

TP 2.3 – Dictionnaires

Objectifs –

- ◇ Comprendre les différences entre une liste Python, dont les éléments sont repérés par des indices entiers, et un dictionnaire, qui associe des clés à des valeurs.
- ◇ Savoir créer un dictionnaire, puis ajouter ou modifier une association clé-valeur.
- ◇ Savoir tester si une clé est présente dans un dictionnaire et récupérer la valeur qui lui est associée le cas échéant.
- ◇ Comprendre qu'une clé est unique dans un dictionnaire et qu'elle ne peut être associée qu'à une seule valeur à un instant donné.
- ◇ Savoir construire et parcourir un dictionnaire pour compter, regrouper ou indexer des informations.
- ◇ Comprendre quand l'utilisation d'un dictionnaire est pertinente, voire indispensable, notamment pour mémoriser des informations déjà rencontrées.

Parcours principal

Exercice 1 – Comptage des éléments

On souhaite écrire une fonction `compte_elements(lst: list) -> dict` qui, pour une liste `lst` donnée, renvoie un dictionnaire. Les clés de ce dictionnaire sont les différents éléments de la liste et la valeur associée à chaque clé est le nombre de fois où cet élément apparaît dans `lst`. Par exemple, pour la liste :

```
| lst = [1, 5, 1, 2, 5, 5, 3, 2]
```

l'élément 1 apparaît 2 fois, l'élément 5 apparaît 3 fois, etc. Ainsi, l'appel `compte_elements(lst)` doit renvoyer le dictionnaire :

```
| {1: 2, 5: 3, 2: 2, 3: 1}
```

Dans cet exemple, les clés sont des entiers et non des chaînes de caractères. Pour concevoir cette fonction, on propose la démarche suivante :

- ◇ initialiser une variable `d` avec un dictionnaire vide, qui se note `{}` ;
- ◇ parcourir chaque élément `elt` de la liste `lst` ;
- ◇ si `elt` n'est pas déjà présent comme clé dans le dictionnaire `d`, ajouter cette clé

en lui associant la valeur 1 ;

- ◇ sinon, augmenter de 1 la valeur déjà associée à cette clé.

L'ordre dans lequel les associations sont écrites dans le dictionnaire n'a pas d'importance.

Question 1. Écrire cette fonction et la tester sur plusieurs listes bien choisies.

Question 2. En s'inspirant de la fonction précédente, écrire une fonction `indices_elements(lst: list) -> dict` renvoyant toujours un dictionnaire dont chaque clé est un élément présent dans `lst`, mais dont la valeur associée est la liste de tous les indices où cet élément apparaît. Par exemple, pour la liste :

```
| lst = [1, 5, 1, 2, 5, 5, 3, 2]
```

l'élément 1 apparaît aux indices 0 et 2, et l'élément 5 apparaît aux indices 1, 4 et 5. L'appel `indices_elements(lst)` doit renvoyer le dictionnaire :

```
| {1: [0, 2], 5: [1, 4, 5], 2: [3, 7], 3: [6]}
```

Exercice 2 – Rechercher des clés particulières

On considère un dictionnaire `d` tel que chaque clé soit une chaîne de caractères et que la valeur associée à chaque clé soit un entier. Par exemple :

```
| d = {"a": 5, "b": 3, "c": 8}
```

Question 1. Écrire une fonction `cles_de_valeur(dct: dict, val: int) -> list` qui renvoie la liste des clés ayant pour valeur `val`. Si aucune clé n'a pour valeur `val`, la fonction renvoie la liste vide.

Question 2. On suppose dans cette question que le dictionnaire passé en argument est non vide. Écrire une fonction `cle_max(dct: dict) -> str` renvoyant une clé pour laquelle la valeur associée est maximale. Si plusieurs clés possèdent cette valeur maximale, la fonction peut renvoyer indifféremment l'une d'entre elles.

On pourra initialiser la plus grande valeur rencontrée avec :

```
| maxi = -float("inf")
```

Question 3. Reprendre la question précédente en écrivant une fonction `cle_min(dct: dict) -> str` renvoyant une clé pour laquelle la valeur associée est minimale. On suppose toujours que le dictionnaire est non vide.

Si le temps le permet

Exercice 3 – Reconstruire une liste

Soit `dct` un dictionnaire dont les clés sont les éléments présents dans une liste `lst`, et dans lequel la valeur associée à chaque clé est la liste des indices où cet élément est présent dans `lst`. On suppose que `dct` a été obtenu par un appel à la fonction `indices_elements`. Ainsi, chaque indice apparaît une et une seule fois dans les listes de valeurs de `dct`.

Écrire une fonction `dictionnaire_vers_liste(dct: dict) -> list` qui reconstruit et renvoie la liste `lst` connaissant `dct`. Si `dct` est vide, la fonction renvoie la liste vide. On pourra commencer par déterminer la longueur de la liste à reconstruire en cherchant le plus grand indice apparaissant dans les valeurs du dictionnaire.

