Dictionnaires

1 Rappels et compléments

Les dictionnaires ont déjà été vus en tant que boîtes noires en première année. Il s'agit, ici, de faire quelques rappels et d'étudier l'implémentation d'une telle structure.

1.1 Définition

Définition

Un **dictionnaire** est un tableau associatif qui, à des clés, associe des valeurs. Ainsi, un dictionnaire `D` sur un ensemble de clés **K** et un ensemble de valeurs **V** est une extension de la structure de tableau.

Intuitivement, on peut voir cela comme un tableau contenant des éléments de l'ensemble **V**, qui au lieu d'être indicé par l'ensemble des entiers de zéro à la longueur du tableau moins un, est indicé par des clés de l'ensemble **K**. Si `k` est une clé apparaissant dans le dictionnaire `D`, `D[k]` sera donc la valeur **v** correspondant à la clé `k`. Mathématiquement, un dictionnaire peut être vu comme une application partielle de **K** dans **V**.

En Python, un dictionnaire est de type `dict` et il se définit à l'aide d'accolades (`dictionnaire = {clé1 : valeur1, clé2 : valeur2}`).

Exemple :

```
Prof_de_physique = {'PC' : 'Décavé', 'PCe' : 'Morello' , 'PSI' : 'Guillot'}
```

Remarque :

- Les valeurs peuvent être de n'importe quelle type (`int`, `float`, `list`, `str`, etc.). En revanche, les clés doivent être des objets non modifiables. En particulier, une liste ne peut pas servir de clé.
- Un dictionnaire vide se définit à l'aide de deux accolades vides : `dico = { }`

1.2 Opérations élémentaires sur les dictionnaires

Les opérations élémentaires qui doivent s'appliquer à un dictionnaire sont :

- INSÉRER OU REMPLACER une association clé/valeur
- SUPPRIMER une association clé/valeur
- RECHERCHER si une clé appartient au dictionnaire et quelle est sa valeur associée.

Attention !

Idéalement, on souhaite que ces trois opérations élémentaires s'exécute avec un temps O(1).

1.3 Syntaxe Python

On suppose qu'une variable `dico` contient un dictionnaire.

1. Pour **accéder aux valeurs associées aux clés** d'un dictionnaire, il n'y a plus la notion de rang comme c'est le cas pour les listes : les associations `(cle, valeur)` ne sont pas ordonnées.
   L'instruction pour obtenir une valeur associée à sa clé est de la forme `dico[cle]`, et elle renvoie `valeur`.

```
In:  Prof_de_physique
Out: {'PC' : 'Décavé', 'PCe' : 'Morello' , 'PSI' : 'Guillot', 'MP' : 'Gérard',
  'MPI' : 'Gérard', 'MPe' : 'Tuloup'}
In:  Prof_de_physique['PCe']
Out: 'Morello'
```

2. La syntaxe `dico[cle] = valeur` permet :
   - lorsque la clé `cle` est présente dans le dictionnaire, de **modifier la valeur associée** à cette clé ;

```
In:  Prof_de_physique
Out: {'PC' : 'Décavé', 'PCe' : 'Morello' , 'PSI' : 'Guillot', 'MP' : 'Gérard',
  'MPI' : 'Gérard', 'MPe' : 'Tuloup'}
In:  Prof_de_physique['PCe'] = 'Frédéric'
In:  Prof_de_physique
Out: {'PC' : 'Décavé', 'PCe' : 'Frédéric' , 'PSI' : 'Guillot', 'MP' : 'Gérard', '
   MPI' : 'Gérard', 'MPe' : 'Tuloup'}
```

   - dans le cas contraire, d'**ajouter l'association** `(cle, valeur)` au dictionnaire.

```
In:  Prof_de_physique
Out: {'PC' : 'Décavé', 'PCe' : 'Morello' , 'PSI' : 'Guillot', 'MP' : 'Gérard',
  'MPI' : 'Gérard', 'MPe' : 'Tuloup'}
In:  Prof_de_physique['MPIe'] =  'Tuloup'
In:  Prof_de_physique
Out: {'PC' : 'Décavé', 'PCe' : 'Morello' , 'PSI' : 'Guillot', 'MP' : 'Gérard',
  'MPI' : 'Gérard', 'MPe' : 'Tuloup', 'MPIe' : 'Tuloup'}
```

