6 Ensembles et dictionnaires

Dans le chapitre 3, vous avez rencontré deux types de *collections* : les listes et les tuples. Vous savez donc qu'il s'agit de collections hétérogènes ordonnées, indexées par des entiers positifs, l'une muable et l'autre immuable. Dans ce chapitre, nous allons découvrir deux nouveaux types de collections :

— les *ensembles*, qui sont des collections hétérogènes **non-ordonnées** et **sans élément en double**,

— les *dictionnaires*, qui sont des collections hétérogènes similaires aux listes, mais dont les éléments sont **indexés par des clés quelconques** et non pas obligatoirement des entiers positifs.

6.1 Ensembles

6.1.1 Définition et propriétés

Pour créer un ensemble :

— soit on saisit l'instruction `set()` pour créer un ensemble vide,

— soit on liste ses éléments entre accolades, séparés par des virgules s'ils sont plusieurs.

Un ensemble ne peut pas contenir de doublons. Si vous répétez plusieurs fois une même valeur lors de la définition de l'ensemble, celle-ci ne sera ajoutée qu'une seule fois à l'ensemble.

À tester. Testez le code ci-dessous et observez les résultats obtenus dans l'explorateur de variables.

```
S1 = set()
S2 = {1, 3, 8}
S3 = {"d", "a", "t", "d", "t", "A", "d"}
```

De combien d'éléments est constitué l'ensemble `S3` ?

Attention. Pour des raisons techniques, un ensemble ne peut pas contenir d'objets muables[1]. Vous pouvez vérifier par exemple que l'instruction `S = { 1, 8, [4, 5] }` provoque une erreur.

Un ensemble `S` possède les propriétés suivantes.

— Il n'est **pas** indiçable (*subscriptable*). Comme ses éléments ne sont pas ordonnés, il est impossible d'accéder à un élément en particulier, et une instruction telle que `S[0]` n'a donc pas de sens et provoque une erreur.

— Il est itérable : on peut parcourir tous ses éléments à l'aide de la syntaxe habituelle `for el in S`, sans garantie toutefois sur l'ordre dans lequel les éléments seront parcourus.

À tester. Essayez le code ci-dessous.

```
S = {18, 5, "abc", 9, (10, 12)}
for x in S:
    print(x)
```

1. En réalité, il y a même une contrainte plus stricte encore sur les éléments d'un ensemble que le lecteur curieux trouvera dans la section 6.3 en fin de ce chapitre.

— Il est muable : il faut donc se méfier des pièges classiques que l'on retrouve déjà avec les listes.

À tester. Dans l'explorateur de variables, observez le contenu des ensembles `A` et `B` à l'issue de chacun des programmes suivants.

```
A = {1, 2, 3}
B = {1, 2, 3}
B.add(4)
```

```
A = {1, 2, 3}
B = A
B.add(4)
```

6.1.2 Opérateurs et méthodes sur les ensembles

Si `A` et `B` sont des ensembles et `x` est un objet quelconque, alors on peut effectuer la plupart des opérations mathématiques usuelles sur les ensembles à l'aide des opérateurs donnés dans la table 6.1.

Opérateur	Type renvoyé	Opération mathématique	Notation math.
`x in A`	`bool`	Appartenance	$x \in A$
`x not in A`	`bool`	Non-appartenance	$x \notin A$
`A | B`	`set`	Union	$A \cup B$
`A & B`	`set`	Intersection	$A \cap B$
`A - B`	`set`	Différence	$A \setminus B$
`A ^ B`	`set`	Différence symétrique	$A \Delta B \overset{\text{déf.}}{=} (A \cup B) \setminus (A \cap B)$
`A == B`	`bool`	Égalité	$A = B$
`A != B`	`bool`	Non-égalité	$A \neq B$
`A <= B`	`bool`	Inclusion (au sens large)	$A \subset B$
`A < B`	`bool`	Inclusion stricte	$A \subsetneq B$
`A >= B`	`bool`	Inclusion (au sens large)	$A \supset B$
`A > B`	`bool`	Inclusion stricte	$A \supsetneq B$

TABLE 6.1 – Opérateurs sur les ensembles

Remarque. Les opérateurs `|`, `&`, `-` et `^` possèdent leurs équivalents `|=`, `&=`, `-=` et `^=`. Par exemple, `A &= B` est équivalent[2] à `A = A & B`.

