Chapitre 1

Dictionnaires

Python propose plusieurs types standard qui permettent de gérer des **collections** de données, on parle aussi de **types construits** : les listes (`list`), les n-uplets (`tuple`), les ensembles (`set`), les dictionnaires (`dict`), les chaines (`str`). Les types `str`, `list`, `tuple` sont des séquences, c'est-à-dire des collections indexées par une suite finie d'entiers. Les ensembles et les dictionnaires ne sont pas des séquences, mais des collections non ordonnées.

1.1 Structure de dictionnaire

Un **dictionnaire**, ou **tableau associatif**, ou table d'association, est un type de données qui associe à un ensemble de **clés** un ensemble de **valeurs**. On le représente par un ensemble de **couples (clé, valeur)**, mais il s'agit fondamentalement d'une application `d : clé` $\mapsto$ `valeur`. Le principal avantage de cette structure de données est l'**accès en temps constant** [1] **à tous les éléments**, comme dans un tableau. Le dictionnaire est donc un conteneur qui généralise la notion de tableau en permettant de s'affranchir de la contrainte d'indicer les éléments par des entiers consécutifs depuis zéro : les clés permettent d'accéder aux éléments du dictionnaire. Le type des clés est presque [2] quelconque : entier, chaîne, tuples, etc. Selon l'implémentation les clés peuvent parfois avoir différents types, de même que les valeurs.

Contrairement aux éléments d'un tableau, **les éléments (ou entrées) d'un dictionnaire ne sont pas ordonnés**, il est donc important de les considérer comme les éléments d'un **ensemble de couples** : l'examen du contenu d'un dictionnaire peut fournir les couples en ordre aléatoire selon l'implémentation interne de la structure.

L'ajout ou la suppression d'un élément d'un dictionnaire fait partie des primitives usuelles de cette structure. Nous devons considérer un dictionnaire comme la généralisation du type « tableau redimensionnable », qui est aussi la nature réelle des listes en Python, mais sans accès séquentiel (un élément d'un dictionnaire n'a pas de prédécesseur ni de successeur).
Le dictionnaire est la structure de données parfaite lorsque l'opération de recherche d'une clé est la plus fréquente, mais l'occupation mémoire est supérieure à celle d'un tableau pour des clés entières.

Structure abstraite de dictionnaire

Un **dictionnaire**, ou *tableau associatif*, ou *table d'association*, en anglais *associative array*, est un ensemble de couples (clé, valeur), en anglais (*keys*, *values*), où les clés, nécessairement distinctes, appartiennent à un ensemble K et où les valeurs sont prises dans un ensemble E.

1. La recherche d'un élément dans un dictionnaire se fait en temps constant par sa clé, la complexité devient linéaire si l'on effectue une recherche sur les valeurs. C'est la même chose pour un tableau. On choisit donc les clés précisément pour être la cible des recherches !
2. Les clés d'un dictionnaire Python doivent être hachables, de type immuable.

Les dictionnaires sont très utilisés en informatique pour la gestion des systèmes de fichier ou la gestion de tables de symboles, d'annuaires téléphoniques, etc. Les dictionnaires et les ensembles sont souvent implémentés de manière similaire. Une table de base de données peut également être vue comme un tableau associatif où une valeur est un enregistrement, en ce sens une base de donnée est un ensemble de dictionnaires. **La structure de dictionnaire est extrêmement efficace, il ne faut pas hésiter à l'utiliser.** Un graphe est par exemple très bien représenté par un dictionnaire de sommets.

1.2 Manipulations des dictionnaires Python

Un dictionnaire Python est de type `dict`, c'est une structure mutable.

1.2.1 Opérations de base

Primitives des dictionnaires

La structure de dictionnaire propose les six primitives (opérations de base) suivantes :

- Création d'un dictionnaire vide
- Ajout d'un élément, c'est-à-dire un couple (clé, valeur).
- Test de l'existence d'une clé dans le dictionnaire (en temps constant)
- Suppression d'un élément dont on connait la clé
- Modification de la valeur associée à une clé
- Lecture (ou recherche) d'un élément à partir de sa clé

L'opération de modification de la valeur associée à une clé existante peut être réalisée un peu moins efficacement par une suppression suivie d'un ajout.

