Lab 2 : Structures de Données Avancées

Overview

Bienvenue ! Aujourd'hui, nous plongeons dans les structures de données fondamentales de Python. L'objectif est de maîtriser les nuances de chaque type et de comprendre quand utiliser lequel.

Rappel important :

- **Listes (`list`) :** Mutables, ordonnées, permettent les doublons.
- **Tuples (`tuple`) :** Immutables, ordonnées, souvent hétérogènes.
- **Sets (`set`) :** Mutables, non ordonnés, **unicité** des éléments (pas de doublons).
- **Dictionnaires (`dict`) :** Mutables, paires clé-valeur (les clés sont uniques comme un set).

Partie 1 : Listes et Tuples (Les Séquences)

Investigating Lists

Les listes sont dynamiques, mais cela a un coût. Expérimentons leur comportement.

Exercice 1 : Slicing Avancé Le "slicing" (découpage) est un outil puissant. Prédisez le résultat de chaque opération.

```
lst = list(range(10)) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(lst[2:5])
print(lst[:3])
print(lst[::2])   # Le "step"
print(lst[::-1])  # Astuce classique
print(lst[7:2:-1]) # Que se passe-t-il ici ?
```

Exercice 2 : Copie de Liste Considérez le code suivant. Pourquoi `b` est-il modifié "en profondeur" ?

```
a = [1, 2, [3, 4]]
b = a[:] # Ceci est une "shallow copy"
b[0] = 99
b[2][0] = 99
print(a) # Que vaut a ? Pourquoi le premier élément n'a-t-il pas changé mais le
troisième oui ?
```

Indice : Recherchez "Shallow copy vs Deep copy".

Problème de codage : Suppression de doublons Écrivez une fonction `remove_duplicates(lst)` qui prend une liste et retourne une nouvelle liste sans doublons, en préservant l'ordre d'origine. *Contrainte : Essayez d'utiliser un `set` pour l'efficacité, mais attention à l'ordre !*

```python
def remove_duplicates(lst):
    # Votre code ici
    pass

remove_duplicates([1, 2, 2, 3, 1, 4]) # => [1, 2, 3, 4]
```

Partie 2 : Sets (Ensembles)

Les sets sont des collections non ordonnées d'éléments uniques. Ils sont incroyablement efficaces pour tester l'appartenance (`in`) et effectuer des opérations mathématiques.

Investigating Sets

Exercice 1 : Création et Unicité Que se passe-t-il si on crée un set avec des doublons ?

```python
s = {1, 2, 2, 3, 3, 3}
print(s)
# Question : Que vaut len(s) ?
```

Attention : Un set vide ne se déclare pas avec `{}` (ceci crée un dictionnaire vide).

```python
empty_dict = {}
empty_set = set()
print(type(empty_dict))
print(type(empty_set))
```

Exercice 2 : Mutabilité et Hash Les éléments d'un set doivent être "hashables" (immutables). Lequel de ces codes provoque une erreur ?

```python
# Cas A
s = {1, 2, (3, 4)} # Un tuple est immutable

# Cas B
s = {1, 2, [3, 4]} # Une liste est mutable
```

Opérations sur les Sets

Imaginez deux ensembles de données : les étudiants qui suivent le cours de Python et ceux qui suivent le cours de Java.

```python
python_students = {"Alice", "Bob", "Charlie", "David"}
java_students = {"Charlie", "David", "Eve", "Frank"}
```

Pour chaque problème ci-dessous, écrivez l'opération set correspondante (souvent un seul opérateur) :

1. **Union :** Tous les étudiants uniques qui suivent au moins un cours. (Opérateur `|`)
2. **Intersection :** Étudiants qui suivent **les deux** cours. (Opérateur `&`)
3. **Différence :** Étudiants en Python mais **pas** en Java. (Opérateur `-`)
4. **Différence Symétrique :** Étudiants qui suivent un seul cours, mais pas les deux. (Opérateur `^`)

Testez vos hypothèses :

```
print(python_students | java_students)
print(python_students & java_students)
print(python_students - java_students)
print(python_students ^ java_students)
```

Problème de codage : Analyse de Texte

Écrivez une fonction `compare_texts(text1, text2)` qui retourne un tuple contenant :

1. Le nombre de mots uniques communs aux deux textes.
2. Le nombre de mots présents dans le premier texte mais pas dans le second.