De plus, la table 6.2 liste les méthodes supportées par le type `set`, permettant de modifier un ensemble après l'avoir créé.

2. À la différence près que dans `A &= B` l'objet lié à `A` est directement modifié en mémoire, alors que dans `A = A & B` un nouvel objet contenant le résultat de `A & B` est créé en mémoire et on redirige ensuite l'identifiant `A` vers cet objet (c'est le même phénomène que pour l'opérateur `+=` sur les listes expliqué dans la section 3.3).

Méthode	Description
`A.add(x)`	Ajoute l'élément `x` à l'ensemble `A`.
`A.remove(x)`	Retire l'élément `x` de l'ensemble `A`. Provoque une erreur si `A` ne contient pas l'élément `x`.
`A.discard(x)`	Retire l'élément `x` de l'ensemble `A`. Ne provoque pas d'erreur si `A` ne contient pas l'élément `x`.
`A.pop()`	Retire et renvoie un élément arbitraire de `A`. Provoque une erreur si l'ensemble `A` est vide.
`A.clear()`	Retire tous les éléments de `A`, qui devient donc un ensemble vide.
`A.update(B)`	Ajoute à `A` tous les éléments contenus dans `B`. *NB : c'est équivalent à* `A |= B`.

TABLE 6.2 – Méthodes du type set

Remarque. L'intérêt de la méthode pop est souvent de parcourir les éléments de l'ensemble en les retirant au fur et à mesure. Dans les deux programmes ci-dessous, on imagine déjà définie un ensemble `S` et une fonction `do_somehting`. Le premier programme ci-dessous applique cette fonction à chaque élément de l'ensemble `S` sans modifier cet ensemble, tandis que le second programme fait la même chose mais en vidant l'ensemble `S` en même temps qu'il traite ses éléments.

```
for x in S:
    do_something(x)

# L'ensemble S n'a pas été modifié
```

```
while len(S) > 0:
    x = S.pop()
    do_something(x)
# L'ensemble S est maintenant vide
```

Exercice 6.1.

1. Créer une fonction `multiples` qui prend en argument un entier `p`, et qui renvoie un ensemble contenant tous les multiples de `p` compris entre 1 et 50.
2. À l'aide de votre fonction et des opérateurs sur les ensembles, afficher tous les entiers compris entre 1 et 50 divisibles …
   a) … par 6,
   b) … par 10,
   c) … à la fois par 6 et par 10,
   d) … par 6 ou par 10,
   e) … par 6 ou par 10 mais pas les deux à la fois,
   f) … par 10 mais pas par 6.

6.1.3 Ensemble immuable

Pour information, il existe une variante du type `set` appelée `frozenset` qui fonctionne de la même manière qu'un ensemble à l'exception qu'il s'agit d'un type immuable. On crée un tel objet en donnant à la fonction `frozenset` un itérable contenant les valeurs qu'on veut stocker dans l'ensemble. Une fois fait, l'ensemble **ne peut plus être modifié**. Cela signifie que :

— on ne peut **pas** lui appliquer les méthodes listées dans le tableau 6.2,

— on peut en revanche effectuer des calculs entre ensembles avec les opérateurs listés dans le tableau 6.1.

À tester. Exécutez le code ci-dessous :

```
S = frozenset({1, 5, 6, 9})
T = frozenset({3, 5, 6, 7})
```

puis observez les résultats des instructions suivantes :

1. `print(2 in S)`
2. `print(5 in S)`
3. `print(S & T)`
4. `S.add(8)`

Notez que la différence entre les types `set` et `frozenset` est exactement la même différence qu'entre les types `list` et `tuple`. On choisira une liste ou un tuple si l'ordre des éléments que l'on veut stocker a une importance, et un ensemble (muable ou immuable) si l'ordre n'a pas d'importance.

