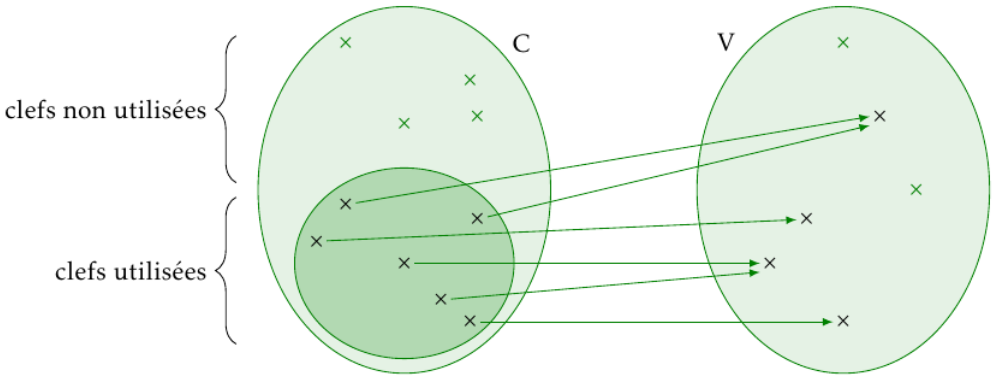
Chapitre 2

Dictionnaires

Cette première partie a pour objet l'étude d'une structure de données que vous avez déjà rencontré en première année : les dictionnaires. Cette structure de données répond à la problématique suivante : comment rechercher efficacement de l'information dans un ensemble géré de façon dynamique (c'est à dire dont le contenu est susceptible d'évoluer au cours du temps).
Les dictionnaires sont couramment utilisées en informatique : c'est par exemple la structure de données utilisée pour gérer les systèmes de noms de domaine (DNS, pour Domain Name System). Les ordinateurs connectés à un réseau comme Internet possèdent une adresse numérique (en IPv4 par exemple, celles-ci sont représentées sous la forme xxx.xxx.xxx.xxx, où xxx est un nombre hexadécimal variant entre 0 et 255). Pour faciliter l'accès aux systèmes qui disposent de ces adresses, un mécanisme a été mis en place pour associer un nom (plus facile à retenir) à une adresse IP. Ce mécanisme utilise un dictionnaire dans laquelle les clefs sont les noms de domaine et les valeurs les adresses IP.

I. Description de la structure de données

Un dictionnaire est un type de données associant un ensemble de clefs à un ensemble de valeurs. Plus formellement, si C désigne l'ensemble des clefs et V l'ensemble des valeurs, un dictionnaire est un sous-ensemble T de $C \times V$ tel que pour toute clef $c \in C$ il existe au plus un élément $v \in V$ tel que $(c, v) \in T$. Les éléments de T sont appelés des associations.



Un dictionnaire supporte en général les opérations suivantes :

- ajout d'une nouvelle association $(c, v) \in C \times V$ dans T ;
- suppression d'une association (c, v) de T ;
- existence d'une association (c, v) dans T pour une clef $c \in C$ donnée ;
- lecture de la valeur v associée à une clef c présente dans T.

II. Utilisation en Python

```
{"nom":"Leonard", "jour":15, "mois":4, "annee":1452}
```

contient 4 clés, ici les chaînes de caractère "nom", "jour", "mois" et "annee", auxquelles sont respectivement associées les valeurs "Leonard", 15, 4 et 1452.

- Un dictionnaire peut être construit en donnant explicitement toutes ses entrées comme dans l'exemple ci-dessus.
  Mais un dictionnaire peut être également construit à partir d'un dictionnaire vide, noté `{}`, en y ajoutant petit à petit des affectation de la forme `d[clé] = valeur`.

```
>>> d = {}
>>> d["Professeur"] = "Lamendin"
>>> d["Matiere"] = "Sciences-Physique"
>>> d["Cours"] = 5
>>> d["TD"] = 1
>>> d["TP"] = 2
>>> len(d)
5
```

- L'ordre d'insertion n'a aucune importance. En particulier, l'affichage est donné dans un ordre arbitraire.
- On accède à la valeur associée à une clé avec la construction `d[clé]`

```
>>> d["Matiere"]
"Sciences-Physique"
```

- On peut tester si le dictionnaire possède une entrée pour une certaine clé avec la construction `clé in d`

```
>>> "Matiere" in d
True
>>> "Salle" in d
False
```

