Chapitre 1
Dictionnaires

1 Rappels sur les dictionnaires

Lorsque l'on a une liste, on accède aux éléments de cette liste grâce aux indices qui sont des entiers. Plus précisément, les indices possibles sont les éléments de $[\![0 ; n-1]\!]$ si n est la longueur de la liste. Ainsi, si `L=[5,"a",["c",3.2]]`, alors

- `L[0]` vaut 5
- `L[1]` vaut `"a"`
- `L[2]` vaut `["c",3.2]`

Évidemment la commande `print(L[3])` provoquera une erreur : `out of range`.

On aimerait généraliser le concept de listes, en nous permettant d'avoir des indices qui ne serait pas forcément des entiers entre 0 et $n-1$. Et c'est précisément ce que font les dictionnaires. La notion d'indices une liste va être remplacée par celle de clé dans un dictionnaire. La notion d'élément d'indice donné va être remplacée par celle de valeur associée à la clé dans un dictionnaire.

Donnons un exemple, imaginons qu'on veuille compter le nombre de pommes de poires et de fraises. On pourrait enregistrer par une liste : `L=[5,2,25]` et on a qu'a retenir que `L[0]` va compter le nombre de pommes, `L[1]` le nombre de poires et `L[2]` le nombre de fraises. Mais ce n'est pas commode, imaginez que la liste est un million d'éléments, vous n'avez pas très envie d'apprendre par cœur que `L[987123]` compte le nombre d'arbouses que vous avez.

Un dictionnaire (ou tableau associatif) est une collection de paires de la forme `(clé,valeur)`, ainsi les différents fruits en votre possessions seront vos clés et les valeurs seront le nombre pour chaque fruit.

1.1 Créer un dictionnaire

Il existe trois façons de créer un dictionnaire.

1. Créer un dictionnaire vide : `D = {}`.
2. Créer un dictionnaire par extension `D = {"pommes":5,"poires":2,"fraises":25}`.
3. Créer un dictionnaire par compréhension `D = {L[i]:M[i] for i in range(len(L))}` avec `M = [5,2,25]` et `L = ["pommes","poires","fraises"]`.

Remarque 1. Il n'y a pas d'ordre dans un dictionnaire :
La valeur du booléen `{"pommes":5,"poires":2,"fraises":25} == {"poires":2,"pommes":5,"fraises":25}` est `True`.

1.2 Vérifier si un élément est une clé dans un dictionnaire

Supposons que l'on ait un dictionnaire `D={"pommes":5,"poires":2,"fraises":25}` La commande `"pommes" in D` est un booléen qui renverra `True` si "pommes" est une clé de D et `False` sinon. Ainsi `"pommes" in D` renverra `True` tandis que `5 in D` et `"kakis" in D` renverront `False`.

1.3 Afficher la valeur d'un dictionnaire avec une clé

La commande `print(D["pommes"])` affichera le nombre de pommes.

1.4 Modifier les valeurs d'un dictionnaire

La commande `D["pommes"]=10` modifie la valeur de `D` pour la clé `"pommes"`

1.5 Rajouter une clé et une valeur à un dictionnaire

Vous venez de recevoir 10 kakis, alors indiquez-le avec la commande `D["kakis"]=10`. À l'inverse, noter qu'on peut supprimer une clé et une valeur dans un dictionnaire grâce à la commande `del D["kakis"]`

1.6 Accéder aux clés/valeurs d'un dictionnaire

Les commandes `D.keys()`, `D.values()` et `D.items()` renvoient respectivement la collection des clés, des valeurs et des couples (clé, valeurs), attention ce ne sont pas des listes, on peut les convertir en liste si besoin grâce aux commandes `list(D.keys())`, `list(D.values())` et `list(D.items())`. On peut aussi boucler sur les clés, valeurs ou items d'un dictionnaire :

```
for cle in D.keys():#for cle in D: fait la même chose en plus court
    print(cle)
    print(D[cle])

for valeur in D.values():
    print(valeur)

for item in D.items():
    print(item)
```

Remarque 2. `len(D)` donne le nombre de couples (clé,valeur) dans le dictionnaire, et donc le nombre de clés.

