

Support de cours

PROGRAMMATION EN PYTHON



CHAPITRE V : LES FICHIERS ET LA PROGRAMMATION MODULAIRE EN PYTHON

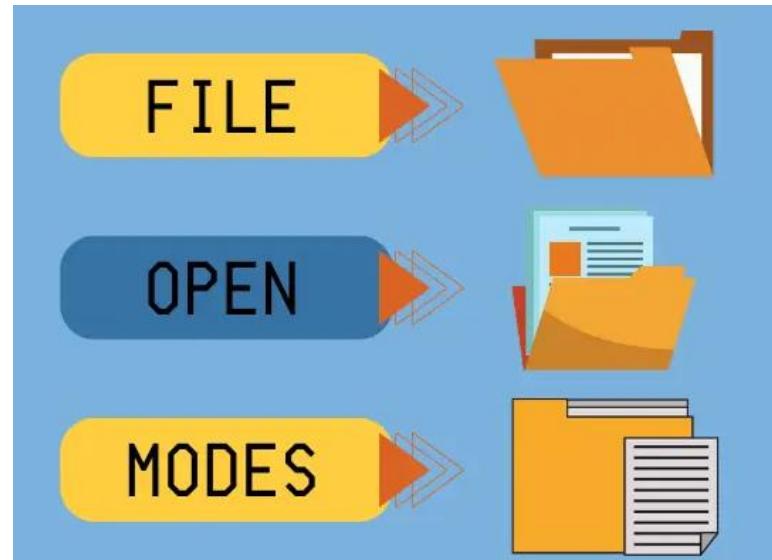
Plan

- INTRODUCTION**
- MANIPULATION DE FICHIERS**
- GESTION DES ERREURS ET EXCEPTIONS**
- PROGRAMMATION MODULAIRE**
- BONNES PRATIQUES DE STRUCTURATION DU CODE**
- EXERCICES**

Introduction

Les fichiers permettent de stocker et de récupérer des données de manière persistante. En Python, on peut manipuler différents types de fichiers, notamment les fichiers texte (.txt) et binaires (.bin).

→ Les fichiers texte contiennent des caractères lisibles par un humain, tandis que les fichiers binaires stockent des données sous forme brute (ex. images, audio, bases de données).



Manipulation de fichiers

Python offre plusieurs méthodes pour interagir avec les fichiers, notamment **`open()`** pour ouvrir un fichier et **`close()`** pour le fermer après utilisation.

→ Ouverture d'un fichier

- La fonction **`open()`** prend en paramètre *le nom du fichier* et le *mode d'ouverture* :

```
f = open("mon_fichier.txt", "r") # Ouvre en mode lecture
```

□ Modes d'ouverture courants :

- **r** : Lecture (erreur si le fichier n'existe pas)
- **w** : Écriture (crée le fichier s'il n'existe pas et écrase son contenu s'il existe)
- **a** : Ajout à la fin du fichier (le fichier est créé s'il n'existe pas)
- **x** : Création (génère une erreur si le fichier existe déjà)
- **b** : Mode binaire (ex. : images, audio)

→ Fermeture d'un fichier

- Il est recommandé de toujours fermer un fichier après usage pour éviter les fuites de mémoire :

```
f.close()
```

Manipulation de fichiers

→ Lecture d'un fichier

- On peut lire un fichier de plusieurs manières :

```
# Lire tout le contenu
txt = f.read()

# Lire ligne par ligne
ligne = f.readline()

# Lire toutes les lignes dans une liste
lignes = f.readlines()
```

- Il est recommandé d'utiliser **with open()** pour une meilleure gestion des ressources :

```
with open("mon_fichier.txt", "r") as f:
    contenu = f.read()
```

Cette approche garantit que le fichier sera automatiquement fermé après utilisation.

Manipulation de fichiers

→ Écriture dans un fichier

On utilise le mode "**w**" ou "**a**" pour écrire dans un fichier :

```
with open("mon_fichier.txt", "w") as f:  
    f.write("Bonjour, Python!\n")
```

➔ Si le fichier n'existe pas, Python le crée automatiquement.

