Structures avancées et gestion des fichiers

Python B2 - Jour 2 - 2025/2026

Structures de données avancées et gestion des fichiers - De l'analyse de données à la persistance

Colin Tenaguillo
formateur_colin.tenaguilo@supdevinci-edu.fr

Plan de la journée

1. **Structures avancées**

   Listes imbriquées, dictionnaires complexes et leurs manipulations

2. **Compréhensions**

   Syntaxe concise pour créer listes et dictionnaires avec filtrage

3. **Fonctions lambda**

   Fonctions anonymes avec map, filter et sorted

4. **Gestion fichiers**

   Formats CSV, JSON et pickle pour la persistance des données

Listes imbriquées : Matrices et tableaux

Création d'une matrice

```
# Matrice 2x3
matrice = [[1, 2, 3], [4, 5, 6]]
print(matrice[1][2])  # 6

# Accès avec boucles
for ligne in matrice:
    for element in ligne:
        print(element)
```

Les matrices permettent de créer des structures bidimensionnelles comme des matrices ou des tableaux de données.

Dictionnaires complexes

```
# Dictionnaire avec valeurs complexes
etudiant = {
    "nom": "Dupont",
    "notes": [15, 12, 18],
    "adresse": {
        "rue": "123 Main St",
        "ville": "Paris"
    }
}
print(etudiant["adresse"]["ville"])  # Paris
```

Les dictionnaires peuvent contenir des listes, d'autres dictionnaires, créant des structures de données riches pour modéliser des informations complexes.

Compréhensions de listes

Méthode traditionnelle

```
carres = []
for i in range(10):
 carres.append(i**2)
```

Compréhension simple

```
carres = [i**2 for i in range(10)]
```

Avec condition

```
pairs = [i for i in range(20) if i % 2 == 0]
```

Les compréhensions offrent une syntaxe concise et lisible pour créer des listes à partir d'autres itérables.

Compréhensions de dictionnaires

Création depuis une liste

```
noms = ["Alice", "Bob", "Charlie"]
longueurs = {nom: len(nom) for nom in noms}
# {'Alice': 5, 'Bob': 3, 'Charlie': 7}
```

Filtrage avancé

```
notes = {"Alice": 15, "Bob": 8, "Charlie": 12}
admis = {nom: note for nom, note
        in notes.items() if note >= 10}
```

Même principe que pour les listes, mais pour créer des dictionnaires avec une syntaxe élégante et des conditions.

Fonctions lambda : L'art de la concision

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

# Utilisation avec map
nombres = [1, 2, 3, 4]
carres = list(map(lambda x: x**2, nombres))
# [1, 4, 9, 16]
```

Les fonctions lambda sont des fonctions anonymes parfaites pour des opérations simples utilisées avec map, filter ou sorted.

filter() et sorted() : Filtrage et tri avancés

1

2

filter() - Filtrage

```
nombres = [1, 2, 3, 4, 5, 6]
pairs = list(filter(lambda x: x % 2 == 0,
nombres))
# [2, 4, 6]
```

sorted() - Tri personnalisé

```
etudiants = [{"nom": "Alice", "note": 15},
 {"nom": "Bob", "note": 12}]
tri_notes = sorted(etudiants,
 key=lambda e: e["note"],
 reverse=True)
```

TP Pratique : Analyse de données étudiants

```
# Données de départ
etudiants = [
    {"nom": "Alice", "note": 15, "annee": 2},
    {"nom": "Bob", "note": 12, "annee": 1},
    {"nom": "Charlie", "note": 18, "annee": 2}
]
```

1. Filtrer les étudiants admis (note ≥ 12)
2. Calculer la moyenne par année
3. Créer un dictionnaire nom → mention

Gestion des fichiers

CSV, JSON, pickle : Trois approches pour sauvegarder et charger vos données Python de manière persistante.

Fichiers CSV : Lecture de données tabulaires

```
import csv

with open('data.csv', 'r', newline='', encoding='utf-8') as file:
    reader = csv.DictReader(file)
    for line in reader:
        print(line['nom'], line['age'])
```

Le module csv permet de lire facilement des fichiers tabulaires. DictReader transforme chaque ligne en dictionnaire avec les en-têtes comme clés.

Fichiers CSV : Écriture de données

```
import csv

donnees = [
    {'nom': 'Alice', 'age': 25},
    {'nom': 'Bob', 'age': 30}
]

with open('output.csv', 'w', newline='',
          encoding='utf-8') as fichier:
    writer = csv.DictWriter(fichier,
                            fieldnames=['nom', 'age'])
    writer.writeheader()
    writer.writerows(donnees)
```

Bonnes pratiques :

- Utilisez newline='' pour éviter les lignes vides
- Spécifiez l'encodage UTF-8
- writeheader() ajoute les en-têtes

Format JSON : Sérialisation et désérialisation

Sérialisation Python → JSON

```
import json

data = {"nom": "Alice", "age": 25}
json_str = json.dumps(data, indent=2)

# Écriture dans fichier
with open('data.json', 'w') as f:
    json.dump(data, f, indent=2)
```

Désérialisation JSON → Python

```
import json

# Lecture depuis fichier
with open('data.json', 'r') as f:
 data = json.load(f)

# Depuis chaîne JSON
json_str = '{"nom": "Alice", "age": 25}'
data = json.loads(json_str)
```

Module pickle : Sérialisation binaire complète

```
import pickle

# Sauvegarde d'objet Python
with open('data.pkl', 'wb') as f:  # 'wb' pour écriture binaire
    pickle.dump(mon_objet_complexe, f)

# Chargement
with open('data.pkl', 'rb') as f:  # 'rb' pour lecture binaire
    mon_objet = pickle.load(f)
```

Pickle peut sérialiser presque tous les objets Python (listes, dictionnaires, classes personnalisées), mais les fichiers ne sont lisibles que par Python.

TP Final : Mini-gestionnaire de données

Objectif : Étendre la bibliothèque avec persistance

Sauvegarde/chargement JSON

Implémenter la persistance des données de la bibliothèque au format JSON lisible

Export CSV du catalogue

Créer une fonction d'export du catalogue complet vers un fichier CSV

Gestion des erreurs

Gérer les cas d'erreur : fichier inexistant, format invalide, permissions insuffisantes