Les dictionnaires

Dernière mise à jour le : 02/11/2023

Préambule

Les dictionnaires Python ont déjà été utilisés et manipulés en classe de Première. En Python, un **dictionnaire** est une structure native de **tableau associatif**. Ici, nous allons définir le type abstrait `Tableau associatif` et surtout voir que la recherche d'une valeur dans un dictionnaire est beaucoup (beaucoup !) plus rapide que la recherche dans une liste.

Le type abstrait `Tableau associatif`

Un **tableau associatif** est un type abstrait de données (commes les piles, les files, les listes, les arbres, etc.).

Ce type abstrait n'est pas linéaire car il associe des **valeurs** à des **clés**.

Voici les opérations usuelles du type `Tableau associatif` :

- création d'un dictionnaire vide
- ajout d'une nouvelle valeur associée à une nouvelle clé (on parlera de nouveau couple clé-valeur)
- modification d'une valeur associée à une clé existante
- suppression d'un couple clé-valeur
- récupération de la valeur associée à une clé donnée

Exemple : un répertoire téléphonique est un exemple de tableau associatif.

Les dictionnaires Python possèden ces 5 opérations (et d'autres évidemment) :

```
# création d'un dictionnaire vide
d = {}

# ajout de nouveaux couples clé-valeur
d['a'] = 2
print(d)   # affiche {'a': 2}
d['b'] = 5
d['c'] = 3
print(d)  # affiche {'a': 2, 'b': 5, 'c': 3}

# modification d'une valeur associée à une clé
d['a'] = 1
print(d)  # affiche {'a': 1, 'b': 5, 'c': 3}

# suppression d'un couple clé-valeur
del d['b']
print(d)  # affiche {'a': 1, 'c': 3}

# récupération de la valeur associée à une clé donnée
valeur = d['c']
print(valeur)  # affiche 3
```

Mesure du temps d'accès à une valeur dans un dictionnaire

Pour tester si une valeur est présente dans un dictionnaire, on peut utiliser le mot clé `in` :

```
>>> d = {'a': 1, 'b': 5, 'c': 3}
>>> 'a' in d
True
>>> 'e' in d
False
```

Protocole

On veut mesurer le temps d'accès à une valeur dans un dictionnaire, et le comparer à celui permettant d'accéder à un élément dans une liste. Pour cela, on va se placer dans le pire cas : chercher une valeur absente (du dictionnaire et de la liste).

Voici le protocole mis en place que vous pouvez analyser :

```
import time

def fabrique_liste(nb):
    lst = [k for k in range(nb)]
    return lst

def fabrique_dict(nb):
    dct = {k: k for k in range(nb)}
    return dct

def mesures(nb):
    lst = fabrique_liste(nb)
    d = fabrique_dict(nb)

    tps_total = 0
    for _ in range(10):
        t0 = time.time()
        test = 'a' in lst # on cherche une donnée inexistante
        delta_t = time.time() - t0
        tps_total += delta_t
    tps_moyen_lst = tps_total / 10

    tps_total = 0
    for _ in range(10):
        t0 = time.time()
        test = 'a' in d # on cherche une donnée inexistante
        delta_t = time.time() - t0
        tps_total += delta_t
    tps_moyen_d = tps_total / 10

    print(f"temps pour une liste de taille {nb}       : {tps_moyen_lst}")
    print(f"temps pour un dictionnaire de taille {nb} : {tps_moyen_d}")
```

Analyse :

- La fonction `mesures` prend en paramètre un nombre `nb` qui sera la taille de la liste ou du dictionnaire. Dans le corps de cette fonction, la liste `lst` et le dictionnaire `d` sont fabriqués *avant le commencement de la mesure du temps*.
- La liste `lst` contient des nombres (de `1` à `nb`), et le dictionnaire `d` associe à un nombre (de `1` à `nb`) sa propre valeur.
- Dans ces deux structures, nous allons partir à la recherche d'une valeur qui n'a aucune chance de s'y trouver : la chaine de caractères `'a'`.
- On effectue la recherche 10 fois de suite (pour avoir un temps moyen le plus juste possible), on va donc mesurer le temps mis pour chercher la chaine `'a'`, qui n'est présente ni dans la liste `lst` ni dans le dictionnaire `d`. On mesure donc une recherche dans **le pire des cas**.