3. Si `cle` n'est pas une clé présente dans `dico`,
   une tentative de lecture de `dico[cle]` déclenche une erreur.
   On peut **tester la présence d'une clé** dans un dictionnaire
   à l'aide de la syntaxe `cle in dico`, qui retourne un booléen.

```
In:  Prof_de_physique['BCPST']
Keyerror: 'BCPST'
In:  'BCPST' in Prof_de_physique
Out: False
```

4. Pour **supprimer une association,** on utilise la syntaxe : `del dico[cle]`.

```
In:  Prof_de_physique
Out: {'PC' : 'Décavé', 'PCe' : 'Morello' , 'PSI' : 'Guillot', 'MP' : 'Gérard',
  'MPI' : 'Gérard', 'MPe' : 'Tuloup', 'MPIe' : 'Tuloup'}
In:  del Prof_de_physique['MPIe']
In:  Prof_de_physique
Out: {'PC' : 'Décavé', 'PCe' : 'Morello' , 'PSI' : 'Guillot', 'MP' : 'Gérard',
  'MPI' : 'Gérard', 'MPe' : 'Tuloup'}
```

1.4 Itération sur un dictionnaire

Principe

Un dictionnaire est une structure de données qui permet de regrouper plusieurs informations dans un même objet : des associations *(clé, valeur)*.
Il sera très souvent nécessaire de traiter toutes les associations *(clé, valeur)* d'un dictionnaire les unes après les autres, de la même manière, autrement dit : de les traiter de façon itérative.
Comme pour les intervalles d'entiers `range` et les listes, cela se fait facilement en Python grâce à une boucle `for`.

Itération sur un dictionnaire

À l'aide de boucles `for`, on peut traiter toutes les associations *(clé, valeur)* d'un dictionnaire. Cela peut se faire de plusieurs manières :

1. Par défaut, **l'itération porte sur les clés du dictionnaire.**
   Dans la boucle suivante,

   ```
   for cle in dico:
       ...
   ```

   ou

   ```
   for cle in dico.keys():
       ...
   ```

   la variable `cle` contiendra successivement chacune des clés présentes dans le dictionnaire `dico`.
2. **Pour parcourir les valeurs d'un dictionnaire,** on peut :
   - itérer sur les clés du dictionnaire, puis accéder aux valeurs associées par la syntaxe habituelle `dico[cle]` :

     ```
     for cle in dico:
         ... dico[cle] ...
     ```

   - utiliser la méthode `.values` du dictionnaire, qui retourne un objet analogue à la liste des valeurs présentes dans le dictionnaire :

     ```
     for valeur in dico.values():
         ....
     ```

3. On peut aussi **accéder simultanément aux clés et aux valeurs** grâce à la méthode `.items` du dictionnaire.
   Cette dernière fournit la liste des associations *(clé, valeur)* présentes dans le dictionnaire :

   ```
   for (cle, valeur) in dico.items():
       ...
   ```

   À chaque itération, les variables `cle` et `valeur` auront pour contenu une clé du dictionnaire et la valeur qui lui est associée.

Remarques

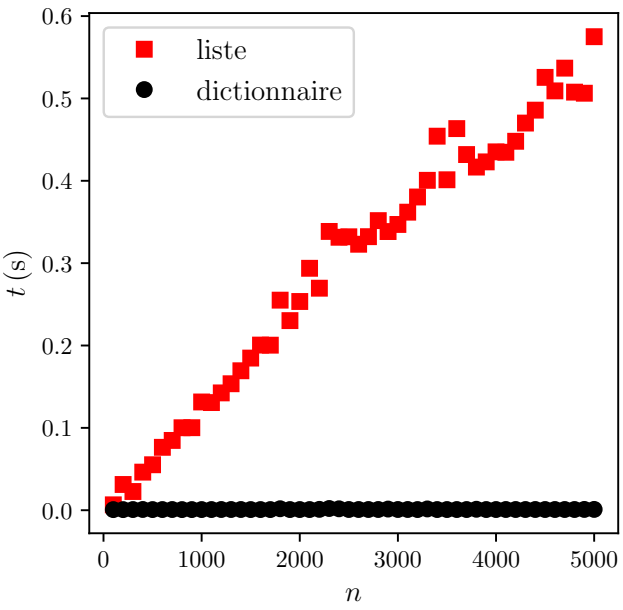
1. L'itération sur les clés permet de toute faire : on pourra s'en contenter.
2. L'ordre dans lequel seront traités les associations *(clé, valeur)* n'est pas prévisible : c'est Python qui choisit pour vous !