Note : Convertissez les textes en minuscules et ignorez la ponctuation basique pour simplifier.

```
def compare_texts(t1, t2):
    # Convertir en liste de mots (split), puis en sets
    pass

t1 = "Le chat mange la souris"
t2 = "Le chien mange le chat"
# Mots communs : {'le', 'chat', 'mange'} -> 3
# Mots uniq t1 : {'la', 'souris'} -> 2
```

Partie 3 : Dictionnaires (Les Mappings)

Investigating Dictionaries

Exercice 1 : Accès aux clés Quelle est la différence entre `d['key']` et `d.get('key')` ? Testez :

```
d = {"a": 1, "b": 2}
print(d['c'])       # Erreur ?
print(d.get('c'))   # Résultat ?
print(d.get('c', 0))# Résultat ?
```

Problème de codage : Inversion de Dictionnaire Inversez un dictionnaire. Si plusieurs clés ont la même valeur, regroupez les clés dans un set (plus approprié qu'une liste ici pour l'unicité).

```
def invert_dict_to_set(d):
    pass

invert_dict_to_set({'a': 1, 'b': 2, 'c': 1})
# => {1: {'a', 'c'}, 2: {'b'}}
```

Partie 4 : Mélange et Structures Imbriquées

Investigating Nested Data

Exercice 1 : Liste de Dictionnaires (Style JSON) Voici une liste d'étudiants. Écrivez une compréhension de liste pour extraire uniquement les noms des étudiants ayant une note supérieure à 15.

```
students = [
    {"name": "Alice", "grade": 18},
    {"name": "Bob", "grade": 12},
    {"name": "Charlie", "grade": 16},
    {"name": "David", "grade": 10}
]

# Résultat attendu : ["Alice", "Charlie"]
```

Exercice 2 : Dictionnaire de Sets Vous avez un dictionnaire représentant des groupes d'amis. Écrivez une fonction pour vérifier si deux personnes sont dans le même groupe (amis communs).

```
groups = {
    "sport": {"Alice", "Bob", "Charlie"},
    "cinema": {"David", "Eve", "Alice"}
}

def are_in_same_group(person1, person2, groups_dict):
    # Retourne True si un groupe contient les deux personnes
    pass
```

Partie 5 : Challenges (Bonus)

Challenge 1 : Déduplication avancée avec Set

Écrivez une fonction qui prend une liste de tuples représentant des coordonnées `(x, y)` et supprime les doublons. *Note : Les tuples sont hashables, donc on peut les mettre directement dans un set.*

```
coords = [(0, 0), (1, 0), (0, 0), (2, 1)]
# Résultat : {(0, 0), (1, 0), (2, 1)} ou une liste équivalente
```

Challenge 2 : Grouper par propriété

Écrivez une fonction `group_by_length(words)` qui prend une liste de mots et retourne un dictionnaire où les clés sont les longueurs des mots et les valeurs sont des **sets** de mots de cette longueur (pour éviter les doublons de mots).

```
def group_by_length(words):
    pass

words = ["apple", "bat", "bar", "code", "pie", "bat"]
group_by_length(words)
# => {5: {'apple'}, 3: {'bat', 'bar', 'pie'}, 4: {'code'}}
# Notez que 'bat' n'apparaît qu'une fois dans le set.
```

Challenge 3 : L'Anagramme Géant

Étant donné une liste de mots, créez un dictionnaire qui regroupe tous les anagrammes ensemble. *Indice : Comment "normaliser" un mot pour qu'il serve de clé commune à tous ses anagrammes ? (Pensez à trier les lettres et utiliser un tuple).*

```
words = ["listen", "silent", "enlist", "google", "goggle"]
# Résultat attendu :
# {('e', 'i', 'l', 'n', 's', 't'): {'listen', 'silent', 'enlist'}, ...}
```

Partie 6 : Challenges Avancés (Listes & Dictionnaires)

Ces exercices visent à consolider votre maîtrise des algorithmes de base tout en exploitant la puissance des structures Python.

