Dictionnaires

I. Dictionnaires de données

I.1. Définition

Définition

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

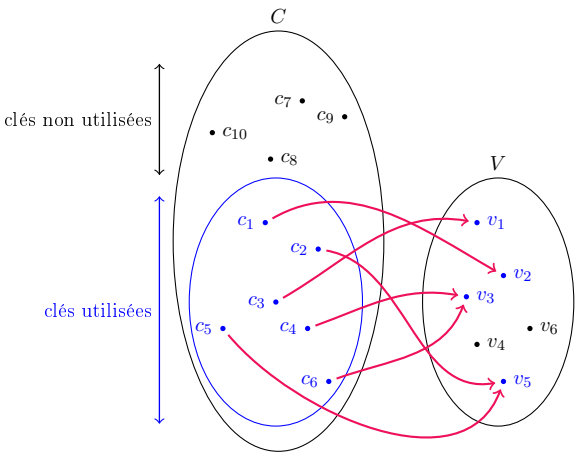

Fig. 1 Représentation informelle d'un dictionnaire

Remarque

- En d'autres termes, un dictionnaire est donc une fonction **injective** de C dans T.
- Un dictionnaire supporte en général les opérations suivantes :
  - × l'*ajout* d'une nouvelles association $(c, v) \in C \times V$ dans T ;
  - × la *suppression* d'une association (c, v) de T ;
  - × l'*existence* d'une association (c, v) dans T pour une clé $c \in C$ donnée ;
  - × la *lecture* de la valeur v associée à une clé c présente dans T.

I.2. Le type `dict`

- Dans les TP précédents, nous avons utilisé des *listes* pour regrouper des données de types élémentaires (`int`, `float`, `bool`). On dit que la liste est un type **construit** (ou séquentiel). L'accès à l'un de ses éléments de fait par son *indice*.
- En **Python**, un dictionnaire est un autre type construit. Il représente un ensemble d'association `clé: valeur`, chaque clé étant unique.
  À la différence des listes, qui sont indexées par des entiers, les dictionnaires sont indexés par des clés qui peuvent être de n'importe quel type usuel non modifiable : des chaînes de caractères (`str`), des tuples (`tuple`) ou des types élémentaires (`int`, `float`, `bool`).

I.3. Création d'un dictionnaire

I.3.a) Définition directe

- La création d'un dictionnaire se réalise en suivant la syntaxe :

  `{c1 : v1, ..., cn : vn}`

  où :
  - `c1`, ..., `cn` sont des clés, nécessairement deux-à-deux distinctes (de type immuable),
  - `v1`, ..., `vn` sont les valeurs qui leur sont associées.

```
1  nombre_de_roues = {'voiture': 4, 'vélo': 2, 'tricycle': 3}
```

  Ici, la valeur associé à la clé `'vélo'` est l'entier 2. Comme précisé précédemment, les clés, comme les valeurs, peuvent être de différents types.

Commentaire

À la différence d'une liste, un dictionnaire n'est pas une collection *ordonnée*.

  Un autre exemple de dictionnaire qui associe à chaque chiffre sa valeur numérique :

```
1  valeur = { 'zéro': 0, 'un': 1, 'deux': 2, 'trois': 3, 'quatre': 4, 'cinq': 5}
```

- La commande `{ }` crée un dictionnaire vide.

Exercice 1
Construire un dictionnaire `jours_par_mois` qui associe à chaque mois d'une année non bissectile, son nombre de jours. On représentera les mois par des chaînes de caractères.

I.3.b) Utilisation de la fonction `dict`

La fonction `dict` permet également de créer des dictionnaires.

```
1  dico_en_fr = dict(yes = 'oui', no = 'non', why = 'pourquoi')
```

I.3.c) À partir de tuples à 2 éléments

On peut également créer un dictionnaire à l'aide d'une liste de tuples à 2 éléments.

```
1  liste_en_nb = [('one', 1), ('two', 2), ('three', 3)]
2  dico_en_nb = dict(liste_en_nb)
```

I.3.d) Définition par compréhension

Comme pour les listes, on peut également définir un dictionnaire par compréhension.

```
1  suite_carres = {x: x**2 for x in range(5)}
```

I.3.e) À l'aide d'une structure itérative

Enfin, un dictionnaire peut également être construit à l'aide d'une structure itérative.

```
1  chiffres = ['zéro', 'un', 'deux', 'trois', 'quatre', 'cinq']
2  valeurs = {}
3  for i in range(6) :
4      c = chiffres[i]
5      valeurs[c] = i
```

I.3.f) En effectuant une copie

Comme pour les listes, on peut effectuer une copie d'un dictionnaire avec la méthode `.copie`.

```
1  suite2 = suite_carres.copie()
```