1.7 Copie d'un dictionnaire

On rappelle que si `L` est une liste alors la commande `M=L` ne créera pas une liste distincte de `L` toute modification de `L` ou de `M` affectera l'autre liste. Il faut avoir en tête le résultat des commandes suivantes :

```
L=[1,2,3]
M=L
M[2]=5
```

Pour parer à ce problème, on pourra utiliser la commande `M=L.copy()`. Ainsi, le code suivant donnera bien le résultat attendu :

```
L=[1,2,3]
M=L.copy()
M[2]=5
```

À noter que cette parade ne fonctionne pas si les éléments de `L` sont eux-même des listes, par exemple :

```
L=[[1,2,4],2,3]
M=L.copy()
M[0][2]=5
```

Pour éviter ce désagrément, on peut effectuer une copie profonde grâce à des commandes (qui ne sont pas au programme). Pour les dictionnaires, c'est le même problème, le code suivant ne donne pas le résultat escompté :

```
D={"a":1,"b":2,"c":3}
Dp=D
D["b"]=5
```

En revanche, la commande `copy` fonctionne de la même manière que pour les listes :

```
D={"a":1,"b":2,"c":3}
Dp=D.copy()
Dp["b"]=5
```

Mais comme pour les listes, cela ne marche pas si les valeurs sont eux-mêmes des listes ou des dictionnaires.

1.8 Un exercice classique sur les dictionnaire

Écrire une fonction qui à, une liste, renvoie un dictionnaire comptant le nombre d'occurrences de chaque élément de la liste. Par exemple, si `L = [1,1,"1",2]`, la fonction doit renvoyer `{1:1,"1":1,2:1}`

```
def CompterOccurences(L):
  """renvoie un dictionnaire D dont les clés sont les éléments de L
  tel que D[x] contient le nombre d'occurences de x dans L"""
```

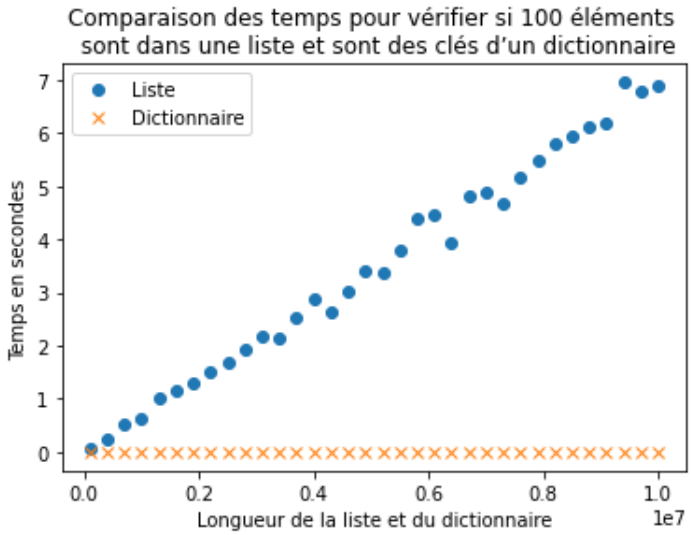
1.9 Quels types de variables peuvent être des clés ou des valeurs ?

Les valeurs d'un dictionnaire peuvent être de tous types : entiers, flottants, chaînes de caractères, listes, tuples et même des dictionnaires. Par contre, pour une raison que l'on verra plus tard, les clés ne peuvent pas être les listes ou des dictionnaires. Pour qu'un tuple puisse être une clé d'un dictionnaire, il ne faut pas non plus que les éléments de ces tuples puissent être eux-même des listes ou des dictionnaires, ou que ces éléments soient eux-mêmes des tuples dont les éléments seraient des dictionnaires ou des listes. Parmi les exemples suivants, dire lesquels sont valides :

```
D = {"3":[1,2,3],"b":3,3:{"a":2,"b":2}}
D = {[1,2,3]:"3"}
D = {(1,2,3):"3"}
D = {(1,2):[1,2,3],"b":2}
D = {([1,2],2):[1,2,3]}
D = {(2,{3:2,4:3}):5}
```

1.10 Complexité des opérations sur un dictionnaire

Plus une liste est grande plus il faudra de temps pour vérifier si un élément donné est dans cette liste. Ce n'est pas le cas pour les dictionnaires, ou le temps pour vérifier si un élément est une clé d'un dictionnaire est quasi-constant. La figure suivante illustre ce principe [1] :



Pour les listes, c'est assez facile à comprendre si un élément n'est pas dans une liste, pour le vérifier il va falloir regarder les éléments un à un de la liste et si une liste contient mille fois plus d'éléments qu'une autre, cela fera mécaniquement mille fois plus de vérifications [2]. On peut rappeler l'algorithme qui vérifie si un élément est dans une liste :

1. Vous pourrez retrouver le script sur Capytale, le lien est sur CDP.
2. On rappelle que lorsqu'on réfléchit en terme de complexité, on regarde souvent le cas le plus défavorable. Si un élément est le premier élément d'une liste, alors cela demandera une seule vérification car on trouvera cette élément du premier coups, si un élément est le deuxième

```python
def EstDansListe(L,x):
  """renvoie True si x est un élément de L et False sinon"""
  for i in range(len(L)):
    if L[i]==x:
      return True
  return False
```

La complexité pour savoir si un élément est dans une liste est donc linéaire en la longueur de la liste. Cela ne semble pas être le cas pour les dictionnaires. Rappelons qu'un dictionnaire n'est pas une liste, il n'y a pas d'ordre dans les clés ou dans les valeurs.