Collection	Ordonnée	Non-ordonnée
Muable	`list`	`set`
Immuable	`tuple`	`frozenset`

6.2 Dictionnaires

Les dictionnaires sont assez similaires aux listes, à la différence près que là où les éléments de la liste sont indexés par des entiers successifs à partir de 0, les éléments d'un dictionnaire sont indexés par des objets quelconques (qu'on appellera *clés* plutôt qu'indices).

6.2.1 Définition et propriétés

Pour créer un dictionnaire, on utilise les syntaxes suivantes.

— **Avec des accolades.** On place entre `{` et `}` des couples de la forme `key : value` avec `value` la valeur que l'on veut stocker dans le dictionnaire et `key` la clé qu'on veut utiliser pour indexer cette valeur. On peut aussi laisser la paire d'accolades vide pour créer un dictionnaire vide.

À tester. Observez les objets créés par le code suivant dans l'explorateur de variables (double-cliquez sur une liste ou un dictionnaire pour révéler plus de détails).

```
L = ["pierre", "papier", "ciseaux"]
D1 = {0:"pierre", 8:"papier", -3:"ciseaux"}
D2 = {"first_name":"Camille", "last_name":"Dupont", "age":32}
```

— **Avec la fonction `dict`.** On donne à `dict` des arguments nommés sous la forme `key=value`. Cette façon de faire est plus restrictive : dans ce cas `key` doit être un identifiant Python valide qui sera transformé en chaîne de caractères. On peut aussi appeler `dict()` sans argument pour créer un dictionnaire vide.

À tester. Vérifiez que l'instruction ci-dessous crée le même dictionnaire `D2` que plus haut.

```
D2 = dict(first_name="Camille", last_name="Dupont", age=32)
```

Aurait-on pu utiliser cette syntaxe pour créer le dictionnaire `D1` ?

Remarque. Dans un dictionnaire, chaque clé doit être unique. Si lors de la construction du dictionnaire on utilise plusieurs fois la même clé, c'est la dernière valeur qu'on lui aura associé qui sera gardée.

Attention. Pour des raisons techniques, on ne peut pas utiliser des objets muables comme *clés* d'un dictionnaire [3]. En revanche, il n'y a pas de contraintes sur les *valeurs* que l'on stocke.

```
D1 = {(1, 2):[1, 2], (3, 4):[3, 4]} # Valide : les clés sont des tuples
D2 = {[1, 2]:(1, 2), [3, 4]:(3, 4)} # Invalide : les clés sont des listes
```

On accède ensuite aux valeurs stockées dans un dictionnaire `D` via la syntaxe `D[key]` avec `key` la clé de l'élément auquel on veut accéder.

À tester. En reprenant les objets créés précédemment, observez le résultat des instructions suivantes :

1. `print(D1[0])`
2. `print(D1[-3])`
3. `print(D1[-3][4])`
4. `print(D1[1])`
5. `print(D2["last_name"])`
6. `print(D2["age"])`

Un dictionnaire `D` vérifie les propriétés suivantes.

— Il est indiçable (*subscriptable* en anglais), puisque c'est précisément la syntaxe `D[x]` qui permet d'accéder à ses éléments.

— Il est itérable : la syntaxe `for x in D` permet de parcourir les **clés** de ce dictionnaire (on récupère alors ensuite aisément ses valeurs avec l'expression `D[x]`). Depuis la version 3.7 de Python, l'itération sur un dictionnaire parcourt toujours les valeurs dans l'ordre où elles ont été ajoutées dans le dictionnaire (ça n'était pas garanti dans les versions précédentes).

À tester. Essayez le code ci-dessous.

```
data = {"first_name":"Camille", "last_name":"Dupont", "age": 32}

for x in data:
    val = data[x]
    print("Clé: " + x + " -> Valeur associée: " + str(val))
```

— Il est muable, on appliquera donc les mêmes précautions que pour les listes et les ensembles.

6.2.2 Itération sur un dictionnaire

En plus de l'itération par défaut sur un dictionnaire `D`, on peut utiliser les méthodes suivantes :

— `D.keys()` qui renvoie un itérable permettant de parcourir les *clés* du dictionnaire `D`,

— `D.values()` qui renvoie un itérable permettant de parcourir les *valeurs* du dictionnaire `D`,

— `D.items()` qui renvoie un itérable permettant de parcourir tous les tuples de la forme `(key, value)` avec `key` une clé de `D` et `value` la valeur qui lui est associée.