En Python les dictionnaires se déclarent en explicitant leur contenu : les délimiteurs sont les accolades, le séparateur la virgule et chaque élément est donné sous forme `cle : valeur`.

```
dico = {'Hugo': 'écrivain', 'Einstein': 'physicien', 'Euler': 'mathématicien'}
```

Le même résultat est obtenu en ajoutant les trois entrées à un dictionnaire vide :

```
dico = {}
dico['Hugo'] = 'ecrivain'
dico['Einstein'] = 'physicien'
dico['Euler'] = 'mathématicien'
print(dico)
```

```
{'Hugo': 'écrivain', 'Einstein': 'physicien', 'Euler': 'mathématicien'}
```

L'ordre d'insertion n'est pas important : un dictionnaire est un ensemble de couples.
Le test d'existence d'une clé est réalisé avec l'opérateur `in` :

```
print('Newton' in dico, 'Euler' in dico)
```

```
False True
```

On lit un dictionnaire comme un tableau :

```
print(dico['Euler'])
```

```
mathématicien
```

Mais cela n'est possible que si la clé existe, sinon l'on provoque l'erreur `KeyError`.
La modification d'une valeur consiste simplement à affecter la nouvelle valeur :

```
dico['Euler'] = 'scientifique'
print(dico)
```

```
{'Hugo': 'ecrivain', 'Einstein': 'physicien', 'Euler': 'scientifique'}
```

La suppression d'un élément se réalise avec `del` :

```
del dico['Hugo']
print(dico)
```

```
{'Einstein': 'physicien', 'Euler': 'scientifique'}
```

Remarque : l'instruction `del` est utilisable avec les dictionnaires car elle agit en temps constant, alors qu'elle est proscrite avec les listes en Python, car elle agit en temps linéaire.

1.2.2 Parcours d'un dictionnaire

La boucle `for` permet de parcourir un dictionnaire facilement :

```
for x in dico:
    print (x, dico[x])
```

ou de manière équivalente, mais plus explicite :

```
for x in dico.keys():
    print (x, dico[x])
```

L'objet `dico.keys()` est de type `dict_keys`. On peut le convertir en liste ou tuple avec le constructeur correspondant (`list` ou `tuple`), mais il est itérable directement comme le montre son usage dans la boucle `for`. Noter qu'il s'agit d'une vue dynamique sur le dictionnaire qui change en même temps que ce dernier.
On dispose de même des objets `dico.values()` et `dico.items()` qui proposent des vues dynamiques sur les valeurs et sur les couples (clé,valeur) respectivement.
Le parcours d'un dictionnaire peut donc se présenter sous la forme

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

Les objets `dico.keys()`, `dico.values()` et `dico.items()` sont facilement convertis en listes par le constructeur `list`. Les clés d'un dictionnaire sont uniques, pas nécessairement les valeurs.
Le nombre d'éléments (ou de clés) d'un dictionnaire est accessible par `len(dico)`. Comme pour les listes, la méthode `copy` ne réalise qu'une copie superficielle (pas de copie récursive des valeurs de type `list` par exemple). La copie profonde se réalise par un parcours complet du dictionnaire ou avec la fonction `deepcopy` du module `copy`. On ne peut pas modifier la taille d'un dictionnaire au sein de la boucle qui le parcourt.

1.3 Implémentation concrète de la structure de dictionnaire

Il n'est pas nécessaire de se poser la question de l'implémentation concrète d'un dictionnaire Python pour l'utiliser ; le programme de première année a montré que l'on peut l'utiliser comme une boîte noire. Le programme de seconde année demande d'en étudier la structure concrète.

Implémentation d'un dictionnaire

Un dictionnaire, comme un ensemble, peut être réalisé par différentes structures de données concrètes : tableau statique, liste d'association, arbre binaire de recherche, arbre équilibré AVL, table de hachage, arbre de préfixes (en anglais *trie*) si les clés sont des chaînes, arbres de Patricia pour les clés entières.

1.3.1 Implémentation d'un dictionnaire par liste d'association

La méthode la plus simple pour réaliser un dictionnaire est d'utiliser une liste d'association (en anglais *association list*). Il suffit de créer une liste de couples (clé, valeur).
La recherche dans une liste d'association ne satisfait pas l'objectif d'une recherche en temps constant. En revanche on peut les utiliser pour contenir les alvéoles (voir après), qui sur le principe contiennent un nombre borné d'entrées.

