Chap.2 : Dictionnaires

Table des matières

1 Dictionnaires

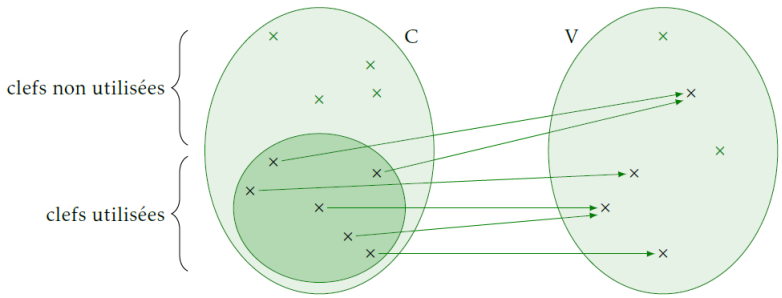
Cette première partie a pour objet l'étude d'une structure de données que vous avez déjà rencontré en première année : les dictionnaires. Cette structure de données répond à la problématique suivante : comment rechercher efficacement de l'information dans un ensemble géré de façon dynamique (c'est à 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.

1.1 Description

Définition 1.1. *Un* ***dictionnaire*** *(ou mieux une table d'association) 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* $\mathrm{C} \times \mathrm{V}$ *tel que pour toute clef* $c \in \mathrm{C}$ *il existe au plus un élément* $v \in \mathrm{V}$ *tel que* $(c, v) \in \mathrm{T}$*. Les éléments de* T *sont appelés des* ***associations****.*



Un dictionnaire supporte en général les opérations suivantes :

- ajout d'une nouvelle association $(c, v) \in \mathrm{C} \times \mathrm{V}$ dans T ;
- suppression d'une association (c, v) de T ;
- existence d'une association (c, v) dans T pour une clef $c \in \mathrm{C}$ donnée ;

- lecture de la valeur v associée à une clef c présente dans T.

En Python, la création d'un dictionnaire se réalise en suivant la syntaxe :

$$\{c1 : \vee 1, \ldots, \text{cn} : \text{vn} \} \text{ où } c_1, \ldots, c_n$$

sont des clefs (nécessairement deux-à-deux distinctes) et $v_1, \ldots, v_n$ les valeurs qui leur sont associées.
Ainsi, { } crée un dictionnaire vide. Si D est un dictionnaire et c une clef :

- ***D[c]=v*** ajoute une nouvelle association si la clef n'est pas présente dans le dictionnaire, et modifie l'association précédente sinon ;
- ***del D[c]*** supprime une association si la clef est présente dans le dictionnaire, et déclenche l'exception KeyError sinon ;
- l'expression ***c in D*** renvoie un booléen indiquant si la clef est présente ou non dans le dictionnaire ;
- ***D[c]*** renvoie la valeur associée à la clef si celle-ci est présente dans le dictionnaire, et déclenche l'exception KeyError sinon.

Exemple 1.2.

```
>>> d = {'clef1':'valeur1', 'clef2':'valeur2', 'clef3':'valeur3'}
>>> type(d)
<class 'dict'>

>>> d['clef1']          #rechercher
'valeur1'
>>> d['clef4']
KeyError: 'clef4'

>>> d['clef2'] = 'nouvellevaleur'          #inserer
>>> d
{'clef3': 'valeur3', 'clef2': 'nouvellevaleur', 'clef1': 'valeur1'}

>>> del d['clef2']          #supprimer
>>> d
{'clef3': 'valeur3', 'clef1': 'valeur1'}
```

1.2 Mise en œuvre pratique d'un dictionnaire

On pourrait être tenté d'implémenter un dictionnaire en utilisant les structures de liste ou de tableau.
Un dictionnaire serait simplement représenté par un tableau de ses éléments ou par une liste chaînée de ses éléments
On obtiendrait alors les complexités suivantes pour les opérations élémentaires :

	Tableau	Liste simplement chaînée
Recherche	$O(n)$	$O(n)$
Insertion	$O(n)$	$O(1)$
Suppression	$O(n)$	$O(n)$

Aucune solution n'est totalement satisfaisante. Dans l'idéal, on souhaiterait atteindre une complexité $O(1)$ pour les trois opérations élémentaires.

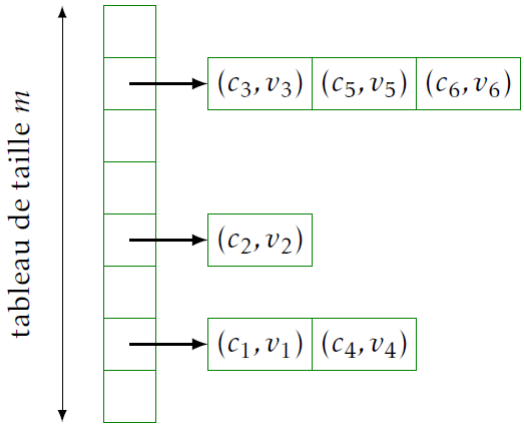
Nous allons étudier une structure de dictionnaire permettant de réaliser ces opérations de base avec une complexité temporelle constante.
Pour comprendre comment une telle performance est possible, nous allons décrire quelques méthodes d'implémentation de cette structure.

