Dictionnaires

Cyril Charignon

22 janvier 2021

Table des matières

1 Dictionnaires

1.1 Description du type

Décrire un type revient essentiellement à donner les opérations permises, voici donc les opérations permises par un dictionnaire.

Nous fixons deux ensembles C et V. L'ensemble C sera appelé l'ensemble des « clefs » et V l'ensemble des « valeurs ». Un dictionnaire permet d'associer à un élément de C un élément de V. Autrement dit, un dictionnaire est une fonction (au sens mathématique) de C dans V.

Selon le point de vue adopté ici, une clef peut ne pas avoir de valeur associée, autrement dit un dictionnaire n'est pas forcément une « application » de C vers V.

Les opérations permises par le type des dictionnaires sont :

- Créer un dictionnaire vide.
- Ajouter une association : étant donné $c \in C$ et $v \in V$, décider que v sera associée à c. Si une valeur était déjà associée à c, la nouvelle écrase l'ancienne.
- Lire la valeur associée à une clef : étant donné $c \in C$, renvoyer la valeur associée à c. Si aucune valeur n'est associée à c, une erreur est déclenchée.

De plus, un dictionnaire doit permettre de réaliser ces trois opérations de base efficacement. Selon les implémentations, le temps d'exécution sera en $O\big(\log(n)\big)$, où n est le nombre de données enregistrées, voire en $O(1)$. Se reporter au paragraphe 1.4.

À ces trois opérations, on rajoute souvent une opération permettant de supprimer une entrée, et une permettant de savoir si une clef est présente dans un dictionnaire (pour éviter un `try`... `except` pour rattraper l'erreur déclenchée par une clef non présente).

En outre, on peut aussi ajouter une opération permettant de renvoyer toutes les clefs d'un dictionnaire, ce qui permettra de parcourir toutes les données du dictionnaire. Ceci dit, si le but principal est de parcourir toutes les données enregistrées, un simple tableau ferait tout aussi bien l'affaire, l'intérêt du dictionnaire étant de pouvoir trouver (ou tester la présence) une clef précise très rapidement.

1.2 En Python

Le type des dictionnaires est présent nativement dans Python. En outre les dictionnaires de Python sont *modifiables*. Voici la syntaxe des opérations de base ci-dessus, où l'on a au besoin $c \in C$ et $v \in V$:

- Créer un dictionnaire vide : `d = {}`.
- Associer v à la clé c : `d[c] = v`.
- Renvoyer la valeur associée à la clé c : `d[c]`.
- Voir si c est une clef de `d` : `c in d`.
- Supprimer l'entrée correspondant à c dans `d` : `del d[c]`. Déclenche une erreur si la clef c n'est pas présente dans `d`.

  Et oui : cette syntaxe est troublante... Un `del(d, c)` aurait été plus clair.

  On peut préférer utiliser `d.pop(c)` qui en plus renvoie l'élément supprimé.
- La méthode `keys` renvoie l'ensemble des clefs de `d`. On peut donc parcourir un dictionnaire à l'aide d'une boucle de la forme `for c in d.keys():`.

NB : il serait maladroit d'utiliser `c in d.keys()` pour tester si la clef `c` est dans le dictionnaire `d`. En effet, ceci conduirait Python à calculer la liste de toutes les clefs de `d`, puis à voir si `c` est parmi elles, pour une complexité en $O(n)$ où n est le nombre de clefs dans `d`, alors que précisément un dictionnaire est optimisé pour rechercher efficacement une clef particulière.

On remarquera la présence de procédures permettant de modifier un dictionnaire. Ainsi les dictionnaires de Python sont ils modifiables, avec les conséquences habituelles (faire attention en les copiant, possibilité de les modifier par une procédure, ...).

En outre, on retiendra que ces opérations de base sont, en gros, en $O(1)$. Le paragraphe 1.4 précisera le sens de ce « en gros ».

1.3 Premier exemple : dépouillement d'une urne

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", ...]
```

Nous voulons déterminer le vainqueur de l'élection, en un seul parcours de l'urne. Le plus pratique est d'utiliser un dictionnaire qui à chaque nom associera son nombre de voix.

Notons qu'un des avantages d'un dictionnaire est qu'il n'y a pas besoin de savoir à l'avance qui sont les candidats, ni même combien il y en a.

On commence par une procédure permettant d'incrémenter la valeur associée à une clef s'il y en a, ou de l'initialiser à 1 dans le cas contraire :

```
def incr_dico(d,c):
    """ Entrée : Un dictionnaire d
                 Une clef c
        Sortie : Rien (ceci est une procédure)
        Effet : - Si c est une clef dans d, la valeur associée est incrémentée
                - Sinon, elle est initialisée à 1.
    """
    if c in d:
        d[c] = d[c] + 1
    else:
        d[c] = 1
```

On écrit alors facilement le programme final :

```
def dépouillement(urne):
    d = {}
    nb_voix_max = 0
    vainqueur = ""
    for nom in urne:
        incr_dico(d, nom)
        if d[nom] > nb_voix_max :
            nb_voix_max = d[nom]
```

```
            vainqueur = nom
    return vainqueur
```

Remarque : Cette fonction ne gère pas les cas d'égalité entre deux candidats vainqueurs.

1.4 Sous le capot : brève description d'une table de hachage

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

1.4.1 Liste d'association

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 que je sois 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, associée 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` est `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.

1.4.2 Table de hachage

Pour ses dictionnaires, Python a choisi 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`.

1. 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.

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"` (mettons que je sois en train de gérer le stock d'une épicerie). Le hash de `"banane"` est -4289889455651360333, ce qui modulo 3 donne 1, donc notre couple `("banane", 3)` ira 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 $O(1)$.

1.5 Deuxième exemple : tri par dénombrement

1.5.1 Programmation

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 :

1. Déterminer l'ensemble des nombres apparaissant dans `t`, et combien de fois chacun apparaît.
2. 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.

Non, l'idéal ici est d'utiliser un dictionnaire, qui à chaque élément de `t` associe son nombre d'occurrence.
Commençons par écrire une fonction pour créer ce dictionnaire. On réutilise `incr_dico` de la partie 1.3.