Mesures

On effectue 4 mesures, les tailles des listes et dictionnaires étant égales et augmentant d'un facteur 10 à chaque fois :

```
>>> mesures(10**4)
temps pour une liste de taille 10000       : 0.000372314453125
temps pour un dictionnaire de taille 10000 : 0.0
```

```
>>> mesures(10**5)
temps pour une liste de taille 100000       : 0.0020259618759155273
temps pour un dictionnaire de taille 100000 : 0.0
```

```
>>> mesures(10**6)
temps pour une liste de taille 1000000       : 0.028008055686950684
temps pour un dictionnaire de taille 1000000 : 0.0
```

```
>>> mesures(10**7)
temps pour une liste de taille 10000000       : 0.2826454401016235
temps pour un dictionnaire de taille 10000000 : 0.0
```

On constate que :

- le temps de recherche dans une liste augmente d'un facteur 10 lorsque la taille de la liste augmente d'un facteur 10
- le temps de recherche dans un dictionnaire reste dans le même ordre de grandeur, proche de 0 seconde, et ce quelle que soit la taille du dictionnaire.

Conclusion

Il y a donc une différence fondamentale entre les temps de recherche d'un élément dans une liste et dans un dictionnaire :

- dans une liste, le coût en temps est **linéaire**, c'est-à-dire qu'il est proportionnel à la taille n de la liste : on écrit que la recherche dans une liste est en $O(n)$.
- dans un dictionnaire, le coût en temps est **constant**, c'est-à-dire qu'il ne dépend pas de la taille du dictionnaire : on écrit que la recherche dans un dictionnaire est en $O(1)$.

Ainsi, si on sait à l'avance que l'on va devoir chercher régulièrement des valeurs dans une structure de données, l'utilisation d'un dictionnaire sera beaucoup plus efficace en temps que pour une liste (à condition que ces structures soient adaptées au stockage de nos données bien sûr).

Remarque : En Python, le **temps d'accès** à un élément d'une liste ou d'un dictionnaire se fait en temps constant dans les deux cas, car les listes de Python sont implémentées par des tableaux dynamiques (voir le cours sur Les listes).

> Comment est-ce possible que la recherche dans un dictionnaire soit en temps constant ? On explique l'idée ci-dessous, mais cela reste hors programme.

Fonctions de hachage (*hors programme*)

Il est important de se rappeler qu'un dictionnaire n'est pas **ordonné** (contrairement à l'objet « dictionnaire » de la vie courante, où chaque mot est classé suivant l'ordre alphabétique).

On n'accède pas à une valeur suivant sa position, mais suivant sa clé.

Dans une liste, lorsqu'on veut savoir si un élément appartient à une liste (problème de la *recherche d'élément*), il n'y a pas (dans le cas général) de meilleure méthode que le parcours exhaustif de tous les éléments de la liste jusqu'à (éventuellement) trouver la valeur cherchée.

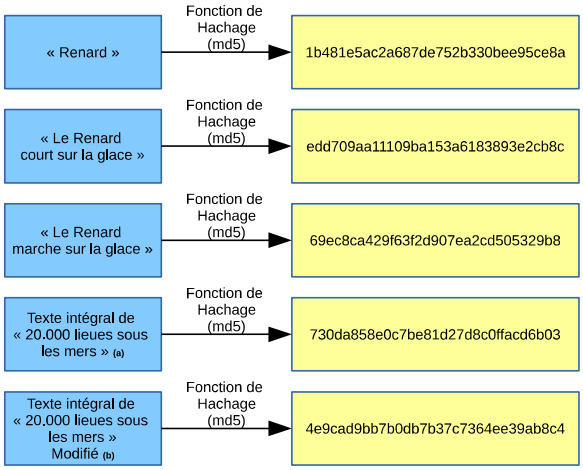
Dans un dictionnaire, on pourrait s'imaginer qu'il va falloir parcourir toutes les clés et regarder les valeurs correspondantes. Il n'en est rien.
Pour comprendre cela nous allons faire un petit détour par les **fonctions de hachage.**

Qu'est-ce qu'une fonction de hachage ?