Si les clefs 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 $f : \mathrm{C} \to [\![0, m-1]\!]$ associant aux différentes clefs $c \in \mathrm{C}$ possibles un entier dans $[\![0, m-1]\!]$.
Cependant, le nombre de clefs possibles étant très important, le cardinal de C est beaucoup plus grand que m et une telle fonction ne sera pas injective. Il existera donc des couples (c_1, c_2) dans C tels que $c_1 \neq c_2$ et $f(c_1) = f(c_2)$. Un tel couple (c_1, c_2) est appelé une collision, et il faudra trouver une solution pour gérer ces collisions lorsqu'elles se produiront. Une solution possible pour résoudre les collisions consiste à stocker les valeurs dont les clefs sont rentrées en collision dans un même "paquet".

Si la répartition entre les différents paquets est équilibrée, chaque paquet ne contiendra qu'un petit nombre de valeurs, et on retrouvera rapidement la valeur associée à une clef en la cherchant dans son paquet.

La figure suivante illustre cette solution avec :

$$f(c_1) = f(c_4) \text{ et } f(c_3) = f(c_5) = f(c_6)$$



On constate sur cette illustration que les clefs qui entrent en collision sont stockées dans le même paquet.

Si on considère que le calcul de $f(c)$ se réalise en temps constant et que chaque paquet est de petite taille, les quatre opérations se réalisent effectivement avec une complexité temporelle constante.

Fonction de hachage

Pour définir la fonction f, on utilise une fonction h, appelée fonction de hachage, associant un entier à une clef de C, et on définit f en posant :

$$f(c) = h(c) \bmod m$$

Pour conduire à une implémentation efficiente, une fonction de hachage doit :

- être facile à calculer ;
- avoir une distribution la plus uniforme possible.

Cette dernière condition est motivée par le souhait de minimiser le nombre de collisions.

Pour que cette fonction assure une bonne répartition des clefs dans les différents emplacements du tableau, il faut, de manière informelle, qu'étant donné un entier $i \in [\![0, m-1]\!]$, la probabilité que $h(c) \bmod m$ soit égal à i soit de l'ordre de $1/m$.

Dans ces conditions, si k désigne le nombre de clefs 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 $m = 1000000$, la probabilité pour qu'il y ait collision dépasse 50% lorsque le nombre de clefs dépasse 1200 ; pour $k = 2500$ 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

C'est bien évidemment un problème très complexe, que nous n'aborderons pas.
Sachez seulement 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.
Ci-dessous, quelques exemples obtenus avec Python 3.2.5 sur une machine 64 bits :

```
# initialisation d'objets
int_obj = 1
string_obj = 'TSI2-Artaud'
float_obj = 5.55
# Afficher les valeurs de hachage
print("La valeur de hachage de l'integer est:"+ str(hash(int_obj)))
print("La valeur de hachage du string est:"+ str(hash(string_obj)))
print("La valeur de hachage du float est:"+ str(hash(float_obj)))
```

On obtient :

```
La valeur de hachage de l'integer est:1
La valeur de hachage dut string est:977323987
La valeur de hachage du float est:2040108651
```

Remarque 1.3. *Autres usages de la fonction de hachage*

Par exemple, lorsque vous choisissez un mot de passe sur un site sécurisé, ce dernier ne va pas stocker votre mot de passe en clair, mais va stocker le résultat de l'application d'une fonction de hachage h à votre mot de passe. Sachant qu'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 de votre 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 (vous connaissez peut-être le MD5(fonction de hachage cryptographique inventée en 1991 par Ronald RIVEST) qui permet de vérifier si un fichier téléchargé a été transmis correctement.

1.3 Résolution des collisions

1.3.1 Résolution des collisions par chaînage

On l'a déjà dit, une première solution consiste, pour deux clefs 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) (voir figure 2).

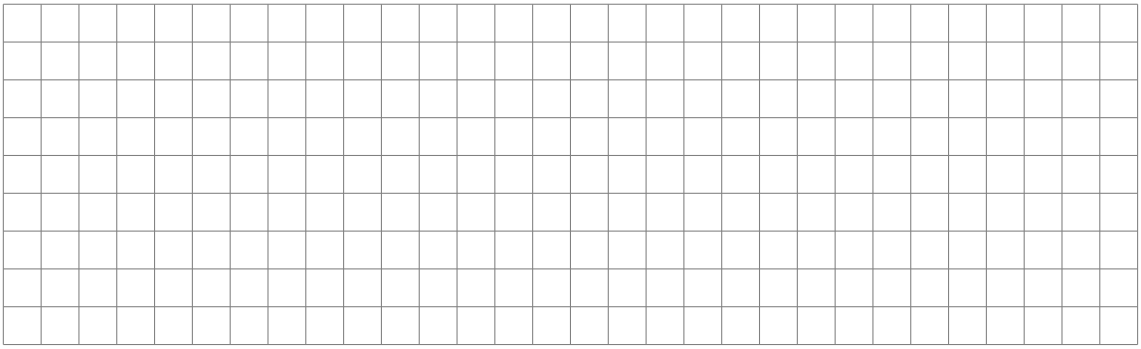
Pour trouver la valeur associée à une clef c présente dans le dictionnaire, on procède alors en deux temps :

- on calcule $f(c) = h(c) \bmod m$ pour déterminer l'emplacement du tableau où se trouve le paquet contenant l'association recherchée ;
- 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 clefs plus grand que le nombre m de cases du tableau.