Challenge 4 : Transposition de Matrice (Listes de Listes)

En mathématiques et en data science, on a souvent besoin de "transposer" une matrice (transformer les lignes en colonnes). Écrivez une fonction `transpose(matrix)` qui prend une liste de listes (une matrice) et retourne sa transposée.

Interdit d'utiliser la librairie `numpy` !

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

```
# Résultat attendu (les lignes deviennent des colonnes) :
# [
#   [1, 4, 7],
#   [2, 5, 8],
#   [3, 6, 9]
# ]

# Indice : La fonction zip() peut faire cela en une ligne si vous savez l'utiliser
avec l'opérateur *.
```

Challenge 5 : Tri de Dictionnaire par Valeur

Les dictionnaires ne sont pas naturellement triés. Souvent, on récupère des données (ex: scores, prix) et on veut les classer. Écrivez une fonction `sort_dict_by_value(d, reverse=False)` qui retourne une liste de tuples `(clé, valeur)` triés par valeur.

```
scores = {"Alice": 85, "Bob": 92, "Charlie": 78, "David": 95}

# Résultat ascendant (du plus petit au plus grand) :
# [('Charlie', 78), ('Alice', 85), ('Bob', 92), ('David', 95)]

# Indice : Regardez du côté de la fonction sorted() et de l'argument key=lambda
...
```

Challenge 6 : Le problème "Two Sum" (Listes vs Dictionnaires)

C'est un grand classique. Étant donné une liste de nombres et un nombre cible (`target`), trouvez les indices des deux nombres de la liste qui s'additionnent pour donner la cible.

On suppose qu'il y a une unique solution et qu'on ne peut pas utiliser le même élément deux fois.

```
nums = [2, 7, 11, 15]
target = 9

# Résultat attendu : [0, 1] (car nums[0] + nums[1] == 9)
```

Niveau 1 (Brute Force) : Essayez d'abord avec deux boucles `for` imbriquées. Complexité : $O(n^2)$. **Niveau 2 (Optimisé) :** Essayez de le faire en un seul passage sur la liste en utilisant un **dictionnaire** pour stocker les nombres déjà vus. Complexité : $O(n)$.

Challenge 7 : Accès Sécurisé aux Dictionnaires Imbriqués

Lorsqu'on travaille avec des données JSON complexes, on a souvent des dictionnaires dans des dictionnaires. Écrire des `if` imbriqués pour vérifier l'existence des clés est fastidieux. Écrivez une fonction `safe_get(data, keys)` qui permet de récupérer une valeur dans une structure imbriquée sans provoquer d'erreur si une clé n'existe pas.

```
data = {
    "user": {
        "id": 1,
        "details": {
            "name": "Alice",
            "city": "Paris"
        }
    }
}

# safe_get(data, ["user", "details", "name"]) => "Alice"
# safe_get(data, ["user", "details", "age"])  => None (au lieu de KeyError)
# safe_get(data, ["user", "admin", "rights"]) => None (au lieu de KeyError)
```

Indice : Utilisez une boucle et un `try/except` ou la méthode `.get()` astucieusement.

Challenge 8 : Fusion de Dictionnaires avec Comptage

Vous avez deux dictionnaires représentant des stocks de fruits. Fusionnez-les pour obtenir le stock total. Si une clé existe dans les deux, additionnez les valeurs.

```
stock_a = {"apple": 5, "banana": 2, "orange": 8}
stock_b = {"banana": 3, "orange": 1, "pear": 4}

# Résultat attendu :
# {"apple": 5, "banana": 5, "orange": 9, "pear": 4}
```

Indice : En Python "moderne" (3.9+), l'opérateur `|` fusionne les dicts, mais il garde la valeur de droite en cas de conflit (écrasement). Ici, on veut une addition. Une boucle ou `collections.Counter` est nécessaire.