Rappels sur les dictionnaires en Python

Structures de données

1 Introduction

Plutôt qu'accéder à une valeur à partir de son indice (comme dans un tableau), on peut souhaiter y accéder à partir d'une **clé**.

Par exemple dans un annuaire téléphonique, on accède à un numéro de téléphone à partir d'un nom. Les noms sont donc les clés, les numéros de téléphone les valeurs. Dans un dictionnaire (au sens traditionnel), on recherche une définition à partir d'un mot. Les mots sont les clés et les définitions les valeurs.

L'accès à la valeur ainsi que la modification ou l'ajout d'une valeur doivent être possibles sans parcourir toute la structure. On parle aussi de **tableaux associatifs**.

Un **dictionnaire** est un ensemble de couples **(clé, valeur)** : on accède à une valeur par sa clé et non par sa position (l'ordre d'insertion est conservé, mais il ne joue aucun rôle). On peut ajouter des couples ; si la clé figure déjà dans le dictionnaire alors le couple est remplacé par le nouveau. Un dictionnaire Python est un objet **mutable** (comme les listes mais contrairement aux chaînes de caractères ou aux tuples).

2 Création de dictionnaires

Plusieurs méthodes permettent de créer un dictionnaire.

2.1 Dictionnaire vide

```
d = dict()
d = {}
print(d)
```

2.2 À partir d'un littéral

```
boxoffice = {'Aladdin': 569439,
             'Godzilla 2': 342574,
             'Rocketman': 277017}
print(boxoffice)
```

2.3 À partir d'une liste de couples

La fonction `dict` transforme une liste de couples `(clé, valeur)` en dictionnaire :

```
couples = [('Aladdin', 569439), ('Godzilla 2', 342574), ('Rocketman', 277017)]
boxoffice = dict(couples)
print(boxoffice)
```

2.4 Par compréhension

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
double_dict1 = {k: v * 2 for (k, v) in dict1.items()}
print(double_dict1)
```

③ Accès aux valeurs, informations

3.1 Accès à une valeur par sa clé

```
print(boxoffice['Aladdin'])
```

3.2 Nombre d'enregistrements (longueur)

```
print(len(boxoffice) == 3)
```

3.3 Test d'appartenance d'une clé

```
print('Rocketman' in boxoffice)
```

④ Ajout, modification, suppression de valeurs

4.1 Ajout d'une valeur

Si la clé n'existe pas encore, l'affectation crée une nouvelle entrée :

```
boxoffice['Avengers'] = 0
print(len(boxoffice) == 4)
print(boxoffice['Avengers'])
```

4.2 Unicité des clés

Si la clé existe déjà, l'affectation **remplace** la valeur existante :

```
boxoffice['Avengers'] = 132260
print(len(boxoffice) == 4)
print(boxoffice['Avengers'])
```

4.3 Les valeurs peuvent ne pas être uniques

```
boxoffice['Mon Film'] = 132260
print(boxoffice['Avengers'])
print(boxoffice['Mon Film'])
```

4.4 Supprimer une valeur

```
del boxoffice['Mon Film']
print('Mon Film' in boxoffice)
```

5 Itérations

5.1 Itération sur les clés

Par défaut, itérer sur un dictionnaire parcourt les **clés** :

```
for film in boxoffice:
    print(f"{film} => {boxoffice[film]} spectateurs")
```

On obtient le même résultat avec `boxoffice.keys()`.

5.2 Itération sur les valeurs

```
for nbre_spectateurs in boxoffice.values():
    print(nbre_spectateurs)
```

5.3 Itération sur les paires (clé, valeur)

```
for film, nbre_spectateurs in boxoffice.items():
    print(f"{film} => {nbre_spectateurs} spectateurs")
```

6 Exceptions

6.1 Accès à une clé inexistante

Accéder à une clé absente provoque une erreur `KeyError` :

```
print(boxoffice['Mon Film'])
```

```
KeyError: 'Mon Film'
```

6.2 Capture d'exceptions

On peut intercepter l'erreur avec `try/except` :

```
def nombre_entrees(titre_film):
    try:
        return boxoffice[titre_film]
    except KeyError:
        return 0

```

```
print(nombre_entrees('Avengers'))
print(nombre_entrees('Mon film'))
```

6.3 La méthode `get`

La méthode `get` permet d'accéder à la valeur d'une clé sans risquer de `KeyError` :

```
print(boxoffice.get('Avengers'))
print(boxoffice.get('Mon Film') == None)
print(boxoffice.get('Mon Film', 0))
```

— `boxoffice.get(key)` est équivalent à `boxoffice[key]` si la clé existe ;

— `boxoffice.get(key)` renvoie `None` par défaut si la clé est absente ;

— `boxoffice.get(key, 0)` renvoie `0` si la clé est absente.

⑦ Recherche dans une liste et dans un dictionnaire

Lorsqu'on cherche une valeur dans une **liste**, il faut, dans le pire cas, parcourir tous les éléments un par un. La complexité est donc en $O(n)$.