Le temps nécessaire pour savoir si un élément est une clé d'un dictionnaire semble constant. Ceci est liée à l'implémentation des dictionnaires dont allons voir quelques détails qui expliqueront ce phénomène.

2 Principe du hachage

Python dispose de fonctions dite de hachage, qui à certains objets, peut associer un nombre (grand en général). Une fonction de hachage est une fonction de la forme $h : U \to [\![0 ; n-1]\!]$, où U est un ensemble de clés possibles (les entiers, les flottants, les chaînes de caractères). Notez que U est ensemble très grand (voire infini), ce qui fait que h n'est pas une fonction injective. Ainsi, il est possible de trouver deux éléments de U : x et x' avec $x \neq x'$ mais $h(x) = h(x')$, on dit que x et x' forment une collision. Lorsque un dictionnaire est crée avec des clés c_1, c_2,..., c_r, Python va calculer $h(c_1)$, $h(c_2)$,..., $h(c_r)$. Python crée aussi un tableau de taille n, ce tableau est appelé table de hachage. Les éléments de ce tableau sont appelés alvéoles. Dans l'alvéole s, Python va stocker tous les couples (c_i, v_i) (où v_i est la valeur associée à la clé c_i) sous forme de liste tels que $h(c_i) = s$.

Dans la pratique, on se restreint au cas où $U = \mathbb{N}$, si on a des objets qui ne sont pas des entiers, on peut les transformer via une fonction en des entiers. Par exemple, si on a une chaîne de caractères, on peut attribuer arbitrairement à chaque lettre un nombre et donc convertir une phrase en un nombre. On dit donc que les chaînes de caractères et les nombres sont haschables (on peut utiliser donc une fonction de hachage). C'est aussi le cas des tuples (à condition que les éléments de ces tuples soient eux-même hashables). Par contre les listes et les dictionnaires ne sont pas haschables. C'est la raison pour laquelle les listes et les dictionnaires ne peuvent être des clés dans un dictionnaire.

Nous avons maintenant les outils pour comprendre pourquoi savoir si un élément est une clé d'un dictionnaire est si rapide. Si on veut savoir si x est une clé du dictionnaire, alors Python calcule $h(x)$. Ainsi, le couple $(x, D[x])$ devrait se trouver dans l'alvéole numéro $h(x)$ si x est une clé du dictionnaire, plusieurs cas peuvent se produire :

1. L'alvéole $h(x)$ est vide, dans ce cas, il est immédiat que x n'est pas une clé.
2. L'alvéole $h(x)$ n'est pas vide, dans ce cas Python va passer un à un tous les couples et voir si le premier élément est x soit x est bien une clé et dans ce cas il va la trouver, soit non.

Le cas numéro 1 est bien sûr réglé assez rapidement, le cas numéro 2 peut poser plus de problème, en effet plus l'alvéole contient de couples (c_i, v_i) plus cela va prendre du temps. Ainsi, tout dépend de la capacité de la fonction de hachage à minimiser les collisions.

Ainsi, une bonne table de hachage aura tendance à répartir le plus uniformément possible les couples (clé, valeurs) dans les alvéoles. Dans le pire cas possible, toutes les alvéoles sont vides, sauf une seule qui contient tous les couples (clé,valeurs). Ainsi, si on veut vérifier que x est une clé, il faudra passer en revue toutes les clés du dictionnaire, ceci prendra donc autant de temps que pour vérifier si un élément est une liste. Heureusement ce cas est hautement improbable. Il y a donc un problème mathématiques, assez intéressant, à créer de bonnes fonctions de hachage.

On comprend aussi (peut-être) pourquoi une liste ne peut pas être une clé d'un dictionnaire. Comme une liste est mutable, si elle venait à évoluer son hachage évoluerait et donc il faudrait déplacer le couple (clé, valeur) d'alvéole ce qui n'est pas pratique.

Remarque 3. Normalement, vos mots de passes ne doivent pas être stocker dans un serveur, quand vous rentrez un mot de passe, votre navigateur calcule sa valeur par une fonction de hachage et le compare à celle enregistrée sur le serveur en question.

élément d'une liste alors il faudra deux vérifications. Par contre si un élément n'est pas dans une liste (ou est le dernier élément de la liste) alors il faudra faire n vérifications avec n la longueur de la liste.