```
def compte(t):
    """ Entrée : un tableau t
        Sortie : un dictionnaire qui à chaque élément de t associe
            son nombre d'occurrences.
    """

    res = {}
    for x in t:
        incr_dico(res, x)
    return res
```

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, modifions la fonction précédente :

```
def compte(t):
    """ Entrée : un tableau t
        Sortie : un triplet (dictionnaire qui à chaque élément de t
            associe son nombre d'occurrences, mir(t), max(t)).
    """

    res = {}
    mini, maxi = t[0], t[0]
    for x in t:
        incr_dico(res, x)
        mini = min(mini, x)
        maxi = max(maxi, x)
    return res, mini, maxi
```

On écrit alors la fonction finale :

```
def tri_dénombrement(t):
    nb, mini, maxi = compte(t)
    res = []
    for i in range(mini, maxi + 1):
        if i in nb:
            # Il faut rajouter nb[i] fois i dans res.
            for k in range(nb[i]):
                res.append(i)
    return res
```

Remarque : On peut par une petite pythonnerie raccourcir un peu le code : la méthode `get` d'un dictionnaire permet de renvoyer la valeur associée à une clef, en précisant une valeur par défaut à renvoyer si cette clef n'est pas présente. Ainsi, dans la fonction précédente, un `nb.get(i, 0)` renverrait le nombre de i dans `t`, si i apparaît effectivement dans `t`, et 0 sinon.

La fonction peut donc se réécrire :

```
def tri_dénombrement(t):
    nb, mini, maxi = compte(t)
    res = []
    for i in range(mini, maxi + 1):
        # Il faut rajouter nb[i] fois i dans res.
        for k in range(nb.get(i, 0)):
            res.append(i)
    return res
```

Autre remarque : Si on veut augmenter encore la clarté du code, on peut externaliser la boucle intérieur, en créant une procédure prenant en entrée deux entiers i et n ainsi qu'un tableau `t` et dont l'effet est de rajouter n fois i à la fin de `t`.

Dernière remarque : On pourrait être tenter de remplacer la boucle par un `for i in nb.items():`. Mais ceci ne fonctionne pas car la méthode `items` n'a aucune raison de renvoyer les clefs du dictionnaire dans l'ordre. Alors bien sûr on pourrait utiliser `sorted(nb.items())`, mais alors on a une complexité de $k \log k$, où k est le nombre de valeurs distincter dans le tableau initial.

1.5.2 Complexité

La terminaison de notre fonction est claire puisqu'elle ne contient ni récursivité, ni boucle conditionnelle. La complexité par contre n'est pas évidente à analyser. Soit `t` un tableau et n son nombre d'éléments, et calculons la complexité de `tri_dénombrement(t)`.

Tout d'abord, `incr_dico` est toujours en $O(1)$, d'où `compte(t)` est en $O(n)$.

On constate ensuite que la complexité de la fonction principale va dépendre du minimum et du maximum de `t`. Notons m et M ces deux nombres.

- La boucle principale s'effectue $M - m$ fois.
- Pour chaque valeur de i, la boucle intérieur tourne au maximum n fois.

Ceci nous indique donc une complexité en $O\big((M-m)n\big)$.

Mais ce résultat, quoique correct, est beaucoup trop grossier ! En effet, la boucle intérieure s'effectue *au total* n fois car à chaque itération elle ajoute un élément à `res`, et celui-ci finit avec n éléments. Ainsi, le coût total généré par cette boucle est en $O(n)$.

Par ailleurs, la boucle principal à un coût en $O(M-m)$.

Donc au total, le coût de notre tri est en $O\left(M-m+n\right)$.

Il s'agit donc d'un tri particulièrement efficace si l'amplitude des nombres à trier n'est pas trop grande.

2 Exercices

Exercices : dictionnaires

Exercice 1. ** Dédoublonnage

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 de dédoublonnage. Quelle est sa complexité ?

Exercice 2. ** 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.

Commentaire : Pour définir le cardinal d'en ensemble E, on trouve un entier n et une bijection $\phi : [\![0, n[\![\to E$. On dit alors que E est de cardinal n, et ϕ s'appelle une « énumération » de E. C'est exactement ce que nous avons fait ici : l'énumération est la fonction $i \mapsto t'[i]$ (injective car t' n'a pas de doublon, surjective à condition de prendre $n = len(t')$).

Exercice 3. ** 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 :
   (a) un dictionnaire tel que celui obtenu dans l'exercice 2, qui permet de numéroter les éléments le `l` ;
   (b) 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.

Les fichiers `around the world.txt` et `vice et versa.txt` vous permettront de tester vos programmes.

Quelques indications

1 La première partie est similaire à celle du tri par dénombrement vu en cours.

On utilise ici le dictionnaire comme ensemble : on peut associer n'importe quelle valeur aux clefs, elle ne sera de toute façon pas lue.