3. De même que pour les ensembles, il s'agit d'un cas particulier d'une contrainte plus stricte expliquée dans la section 6.3.

À tester. Étant déjà défini un dictionnaire `data`, observez les affichages produits par le code suivant.

```
print(">> Méthode 'keys'")
for x in data.keys():
    print(x)

print(">> Méthode 'values'")
for x in data.values():
    print(x)

print(">> Méthode 'items'")
for x in data.items():
    print(x)
```

6.2.3 Modification des entrées d'un dictionnaire

Le type `dict` supporte les méthodes listées dans la table 6.3.

Méthode	Description
`D.get(key)`	Renvoie la valeur associée à la clé `key` dans le dictionnaire `D`. Provoque une erreur si `D` ne contient pas la clé `key`.
`D.get(key, default)`	Renvoie la valeur associée à la clé `key` dans le dictionnaire `D`. Renvoie la valeur `default` si `D` ne contient pas la clé `key`.
`D.pop(key)`	Supprime et renvoie la valeur associée à `key` dans le dictionnaire `D`. Provoque une erreur si `D` ne contient pas la clé `key`.
`D.pop(key, default)`	Supprime et renvoie la valeur associée à `key` dans le dictionnaire `D`. Renvoie la valeur `default` si `D` ne contient pas la clé `key`.
`D.clear()`	Retire tous les éléments de `D`, qui devient donc un dictionnaire vide.
`D.update(D2)`	Ajoute à `D` tous les couples `(key, value)` contenus dans `D2`. Si `D[key]` existait déjà, il est remplacé par `D2[key]`.

TABLE 6.3 – Méthodes du type `dict`

Il existe aussi les syntaxes suivantes :

— `key in D` renvoie un booléen qui indique si le dictionnaire `D` contient la clé `key`, [4]

— `D[key] = value` ajoute dans `D` une valeur associée à la clé `key` (ou la modifie si elle existait déjà),

— `del D[key]` supprime la clé `key` du dictionnaire (et la valeur qui lui était associée).

Remarque. Attention à ne pas oublier que les opérateurs `in` et `not int` opèrent sur les *clés* d'un dictionnaire. Pour étudier la présence ou non d'un élément parmi les *valeurs* d'un dictionnaire `D`, on utilisera `in` ou `not in` sur l'itérable `D.values()`.

4. Comme pour les listes et les ensembles, il existe l'opérateur `not in` qui renvoie la négation de ce que renverrait `in`.

Exercice 6.2. Jeu du Tic-Tac-Toe. Le jeu du *Tic-Tac-Toe* (aussi appelé jeu du morpion) se joue à deux joueurs sur une grille de taille 3×3. Chacun leur tour, les deux joueurs inscrivent leur symbole (`O` ou `X`) dans une case encore inoccupée du plateau. Si un joueur parvient à aligner trois fois son symbole horizontalement, verticalement ou en diagonale, alors il remporte la partie. Si la grille est complètement remplie et qu'aucun joueur n'a gagné, alors il y a match nul.

Dans cet exercice, on va modéliser une partie de Tic-Tac-Toe par un dictionnaire représentant l'état du plateau à un instant donné. Les clés de ce dictionnaire seront les tuples de la forme (i, j) avec $0 \leq i < 3$ le numéro de ligne de la case et $0 \leq j < 3$ son numéro de colonne. La valeur associée à chacune de ces cases sera une chaîne de caractères soit vide, soit égale à `'O'`, soit égale à `'X'`.