```
def recherche_liste(element, liste):
    for e in liste:
        if e == element:
            return True
    return False

animaux = ['chat', 'chien', 'poule', 'vache', 'cochon']
print(recherche_liste('vache', animaux))
print(recherche_liste('tigre', animaux))
```

Lorsqu'on cherche une clé dans un **dictionnaire**, Python utilise une **table de hachage**. Une fonction de hachage transforme la clé en un indice de tableau, ce qui permet d'accéder **directement** à la valeur associée. La complexité est donc en $O(1)$ en moyenne.

```
ages = {'Alice': 17, 'Bob': 16, 'Charlie': 18}
print('Bob' in ages)
print('Diana' in ages)
```

On peut vérifier cette différence de performance par l'expérience :

```
import time

n = 10_000_000
grande_liste = list(range(n))
grand_dict = {i: None for i in range(n)}

t0 = time.time()
print(n - 1 in grande_liste)
t1 = time.time()
```

```
print(f"Liste : {t1 - t0:.4f} s")

t0 = time.time()
print(n - 1 in grand_dict)
t1 = time.time()
print(f"Dictionnaire : {t1 - t0:.6f} s")
```

La différence est spectaculaire : la recherche dans la liste prend quelques centièmes de seconde tandis que celle dans le dictionnaire prend environ une microseconde, soit un facteur de l'ordre de 10 000 pour $n = 10\,000\,000$. Ce facteur augmente avec la taille de la structure : la durée de la recherche dans le dictionnaire ne dépend pas de n, alors que celle de la recherche dans la liste est proportionnelle à n.

Structure	**Complexité de la recherche**	**Test `in`**
Liste	$O(n)$	Parcours séquentiel
Dictionnaire	$O(1)$ en moyenne	Table de hachage

Remarque. Le choix entre une liste et un dictionnaire a un impact majeur sur les performances d'un programme. Si l'on a besoin de tester fréquemment l'appartenance d'un élément, un dictionnaire est bien plus adapté qu'une liste.

⑧ Type des clés et valeurs

8.1 Les clés doivent être non mutables

Les clés d'un dictionnaire doivent être **hachables**, c'est-à-dire non mutables. Les types autorisés comme clés sont : chaînes de caractères, entiers, flottants, tuples (s'ils ne contiennent que des éléments non mutables).

Les entiers comme clés :

```
chiffres = {1: 'I', 10: 'X', 100: 'C'}
print(chiffres[10])
```

Les tuples comme clés :

```
grille = {(0, 0): 'T', (0, 1): 'S', (1, 0): ' ', (1, 1): 'T'}
print(grille[(0, 0)] == 'T')
```

Les listes ne sont **pas** autorisées comme clés car elles sont mutables :

```
d1 = {'s': 3, 8: 'chat', [1, 'chien']: (2, 7)}
```

```
TypeError: unhashable type: 'list'
```

En revanche, un tuple est hachable et fonctionne comme clé :

```
d2 = {'s': 3, 8: 'chat', (1, 'chien'): (2, 7)}
print(d2)
```

8.2 Les valeurs peuvent être de n'importe quel type

Les valeurs d'un dictionnaire peuvent être de n'importe quel type, y compris d'autres dictionnaires :

```
thon = {'espece': 'T', 'gestation': 3}
requin = {'espece': 'S', 'gestation': 2, 'energie': 2}
mer = {(0, 0): thon, (0, 1): None, (1, 0): None, (1, 1): requin}

print(mer[(0, 0)]['espece'])
print(mer[(1, 1)]['energie'])
```

Exercice 1 *Comptage d'occurrences* On dispose d'un texte sous forme de liste de mots. Écrire une fonction `compter(mots)` qui renvoie un dictionnaire associant chaque mot à son nombre d'occurrences.

```
texte = ['le', 'chat', 'mange', 'le', 'poisson', 'le', 'chat']
print(compter(texte))
```

Exercice 2 *Choix de la bonne structure* On souhaite vérifier si un mot appartient à un dictionnaire de la langue française contenant 300 000 mots.

1. Quelle structure de données choisir ? Justifier.
2. Écrire le code Python correspondant, en rangeant les mots comme clés d'un dictionnaire.
3. Quelle serait la complexité avec une liste ? Et avec un dictionnaire ?