Travail à la maison

Exercice 4 – Détection rapide de doublons

Dans un exercice précédent, on a écrit une fonction `sans_doublon` qui teste si une liste contient ou non des doublons. La méthode utilisée consistait à comparer les éléments deux à deux : pour chaque couple d'indices (i, j) avec $i < j$, on testait si `lst[i] == lst[j]`. Cette méthode fonctionne, mais elle peut faire beaucoup de comparaisons lorsque la liste est longue.

Dans cet exercice, on utilise un dictionnaire pour mémoriser les éléments déjà rencontrés. On suppose que les éléments de la liste peuvent être utilisés comme clés d'un dictionnaire. On parcourt la liste de gauche à droite et on maintient un dictionnaire `deja_vus` :

- ◊ si l'élément courant est déjà une clé de `deja_vus`, alors cet élément a déjà été rencontré : la liste contient donc un doublon ;
- ◊ sinon, on ajoute cet élément comme clé de `deja_vus`, puis on poursuit le parcours.

Les clés de `deja_vus` sont donc les éléments déjà rencontrés. La valeur associée à chacune de ces clés sert seulement de marqueur : on pourra choisir la valeur `True`. Seule la présence de la clé sera utilisée par l'algorithme.

Ainsi, on ne compare plus directement chaque élément avec tous les autres. En moyenne, tester la présence d'une clé dans un dictionnaire se fait en temps constant.

Question 1. Écrire une fonction `sans_doublon_memo(lst: list) -> bool` qui renvoie `True` si les éléments de `lst` sont tous distincts, et `False` sinon, en utilisant le principe décrit ci-dessus. On pourra utiliser un dictionnaire initialement vide :

```
| deja_vus = {}
```

Puis faire les tests suivants :

```
| assert(sans_doublon_memo([2, 3, 1, 5]))
| assert(not sans_doublon_memo([2, 3, 3, 1, 2]))
| assert(sans_doublon_memo([]))
```

Question 2. On souhaite maintenant construire une liste sans doublon à partir d'une liste donnée. Lorsqu'un élément apparaît plusieurs fois, on ne conserve que sa première apparition. Écrire une fonction `liste_sans_doublon_memo(lst: list) -> list` qui renvoie une nouvelle liste contenant les éléments de `lst`, mais sans doublon, en conservant l'ordre de leur première apparition. Par exemple :

```
| liste_sans_doublon_memo([2, 3, 3, 1, 2])
```

doit renvoyer :

```
| [2, 3, 1]
```

On pourra utiliser à la fois un dictionnaire `deja_vus` pour mémoriser les éléments déjà rencontrés et une liste `res` pour construire le résultat.

Le gain est important : au second semestre, nous montrerons que cette méthode effectue un nombre d'opérations proportionnel à la longueur n de la liste, en moyenne, alors que la méthode naïve par comparaison de tous les couples effectue un nombre d'opérations proportionnel à n^2 dans le pire cas. Ce gain de temps est obtenu au prix d'un espace mémoire supplémentaire pour stocker le dictionnaire `deja_vus` ; cet espace peut lui aussi être proportionnel à n .

Exercice 5 – Construction d'un index pour un texte

Soit `texte` une chaîne de caractères contenant des mots séparés par des espaces. Dans cet exercice, les chaînes étudiées ne contiennent pas de ponctuation.

La méthode `split` construit la liste des mots d'une chaîne en la découpant aux espaces. Elle ne supprime pas la ponctuation : si le texte contenait une virgule ou un point d'interrogation, ceux-ci resteraient dans les chaînes obtenues. Par exemple, avec :

```
| texte = "itc est vraiment genial"
| liste_mots = texte.split()
```

on obtient :

```
| ["itc", "est", "vraiment", "genial"]
```

Dans cet exercice, on s'intéresse aux positions des mots dans la liste obtenue après découpage par `split`. Ces positions sont numérotées à partir de 0.

On souhaite écrire une fonction `index_mots(texte: str) -> dict` qui renvoie un dictionnaire dont :

- ◊ les clés sont les mots de `texte` ;
- ◊ la valeur associée à un mot est la liste des positions auxquelles ce mot apparaît dans la liste obtenue avec `split`.

Par exemple, pour la chaîne suivante :

```
| texte = "le temps est beau le ciel est bleu le lycee est beau et  
|         bleu"
```

la liste des mots est :

```
| ["le", "temps", "est", "beau", "le", "ciel", "est",  
|  "bleu", "le", "lycee", "est", "beau", "et", "bleu"]
```

Le mot `"le"` apparaît aux positions 0, 4 et 8. Le mot `"beau"` apparaît aux positions 3 et 11.