I.4. Opérations sur les dictionnaires

I.4.a) Ajout de clés et de valeurs

L'instruction `d[clé] = valeur` peut avoir deux effets :

× si la clé `clé` n'est pas présente dans le dictionnaire `d`, alors cette instruction ajoute l'association `clé: valeur` dans le dictionnaire `d`.

```
1  dico_en_fr['because'] = 'seulement'
```

× si la clé `clé` est présente dans le dictionnaire `d`, alors cette instruction modifie la valeur associée à la clé `clé` dans le dictionnaire `d`.

```
1  dico_en_fr['because'] = 'parce que'
```

I.4.b) Supprimer des entrées

- Comme pour les listes, c'est la commande `del` qui permet de supprimer des entrées d'un dictionnaire.

```
In [1] :   del dico_en_fr['no']
In [2] :   print(dico_en_fr)
Out[2] :   {'yes': 'oui', 'why': 'pourquoi', 'because': 'parce que'}
```

- La méthode `.clear` permet de vider complètement un dictionnaire.

```
In [3] :   suite_carres.clear()
In [4] :   print(suite_carres)
Out[4] :   {}
```

I.4.c) Tester l'appartenance à un dictionnaire

L'instruction `in` permet de tester l'appartenance d'une clé à un dictionnaire.

```
In [5] :   print('yes' in dico_en_fr)
Out[5] :   True
```

L'instruction `in` `NE` permet `PAS` de tester l'appartenance d'une valeur à un dictionnaire.

```
In [6] :   print('oui' in dico_en_fr)
Out[6] :   False
```

I.4.d) Consultation d'un dictionnaire

Comme un dictionnaire est une structure séquentielle, on retrouve les commandes spécifiques à ce type de données.

- La fonction `len` permet de calculer la longueur d'un dictionnaire.

```
In [7] :   len(dico_en_fr)
Out[7] :   3
```

- On accède à la valeur associée à une clé avec la syntaxe `[clé]`.

```
In [8] :   dico_en_fr['why']
Out[8] :   'pourquoi'
```

- La méthode `.keys` donne la liste de toutes les clés du dictionnaire.

```
In [9] :   dico_en_fr.keys()
Out[9] :   dict_keys(['yes', 'why', 'because'])
```

- La méthode `.values` donne la liste de toutes les valeurs du dictionnaire.

```
In [10] :   dico_en_fr.values()
Out[10] :   dict_keys(['oui', 'pourquoi', 'parce que'])
```

- La méthode `.items` donne la liste de tous les couples `(clé, valeur)`.

```
In [11] :   dico_en_fr.items()
Out[11] :   dict_items([('yes', 'oui'), ('why', 'pourquoi'), ('because',
            'parce que')])
```

I.5. Parcours d'un dictionnaire

On peut enfin utiliser un parcours de dictionnaire pour effectuer une énumération. Il peut être utile de parcourir :

- l'ensemble des clés d'un dictionnaire :

```
for c in dico_en_fr.keys :
    print(c)
```

  On pourra aussi écrire plus simplement :

```
for c in dico_en_fr :
    print(c)
```

- l'ensemble des valeurs d'un dictionnaire :

```
for v in dico_en_fr.values :
    print(v)
```

- l'ensemble des couples (clé, valeur) d'un dictionnaire :

```
for (c,v) in dico_en_fr.items :
    print(c,v)
```

II. Mise en oeuvre pratique d'un dictionnaire

II.1. Complexité temporelle

Il est communément admis que les quatre opérations de base d'un dictionnaire (ajout, suppression, existence d'une association, lecture d'une valeur associée à une clé) se réalisent avec une complexité temporelle constante, c'est-à-dire en $\underset{\to(}{O}1)$. Pour comprendre comment une telle performance est possible, nous allons décrire quelques méthodes d'implémentation de cette structure.

- Si les clés étaient des entiers compris entre 0 et $m-1$, le problème serait simple : il suffirait d'utiliser un tableau de taille m. Comme ce n'est en général pas le cas, on se ramène à cette situation en utilisant une fonction

$$f : C \to [\![0, m-1]\!]$$

  Cette fonction f associe à chaque clé $c \in C$ un entier dans $[\![0, m-1]\!]$.

- Cependant, le nombre de clés dans C étant souvent très important, le cardinal de C est alors très supérieur à la valeur de m. La fonction f ne peut alors être injective. Autrement dit, il existe forcément des couples $(c_1, c_2) \in C^2$ tels que :

$$c_1 \neq c_2 \qquad \texttt{ET} \qquad f(c_1) = f(c_2)$$

  Un tel couple (c_1, c_2) est appelé une **collision**. Il faut alors trouver une solution pour gérer ces collisions lorsqu'elles se produisent.