→ Écriture de plusieurs lignes

```
with open("mon_fichier.txt", "w") as f:  
    f.writelines(["Ligne 1\n", "Ligne 2\n", "Ligne 3\n"])
```

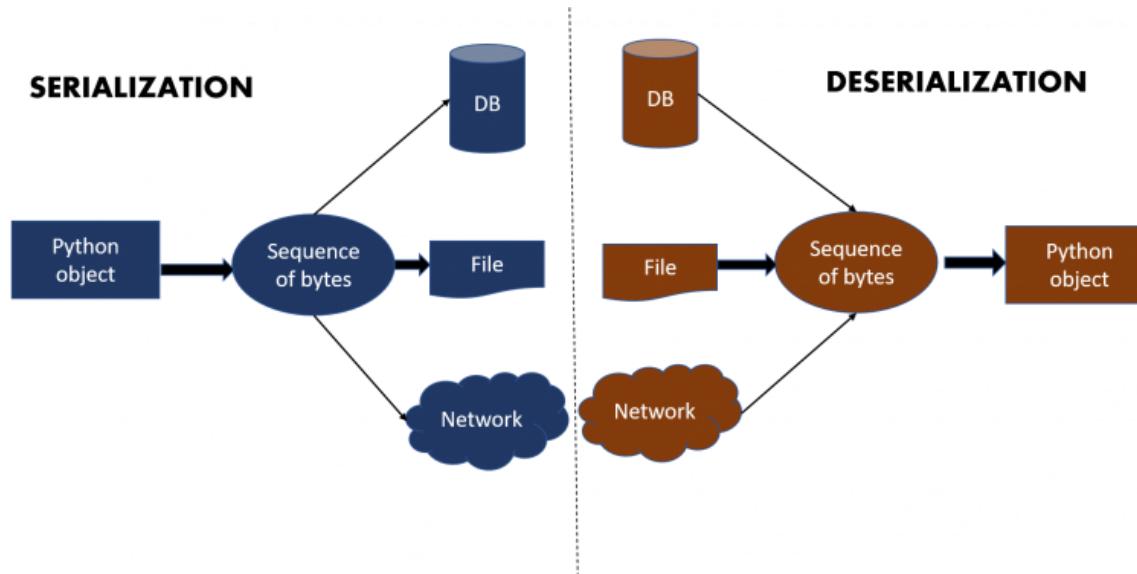
Manipulation de fichiers

→ Sérialisation et désérialisation des données

- La **sérialisation** est le processus de conversion d'un objet Python en un format qui peut être stocké dans un fichier ou transmis sur un réseau.
- La **désérialisation** est l'opération inverse, consistant à reconstituer l'objet à partir du fichier ou du flux de données.

Python propose deux principaux modules pour la sérialisation :

- **pickle** : utilisé pour stocker des objets Python sous forme binaire.
- **json** : utilisé pour convertir des objets en une représentation textuelle lisible et compatible avec d'autres langages de programmation.



Manipulation de fichiers

→ Sérialisation et désérialisation des données

□ Enregistrer un objet avec JSON

```
import json

donnees = {"nom": "Sara", "age": 19, "ville": "Beni Mellal"}
with open("donnees.json", "w") as f:
    json.dump(donnees, f) # Sérialisation en JSON
```

□ Charger un objet sérialisé en JSON

```
with open("donnees.json", "r") as f:
    donnees_chargees = json.load(f) # Désérialisation depuis JSON
    print(donnees_chargees)
```

Manipulation de fichiers

→ Sérialisation et désérialisation des données

□ Enregistrer un objet avec pickle

```
import pickle

donnees = {"nom": "Alice", "age": 25, "ville": "Paris"}
with open("donnees.pkl", "wb") as f:
    pickle.dump(donnees, f) # Sérialisation de l'objet
```

□ Charger un objet sérialisé en pickle

```
with open("donnees.pkl", "rb") as f:
    donnees_chargees = pickle.load(f) # Désérialisation de l'objet
    print(donnees_chargees)
```

Cette méthode est utile pour stocker et récupérer des objets complexes sans avoir à les convertir en texte.

Manipulation de fichiers

→ Vérification de l'existence du fichier avant ouverture

Une bonne pratique consiste à vérifier si un fichier existe avant de l'ouvrir pour éviter les erreurs :

Exemple

```
import os
if os.path.exists("mon_fichier.txt"):
    with open("mon_fichier.txt", "r") as f:
        print(f.read())
else:
    print("Le fichier n'existe pas.")
```

Le module `os` permet d'interagir avec le système d'exploitation, notamment pour vérifier l'existence d'un fichier.

- ➔ Lors de la manipulation des fichiers, plusieurs erreurs peuvent survenir, notamment si un fichier est inexistant, s'il y a un problème d'accès, ou si une erreur d'écriture ou de lecture se produit.

Gestion des exceptions

En Python, les exceptions permettent de gérer les erreurs qui peuvent survenir lors de l'exécution du programme. Elles empêchent l'arrêt brutal du programme et permettent d'afficher des messages d'erreur appropriés.

□ Syntaxe de base des exceptions

- **try**: Contient le code susceptible de provoquer une erreur.
- **except**: Capture l'erreur et exécute un code de secours.
- **finally**: Exécute du code, qu'il y ait eu une exception ou non.
- **raise**: Permet de générer une exception manuellement.