2 Table de hachage

2.1 Complexité

Commençons par une étude empirique des coûts de la recherche d'un élément dans une liste et d'une clé dans un dictionnaire.
Le graphique ci-contre, qui permet la comparaison de ces coûts, est issu d'un script qui :

- Pour chaque entier $n = 100, 200, 300, \cdots, 5\,000$ crée un dictionnaire `D` vide et une liste `L` vide.
- Tire au sort n entiers aléatoires compris entre 0 et 10 000 et qu'il range dans la liste (`L.append(j)`) et dans le dictionnaire (`D[j] = True`).
- Une fois ces deux conteneurs créés, ce script teste, pour chaque entier k entre 0 et 10 000, l'appartenance de k à la liste et au dictionnaire. Les temps de calculs cumulés de chaque instruction `k in L` et `k in D` sont relevés et reportés sur le graphique.



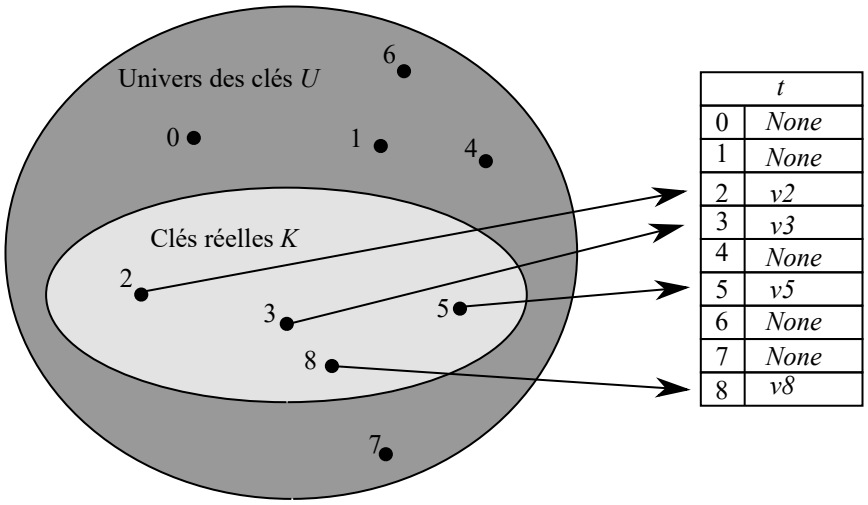
Bilan

Ce graphique met en évidence un coût moyen linéaire en O(n) pour la recherche d'un élément dans la liste, ce qui est bien conforme à nos attentes. Mais le coût de recherche d'une clé dans le dictionnaire est constant (en O(1)) pour le dictionnaire.

Dans la suite, nous verrons qu'une recherche en O(1) au sein d'un dictionnaire est possible grâce à l'utilisation d'une **table de hachage**.

2.2 Implémentation naïve d'un dictionnaire

Supposons que l'on souhaite implémenter un dictionnaire dont les clés sont prises dans un ensemble U, appelé **univers des clés**, tel que $U = \{0, 1, \cdots, n-1\}$, où n est un entier positif *pas trop grand*. Alors, on peut utiliser un simple tableau `t` de taille n, dont les case, appelées **alvéoles**, sont initialisées à une valeur conventionnelle, par exemple `None`. Ainsi, si `k` et `v` sont respectivement une clé et sa valeur associée alors `t[k] = v`.



Dans l'exemple illustré ci-dessus, chaque clé de l'univers $U = \{0, 1, \cdots, 8\}$ correspond à un indice du tableau `t`. L'ensemble $K = \{2, 3, 5, 8\}$ des clés réelles détermine les alvéoles du tableau qui contiennent les valeurs associées.

L'implémentation des opérations de dictionnaire est triviale :

- INSÉRER L'ASSOCIATION `k, v` :

```
def inserer(k, v, t):
  """insere le couple clé/valeur k -> v dans le
   dictionnaire t"""
  t[k] = v
```

- SUPPRIMER :

```
def supprimer(k, t):
  """supprime l'association de clé k du dictionnaire t"""
  t[k] = None
```

- RECHERCHER :

```
def valeur(k, t):
  """renvoie la valeur correspondante à la clé k dans le
   dictionnaire t
  ou None si la clé n'appartient pas à t"""
  return t[k]
```

Chacune des ces opérations est rapide, elles s'exécutent en un temps O(1).

Remarque :

Dans le cas ou l'univers des clés U n'est pas de la forme $\{0, 1, \cdots, n-1\}$, mais par exemple l'ensemble des chaînes de caractères de longueurs 4 ; il suffit de disposer d'une fonction f bijective transformant une telle chaîne de caractère en un entier.
Le problème est que U peut devenir très grand, voire infini. Par exemple, si l'on considère l'ensemble des chaînes constituées d'au plus 4 caractères choisis parmi les 52 minuscules et majuscules, U contient déjà 7 454 981 éléments.