1. Écrire une fonction `new_game` sans argument qui renvoie un dictionnaire représentant un plateau de jeu de Tic-Tac-Toe en début de partie (c'est-à-dire dont les 9 cases sont vides).
2. Écrire une fonction `is_valid` qui prend en arguments :
   - `grid` (`dict`) : le plateau de jeu,
   - `i` (`int`) : un numéro de ligne,
   - `j` (`int`) : un numéro de colonne,

   et qui renvoie un booléen indiquant s'il est actuellement possible de jouer dans la case (i, j). Autrement dit, votre fonction devra renvoyer `True` si les coordonnées (i, j) sont valides **et** si la case (i, j) est vide, et `False` dans tous les autres cas.
3. Écrire une fonction `check_grid` qui prend en argument un dictionnaire `grid` représentant le plateau de jeu, et qui renvoie :
   - une chaîne vide si aucun joueur n'a gagné pour le moment,
   - le symbole du joueur qui a réussi à créer une ligne, colonne ou diagonale complète sinon.
4. Écrire une fonction `draw_grid` qui prend en argument un dictionnaire `grid` représentant un plateau de Tic-Tac-Toe, et qui affiche ce plateau dans la console.
   Exemple d'affichage possible :

   ```
   O O .
   . . X
   . . .
   ```

5. Écrire une fonction `play` qui prend en arguments :
   - `grid` (`dict`) : le plateau de jeu,
   - `S` (`str`) : un symbole (`O` ou `X`)
   - `i` (`int`) : un numéro de ligne,
   - `j` (`int`) : un numéro de colonne,

   et qui utilise les fonctions précédentes de sorte à réaliser les tâches suivantes :
   - si le coup en (i, j) n'est pas valide, alors afficher un message d'erreur,
   - sinon, modifier le plateau pour ajouter le symbole `S` dans la case de coordonnées (i, j), afficher le plateau dans la console, puis :
     - si un joueur a gagné, afficher le message « Gagné ! »,
     - sinon, ne rien faire de plus.

Une fois que vous avez créé toutes les fonctions de cet exercice, vous pouvez jouer une partie complète de Tic-Tac-Toe en saisissant l'instruction

```
grid = new_game()
```

puis en appelant (à tour de rôle avec votre adversaire) la fonction `play` en lui donnant le prochain coup à jouer.

6.3 Pour aller plus loin : objets *hashables* (section facultative)

Au fil des chapitres précédents, vous avez appris la différence entre les types *muables* et *immuables*. Concrètement, un objet de type *immuable* ne peut pas être modifié en mémoire : si on essaie de changer sa valeur, soit on obtient une erreur, soit un nouvel objet est créé en mémoire de manière invisible.

À tester. La fonction `id` renvoie une valeur permettant d'identifier de manière unique un objet dans la mémoire. Autrement dit, si `id(a)` et `id(b)` renvoient la même valeur, cela signifie que les deux identifiants `a` et `b` renvoient vers un seul même objet en mémoire.

1. Testez le code suivant. Observez les affichages produits, comparez-les avec les valeurs finales de `A` et de `B` données dans l'explorateur de variables, et essayez de décrire ce qui s'est passé en mémoire.

```
A = 3
B = A
print("Avant modification")
print(" - A pointe vers l'objet", id(A))
print(" - B pointe vers l'objet", id(B))

B += B
print("Après modification")
print(" - A pointe vers l'objet", id(A))
print(" - B pointe vers l'objet", id(B))
```

2. Même question en remplaçant la première ligne par `A = [4, 5, 6]`, puis par `A = (4, 5, 6)`.

En particulier, le type `tuple` est immuable. Cela signifie qu'une fois qu'un tuple `T` est créé, chaque `T[i]` pointe vers un objet fixé en mémoire et on ne pourra pas modifier ce lien. Toutefois, si `T[i]` pointe vers un objet qui, lui, est muable (par exemple une liste) alors rien n'empêche que cet objet change de valeur au cours du programme.

Par exemple, le code ci-dessous provoque une erreur car on essaie de modifier le lien entre `T[1]` et l'objet auquel il est rattaché, ce qui reviendrait à modifier l'objet `T` en mémoire.

```
T = (4, 8, 2)
T[1] = 3
print(T)
```

En revanche le code ci-dessous ne provoque pas d'erreur. En effet, on ne fait que modifier la liste vers laquelle pointe `T[1]`, mais à aucun on ne modifie l'objet `T` lui-même dans la mémoire.

```
T = ([1, 3], [12, 8], [24])
T[1].append(9)
print(T)
```

Autrement dit, le fait qu'un tuple soit immuable n'empêche **pas** que les objets qu'il contient changent de valeur, sauf s'ils sont eux-mêmes immuables. Pour cela, il existe un autre concept plus strict : on dit qu'un objet est ***hashable*** si, pour simplifier, *tout* son contenu n'est pas modifiable et restera donc *exactement* le même tout au long du programme. Dans les grandes lignes, on a donc les cas suivants.