Question 1. À la main, donner les positions d'apparition des mots `"est"`, `"bleu"` et `"temps"` dans la chaîne précédente, puis construire le dictionnaire complet attendu.

Question 2. Écrire la fonction `index_mots(texte: str) -> dict`. On pourra commencer par créer un dictionnaire vide :

```
| index = {}
```

Puis parcourir les positions `i` de la liste `liste_mots`. Pour chaque mot rencontré :

- ◊ si le mot n'est pas encore une clé du dictionnaire, on l'ajoute avec une liste vide comme valeur ;
- ◊ on ajoute ensuite la position `i` à la liste associée à ce mot.

Question 3. Tester la fonction sur plusieurs chaînes, en particulier sur une chaîne vide et sur une chaîne contenant plusieurs fois le même mot. Pour une chaîne vide, le dictionnaire renvoyé doit être vide.

Approfondissement facultatif

Exercice 6 – Dictionnaire de listes

On considère des données représentées par un dictionnaire Python. Les clés sont des chaînes de caractères, par exemple `"a"`, `"b"`, `"c"`, etc. À chaque clé est associée une liste contenant certaines des autres clés.

Par exemple :

```
| network = {  
|     "a": ["b", "d"],  
|     "b": ["a", "c"],  
|     "c": ["b", "d", "e"],  
|     "d": ["a", "c"],  
|     "e": ["c"]  
| }
```

Dans cet exercice, on adopte les conventions suivantes :

- ◊ toutes les clés qui apparaissent dans les listes sont aussi des clés du dictionnaire ;
- ◊ une clé n'apparaît pas dans sa propre liste ;
- ◊ les listes ne contiennent pas de doublons ;
- ◊ les relations sont réciproques : si `"b"` apparaît dans la liste associée à `"a"`, alors `"a"` apparaît dans la liste associée à `"b"`. En vocabulaire mathématique, cette relation est dite *symétrique*.

Question 1. Écrire une fonction `number_of_keys(network: dict) -> int` qui renvoie le nombre de clés du dictionnaire.

Question 2. Écrire une fonction `number_of_links(network: dict) -> int` qui renvoie le nombre de relations réciproques distinctes représentées par le dictionnaire. Par exemple, avec le dictionnaire précédent, la fonction doit renvoyer 5, car les relations représentées sont :

`"a" — "b", "a" — "d", "b" — "c", "c" — "d", "c" — "e"`

Question 3. Écrire une fonction `contains_key(network: dict, key: str) -> bool` qui renvoie `True` si `key` est une clé du dictionnaire, et `False` sinon.

Question 4. Écrire une fonction `has_link(network: dict, key1: str, key2: str) -> bool` qui renvoie `True` si `key2` apparaît dans la liste associée à `key1`, et `False` sinon. La fonction doit aussi renvoyer `False` si `key1` n'est pas une clé du dictionnaire.

Question 5. Écrire une fonction `add_link(network: dict, key1: str, key2: str) -> None` qui ajoute une relation réciproque entre `key1` et `key2`.

On suppose que `key1` et `key2` sont déjà des clés du dictionnaire, et que `key1 != key2`. La fonction ne doit pas créer de doublon dans les listes.

Question 6. Écrire une fonction `remove_link(network: dict, key1: str, key2: str) -> None` qui supprime la relation réciproque entre `key1` et `key2` lorsqu'elle existe. La fonction ne doit pas produire d'erreur si cette relation n'existe pas. Si l'une des deux clés n'est pas présente dans le dictionnaire, la fonction ne modifie pas le dictionnaire.

Question 7. Écrire une fonction `add_key(network: dict, key: str) -> None` qui ajoute une nouvelle clé au dictionnaire, associée à une liste vide. Si la clé est déjà présente, la fonction ne modifie pas le dictionnaire.

Question 8. Écrire une fonction `remove_key(network: dict, key: str) -> None` qui supprime une clé du dictionnaire. La fonction doit aussi supprimer cette clé de toutes les listes où elle apparaît. Si la clé n'est pas présente, la fonction ne modifie pas le dictionnaire.

Question 9. Tester les fonctions précédentes sur le dictionnaire `network`. On pourra notamment faire les tests suivants :

```
assert(number_of_keys(network) == 5)
assert(number_of_links(network) == 5)

assert(contains_key(network, "a"))
assert(not contains_key(network, "f"))

assert(has_link(network, "a", "b"))
assert(not has_link(network, "a", "c"))

add_key(network, "f")
add_link(network, "e", "f")
assert(contains_key(network, "f"))
assert(has_link(network, "e", "f"))
assert(has_link(network, "f", "e"))

remove_link(network, "a", "b")
assert(not has_link(network, "a", "b"))
assert(not has_link(network, "b", "a"))
```

```
remove_key(network, "c")
assert(not contains_key(network, "c"))
assert("c" not in network["b"])
assert("c" not in network["d"])
assert("c" not in network["e"])
```