Bilan

L'utilisation d'un tableau de la taille de U risque d'être trop gourmande en mémoire et infaisable en pratique. Par ailleurs, l'ensemble K des clés réellement utilisées peut être tellement petit par rapport à U que la majeure partie de l'espace alloué pour `t` est gaspillée. Ainsi, l'implémentation naïve par adressage direct n'est pas une solution à retenir.

2.3 Tables de hachage

Méthode

L'idée des tables de hachage est de renoncer à la fonction f bijective et de se contenter d'une application de U dans $\{0, 1, m-1\}$, où m est un entier pas trop grand.
En pratique, on se donne une fonction h de U dans $\mathbb{Z}$ que l'on appelle **fonction de hachage**.
Ensuite, dans un tableau, appelé **table de hachage**, constitué de m alvéoles, une clé k et sa valeur associée seront stockées dans l'alvéole $h(k)\%m$, où % donne le reste de la division euclidienne. Les cases non utilisées contiennent une valeur arbitraire (par exemple, `None`).

En Python, il existe une fonction de hachage `hash` pour les types usuels.

```
In [1]: hash('a')
Out[1]: -3992963190910735510

In [2]: hash('b')
Out[2]: -6130452822597348851

In [3]: hash('Victor Hugo')
Out[3]: -8726737909994322031
```