1.3.2 Implémentation d'un dictionnaire dans un tableau

Un dictionnaire dont les clés sont des entiers consécutifs peut être stocké dans un tableau (reflet direct de la mémoire physique, donc rapide à manipuler). Une approche naïve consiste donc à mettre l'ensemble des n clés d'un dictionnaire en bijection avec $[\![0, n-1]\!]$, puisque l'utilisation d'un tableau devient évidente. Ce n'est pas très utile en pratique : pour stocker les mots du dictionnaire de la langue française (clés) accompagnés de leur définition (valeur), il ne suffit pas de les numéroter et de les mettre dans un tableau, car la recherche d'un mot particulier serait trop complexe (on ne connaît pas a priori le numéro d'ordre d'un mot dans le dictionnaire !), temporellement en $O(\log n)$ avec une recherche dichotomique si un ordre des clés peut être exhibé et en $O(n)$ sinon. Mais la complexité logarithmique s'obtiendrait au prix d'une insertion et d'un suppression en $O(n)$ pour maintenir le tableau trié. On serait donc condamné à travailler avec des opérations d'ajout et de suppression en $O(n)$.

Pour être féconde et s'étendre au cas de clés non ordonnées, cette idée doit être complétée par la notion de **hachage**. Le principe d'une **fonction de hachage** est d'associer un entier (la **valeur de hachage** de la clé) à une clé. C'est le fondement de l'efficacité de la recherche dans un dictionnaire : on ne parcourt pas toutes les clés pour en trouver une particulière, mais on calcule son haché (entier) qui permet de retrouver la cellule du tableau où est stockée la valeur associée à cette clé, en temps constant, quitte à ne pas utiliser toutes les cellules du tableau.

Trouver une fonction de hachage bijective de l'ensemble des clés vers l'ensemble des cellules d'un tableau est généralement impossible car les clés appartiennent à un ensemble infini. On utilise donc des fonctions de hachage qui ne sont pas injectives mais qui répartissent au mieux les valeurs de hachage dans différentes **alvéoles** ou paquets ou **seaux** (en anglais *buckets* ou *slots*). Lorsque deux clés possèdent la même valeur de hachage, on parle de **collision**.

1.3.3 Table de hachage

On réalise des recherches avec une complexité constante en moyenne grâce à une table de hachage, qui est une réalisation concrète de la structure abstraite de tableau associatif (dictionnaire).

Implémentation d'un dictionnaire par table de hachage

Un tableau contenant les couples (clé, valeur) peut fournir une implémentation impérative efficace de la structure abstraite de dictionnaire grâce à la structure de **table de hachage**. Cette table est implémentée dans un tableau t de w listes (appelées *alvéoles* ou *listes d'association*, en anglais *buckets*) de couples (clé, valeur) notés (k, v). L'ensemble K des clés peut être plus grand que $[\![0, w-1]\!]$ et chaque alvéole peut contenir des éléments distincts, on parle de *collisions*. La table est organisée de manière à ce que la liste d'indice i contienne tous les couples (k, v) tels que $h_w(k) = i$, où $h_w : K \to [\![0, w-1]\!]$ s'appelle la **fonction de hachage**. L'entier w représente la *largeur de la table* de hachage et on dit que $h_w(k)$ est le *haché* de k.

Pour rechercher ou supprimer l'élément de clé k de la table t, on commence par calculer le haché $i = h_w(k)$ de la clé k pour déterminer l'alvéole i concernée, puis on parcourt la liste $t[i]$ depuis son début. Pour ajouter un nouvel élément au dictionnaire, on l'ajoute en tête de l'alvéole indiquée par le haché de sa clé. La table de hachage est souvent comparée à une commode contenant w tiroirs, dans lesquels la fonction de hachage envoie les vêtements à ranger, par dessus ceux qui sont déjà rangés.

1.3.4 Fonction de hachage

La fonction de hachage n'est pas injective, mais doit être choisie de manière à répartir les clés de manière apparemment uniforme entre les différentes alvéoles. Dans l'hypothèse d'un hachage uniforme, la probabilité qu'une clé k soit hachée dans l'alvéole i est $1/w$, indépendante des autres clés. Si l'on note n le nombre d'éléments dans la table t, on appelle facteur de remplissage de la table le rapport $\alpha = n/w$. C'est le nombre moyen de clés par alvéole et l'on obtiendra donc une complexité moyenne pour insérer ou rechercher un élément en $O(1+\alpha)$. Si le facteur de remplissage α est borné, on garantit ainsi que les recherches, insertions et suppressions se font en temps constant dans le cas moyen. Mais il faut que le nombre de clés à insérer dans la table soit raisonnable comparé à la taille du tableau pour que le facteur de remplissage reste modéré.