Exemple

```
try:  
    x = 10 / 0 # Provoque une erreur de division par zéro  
except ZeroDivisionError:  
    print("Erreur : Division par zéro !")  
finally:  
    print("Bloc finally exécuté, quel que soit le résultat.")
```

Gestion des exceptions

Lors de la manipulation des fichiers, plusieurs erreurs peuvent survenir, notamment si un fichier est inexistant, s'il y a un problème d'accès, ou si une erreur d'écriture ou de lecture se produit.

→ Gestion des erreurs courantes

On peut gérer ces erreurs à l'aide des exceptions avec le bloc **try-except** :

```
try:  
    with open("inexistant.txt", "r") as f:  
        contenu = f.read()  
except FileNotFoundError:  
    print("Erreur : Le fichier n'existe pas.")  
except PermissionError:  
    print("Erreur : Permission refusée.")  
except IOError:  
    print("Erreur d'entrée/sortie lors de la manipulation du fichier.")
```

Dans cet exemple :

FileNotFoundException se produit si le fichier n'existe pas.

PermissionError survient si l'on n'a pas les droits d'accès au fichier.

IOError couvre d'autres erreurs d'entrée/sortie (lecture/écriture).

Gestion des exceptions

→ Utilisation de **finally**

Le bloc **finally** permet de s'assurer que le fichier est bien fermé, même si une exception se produit :

Exemple:

```
try:  
    f = open("mon_fichier.txt", "r")  
    contenu = f.read()  
except Exception as e:  
    print(f"Une erreur est survenue : {e}")  
finally:  
    f.close()  
    print("Fichier fermé correctement.")
```

Gestion des exceptions

→ Utilisation de **raise**

Le mot-clé **raise** permet de lever une exception manuellement lorsqu'une condition spécifique est rencontrée.

Exemple:

```
x = -1
if x < 0:
    raise ValueError("x ne peut pas être négatif")
```

→ Définition

La **programmation modulaire** consiste à diviser un programme en plusieurs fichiers ou modules réutilisables afin d'améliorer la clarté et la maintenabilité du code.

→ Crédit et importation de modules

On peut créer un module Python en sauvegardant des fonctions dans un fichier **.py**.

Exemple : *mon_module.py*

```
def saluer(nom):
    return f"Bonjour, {nom}!"
```

→ Importation dans un autre script :

```
import mon_module
print(mon_module.saluer("Sara"))
```

→ On peut aussi importer une fonction spécifique :

```
from mon_module import saluer
print(saluer("Sara"))
```

→ Utilisation des bibliothèques standards

Python propose de nombreuses bibliothèques standards comme *math*, *os*, *random*, *sys*, etc.

Exemple d'utilisation de math

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

4.0

Exemple d'utilisation de os pour manipuler des fichiers

```
import os  
if os.path.exists("mon_fichier.txt"):  
    os.remove("mon_fichier.txt") # Supprime le fichier
```

Bonnes pratiques de structuration du code

- Utiliser des noms de variables explicites pour améliorer la lisibilité
- Diviser le code en fonctions et modules pour éviter la redondance
- Utiliser `if __name__ == "__main__"` pour exécuter uniquement le script principal

```
def main():
    print("Ce script s'exécute directement")

if __name__ == "__main__":
    main()

Ce script s'exécute directement
```

- Documenter le code avec des *commentaires* et des *docstrings*
- Respecter la **PEP8** (*Python Enhancement Proposal 8*), la convention de style de code en Python qui définit les règles de lisibilité et de formatage du code (indentation, nommage, longueur des lignes, etc.). Ce document est disponible sur le site officiel de Python : [PEP8](#).

Exercice 1

1. Crée un fichier ***donnees.txt***.
2. Écris trois lignes de texte dans ce fichier.
3. Ouvre le fichier en mode lecture.
4. Affiche son contenu ligne par ligne.
5. Ferme le fichier après lecture.

Exercice 2

1. Demander à l'utilisateur de saisir un nom de fichier.
2. Vérifier si ce fichier existe.
 - Si le fichier existe, l'ouvrir en mode lecture.
 - Afficher son contenu ligne par ligne.
 - Si le fichier n'existe pas, afficher un message d'erreur.

Exercice 3

1. Créer un fichier operations.py contenant :
 - Une fonction addition(a, b) qui retourne la somme de a et b.
 - Une fonction soustraction(a, b) qui retourne la différence entre a et b.
2. Créer un fichier main.py qui :
 - Importe le module operations.py.
 - Teste les deux fonctions avec des valeurs de ton choix.
 - Affiche les résultats à l'écran.