- Une solution possible pour résoudre les collisions consiste à stocker, dans un même « paquet », les valeurs dont les clés sont rentrées en collision. Si chaque paquet ne contient qu'un petit nombre de valeurs, on retrouvera rapidement la valeur associée à une clé en la cherchant dans son paquet.

- Si on considère que, pour toute clé $c \in C$, le calcul de $f(c)$ se réalise en temps constant et que chaque paquet est de petite taille, les quatre opérations de base d'un dictionnaire se réalisent effectivement avec une complexité temporelle constante.

Exemple
Illustrons cela sur un exemple. On considère 6 associations (c_1, v_1), (c_2, v_2), …, (c_6, v_6). On rappelle que les clés c_1, …, c_6 sont deux à deux distinctes.
Supposons que l'on dispose d'une fonction f telle que :

× $f(c_1) = f(c_4)$,

× $f(c_3) = f(c_5) = f(c_6)$.

Dans ce cas :

× les valeurs v_1 et v_4 seront stockées dans le même paquet,

× la valeur v_2 sera stocké dans un paquet,

× les valeurs v_3, v_5, v_6 seront stockées dans le même paquet.

On peut représenter cette situation avec le dessin suivant.

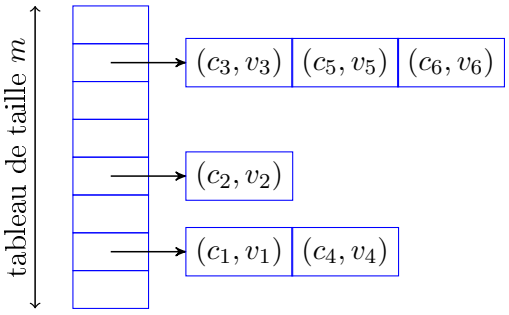

Fig. 2 Les clés qui entrent en collision sont stockées dans le même paquet

II.2. Fonction de hachage

Pour définir la fonction f, on utilise une fonction h, appelée **fonction de hachage**. La fonction h associe à chaque clé $c \in C$ un entier relatif (pas forcément entre 0 et $m-1$). On définit alors la fonction f en posant :

$$f(c) \equiv h(c)\ [m]$$

Pour obtenir une implémentation efficiente, une fonction de hachage doit :

× être facile à calculer, pour conserver une complexité temporelle constante.

× avoir une distribution la plus uniforme possible,pour minimiser le nombre de collisions.

Remarque

- Pour tout $c \in C$, on note $r_m(c)$ le reste de la division euclidienne de $h(c)$ par m.
  Pour que la fonction h fournisse une bonne répartition des clés dans les différents emplacements du tableau (peu de valeurs par case du tableau), il faut, à la louche, que pour tout $i \in [\![0, m-1]\!]$, la probabilité que $r_m(c)$ soit égale à i soit de l'ordre de $\dfrac{1}{m}$.
- Dans ces conditions, si on note k le nombre de clés distinctes de T, la probabilité pour qu'il y ait au moins une collision est égale à :
  $$1 - \frac{m!}{m^k\,(m-k)!}$$
  Par exemple, pour $m = 1\,000\,000$, la probabilité pour qu'il y ait collisoin dépasse 50% lorsque le nombre de clés dépasse 1 200. Si ce nombre de clés vaut 2500, alors la probabilité qu'il y ait collision dépasse 95%.
  Ces chiffres montrent que les collisions sont, quoi qu'on fasse, rapidement inévitables.

Choix de la fonction de hachage
Le choix de la fonction de hachage est un problème complexe que l'on abordera pas. On peut par contre noter que **Python** propose une fonction de hachage : si `x` est un objet immuable (`int`, `str`, `float`, ...) alors `hash(x)` renvoie un entier répondant peu ou prou aux exigences d'une fonction de hachage.

```
In [12] : hash(12345)   # un entier est immuable
Out[12] : 12345
In [13] : hash('12345')   # une chaîne de caractère est immuable
Out[13] : 3733634754630841124
In [14] : hash((1, 2, 3, 4, 5))   # un tuple est immuable
Out[14] : -5659871693760987716
In [15] : hash([1, 2, 3, 4, 5])   # une liste est mutable
TypeError : unhashable type: 'list'
```

Remarque

- Une fonction de hachage a d'autres usages. Par exemple, lorsque l'on choisit un mot de passe sur un site sécurisé, ce dernier ne va pas stocker le mot de passe en clair. Il va stocker le résultat de l'évaluation d'une fonction de hachage h au mot de passe. Comme il est très difficile de trouver les antécédents d'un entier par la fonction h, pirater le site ne mettra pas en cause la sécurité du mot de passe.
- Une autre application des fonctions de hachage est le contrôle d'intégrité d'un fichier informatique : à chaque fichier est associé une valeur par une fonction de hachage (le MD5 (Message Digest 5) par exemple) qui permet de vérifier si un fichier téléchargé a été transmis correctement.