Une **fonction de hachage** est une fonction qui va calculer une empreinte unique à partir de la donnée fournie en entrée. Elle doit respecter les règles suivantes :

- la longueur de l'empreinte (valeur retournée par la fonction de hachage) doit être toujours la même, indépendamment de la donnée fournie en entrée.
- connaissant l'empreinte, il ne doit pas être possible de reconstituer la donnée d'origine
- des données différentes doivent donner dans la mesure du possible des empreintes différentes.
- des données identiques doivent donner des empreintes identiques.

Exemple : la fonction de hachage `md5` permet de convertir un mot binaire (une chaîne, un fichier, ...) de taille quelconque en un mot de 128 bits représenté par une chaîne hexadécimale de 32 caractères (il y a donc $2^{128} \simeq 10^{39}$ empreintes MD5 différentes).



Exemples de hachages de textes par la fonction md5

Crédits : Unique Nitrogen, CC BY-SA 4.0, via Wikimedia Commons

Remarques :

- Le mécanisme de calcul de la fonction MD5 est complexe, pour en savoir plus allez voir cette page.
- Il est évidemment **impossible** de revenir en arrière, sinon on serait capable de compresser sans perte n'importe quel fichier en une chaîne de 128 bits. Cette impossibilité de trouver une fonction réciproque à la fonction de hachage est très important en cryptographie.
- La fonction MD5 n'est plus utilisée depuis 2005 pour des applications sensibles car des chercheurs ont prouvé qu'il était possible de créer facilement deux messages ayant la même empreinte, ce qui contredit la troisième règle donnée au-dessus.
- Aujourd'hui, on utilise principalement la fonction de hachage SHA-256 (voir ici)

Utilisations concrètes de fonctions de hachage

Vérifier l'intégrité d'un fichier

Lorsque vous téléchargez un fichier important et que vous souhaitez vérifier qu'il n'a pas été corrompu lors du téléchargement (ou avant), vous avez parfois la possibilité de vérifier l'intégrité de votre fichier téléchargé, en calculant une «empreinte» de votre fichier et en la comparant avec celle que vous êtes censée obtenir :

Voilà par exemple ce qui apparaît sur la page de téléchargement de l'image de Linux Mint 21.1 "Vera" (voir ici) :

Capture d'écran du site Linux Mint

En cliquant sur le bouton `sha256sum.txt` on peut voir l'empreinte hachée de l'image de Linux :

```
2df322f030d8ff4633360930a92d78829d10e515d2f6975b9bdfd1c0de769aca *linuxmint-21.1-cinnamon-64bit.iso
f7fb9c0500e583c46587402578547ea56125e0a054097f9f464a2500830c8b25 *linuxmint-21.1-mate-64bit.iso
6fea221b5b0272d55de57f3d31498cdf76682f414e60d28131dc428e719efa8b *linuxmint-21.1-xfce-64bit.iso
```

Il est alors possible de vérifier que le fichier iso téléchargé n'est pas corrompu, en calculant son empreinte avec la fonction de hachage SHA-256 et en vérifiant qu'elle est bien égale à celle indiqué au-dessus. On peut appliquer la fonction de hachage directement dans un terminal comme suit :

```
$ sha256sum -b yourfile.iso
```

Stockage des mots de passe

L'utilisation la plus courante est le stockage des mots de passe dans un système informatique un peu sécurisé. En effet, lorsqu'on crée un compte sur un service en ligne, le mot de passe **ne doit pas être stocké en clair**, une empreinte est générée (le hash du mot de passe) et c'est ce hash du mot de passe qui est stocké sur le serveur. Cela permet d'éviter, en cas de piratage, de protéger les comptes car il n'est pas possible de reconstituer les mots de passe à partir et des empreintes.

Des hackers possèdent des tables avec des empreintes des mots de passe les plus courants (rainbow table) ce qui leur permet de chercher par force brute le mot de passe correspondant à une empreinte dérobée. En particulier, la fonction de hachage MD5 ne résiste pas longtemps à ce type d'attaque : voyez que le mot de passe correspond à l'empreinte `bdc87b9c894da5168059e00ebffb9077` ne résiste pas longtemps sur ce site : https://md5.gromweb.com/. C'est pourquoi, cette fonction de hachage ne doit plus être utilisée !

