La manipulation de fichiers

En Python, la manipulation de fichiers repose sur la fonction intégrée `open()`, qui permet d'ouvrir un fichier et de choisir le mode d'accès (lecture, écriture, ajout, etc.). L'ouverture d'un fichier se fait souvent avec le mot-clé `with`, qui garantit la fermeture automatique du fichier, même en cas d'erreur.

```
chemin = r"/Users/VotreNom/Documents/mon_fichier.txt"
print(f"Chemin du fichier : {chemin}")

# Ouvrir un fichier en mode écriture et y écrire trois lignes de texte
with open(chemin, 'w') as fichier:
    fichier.write("Ligne 1 : Ceci est la première ligne.\n")
    fichier.write("Ligne 2 : Ceci est la deuxième ligne.\n")
    fichier.write("Ligne 3 : Ceci est la troisième ligne.\n")
```

Ici, le mode `'w'` signifie « écriture ». S'il existe déjà un fichier à cet emplacement, son contenu est écrasé. Ensuite, on peut lire le contenu avec le mode `'r'` :

```
# Ouvrir le fichier en mode lecture
with open(chemin, 'r') as fichier:
    contenu = fichier.read()
    print("Contenu du fichier :")
    print(contenu)
```

La méthode `read()` lit tout le fichier en une seule fois. Si l'on souhaite lire ligne par ligne, on utilise `readlines()` qui renvoie une liste :

```
with open(chemin, 'r') as fichier:
    lignes = fichier.readlines()
    print("Liste des lignes :")
    print(lignes)
```

Le curseur de lecture se déplace automatiquement pendant la lecture. On peut le repositionner au début du fichier grâce à `seek(0)` :

```
with open(chemin, 'r') as fichier:
    fichier.seek(0)  # Remet le curseur au début
    contenu = fichier.read()
    print("Contenu après repositionnement du curseur :")
    print(contenu)
```

On peut aussi lire seulement une portion du fichier, par exemple les dix premiers caractères :

```
with open(chemin, 'r') as fichier:
    fichier.seek(0)
    premiers_caracteres = fichier.read(10)
    print(f"Les 10 premiers caractères : {premiers_caracteres}")
```

Cette capacité de lecture partielle est utile pour le traitement de gros fichiers.

Sérialisation et désérialisation

La **sérialisation** et la **désérialisation** sont deux notions essentielles en informatique, qui permettent de faire circuler et de sauvegarder des données de manière durable ou transportable.

En Python, elles sont particulièrement importantes dès qu'on souhaite conserver l'état d'un programme, échanger des informations entre systèmes ou enregistrer des données d'un projet (comme dans le TP de gestion des patients du syllabus).

Qu'est-ce que la sérialisation ?

La sérialisation consiste à **convertir un objet Python en une suite de caractères ou d'octets** pouvant être enregistrée dans un fichier, envoyée sur un réseau ou stockée dans une base de données.

L'idée est simple : un objet Python (dictionnaire, liste, classe, etc.) existe seulement en mémoire pendant l'exécution du programme. Si le programme s'arrête, ces objets disparaissent. Pour les conserver, il faut les **transformer dans un format pérenne** (texte ou binaire).

Prenons un exemple :

```
import json

settings = {
    "fontsize": 12,
    "theme": "dark",
    "autosave": True
}

# Sérialisation : conversion du dictionnaire Python en chaîne JSON
chaine_json = json.dumps(settings, indent=4)
print(chaine_json)
```

Résultat affiché dans le terminal :

```
{
    "fontsize": 12,
    "theme": "dark",
    "autosave": true
}
```

Ici, `json.dumps()` (le « s » de *string*) transforme le dictionnaire en une **chaîne JSON**. Cette chaîne peut ensuite être **écrite dans un fichier** pour être conservée :

```
with open('settings.json', 'w') as fichier_json:
    json.dump(settings, fichier_json, indent=4)
```

Le fichier `settings.json` contient maintenant une version textuelle du dictionnaire Python. On dit que l'objet a été **sérialisé** dans un format standard, compréhensible par d'autres langages (JavaScript, Java, etc.), ce qui facilite l'échange de données entre programmes hétérogènes.

Qu'est-ce que la désérialisation ?

La désérialisation est l'opération inverse. Elle consiste à **reconstruire un objet Python à partir d'une représentation stockée ou transmise**.

Ainsi, lorsqu'on relit le fichier `settings.json`, Python doit convertir le texte du fichier (le JSON) en véritable dictionnaire Python.

```
# Désérialisation : lecture et reconstruction de l'objet Python
with open('settings.json', 'r') as fichier_json:
    settings = json.load(fichier_json)

print(settings)
```

Résultat :

```
{'fontsize': 12, 'theme': 'dark', 'autosave': True}
```

L'objet `settings` est à nouveau un dictionnaire Python utilisable dans le programme. Autrement dit, la désérialisation **ramène à la vie** les objets enregistrés.

Pourquoi utiliser la sérialisation ?

La sérialisation intervient dans de nombreux contextes :

1. **Sauvegarde de l'état d'une application** Exemple : un jeu qui sauvegarde la progression du joueur dans un fichier JSON.
2. **Communication entre programmes** Par exemple, un serveur web écrit en Python peut envoyer des données JSON à un client JavaScript.
3. **Stockage temporaire dans un fichier ou cache** Certaines applications sauvegardent leurs paramètres, préférences ou résultats dans un format sérialisé.
4. **Transfert de données sur un réseau** Les API REST échangent presque toujours des données au format JSON, qui est le fruit d'une sérialisation.

Autres formats possibles

Bien que **JSON** soit le format le plus courant (lisible par l'humain, simple à manipuler), Python propose d'autres méthodes selon les besoins :

- **Pickle** Le module `pickle` permet de sérialiser des objets Python plus complexes (comme des classes, fonctions, etc.). Cependant, il ne doit jamais être utilisé avec des données non fiables, car il peut exécuter du code arbitraire lors de la désérialisation.

  Exemple :

  ```
  import pickle

  data = {"nom": "Dupont", "age": 35}
  with open('data.pkl', 'wb') as fichier:
      pickle.dump(data, fichier)  # Sérialisation binaire

  with open('data.pkl', 'rb') as fichier:
      donnees = pickle.load(fichier)  # Désérialisation
      print(donnees)
  ```

- **CSV** Pour des données tabulaires (listes ou tableaux), le format CSV (*Comma-Separated Values*) est souvent utilisé. On peut le lire et l'écrire facilement avec le module `csv`.

- **YAML, XML, Avro, Protobuf** Utilisés dans des contextes spécifiques (configuration, interopérabilité, systèmes distribués).