Dictionnaires

Cours de CPGE PSI, 2 octobre 2024

Adeline Pierrot

Un dictionnaire permet de stocker des couples (clé,valeur):

`{"flower":"fleur", "sun":"soleil", "moon":"lune"}`

Les clés doivent être distinctes (mais les valeurs peuvent se répéter)
Les clés doivent être d'un type immuable (i.e. dont on ne peut pas changer seulement une partie) $\rightarrow$ tuple ok mais pas liste.
Les valeurs peuvent être de n'importe quel type (y compris listes).
Les types des différentes clés et valeurs peuvent être différents.

Exemple correct: `{'a':2, 'b':2, (2,5):True, 0.5:[3,2,6]}`

Exemple incorrect: `{'a':2, (2,5):True, [3,2,6]:0.5}`
(une liste peut être une valeur mais ne peut pas être une clé).

Autre exemple incorrect: `{'a':2, 'a':3, (2,5):True}`
(les clés doivent être distinctes).

Syntaxe

- Un dictionnaire s'écrit entre accolades { }, les couples (clé, valeur) indiqués sous la forme `clé:valeur`, séparés par des virgules.
```
>>> d = {'a':2, (2,5):True, 0.5:[3,2,6]}
```
- La fonction `len( )` renvoie le nombre de couples d'un dictionnaire:
```
>>> len(d)
3
```
- On accède aux éléments par leur clé, entre crochets: `d[c]` renvoie la valeur associée à la clef c si celle-ci est présente dans le dictionnaire, et provoque une erreur sinon:
```
>>> d['a']
2
>>> d['b']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError:  'b'
```

Opérations sur les dictionnaires

- On peut modifier la valeur associée à une clé: `D[c] = v` modifie la valeur associée à la clé *c* si la clé *c* est présente dans le dictionnaire.

```
>>> d['a']=-1; print(d)
{'a':-1, (2,5):True, 0.5:[3,2,6]}
```

- On peut ajouter un nouveau couple (clé,valeur) à un dictionnaire: `D[c] = v` ajoute un nouveau couple (c,v) si la clé *c* est absente du dictionnaire:

```
>>> d['b']=-1; print(d)
{'a':-1, 'b':-1, (2,5):True, 0.5:[3,2,6]}
```

- L'expression `c in D` renvoie un booléen indiquant si `c` est une clé du dictionnaire `d` (pour tester si une *valeur* est présente il faut parcourir le dictionnaire).

```
>>> 'a' in d
True
>>> -1 in d
False
```

Parcours d'un dictionnaire

Une boucle `for c in d` permet de parcourir les clés d'un dictionnaire (on obtient alors les valeurs par `d[c]`)

`d.keys()` est la liste des clés de d
`d.values()` est la liste des valeurs de d
`d.items()` est la liste des couples (clé,valeur) de d.

Par conséquent
`for c in d.keys()` est équivalent à `for c in d`
`for c in d.values()` permet de parcourir les valeurs de d
`for c,v in d.items()` permet de parcourir les couples clé,valeur de d

Création d'un dictionnaire

- On peut mettre directement les éléments dans le dictionnaire:

```
d = {1:2, 2:4, 3:6}
```

- On peut partir du dictionnaire vide et lui ajouter des éléments un par un:

```
d = {}
for i in range(1,n):
     d[i] = 2*i
```

- On peut créer un dictionnaire "par compréhension" (comme un ensemble mathématique):

```
d = { i:2*i for i in range(1,n)}
```

Copie de dictionnaires

- Attention à l'affectation de dictionnaires :

```
>>> d = {'a':2, 'b':7, 0.5:"ok"}
>>> d2 = d
>>> d2['a'] = 0   # Modification de d2
>>> print(d2)
{'a':0, 'b':7, 0.5:"ok"}
>>> print(d)
{'a':0, 'b':7, 0.5:"ok"} # Modifier l2 a aussi changé L
```

  L'affectation `d2 = d` a juste créé un alias : un nouveau nom référant au même dictionnaire, dont une seule copie figure en mémoire.

- Pour recopier effectivement le dictionnaire, utiliser la méthode `copy`

```
>>> d = {'a':2, 'b':7, 0.5:"ok"}
>>> d2 = d.copy()
>>> d2['a'] = 0   # Modification de d2
>>> print(d2)
{'a':0, 'b':7, 0.5:"ok"}
>>> print(d)
{'a':2, 'b':7, 0.5:"ok"} # Modifier d2 n'a pas changé d
```

Avantage des dictionnaires

On a les complexités (amorties) suivantes:

Pour une liste de taille n:

- Ajout d'un élément: $\mathcal{O}(1)$
- Accès à (= lecture de) la valeur d'un élément d'indice donné: $\mathcal{O}(1)$
- Modification de la valeur d'un élément d'indice donné: $\mathcal{O}(1)$
- Test de la présence d'un élément dans la liste: $\mathcal{O}(n)$

Pour un dictionnaire de taille n:

- Ajout d'un couple (clé,valeur): $\mathcal{O}(1)$
- Accès à (= lecture de) la valeur associée à une clé donnée: $\mathcal{O}(1)$
- Modification de la valeur associée à une clé donnée: $\mathcal{O}(1)$
- Test de la présence d'une clé dans le dictionnaire: $\mathcal{O}(1)$

Hachage

Ce qui permet l'efficacité de la recherche d'une clé dans un dictionnaire est le principe du hachage:

On a une fonction de hachage `h` qui à chaque clé possible associe un entier.

On stocke les couples (clés,valeur) dans un tableau de taille `m` donnée, en mettant le couple `(c,v)` à l'indice `h(c) % m` du tableau.

Le nombre de clés possibles est beaucoup plus grand que le nombre de clés utilisées en pratique dans un dictionnaire, donc pour ne pas gaspiller trop de place on prend `m` plus petit que le nombre de clés possibles.

Par conséquent, il peut y avoir deux clés `c1` et `c2` d'un même dictionnaire telles que `h(c1) % m = h(c2) % m`. On appelle cela une collision.

Pour résoudre les collisions, on peut soit chercher une autre place pour mettre une clé dont la place est déjà occupée, soit mettre plusieurs clés à la même place du tableau en utilisant des listes: le dictionnaire est alors implémenté par un tableau de listes de couples (clés,valeurs).

Fonction de hachage

Une bonne fonction de hachage doit:

- Etre rapide à calculer
- Permettre de bien répartir les éléments du dictionnaire dans la table de hachage. Il faut donc qu'il soit difficile de construire des collisions, c'est-à-dire de trouver deux clés différentes dont l'image par la fonction de hachage est la même.

Les fonctions de hachage sont utiles dans d'autres contextes que l'implémentation des dictionnaires, par exemple en cryptographie.