Retour aux dictionnaires Python

> Quel est le lien entre les fonctions de hachage et les dictionnaires ?

L'idée essentielle est que chaque clé est hachée pour donner une empreinte unique, qui est ensuite transformée en un indice de positionnement dans un tableau.

Le dictionnaire :

```
d = {"pommes": 3, "poires": 0, "bananes": 5}
```

serait donc par exemple implémenté dans un tableau comme celui-ci :

indice	hash	clé	valeur
0	None	None	None
1	23e261452	"bananes"	5
2	None	None	None
3	None	None	None
4	7e1fd3345	"pommes"	3
5	None	None	None
6	8ab561284	"poires"	0
7	None	None	None

On peut remarquer que ce tableau laisse beaucoup de cases vides.

Si je souhaite ensuite accéder à l'élément `d["kiwis"]` :

- le hash de la chaîne `"kiwis"` est calculé. Par exemple, `4512d2202`.
- l'indice de la position (éventuelle) de la clé `"kiwis"` dans mon dictionnaire est calculé à partir de ce hash `4512d2202`. Dans notre exemple, cela pourrait donner l'indice 3.
- Python accède **directement** à cet indice du tableau :
  - si la valeur de la clé sur cette ligne du tableau est `None`, cela signifie que `"kiwis"` n'est pas une clé existante du tableau. C'est notre cas ici car il n'y a rien à la ligne 3.
  - si la valeur de la clé sur cette ligne du tableau est bien `"kiwis"`, la valeur correspondante est renvoyée.

En résumé, Python sait toujours où aller chercher un élément de son dictionnaire : soit il le trouve à l'endroit calculé, soit il n'y a rien à cet endroit calculé, ce qui veut dire que l'élément ne fait pas partie du dictionnaire.

Par ce mécanisme, l'accès à un élément du dictionnaire se fait toujours en temps **constant**.

Il existe une manière de « voir » que Python utilise une fonction de hachage pour implémenter un dictionnaire :

```
>>> mondico = {}
>>> mondico[4] = "foo"  # un nombre peut-il être une clé ?
>>> mondico  # OUI !
{4: 'foo'}
>>> mondico["riri"] = "fifi"  # une chaîne de caractères peut-elle être une clé ?
>>> mondico  # OUI !
{4: 'foo', 'riri': 'fifi'}
>>> mondico[[2,5]] = "loulou"   # une liste peut-elle être une clé ? --> NON
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-9-e3c61f37c6b5> in <module>
        8
        9 # une liste peut-elle être une clé ? --> NON
---> 10 mondico[[2,5]] = "loulou"

TypeError: unhashable type: 'list'
```

Le message d'erreur est explicite : le type `list` que nous avons voulu utiliser comme clé n'est pas hachable, car c'est un type d'objet pouvant être modifié a posteriori tout en gardant la même référence (on dit que c'est un objet **mutable**) :

```
>>> a = [3, 6, 8]
>>> id(a)
2819681505736
>>> a.append(12)
>>> id(a)
2819681505736
```

Ce changement de valeur tout en gardant la même référence détruirait le principe associant à une clé unique une position unique dans le tableau implémentant le dictionnaire.

Ce problème ne se pose pas si la variable désigne une chaîne de caractères, ou un nombre :

```
>>> a = 2023
>>> id(a)
2819682534256
>>> a = a + 1
>>> id(a)
2819682534480
```

Une variable contenant un entier est donc un objet **immuable** car si on modifie la valeur de l'entier, la référence de la variable changera aussi. Comme un dictionnaire a besoin d'avoir des clés dont les références soient définitives, seuls les objets **immuables** peuvent donc servir de clés dans les dictionnaires.