III. Résolution des collisions

III.1. Résolution des collisions par chaînage

Comme dit précédemment et vu en Figure 2, une première solution consiste, pour deux clés c_1 et c_2 entrant en collision, à stocker les associations (c_1, v_1) et (c_2, v_2) dans un même paquet (une liste par exemple).
Pour trouver la valeur associée à une clé c présente dans le dictionnaire, on procède alors en deux temps :

1) on calcule $f(c)$ (on rappelle : $f(c) \equiv h(c)\ [m]$) pour déterminer l'emplacement du tableau où se trouve le paquet contenant l'association recherchée.

2) on procède à une recherche naïve dans le paquet pour trouver cette association.

Cette méthode présente l'avantage de pouvoir contenir un nombre k de clés plus grand que le nombre de cases du tableau.

Remarque
Si on note $\alpha = \dfrac{k}{m}$ le taux de remplissage du dictionnaire, sous une hypothèse de hachage uniforme, les paquets auront une taille moyenne de α. La recherche d'une association dans le dictionnaire utilisera donc un nombre de comparaison en moyenne de l'ordre de α.

III.2. Adressage ouvert

Une autre solution consiste, en cas de collision, à chercher un emplacement libre dans lequel déposer la nouvelle association. La recherche d'un emplacement libre porte le nom de **sondage**.

× Le sondage **linéaire** consiste à partir de $i = f(c)$ et à chercher une place libre en testant successivement les cases d'indices $(i+1)$ $[m]$, $(i+2)$ $[m]$, $(i+3)$ $[m]$, ... jusqu'à trouver un emplacement libre.

× Le sondage **quadratique** procède de même mais en sondant les cases d'indices $(i+1)$ $[m]$, $(i+1+2)$ $[m]$, $(i+1+2+3)$ $[m]$, ...

Remarque

- L'inconvénient d'un sondage linéaire est qu'il y a un risque de former des « agrégats », c'est-à-dire de longues successions de cases contigües occupées, ce qui nuit à la répartition uniforme recherchée. On illustre ci-dessous un exemple de sondage linéaire. On dispose du tableau suivant où l'on a déjà stocké les associations (c_1, v_1) à (c_5, v_5).

0	1	2	3	4	5	6	7	8
(c_5, v_5)		(c_2, v_2)	(c_4, v_4)	(c_3, v_3)			(c_2, v_2)	

Fig. 3 Un exemple de sondage linéaire avant l'association (c_6, v_6)

On doit maintenant stocker l'association (c_6, v_6) pour laquelle on suppose : $f(c_6) = 2$. Ici, on a donc : $m = 9$ et $i = f(c_6) = 2$.

× La case d'indice $i+1$ $[m]$ (c'est-à-dire la case d'indice 3) n'est pas libre. On poursuit donc l'exploration.

× La case d'indice $i+2$ $[m]$ (c'est-à-dire la case d'indice 4) n'est pas libre. On poursuit donc l'exploration.

× La case d'indice $i+3$ $[m]$ (c'est-à-dire la case d'indice 5) est libre. On y stocke donc l'association (c_6, v_6).

0	1	2	3	4	5	6	7	8
(c_5, v_5)		(c_2, v_2)	(c_4, v_4)	(c_3, v_3)	(c_6, v_6)		(c_2, v_2)	

Fig. 4 Un exemple de sondage linéaire après l'association (c_6, v_6)

D'autres solutions plus complexes existent, comme par exemple sonder les cases $i + h'(c)$ $[m]$, $i + 2\,h'(c)$ $[m]$, $i + 3\,h'(c)$ $[m]$, ... où h' est une autre fonction de hachage, mais aucune ne résout complètement le problème de la création d'agrégats.

Évidemment, un adressage ouvert exige que le nombre k de clés soit inférieur à la taille de la table m. En outre, on imagine aisément que, lorsque le rapport $\alpha = \dfrac{k}{m}$ se rapproche de 1, il devient de plus en plus difficile de trouver un emplacement vide. En effet :

× si $\alpha = \dfrac{1}{2}$, un ajout nécessite en moyenne deux sondages,

× si $\alpha = \dfrac{9}{10}$, un ajout nécessite en moyenne dix sondages.

Aussi, lorsque α est trop grand, il est nécessaire de créer une table plus grande (la taille est en général doublée) pour préserver les performances.