- Comme pour une liste, le contenu d'un dictionnaire peut être modifié :

```
>>> d["Professeur"] = "Champault"
>>> len(d)
5
>>> d["Professeur"]
"Champault"
```

- On peut supprimer une entrée du dictionnaire avec l'instruction `del` :

```
>>> del d["TP"]
>>> "TP" in d
False
>>> len(d)
4
```

1. Parcours d'un dictionnaire

- On peut parcourir toutes les clés d'un dictionnaire avec la boucle `for`. L'ordre de parcours est arbitraire :

```
for x in d :
    print("la cle", x, "est associee a la valeur", d[x])
```

- On peut obtenir une liste contenant toutes les valeurs de d avec l'instruction `d.keys())` ou plus simplement avec `list(d)`

```
>>> d = {"a":1, "b":2, "c":1}
>>> list(d.keys())
['b', 'a', 'c']
```

- De la même façon, on peut obtenir une liste contenant toutes les valeurs avec `list(d.values())`

```
>>> list(d.values())
[1, 1, 2]
```

- On peut obtenir un tableau contenant toutes les entrées du dictionnaire sous la forme `(cle, valeur)` avec `list(d.items())` :

```
>>> list(d.items())
[('a', 1), ('b', 2), ('c', 1)]
```

- Enfin, on peut construire un dictionnaire par compréhension :

```
>>> { x*x : x for x in range(4)}
{0: 0, 1: 1, 4: 2, 9: 3}
```

Exercice 2.1

Ecrire une fonction `occurrences(liste)` qui prend en argument une liste `liste` et renvoie un dictionnaire donnant, pour chaque valeur de `liste`, le nombre de fois qu'elle apparaît dans `liste`.

```
>>> occurrences([1, 3, 4, 2, 3, 2, 2])
{1 : 1, 2 : 3, 3 : 2, 4 : 1}
```

Exercice 2.2

Ecrire une fonction permettant de savoir si deux listes comportent exactement les mêmes éléments (avec pour chacun le même nombre d'occurrences).

Exercice 2.3

À l'issue d'une élection à scrutin uninominal, on récupère un tableau contenant tous les noms inscrits sur les bulletins trouvés dans l'urne. Par exemple :

```
urne = [" Maurice ", " Roger ", " Maurice ", " Marie ", " Marie ", " Jeanne ", ...]
```

Ecrire une fonction `depouillement(urne)` qui permet de déterminer le vainqueur de l'élection, en un seul parcours de l'urne.

III. description de l'implémentation des dictionnaires

Or décrit ici grossièrement comment est implémenté un dictionnaire dans Python. Cela vous permettra d'avoir une meilleure idée de la complexité des opérations élémentaires.

1. Liste d'associations

Une première méthode quelque peu naïve d'implémenter un dictionnaire utilise ce qu'on nomme une « liste d'association ». Il s'agit d'une liste de couples de la forme `(clef, valeur)`. Par exemple

la liste d'association suivante :

```
l = [ ('bananes ', 6), ('carottes ', 5), ('topinambours ', 7) ]
```

représente le dictionnaire qui à la chaîne `'bananes'` associe l'entier 6, qui à `'carottes'` associe 5 et à `'topinambours'` associe 7 (mettons qu'on soit en train de gérer le stock d'une épicerie). Rédigeons les fonction de bases pour cette implémentation des dictionnaires.

```
def nouveau_dico ():
    return []

def ajout(d,c,v):
    ''' Insère la valeur v, associ ée à la clef c, dans le dictionnaire d.'''
    d. append ((c,v))

def assoc(d,c):
    ''' Renvoie la valeur associée à la valeur c dans le dictionnaire d, ou None
    si aucune valeur n'est associée à c.'''
    for (clef , val) in d:
        if c== clef:
            return val

def estUneClef (d,c):
    for (clef, _) in d:
    if clef ==c:
        return True
    return False
```

Le défaut est que la complexité de `assoc` et estUneClef est en $O(n)$, où n est le nombre d'associations dans le dictionnaire. On pourrait trier les entrées (si C est muni d'une relation d'ordre, mais quasiment tous les types de bases le sont), ce qui nous ferait une complexité en $O(log(n))$ pour ces deux fonctions, au prix d'un $O(n)$ pour l'insertion.