1. Tout objet de type muable (par exemple `list`, `set` ou `dict`) est *non-hashable*, puisque son contenu peut clairement changer à tout moment.
2. Tout objet de type immuable « basique » (`int`, `float`, `bool`, `str`) est *hashable*.

3. Pour une collection immuable tel qu'un tuple ou un *frozen set*, il faut être prudent : cette collection ne sera *hashable* que si elle contient uniquement des objets eux-même *hashables*. Par exemple :
   — le tuple `(4, 8, 2)` est *hashable* : il ne contient que des entiers (immuables) et donc il est impossible de changer son contenu,
   — le tuple `(4, 8, [2, 1])` n'est pas *hashable* : `T[2]` pointe vers une liste qui pourrait très bien être modifiée plus tard dans le programme.

Cette nouvelle notion nous permet de préciser un peu les contraintes évoquées dans les sections précédentes. Pour être tout à fait rigoureux :

— les éléments d'un ensemble doivent impérativement être des objets *hashables*,
— les clés d'un dictionnaire doivent impérativement être des objets *hashables* (en revanche il n'y a pas de contrainte sur les valeurs).

À tester. Testez les instructions suivantes.

1. `S = { 1, 8, (4, 5) }`
2. `S = { 1, 8, (4, [5]) }`
3. `D = {"a": [1, 2], "b": [3, 4]}`
4. `D = {[1, 2]: "a", [3, 4]: "b"}`

Exercices supplémentaires

Exercice 6.3. Écrire une fonction `stats` qui prend en argument une liste `L` quelconque, et qui renvoie un dictionnaire dont :

— les clés sont les valeurs qui apparaissent dans la liste `L`,

— la valeur associée à chaque clé est le nombre de fois où cette valeur apparaît dans la liste `L`.

N'utilisez pas la méthode `L.count` mais parcourez plutôt la liste valeur par valeur en mettant à jour votre dictionnaire au fur et à mesure.

Par exemple, `stats(['d', 'a', 'a', 'e', 'c', 'b', 'e', 'd', 'e', 'a', 'a', 'a'])` *devra renvoyer le dictionnaire* `{'a': 5, 'b': 1, 'c': 1, 'd': 2, 'e': 3}`.

Exercice 6.4. Ajoutez à votre programme de Tic-Tac-Toe (exercice 6.2) une fonction `random_play` qui prend en arguments :

— `grid` (`dict`) : le plateau de jeu,

— `S` (`str`) : un symbole (`O` ou `X`)

et qui place le symbole `S` dans l'une des cases vides restantes du plateau choisie aléatoirement. Ensuite, votre fonction effectuera les mêmes tâches qu'effectuait la fonction `play` (affichage de la grille puis éventuel affichage du message « Gagné ! »).

À retenir

— Connaître et comprendre le vocabulaire : *indexable/subscriptable*, *iterable*, *(im)mutable*.

— Connaître les propriétés des types `set`, `frozenset` et `dict`.

— Savoir initialiser un ensemble ou un dictionnaire en Python.

— Connaître toutes les façons de parcourir les éléments d'un dictionnaire ou d'un ensemble.

— Connaître les opérateurs et méthodes usuelles sur les ensembles.

— Connaître les méthodes usuelles sur les dictionnaires.

— Savoir mettre en place un algorithme pour remplir un ensemble ou un dictionnaire pas à pas.

— Savoir accéder à un élément d'un dictionnaire, connaître et comprendre le vocabulaire : *clé*, *valeur*.

	`list`	`tuple`	`set`	`frozenset`	`dict`
Subscriptable	✓	✓	✗	✗	✓
Iterable	✓	✓	✓	✓	✓
Muable	✓	✗	✓	✗	✓