affaires total et le nombre de clients par catégorie.
- Connectez ces fonctions à l'interface utilisateur pour afficher les statistiques mises à jour dans les labels `label_total_clients` et `label_chiffre_affaire`.

### 2. Intégration :

- Dans `crm_ui.py`, intégrez les méthodes `afficher_clients()` et `afficher_statistiques()` pour qu'elles soient appelées dès l'ouverture de l'interface, permettant ainsi de charger les données actuelles de la base de données.

## Livrables Attendus :

- Un projet CRM complet contenant les modules suivants :
  - `client.py` : Classe `Client` et méthodes associées.
  - `db.py` : Fonctions pour interagir avec la base de données SQLite.
  - `admin.py` : Fonctions pour gérer les clients (ajout, modification, suppression).
  - `crm_ui.py` : Interface graphique Tkinter pour l'interaction utilisateur.
  - `statistiques.py` : Fonctions pour calculer et afficher les statistiques.
- Une application graphique qui permet :
  - L'ajout, la modification, et la suppression des clients.
  - L'affichage des clients dans un tableau (Treeview).
  - La mise à jour des statistiques en temps réel.

# Les Bonus du Formateur Jules Galian



Chers stagiaires, en guise de remerciement pour votre participation et votre engagement lors de cette formation, voici quelques ressources exclusives pour continuer à progresser dans vos projets :

## Livres et eBooks sur Python :

- **"Apprendre à programmer avec Python"** de Gérard Swinnen – Un excellent livre pour débiter en Python avec des explications claires et des exercices pratiques.
- **"Automate the Boring Stuff with Python"** de Al Sweigart (Version française disponible) – Apprenez à automatiser des tâches du quotidien avec des exemples concrets.

## Sites Web et Tutoriels Gratuits :

- **OpenClassrooms** : [openclassrooms.com](https://openclassrooms.com) – Cours complets et gratuits sur Python, allant des bases à des concepts plus avancés.
- **Docstring (ancien Zeste de Savoir)** : [zestedesavoir.com](https://zestedesavoir.com) – Des tutoriels en français pour approfondir votre compréhension de Python.
- **Python.org (Documentation Officielle)** : [docs.python.org/fr/](https://docs.python.org/fr/) – Pour consulter la documentation officielle en français.
- **Real Python** : [realpython.com](https://realpython.com) – De nombreux tutoriels, guides, et astuces sur Python (articles majoritairement en anglais, mais certaines ressources traduites).

## Ressources d'Apprentissage Complémentaires :

- **MOOCs et Cours Vidéo** :
  - **Udemy – "Apprendre Python"** : Une série de vidéos pour vous guider étape par étape.
  - **YouTube – Graven Développement** : Chaîne francophone avec des tutoriels Python et des projets concrets.
- **Plateformes Pratiques** :
  - **LeetCode et HackerRank** (avec des exercices disponibles en français) – Pratiquez et améliorez vos compétences en résolvant des défis de programmation en Python.

## Liens vers des IA et Outils d'Intelligence Artificielle :

- **ChatGPT** – Une IA conversationnelle pour répondre à vos questions sur Python et d'autres sujets.
- **Google Colab** – Un environnement de développement Python gratuit, hébergé en ligne pour l'exécution et le partage de vos notebooks.

# Dossier pédagogique

- Feuilles d'émargement signées pour chaque journée.
- Feuilles d'émargement signées pour les passages de certifications (si certifications)
- Évaluations formateur

*Ce slide le dernier jour de la formation au retour de la pause déjeuner permet de ne rien oublier. M2I doit transmettre les évaluations à son client avant 15h.*

*Pensez à vous connecter à votre espace formateur pour vous assurer que les évaluations de chaque stagiaires ont bien été remplies.*

*Vous pouvez dire aux stagiaires que vous aurez leurs évaluations dans quelques jours et que les commentaires sur le formateur sont agréables à lire, des évaluations bien remplies sont valorisantes.*

*Pensez à vérifier sur M2I Sign qu'aucune signature ne manque pour vous et les stagiaires, pensez aussi aux émargements supplémentaires en cas de certifications.*



# Formation Django

## Objectifs de Formation :

- Prendre en main le framework Django.
- Construire un site Web complet.
- Fournir une API REST pour les clients mobiles/front-end.
- Personnaliser Django en fonction des besoins.
- Tester et déployer un site en production.