2. Tables de hachage

Pour ses dictionnaires, Python a choisi d'utiliser les tables de hachage, ce qui est la manière usuelle de créer des dictionnaires modifiables. Au programme de terminale en NSI figurent les arbres de recherche qui sont une autre manière d'implémenter des dictionnaires, plutôt persistants.
Voici en quelques mots le principe d'une table de hachage. Tout d'abord, Python dispose d'une fonction qui à tout objet persistant `1` associe un nombre entier. Cette fonction s'appelle « fonction de hachage » et nous n'allons pas nous préoccuper de savoir comment elle fonctionne. Par contre nous supposerons que son temps d'exécution est en $O(1)$. Remarquons tout de même qu'il est évident qu'une telle fonction existe puisque tout objet est finalement enregistré comme une suite de bits, laquelle peut toujours être interprétée comme un nombre écrit en base deux. Cette fonction s'appelle `hash`.
Ensuite, une table de hachage est un tableau de listes d'association. Notons d la longueur du tableau principal. Pensons-y comme à une commode contenant d tiroirs. Chacun de ces tiroirs va contenir une liste de couples `(clef, valeur)`. Et c'est la fonction de hachage qui va indiquer dans quel tiroir doit être rangée chaque donnée. Pour être précis, soit $(c, v) \in C \times V$ un couple `(clef, valeur)`. Celui-ci sera ajouté dans la case d'indice `hash(c) % d` de notre table. La réduction modulo d nous assure de tomber sur une case valide de la table. Voici un exemple pour d = 3. Initialement,

une table de hachage vide ressemblera à :

```
[ [],
  [],
  []
]
```

Je veux associer la valeur 2 à la clef `"bananes"`. Le `hash` de `"banane"` est `-1929216976289115725`, ce qui modulo 3 donne 1, donc notre couple `("banane", 3)` ira dans la case 1. Et notre table devient :

```
[ [],
  [(" banane ", 3)],
  []
]
```

J'associe à présent la valeur 5 à la clef `"celeri"`. Sachant que `hash("celeri") % 3` vaut 2, j'obtiens :

```
[ [],
  [("banane", 3)],
  [("celeri", 5)]
]
```

Enfin, j'ajoute 6 navets. Mais `hash("navet") % 3` vaut 1 : ils arrivent dans la même case que les bananes :

```
[ [],
  [("banane", 3), ("navet", 6)],
  [("celeri", 5)]
]
```

On comprend le principe de la recherche : étant donnée une clef, il suffit de calculer son `hash` pour savoir dans quelle case de la table de hachage la chercher. Si par malheur il y avait beaucoup d'éléments dans cette case, la recherche pourrait être lente… C'est pourquoi une « bonne » table de hachage devra faire en sorte que ces cases ne soient pas trop remplies. D'une part, une « bonne » fonction de hachage devra faire en sorte de bien répartir les éléments, et d'autre part, la table sera automatiquement agrandie dès que le ratio nombre d'éléments sur nombre de cases devient trop important (supérieur à 2/3 en Python). Cette agrandissement coûtera du temps mais arrive rarement… Au final, la complexité d'une insertion et d'une lecture dans une table de hachage sera « en moyenne » en O(1).

> **Remarque**
> Python ne permet pas que les clefs soient mutables, car la moindre modification d'une clef empêcherait par la suite de retrouver la valeur associée.

> **Remarque**
> Le programme suivant permet de comparer l'efficacité de l'utilisation d'un dictionnaire avec celle d'une liste lorsqu'il nécessaire de rechercher une valeur :

```python
import random
import time

L = []

dict = {}

for i in range(2000000):
    elt = random.randint(0, 10**5)
    L.append(elt)
    if elt in dict:
        dict[elt] += 1
    else :
        dict[elt] = 1

elt = random.randint(0, 10**5)
t0 = time.time()
print(elt in L)
t1 = time.time()
print(t1 - t0)

t0 = time.time()
print(elt in dict)
t1 = time.time()
print(t1 - t0)
```

Exercice 2.4 (Tri par dénombrement)

La méthode de tri que nous allons présenter ici ne s'applique qu'aux tableaux d'entiers. En outre, ce tri n'est pas en place : nous allons créer un tableau distinct du tableau initial, qui contiendra les mêmes éléments mais dans l'ordre croissant.
Soit `t` un tableau d'entiers. Pour trier `t`, la méthode consiste en :

- Déterminer l'ensemble des nombres apparaissant dans t, et combien de fois chacun apparaît.
- Grâce à ces informations, construire un tableau `ttrié` contenant les mêmes éléments, mais dans l'ordre.