La fonction de hachage consiste généralement à réaliser une transformation de la clé en un entier naturel assez grand, puis à considérer le reste modulo w du résultat pour garantir l'existence de l'indice $i = h_w(k) \in [\![0, w-1]\!]$. **Le choix de la fonction de hachage est important pour assurer une relative uniformité de la distribution des éléments dans les alvéoles** (faible dispersion de la population des alvéoles par rapport à l'occupation moyenne α). Si l'occupation mémoire n'est pas un problème, **il faut choisir w de l'ordre de n**, on conseille généralement de choisir pour w un entier premier éloigné d'une puissance de 2. Il est possible de concevoir des tables de hachage dynamiques dont on peut faire varier la largeur w en fonction du nombre n d'éléments présents dans le dictionnaire, c'est la raison de la notation h_w pour la fonction de hachage. Cela est nécessaire si l'on ne connait pas l'ordre de grandeur du nombre d'éléments du dictionnaire à l'avance.

Avantages et inconvénients d'une table de hachage

Avantages : elle fournit une complexité constante pour l'insertion et la recherche, garantit une représentation en mémoire compacte et ne nécessite pas un ordre sur les clés. **Inconvénients** : elle nécessite une bonne fonction de hachage, parfois difficile à trouver, elle peut présenter une complexité dégradée dans le pire des cas, en $O(n)$, elle ne préserve pas l'ordre des clés, s'il existe.

La **complexité temporelle** dépend du nombre de clés n. Si l'on suppose que la fonction de hachage agit en $O(1)$, le pire des cas est celui où toutes les clés de K sont envoyées dans la même alvéole : la recherche et la suppression sont alors en $O(n)$ et l'insertion en $O(1)$. Si l'on recherche la présence

d'une clé avant de l'ajouter, l'insertion est aussi en $O(n)$. Dans le meilleur des cas, les clés sont réparties uniformément dans toutes les alvéoles, la suppression et la recherche sont en $O(1+\alpha)$ où $\alpha = n/w$ est le facteur de remplissage (ou *charge*) de la table de hachage, ce qui est proche du temps constant si $n = O(w)$. La complexité spatiale est en $O(n+w)$.

Dictionnaire et table de hachage

Une table de hachage est une implémentation très efficace de la structure de dictionnaire. Une table de hachage n'a d'intérêt que si la taille de chaque alvéole est majorée par une petite constante, la complexité moyenne des différentes opérations est alors constante. Pour maintenir un facteur de remplissage borné, on modifie dynamiquement la taille w de la table de hachage en pratique. Le redimensionnement a un coût en $O(n)$ car tous les éléments doivent être copiés dans le nouveau tableau. Si l'on choisit une stratégie consistant à doubler la taille du tableau à chaque redimensionnement, alors le coût s'amortit sur l'ensemble des opérations et l'on maintient un **coût amorti** en $O(1)$.

1.3.5 Exemple de fonction de hachage

Si l'on construit une table de hachage pour stocker le dictionnaire français, les clés sont de type chaîne (`str`). Chaque alvéole contient un couple (mot, définition). Si l'on considère qu'il y a environ 60000 mots dans le dictionnaire, on prend par exemple $w = 59999$.
Comment convertir un mot en un entier de $i = h_w(k) \in [\![0, w-1]\!]$? On pourrait prendre la longueur du mot, mais on occuperait que les 25 première alvéoles. Pour obtenir de plus grands entiers, on peut penser à faire la somme des codes ASCII des lettres du mot. La fonction obtenue est meilleure, mais $255 \times 10 = 2550$ en prenant des mots de longueur 10. C'est encore trop peu d'alvéoles utilisées. Voici une fonction plus satisfaisante :

$$h(s) = \sum_{0 \leqslant i < n} a^i \operatorname{ord}(s_i) \mod w \tag{1.1}$$

La chaîne s est constituée des n caractères s_i. La fonction `ord` renvoie le code ASCII sur un octet du caractère transmis et le paramètre a est un entier choisi selon la taille du dictionnaire ($a = 31$ par exemple). Python intègre une fonction de hachage `hash` qui est très efficace également.
La première qualité d'une fonction de hachage est de produire peu de collisions, la seconde est d'être rapide à calculer. La **gestion des collisions** consistant à utiliser une liste par alvéole constitue une résolution des collisions par chaînage, ou **adressage ouvert**. On peut aussi rechercher une autre alvéole si elle est déjà occupée, on parle alors d'**adressage fermé** pour la résolution des collisions : on utilise une formule simple pour **sonder** les alvéoles à partir de l'alvéole déjà occupée.

Dictionnaires Python et types

Les dictionnaires Python (`dict`) sont implémentés par des tables de hachage. Il en résulte que les clés doivent être hachables : seuls les types immuables peuvent servir de clés, en particulier les tuples d'éléments hachables sont hachables. Les valeurs peuvent être d'un type quelconque.