## Prérequis :

- Connaissance de base en Python et des notions en HTML/CSS/JavaScript.

## Public Concerné :

- Développeurs web et chefs de projet.

## Programme (4 jours) :

- **Jour 1** : Introduction à Django, programmation Python, structure d'un projet Django, URL et vues, chargement des templates.
- **Jour 2** : Interactions avec la base de données, ORM, relations de modèles (OneToOne, ForeignKey, ManyToMany), migrations, requêtes SQL et QuerySets.
- **Jour 3** : Gestion des formulaires, administration Django, structurer les vues avec les classes, mise en place d'une API REST avec Django REST Framework.
- **Jour 4** : Notions avancées (versioning, caching, authentification), tests avec Django, configuration en production, déploiement avec Docker.

## Méthodes Pédagogiques :

- Formation présentielle/distancielle.
- Alternance entre méthode démonstrative, interrogative, et pratique.

# Formation Big Data avec Python et Spark

## Objectifs de Formation :

- Développer des applications de Machine Learning et d'IA avec Spark et Python.
- Exploiter la programmation parallèle sur un cluster.
- Optimiser des algorithmes de Machine Learning.
- Utiliser les bibliothèques Python pour l'IA.
- Comprendre le cycle de vie d'un projet Data Science.

## Programme (5 jours) :

- **Jour 1** : Introduction au Big Data, rappel des bases en Python et Data Science.
- **Jour 2** : Concepts du Machine Learning, algorithmes (régression, classification, K-NN, etc.).
- **Jour 3** : Algorithmes avancés (Réseaux de neurones, SVM, etc.).
- **Jour 4** : Développement avec Apache Spark, Spark Streaming, Spark SQL, GraphFrames.
- **Jour 5** : Visualisation des données (Dataviz) avec Python, Tableau, Power BI, introduction à MLOps.



# Formation Python perfectionnement

## Objectifs de formation :

- Maîtriser les subtilités avancées de Python.
- Écrire des programmes robustes, structurés et efficaces.
- Approfondir la gestion du développement Python.

## Prérequis :

- Avoir suivi **Python - Par la pratique** ou disposer des compétences équivalentes.

## Public concerné :

Développeurs, administrateurs, architectes techniques.

## Programme :

### Jour 1 :

- Programmation avancée (\*args, \*\*kwargs, threading, sockets).
- POO avancée.

### Jour 2 :

- XML (SAX, DOM, Xpath).
- Interfaces graphiques (Tkinter).
- Persistance (JSON, Pickle, bases de données).

### Jour 3 :

- Intégration Python avec C et Java.
- Débogage et profiling.



# Formation Intégrer les Modèles ChatGPT et GPT-4 dans les Applications Python

## Objectifs de Formation :

- Comprendre le fonctionnement des modèles ChatGPT et GPT-4.
- Intégrer ces modèles dans des applications de traitement du langage naturel (NLP) en Python.
- Développer des applications de génération de texte, Q&R, résumé de contenu, et prompting.

## Programme (3 jours) :

- **Jour 1** : Bases de ChatGPT et GPT-4, apprentissage des modèles, études de cas d'utilisation.
- **Jour 2** : Approfondissement des API OpenAI, utilisation des modèles via la bibliothèque Python d'OpenAI, modération et sécurité des données.
- **Jour 3** : Fonctionnalités avancées, techniques de prompt engineering, adaptation des modèles GPT à des domaines spécifiques.

# Compiler un fichier exécutable en python

## Introduction :

- **PyInstaller** est une bibliothèque Python qui permet de convertir des scripts Python en fichiers exécutables (.exe) sur Windows, ou en exécutables autonomes pour Mac et Linux.
- Cette conversion permet de distribuer des applications Python sans avoir besoin d'installer Python sur la machine cible.

```
pyinstaller --onefile example.py  
pip install pyinstaller
```

```
pyinstaller --onefile  
--icon=mon_icone.ico example.py
```

```
pyinstaller --onefile --add-data  
"path_to_folder/*;target_folder"  
main.py
```

# Comprendre et Manipuler JSON avec Python

**JSON** signifie **JavaScript Object Notation**.

Format léger pour l'échange de données.

Lisible par les humains et facilement interprété par les machines.