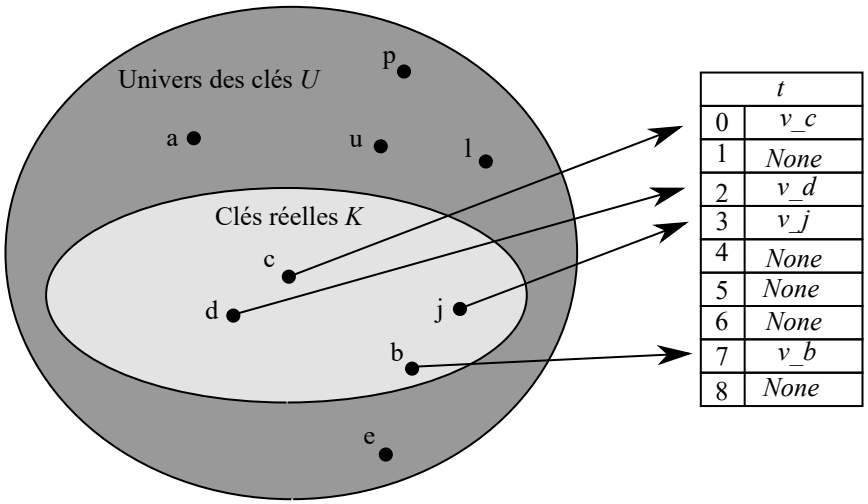
```
In [4]: hash(7)
Out[4]: 7

In [5]: hash(1e20)
Out[5]: 848750603811160107

In [5]: hash([0, 1])
Out[5]: Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

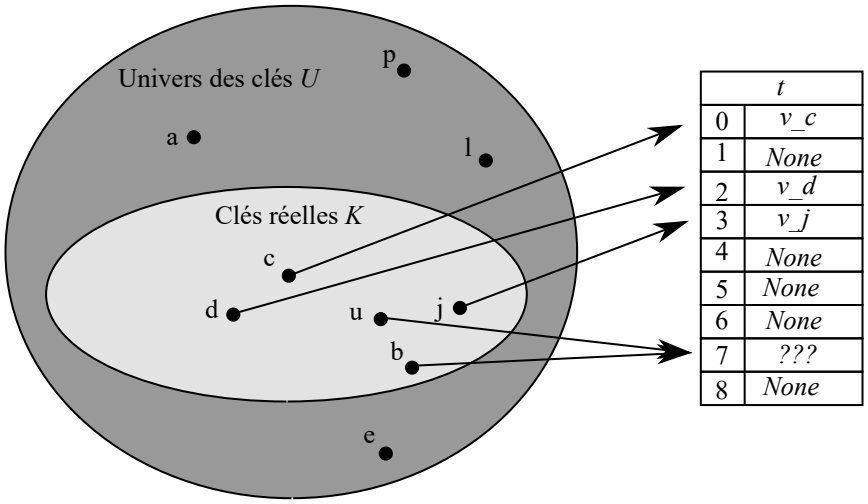
Exemple :

Pour illustrer, considérons un univers U contenant les clés a, b, c, d, e, j, l, p et u. On souhaite stocker les clés b, c, d et j et leurs valeurs associées dans une table de hachage de taille 9. L'instruction `hash('b') % 9` permet de trouver l'indice de l'alvéole qui stockera la clé b et sa valeur associée, ici c'est l'alvéole n°7. On remplit de la sorte la table de hachage.



Remarque :

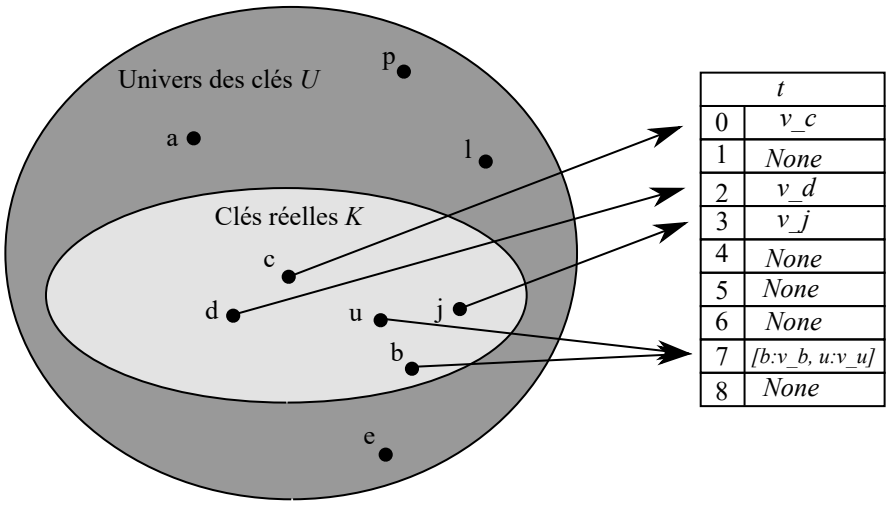
On remarque tout de suite une limitation des tables de hachage. Comme m est plus petit que la taille de U, alors deux clés peuvent être hachées sur la même alvéole : on dit qu'il y a **collision**.
Sur l'exemple précédent, si l'on rajoute la clé u à l'ensemble des clés utilisées, comme `hash('u')%9` renvoie `7`, la clé u entre en collision avec la clé b.



2.4 Gestion des collisions

Méthode

On peut gérer les collisions en utilisant des listes. Ainsi, une alvéole contiendra une liste de toutes les clés et valeurs associées qui sont hachées sur celle-ci.



Chaque alvéole ne contient désormais plus une valeur, mais une liste de valeurs. On peut donc stocker dans la table de hachage un nombre de clés n plus grand que le nombre d'alvéole m. On note $\alpha = \frac{n}{m}$ le taux de remplissage de la table de hachage, c'est-à-dire le nombre moyen d'éléments stockés dans une liste.

Exercice

Implémenter les opérations de dictionnaire INSÉRER, SUPPRIMER et RECHERCHER sur une table de hachage lorsque les collisions sont résolues par chaînage.

Le comportement, dans le cas le plus défavorable, du hachage avec chaînage est très mauvais : les n clés sont toutes hachées vers la même alvéole, y formant une liste de longueur n. Le temps d'exécution de la recherche est alors $O(n)$ plus le temps de calcul de la fonction de hachage ; pas mieux que si l'on avait utilisé une seule liste pour tous les éléments.
Ainsi, les performances moyennes du hachage dépendent de la manière dont la fonction de hachage h répartit en moyenne l'ensemble des clés à stocker parmi les m alvéoles.

Théorème

Dans le cas le plus favorable, on suppose que chaque élément a la même chance d'être haché vers l'une quelconque des alvéoles, indépendamment des endroits où les autres éléments sont allés. Cette hypothèse est dite de hachage uniforme simple. On peut montrer que, dans cette situation, une recherche prend un temps $O(1+\alpha)$. Ainsi, si le nombre d'alvéoles de la table de hachage est au moins proportionnel au nombre d'éléments de la table, on a $n = O(m)$ et, par conséquent, $\alpha = n/m = O(m)/m = O(1)$. La recherche prend un temps constant en moyenne, ce qui correspond bien à notre attente.

Bilan

En conclusion, une bonne fonction de hachage doit vérifier (approximativement) l'hypothèse du hachage uniforme simple : chaque clé a autant de chances d'être hachée vers l'une quelconque des m alvéoles, indépendamment des endroits où sont allées les autres clés.