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

Application 1.4. *On dispose d'une table de hachage de taille m dont les collisions sont résolues par chaînage.*
Exécuter manuellement l'algorithme d'insertion dans la table pour $m = 9$ et $f : c \mapsto c \bmod m$ des clefs suivantes : $5, 28, 19, 15, 20, 33, 12, 17, 10$.



1.3.2 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 $i = h(c)$ et à chercher une place libre en testant successivement les cases d'indices $(i+1) \bmod m, (i+2) \bmod m, (i+3) \bmod m, \ldots$ jusqu'à trouver un emplacement libre.
- le sondage quadratique procède de même mais en sondant les cases d'indices $(i+1) \bmod m, (i+1+2) \bmod m$, $(i+1+2+3) \bmod m, \ldots$

L'inconvénient d'un sondage linéaire est qu'il y a un risque de former des « agrégats", autrement dit de longues successions de cases contigües occupées, qui nuisent à la répartition uniforme recherchée.

Exemple 1.5. *Voici un exemple de sondage linéaire : $f(c_6) = 2$ mais l'association (c_6, v_6) sera stockée dans la case 5 .*

0	1	2	3	4	5	6	7	8
(c_5, v_5)		(c_1, v_1)	(c_4, v_4)	(c_3, v_3)			(c_2, v_2)	

Évidemment, un adressage ouvert exige que le nombre k de clefs soit inférieur à la taille de la table m.
En outre, on imagine aisément que lorsque le rapport $\alpha = \frac{k}{m}$ se rapproche de 1 , il devient de plus en plus difficile de trouver un emplacement vide : si $\alpha = 0,5$ un ajout nécessite en moyenne deux sondages, contre dix lorsque $\alpha = 0,9$.
Aussi, lorsque α devient trop grand il est nécessaire de créer une table plus grande (la taille est en général doublée) pour préserver les performances.

Application 1.6. *Dans cet exercice, on s'intéresse à la méthode de résolution des collisions par adressage ouvert à l'aide d'un sondage linéaire.*

1. *Exécuter manuellement l'algorithme d'insertion dans une table de taille $m = 9$ des clefs 5, 28, 19, 15, 20,* $33, 12, 17, 10$ *avec la fonction de hachage $f(c) = c \bmod 9$.*
2. *On représente le dictionnaire par un tableau $D =$ [None, None, ..., None] de m cases et on prend toujours pour fonction de hachage $f(c) = c \bmod m$. les clefs sont des entiers et les valeurs des chaînes de caractères.*
   Rédiger les fonctions de lecture et d'ajout associées :
   - ***add*** (D, C, v) *ajoute au dictionnaire D l'association (c, v)*
   - ***find*** (D, C) *renvoie la valeur associée à la clef c.*

   On supposera que le dictionnaire n'est pas complètement rempli.
3. *Quel problème se pose lors de la suppression de certaines associations de la table ? Comment peut-on le résoudre ?*
   Rédiger la fonction ***remove***(D, C) *correspondante.*

2 Ensembles

Les dictionnaires permettent aussi de représenter des **ensembles** : il suffit de supprimer les valeurs associées aux clefs pour faire d'un dictionnaire un ensemble de clefs.

En Python, le type correspondant s'appelle ***set***.
Un ensemble peut être défini par l'énumération de ses éléments initiaux : $s = \{2, 3, 5, 7, 11, 13\}$ définit un ensemble.
En revanche, pour qu'il n'y ait pas de confusion avec le dictionnaire vide il faut, pour définir l'ensemble vide, écrire ***set ( )***.

Exemple 2.1. *La conversion d'une liste en ensemble est un moyen simple de supprimer les doublons.*
Dans le script suivant, je tire au hasard 1000 nombres compris entre 0 et 999 puis je supprime les doublons :

```
>>> import numpy.random as rd
>>> s = set(rd.randint(1000, size=1000))
>>> len(s)
643
```

Après suppression des doublons il n'en reste que 630.

Sous Python, les instructions associées sont :

```
>>> e = {'objet1', 'objet2'}
>>> type(e)
<class 'set'>

>>> 'objet1' in e        #rechercher
True
>>> 'objet3' in e
False

>>> e.add('objet3')        #inserer
>>> e
{'objet2', 'objet3', 'objet1'}

>>> e.add('objet3')        #inserer de nouveau
>>> e
{'objet2', 'objet3', 'objet1'}

>>> e.remove('objet2')        #supprimer
>>> e
{'objet3', 'objet1'}
```