Structure basée sur les paires **clé-valeur** :

```
{  
  "nom": "Alice",  
  "age": 25,  
  "actif": true,  
  "compétences": ["Python", "Django", "SQL"]  
}
```

## Manipuler JSON en Python

### 1. Importer le module JSON :

```
import json  
  
data = '{"nom": "Alice", "age": 25}'  
dict_data = json.loads(data) # Convertit en  
dictionnaire Python  
print(dict_data["nom"]) # Alice  
with open('data.json', 'r') as f:  
    dict_data = json.load(f)  
print(dict_data["age"]) # 25  
data = {"nom": "Bob", "actif": False}  
json_str = json.dumps(data, indent=4) #  
Beautification avec indent  
print(json_str)
```

# Hicham,Pascal, toufik & Fethi / Initiation à l'exécution de commandes Shell avec Python

- Le module `subprocess` permet de démarrer de nouveaux processus, d'exécuter des commandes Shell, et de gérer les entrées et sorties de ces processus. Son utilisation est recommandée pour remplacer les modules plus anciens comme `os.system()` et `os.popen()`.

```
import subprocess
# Commande qui liste les fichiers
dans le répertoire courant
subprocess.run(["ls", "-l"])
result = subprocess.run(["ls", "-l"],
capture_output=True, text=True)
print(result.stdout) # Affiche la
sortie de la commande
```

## . Exécuter des Commandes Complexes

Les commandes Shell complexes (comme les pipes `|`) nécessitent de lancer le Shell explicitement (`shell=True`). Toutefois, cette méthode doit être utilisée avec précaution pour éviter des vulnérabilités de sécurité.

```
subprocess.run("ls -l | grep .py", shell=True)
```

# Hicham,Pascal, toufik & Fethi / Gestion serveur avec Python

- Le module `subprocess` permet de démarrer de nouveaux processus, d'exécuter des commandes Shell, et de gérer les entrées et sorties de ces processus. Son utilisation est recommandée pour remplacer les modules plus anciens comme `os.system()` et `os.popen()`.

```
import paramiko

# Connexion à un serveur distant via SSH
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.connect("hostname", username="user", password="password")

# Exécution d'une commande sur le serveur
stdin, stdout, stderr =
client.exec_command("ls -l /var/log")
print(stdout.read().decode())
client.close()
```

## . Exécuter des Commandes Complexes

Les commandes Shell complexes (comme les pipes `|`) nécessitent de lancer le Shell explicitement (`shell=True`). Toutefois, cette méthode doit être utilisée avec précaution pour éviter des vulnérabilités de sécurité.

```
subprocess.run("ls -l | grep .py", shell=True)
```

# Jean François : Développement de Modules Ansible Personnalisés

Ansible utilise des modules pour exécuter des tâches spécifiques, et vous pouvez développer des modules personnalisés en Python pour des opérations uniques ou des environnements particuliers. Par exemple :

- Créer un module Python pour automatiser des tâches spécifiques non couvertes par les modules Ansible natifs.
- Écrire des modules pour interagir avec des API tierces ou pour gérer des applications qui n'ont pas de module Ansible par défaut.

```
from ansible.module_utils.basic import AnsibleModule

def main():
    module_args = dict(
        name=dict(type='str', required=True)
    )
    module = AnsibleModule(argument_spec=module_args)

    response = f"Hello, {module.params['name']}!"
    module.exit_json(changed=False, message=response)

if __name__ == '__main__':
    main()
```

# Jean François : Utiliser Copilot avec VSC

## Installer l'extension GitHub Copilot :

- Ouvrez VS Code.
- Accédez à l'onglet **Extensions** (Ctrl+Shift+X).
- Recherchez "GitHub Copilot" et cliquez sur **Installer**.

## Se connecter à GitHub :

- Une fois l'extension installée, VS Code vous demandera de vous connecter.
- Suivez les instructions pour autoriser votre compte GitHub.

## Activer GitHub Copilot :

- Dans VS Code, vérifiez que l'extension est activée.
- Testez dans un fichier en tapant du code pour voir les suggestions de Copilot.

## Configurer les préférences (facultatif) :

- Accédez aux paramètres (Ctrl+,).
- Recherchez "Copilot" pour ajuster les comportements (par exemple, activer ou désactiver des suggestions automatiques).