Utilisation de quelques fonctions de hachage

En Python

En Python, la fonction `hash()` est accessible nativement, mais n'est pas à utiliser pour des applications sensibles :

```
>>> hash("Vive la NSI")
-2945528429037095996
>>> hash("vive la NSI")  # un seul caractère différent --> hash totalement différent
-1612067756032566050
```

Pour utiliser la fonction SHA-256, on peut utiliser le module `hashlib` (voir documentation officielle). Par exemple, pour hacher un mot de passe (destiné à être écrit en base de données) :

```
import hashlib

def hash_mdp(mdp):
    mdp_utf8 = str(mdp).encode('utf-8')
    return hashlib.sha256(mdp_utf8).hexdigest()
```

```
>>> hash_mdp("password1234")
'b9c950640e1b3740e98acb93e669c65766f6670dd1609ba91ff41052ba48c6f3'
```

Dans le terminal

On peut utiliser la fonction `md5sum` sous Linux de la façon suivante dans un **terminal** :

```
$ echo "Vive la NSI" | md5sum
0578f6f1845600d4eeea883d610fccfe  -
$ echo "vive la NSI" | md5sum
b3596079dc6c389c62a62f62e83d8aee  -
$ echo Vive la NSI | md5sum
0578f6f1845600d4eeea883d610fccfe  -
```

On peut également utiliser cette fonction pour calculer l'empreinte d'un fichier :

```
$ md5sum mon_fichier.txt
```

Il est aussi possible d'utiliser la fonction `sha256sum` pour appliquer la fonction de hachage SHA-256 :

```
$ sha256sum -b mon_fichier.txt
```

À faire

1. Créez une fichier `mon_fichier.txt` avec plusieurs paragraphes (vous pouvez utiliser https://fr.lipsum.com/) puis calculer son empreinte avec les fonctions `md5sum` et `sha256sum`.
2. Modifiez légèrement le contenu du fichier (ajoutez un espace par exemple) et recalculez les deux empreintes. Vous devriez constater qu'elles sont totalement différentes que celles précédentes (pour un seul caractère qui diffère).

Conclusion

- Les dictionnaires sont des structures de données de type *tableau associatif* qui permettent d'associer des valeurs à des clés.
- La recherche d'un élément dans un dictionnaire est très rapide : elle se fait en **temps constant**. En particulier, elle est beaucoup plus rapide que la recherche dans un liste, qui elle se fait en temps linéaire.
- Cette rapidité de recherche dans un dictionnaire Python est possible car ces derniers sont implémentés par des *tables de hachages* (hors programme).
- Les tables de hachage utilisent des *fonctions de hachage* qui permettent de créer une empreinte de taille fixe à partir d'un objet de départ. Il est impossible de retrouver l'objet de départ à partir de l'empreinte, et deux objets différents donnent deux empreintes différentes.
- Dans le cas des dictionnaires, pour chercher une valeur à partir d'une clé, Python calcule l'empreinte de la clé (rapide), puis déduit de cette empreinte (rapide) l'indice du tableau dans lequel est stocké la valeur. Il peut ensuite accéder *directement* à cette valeur. Il n'est donc pas nécessaire de parcourir chaque paire clé-valeur pour chercher une valeur dans un dictionnaire, d'où la rapidité de recherche.
- Les fonctions de hachage sont très utilisés en informatique : ce sont les valeurs hashées des mots de passe qui sont stockées en base de données, on peut vérifier l'intégrité et l'authenticité d'un fichier avec une fonction de hachage. Les fonctions de hachage sont à la base de la technologie de la blockchain sur laquelle s'appuient les cryptomonnaies.
- Pour voir un résumé sur les tables de hachage, cette vidéo est très bien faite : https://youtu.be/IhJo8sXLfVw.

Références :

- Cours de Gilles Lassus sur les dictionnaires : https://glassus.github.io/terminale_nsi/T1_Structures_de_donnees/1.2_Dictionnaires/cours/
- Cours d'Olivier Lécluse sur les dictionnaires : https://www.lecluse.fr/nsi/NSI_T/donnees/dico/
- Vidéo de la chaîne *Cours Python* sur les dictionnaires : https://youtu.be/VnhBoQAgIVs

---

Germain BECKER, Lycée Mounier, ANGERS

Voir en ligne : info-mounier.fr/terminale_nsi/structures_donnees/dictionnaires.php