Comme toujours la première question à se poser est la manière d'enregistrer les données nécessaires. Ici, il nous faut enregistrer pour chaque valeur apparaissant dans `t` le nombre de fois qu'elle y apparaît.
On pourrait utiliser un tableau `nb` tel que pour tout `i`, `nb[i]` contienne le nombre de `i` dans `t`. Ceci aurait plusieurs défauts :

- Éventuellement de nombreuses cases inutiles (si `t` contient 12, le tableau `nb` devra avoir au moins 13 cases, et tout $i \in [0, 12]$ qui n'apparaît pas dans `t` donne lieu à une case inutile).
- Que faire si `t` contient des nombres négatifs ?
- La création de `nb` va être laborieuse car il faudra l'agrandir à chaque fois qu'on rencontre un élément de `t` plus grand que les précédents.

Au final, l'idéal ici est d'utiliser un dictionnaire, qui à chaque élément de `t` associe son nombre d'occurrence.

1. Ecrire une fonction pour créer ce dictionnaire.

2. Maintenant, il s'agit de construire le tableau trié à l'aide du dictionnaire. pour parcourir dans l'ordre toutes les valeurs prises par t. Le plus simple est d'utiliser une boucle for, depuis le minimum jusqu'au maximum de t. Pour disposer de ces extrema, modifier la fonction précédente pour qu'elle retourne les extremas en plus du dictionnaire.
3. Ecrire la fonction finale.

Exercice 2.5 (Doublons)

1. Écrire une fonction qui prend en entrée un tableau `t` et qui renvoie un dictionnaire dont les clefs sont les éléments de `t`.
2. En déduire une fonction qui prend en entrée un tableau `t` et qui renvoie un tableau avec les mêmes éléments mais sans doublon. Quelle est sa complexité ?

Exercice 2.6 (Numérotation)

Écrire une fonction `numerotation` prenant en entrée un tableau `t` et renvoyant un couple `(t', d)` tel que :

- t' contient les mêmes éléments que t mais sans doublon ;
- d est un dictionnaire associant à chaque élément de t' son indice dans t'.

Cette fonction sert régulièrement car elle permet d'associer à n'importe quelle liste de n'importe quel type d'éléments des numéros qui les identifient de manière unique. Par exemple nous pourrons numéroter les sommets d'un graphe afin de créer la matrice qui indique comment sont reliés ces sommets.

Exercice 2.7 (compression zip très simplifiée)

1. Écrire une fonction `compression` qui prendra en entrée un tableau de mots `t` et qui renverra un couple formé de :
   - un dictionnaire tel que celui obtenu dans l'exercice précédent, qui permet de numéroter les éléments le `lt` ;
   - un tableau d'entiers obtenu en remplaçant chaque élément de t par son numéro.
2. Écrire une fonction `decompression` réciproque de la précédente.
3. *Mise en pratique* : À présent, on veut écrire et lire les données compressées dans un fichier. Nous allons donc travailler la lecture et l'écriture dans un fichier.
   (a) Écrire une procédure `tab_vers_fichier` permettant d'enregistrer le contenu d'un tableau dans un fichier. On convient d'écrire un élément par ligne.
   (b) Écrire une procédure analogue `dico_vers_fichier`. On convient d'utiliser une ligne par entrée du dictionnaire, chaque ligne étant de la forme `mot:numéro`.
   (c) Écrire une fonction `tab_de_mots_of_fichier` qui prend un fichier enregistré dans un fichier .txt et renvoie la liste de mots correspondants.
   On rappelle que si ligne est une ligne d'un fichier texte, alors `ligne.strip().split(' ')` renvoie le tableau des mots dans cette ligne.
   (d) Rassembler tous les morceaux pour obtenir une procédure prenant en entrée un fichier texte et créant un fichier contenant le texte compressé. Vérifier si le fichier obtenu est plus léger que le fichier source.
   (e) Écrire la fonction de décodage correspondante.
   Le fichier `ltdme80j-p.txt` vous permettra de tester vos programmes.