# Création de Playbooks ou de Tâches Ansible Dynamique avec Python

Python permet de créer des playbooks dynamiques, en fonction des besoins d'un environnement ou des conditions. Par exemple, un script Python pourrait générer un playbook Ansible basé sur des paramètres ou des configurations spécifiques, comme une liste de services à redémarrer sur plusieurs serveurs.

```
import yaml

def generate_playbook(services):
    playbook = [{
        'hosts': 'web_servers',
        'tasks': [{ 'name': f'Restart {service}', 'service': { 'name':
service, 'state': 'restarted' }} for service in services]
    }]

    with open('generated_playbook.yml', 'w') as f:
        yaml.dump(playbook, f)

services_to_restart = ['nginx', 'mysql']
generate_playbook(services_to_restart)
```

# David : Analyse d'images médicales au format DICOM

**pydicom** est une bibliothèque Python dédiée à la gestion des fichiers DICOM, permettant de lire, écrire et modifier des fichiers DICOM facilement.

## Principales fonctionnalités :

- Chargement des fichiers DICOM et extraction des métadonnées.
- Accès aux images et possibilité de les manipuler ou les transformer.
- Intégration avec d'autres bibliothèques pour l'analyse approfondie des images (par exemple, avec NumPy).

```
import pydicom
from pydicom.data import get_testdata_files

# Charger un fichier DICOM
dicom_file = get_testdata_files("CT_small.dcm")[0]
ds = pydicom.dcmread(dicom_file)

# Accéder aux métadonnées et afficher une image
print(ds.PatientName)
image_array = ds.pixel_array
```

# David : Analyse d'images médicales au format DICOM

**SimpleITK** est une bibliothèque puissante pour l'analyse d'images médicales, compatible avec DICOM, NIfTI, et d'autres formats spécialisés. Elle est utilisée pour les opérations complexes sur les images, comme la segmentation, la détection de bords, et la transformation d'images.

## Principales fonctionnalités :

- Prend en charge divers formats d'images, y compris DICOM.
- Algorithmes de traitement d'image pour la détection de contours, la segmentation, le filtrage.
- Intégration avec **pydicom** pour lire les métadonnées.

```
import SimpleITK as sitk

# Lecture de l'image DICOM
image = sitk.ReadImage("path/to/image.dcm")

# Appliquer un filtre pour le lissage
smoothed_image = sitk.SmoothingRecursiveGaussian(image, sigma=2.0)
```

# David : Analyse d'images médicales au format DICOM

**MONAI** est une bibliothèque avancée spécialisée pour les applications d'intelligence artificielle dans l'imagerie médicale. Elle intègre PyTorch pour les modèles d'apprentissage profond, avec des fonctionnalités spécifiques pour DICOM et d'autres formats médicaux.

## Principales fonctionnalités :

- Outils avancés pour la segmentation, le diagnostic automatique, et la classification.
- Transformation des images pour l'entraînement et l'inférence de modèles.
- Conçu pour le deep learning et les architectures de réseaux neuronaux dans l'analyse d'images médicales.

```
from monai.transforms import LoadImage

# Charger une image DICOM et la convertir en format compatible pour deep
learning
dicom_img = LoadImage(image_only=True) ("path/to/image.dcm")
```

Conclusion :

Charger et visualiser les données DICOM avec pydicom.

Appliquer des transformations et des filtrages d'image avec SimpleITK ou OpenCV.

Détecter des patterns spécifiques grâce à des techniques de machine learning avec MONAI.

# David : SMTP avec python

## Qu'est-ce que SMTP ?

- **SMTP** (Simple Mail Transfer Protocol) est le protocole utilisé pour envoyer des emails via Internet.
- Python offre une bibliothèque intégrée, `smtplib`, pour envoyer des emails de manière simple.

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart

# Informations du serveur SMTP
smtp_server = "smtp.gmail.com"
port = 587 # Port pour TLS

# Informations de connexion
email = "votre_email@gmail.com"
password = "votre_mot_de_passe"

# Destinataire
destinataire = "destinataire@example.com"

# Contenu du message
sujet = "Exemple d'Email avec Python"
contenu = "Bonjour, ceci est un email envoyé avec Python."

# Création du message
message = MIMEMultipart()
message['From'] = email
message['To'] = destinataire
message['Subject'] = sujet
message.attach(MIMEText(contenu, 'plain'))

# Connexion au serveur et envoi
try:
    with smtplib.SMTP(smtp_server, port) as server:
        server.starttls() # Sécurisation de la connexion
        server.login(email, password)
        server.sendmail(email, destinataire, message.as_string())
        print("Email envoyé avec succès !" )
except Exception as e:
    print(f"Erreur lors de l'envoi : {e}")
```

# David : Tâche CRON associé à un fichier python

## Qu'est-ce qu'une tâche CRON ?

- Une tâche **CRON** est un programme ou script exécuté automatiquement à des intervalles définis.
- Utilisé pour automatiser des tâches répétitives, comme des sauvegardes ou des notifications.

Configurer une tâche:

- **Dépend de l'environnement serveur.**

# Othmane : Utilisation de Python en Business Intelligence avec IA

## Pourquoi Python pour la BI et l'IA ?

Python est un langage de choix pour l'Intelligence Artificielle (IA) en Business Intelligence (BI) grâce à sa flexibilité, ses bibliothèques spécialisées, et sa facilité d'intégration avec des outils de gestion et d'analyse de données. Les applications en BI incluent :

- **Prévisions** : Analyser les tendances de données pour prédire les ventes, la demande, etc.
- **Segmentation de clients** : Identifier des segments dans les données pour un ciblage plus efficace.
- **Détection des anomalies** : Repérer les écarts dans les données pour la gestion des risques.

## Pourquoi intégrer Python dans Power BI ?

- **Visualisations avancées** : Créez des graphiques personnalisés (matplotlib, seaborn).
- **Traitement de données** : Utilisez des bibliothèques comme pandas ou numpy pour manipuler les données.
- **Modèles d'IA** : Implémentez des modèles de machine learning (scikit-learn, TensorFlow).
- **Automatisation** : Automatisez les processus analytiques et prédictifs directement dans Power BI.

## Étapes pour Utiliser Python comme Connecteur

1. **Récupérez les données avec Python :**
  - Utilisez les bibliothèques pour interroger des bases, APIs, ou fichiers non standards.
2. **Retournez les données dans un DataFrame pandas :**
  - Power BI s'attend à un objet **DataFrame** pour manipuler les données.
3. **Exécutez le script Python dans Power BI :**
  - Chargez le script Python dans Power Query.
  - Les résultats du DataFrame sont ajoutés à votre modèle de données.

# Othmane : Utilisation de Python en Business Intelligence avec IA

Critères	Relationnelle (SQL)	Non Relationnelle (NoSQL)
Structure	Fixe, basée sur des tables	Flexible, JSON, clés-valeurs, etc.
Relations	Gérées avec des clés primaires/étrangères	Relationnelles simulées (imbriquées)
Langage	SQL	API ou requêtes spécifiques (ex : MongoDB)
Exemples d'utilisation	Gestion des données complexes et liées	Données volumineuses, flexibles



# Othmane : Utilisation de l'IA pour agrémenter une base de donnée SQL

Ajoutez SQLite pour stocker des messages et les associer au contexte GPT-4 :

```
import sqlite3

# Connexion à la base de données
conn = sqlite3.connect('chat_memory.db')
cursor = conn.cursor()

# Création d'une table pour stocker les messages
cursor.execute('''
CREATE TABLE IF NOT EXISTS chat_history (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    role TEXT NOT NULL,
    content TEXT NOT NULL,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP
)
''')
conn.commit()

# Fonction pour enregistrer un message dans la base de données
def save_message(role, content):
    cursor.execute('INSERT INTO chat_history (role, content) VALUES (?, ?)', (role, content))
    conn.commit()

# Fonction pour récupérer l'historique des messages
def get_chat_history():
    cursor.execute('SELECT role, content FROM chat_history ORDER BY timestamp ASC')
    return cursor.fetchall()
```

# Othmane : Utilisation de l'IA pour agrémenter une base de donnée Non Relationnelle

Ajoutez MongoDB pour stocker les messages de manière flexible.

```
from pymongo import MongoClient

# Connexion à la base MongoDB
client = MongoClient('mongodb://localhost:27017/')
db = client['chatgpt']
collection = db['chat_history']

# Fonction pour enregistrer un message
def save_message(role, content):
    collection.insert_one({'role': role, 'content': content,
'timestamp': datetime.datetime.now()})

# Fonction pour récupérer l'historique des messages
def get_chat_history():
    return list(collection.find({}, {'_id': 0, 'role': 1, 'content':
1}).sort('timestamp', 1))
```

# Question hors plan de cours : map()

## Description :

- `map()` est une fonction intégrée de Python.
- Elle applique une fonction spécifiée à chaque élément d'un iterable (comme une liste, un tuple, etc.).
- `map()` retourne un objet de type `map` (un iterable) que l'on peut convertir en liste, tuple, etc.

```
numbers = [1, 2, 3, 4, 5]
result = map(lambda x: x * 2, numbers)
print(list(result)) # Output : [2, 4, 6, 8, 10]
```

## Avantages de `map()` :

- Plus rapide et élégant que l'utilisation de boucles pour transformer des éléments.
- Permet de réduire le nombre de lignes de code.
- Fonctionne bien avec des fonctions lambda pour des transformations rapides.

# Traitement d'une requête HTTP

## Cycle de Vie d'une Requête HTTP dans Django

**Client :** L'utilisateur envoie une requête HTTP depuis un navigateur ou une application.

**Serveur Web :** Le serveur web (ex: Nginx, Apache) reçoit la requête et la redirige vers l'application Django via WSGI ou ASGI.


**Middleware :** La requête passe à travers une série de middlewares Django, qui peuvent modifier ou valider la requête avant qu'elle n'atteigne la vue. Exemple : Authentification, gestion de sessions, sécurité (CSRF).


**URL Routing, Vue, Modèle, Template :** Django utilise le fichier urls.py pour faire correspondre l'URL de la requête à la vue appropriée. La vue associée traite la requête. Elle peut interagir avec le modèle pour récupérer des données ou effectuer des opérations logiques. Si nécessaire, la vue interroge les modèles pour accéder ou manipuler les données stockées dans la base de données via l'ORM. La vue peut rendre un template pour générer du HTML dynamique.

**Réponse HTTP:** Django retourne une réponse HTTP au client, qui peut être du HTML, JSON, XML, etc., en fonction du type de requête et du traitement effectué par la vue.

**Client :** Le client reçoit la réponse HTTP et l'affiche dans le navigateur ou l'application, complétant ainsi le cycle de vie de la requête.

# Exercice Requête HTTP API OPENAI

 **Objectif** : Dans cet exercice, nous allons créer un programme Python qui interagit avec l'API OpenAI pour générer des réponses dynamiques en fonction des messages de l'utilisateur. L'objectif est d'exploiter les concepts des chapitres 1 à 6, en particulier l'utilisation de fonctions, d'arguments dynamiques **`**kwargs`**, et de manipulation de données JSON.

 **Contexte** : En utilisant les connaissances acquises sur les fonctions Python, les arguments dynamiques (**`*args`**, **`**kwargs`**), et la gestion des entrées utilisateur, nous allons concevoir un script permettant de :

1. Saisir un message utilisateur et l'envoyer à OpenAI pour obtenir une réponse.
2. Récupérer la réponse sous forme de JSON.
3. Extraire les informations renvoyées et les afficher de manière structurée.

## Énoncé de l'exercice :

1. **Créer une fonction `openai_request(prompt, **kwargs)`** qui prend un message `prompt` de l'utilisateur et envoie une requête à l'API OpenAI.
  - **Paramètre `prompt`** : message de l'utilisateur.
  - **Paramètres dynamiques `**kwargs`** : permettent d'ajouter des informations supplémentaires au besoin.
  - La fonction retourne la réponse d'OpenAI sous forme de texte brut.
2. **Créer une fonction `afficher_reponse_json(response_text)`** qui :
  - Convertit la réponse d'OpenAI en dictionnaire JSON.
  - Utilise **`**kwargs`** pour afficher chaque clé et valeur du JSON en structurant l'affichage pour qu'il soit facile à lire.
3. **Dans la fonction principale `main()`** :
  - Utiliser une boucle qui demande un message à l'utilisateur.
  - Envoyer ce message à `openai_request`.
  - Afficher la réponse en JSON en utilisant `afficher_reponse_json`.
  - Permettre à l'utilisateur de taper `exit` pour quitter.

# Questions / réponses

- Revenons sur les questions hors plan de cours que vous m'avez posé durant la formation pour y répondre

**Merci d'avoir suivi cette formation  
M2I et à très bientôt !**

---



# Bilan formation et remerciements

- Merci d'avoir participé à cette formation M2I.
- Envoie du Bilan formation.





# Votre formateur

Jules Galian

Formateur externe M2I

j.galian@com

<https://www.linkedin.com/in/jules-galian-929686198/>

Et encore merci !