Dictionnaires

Capacités exigibles

Dictionnaires, clés et valeurs.
→ On présente les principes du hachage, et les limitations qui en découlent sur le domaine des clés utilisables.

Usage des dictionnaires en programmation Python.
→ Syntaxe pour l'écriture des dictionnaires. Parcours d'un dictionnaire.

Les dictionnaires sont utilisés en boîte noire dès la première année ; les principes de leur fonctionnement sont présentés en deuxième année. Ils peuvent être utilisés afin de mettre en mémoire des résultats intermédiaires quand on implémente une stratégie d'optimisation par programmation dynamique (cf. chapitre suivant).

1. Rappels de première année

1.1. Définition

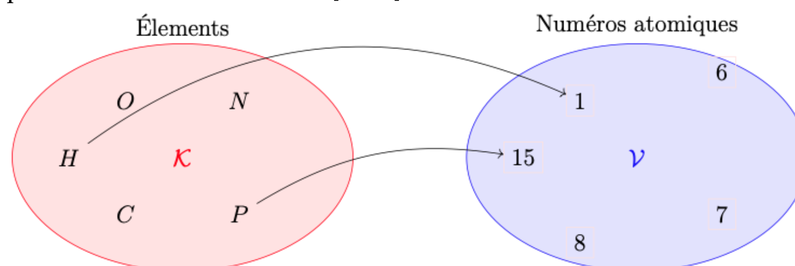
Un **dictionnaire** est une structure de données, alternative aux listes.

Les éléments $\mathscr{V}$, au lieu d'être indicés par un entier sont indicés par des clés appartement à un ensemble $\mathscr{K}$. Soit $c \in \mathscr{K}$, une clé d'un dictionnaire `d`, alors `d[c]` est la valeur v de $\mathscr{V}$ qui correspond à la clé c.

Les opérations sur un dictionnaire sont :
- → rechercher la présence d'une clé dans le dictionnaire,
- → accéder à la valeur correspondant à une clé,
- → insérer une valeur associée à une clé dans le dictionnaire,
- → supprimer une valeur associée à une clé dans le dictionnaire.

Un dictionnaire relie donc directement une clé, qui n'est pas nécessairement un entier, à une valeur : pas besoin d'index intermédiaire pour rechercher une valeur comme dans une liste. Par contre, cette **clé est nécessairement d'un type immuable**, qui ne peut pas être modifiée.

Exemple - Élément chimique On suppose que les éléments chimiques sont enregistrés via une chaîne de caractères : "C", "O" , "H", "Cl", "Ar", "N". Soit `d`, un dictionnaire correspondant à la figure ci-dessous. Accéder au numéro atomique de l'élément C s'écrit : `d ["C"]`.



Attention! Un dictionnaire n'est pas une structure ordonnée, à la différence des listes ou des tableaux.

Les dictionnaires sont utiles notamment dans le cadre de la programmation dynamique (cf chapitre suivant) pour la mémoïsation, c'est à dire l'enregistrement des valeurs d'une fonction selon ses paramètres d'entrée. Par exemple :
- → a-t-on déjà rencontré un sommet lorsqu'on parcourt un graphe ?
- → a-t-on déjà calculé la suite de Fibonacci pour $n = 4$?

Répondre à ces questions exige de savoir si pour une clé donnée il existe une valeur.

Si on utilise une liste pour stocker ces informations, par exemple (n, fibo(n)), les performances de l'exécution d'un algorithme peuvent être mauvaises : en effet, rechercher un élément dans une liste peut présenter dans le cas le pire une complexité linéaire.

Si on implémente bien un dictionnaire, tester l'appartenance à un dictionnaire peut être de complexité constante, ce qui peut accélérer grandement l'exécution d'un algorithme.

1.2. Création d'un dictionnaire

Les dictionnaires sont définis par des accolades et on met dans l'ordre la clé et la valeur correspondante séparés par un double point : `dico={clé1:valeur1,clé12:valeur2}`

```
>>> dico={ } # création d'un dictionnaire vide
>>> type(dico)
< class 'dict'>
>>> dico={'H':1,'C':6,'N':7,'O':8}
>>> print (dico)
{'H':1,'C':6,'N':7,'O':8}
```

Les clés d'un dictionnaire ne sont pas forcément toutes du même type. Nous pouvons utiliser tout objet non mutable (immuable) pour les clés : des **entiers** (type `int`), des **flottants** (type `float`), des **chaînes de caractères** (type `str`), ou bien des **n-uplets** (type `tuple`). **À l'inverse, une liste ne peut pas être une clé.**

Les valeurs peuvent être de n'importe quel type de données.

```
dico2{3:[], 'C':6, (15,15):[1,2,3,4]}
```

On peut également créer un dictionnaire en compréhension comme pour les listes :

```
>>>dico3=\{x:x**2 for x in range(1,6)}
>>>dico3
{1:1, 2:4, 3:9, 4:16, 5:25}
```

Cette méthode de création est pratique mais difficile à manipuler (sauf dans un cas très simple comme ci-dessus).

1.3. Opérations sur un dictionnaire

1.3.1. Nombre d'éléments

La fonction `len` renvoie le nombre d'éléments associés d'un dictionnaire.

```
>>>len(dico)
4
```

1.3.2. Accès à un élément

On accède aussi à un élément avec la même notation entre crochet que pour les listes, mais en donnant cette fois la clé d'accès en argument : `dictionnaire[clé]`

```
>>> dico['C'] # accès à la valeur de la clé 'C'
6
>>> dico['P'] # erreur quand on demande l'accès à une clé non présente dans le
    dictionnaire
Traceback (most recent call last):
        File "<console>", line 1 , in <module>
KeyError: 'P'
```

1.3.3. Ajout et suppression d'un élément

L'opérateur `[]` est nécessaire pour ajouter un élément.

```
>>> dico['P']=14 # ajoute l'élément de clé 'P' de valeur 14
>>> print (dico)
{'H':1, 'C':6, 'N':7, 'O':8,' P':14}
>>> dico['P']=15 # la clé 'P' existe, on la modifie avec la valeur 15
>>> print (dico)
{'H':1, 'C':6, 'N':7, 'O':8, 'P':15}
>>> del dico['C'] # supprime l'élément de clé 'C'
>>> print (dico)
{'H':1, 'N':7, 'O':8, 'P':15}
```

1.3.4. Test d'appartenance

Pour tester si une clé est présente dans le dictionnaire, on utilise `clé in dictionnaire` qui renvoie un booléen : `True` si la clé est présente, `False` sinon.

```
>>> 'O' in dico
True
>>> 'Ar' in dico
False
```

1.3.5. Parcours

Pour parcourir les clés d'un dictionnaire avec une boucle `for`, on utilise la syntaxe :

```
for cle in dictionnaire :
```

Par exemple, avec le dictionnaire précédent, si on veut construire une liste `L` contenant les valeurs du dictionnaire `dico` :

```
L=[]
for cle in dico:
L.append(dico[cle])
```

On peut aussi le faire en définissant la liste par compréhension.

```
L=[dico[c] for c in dico]
```

On peut accéder à l'ensemble des clés à l'aide de la méthode `keys()`, et à la liste des couples `clé:valeur` par la méthode `items()`

```
>>> cles=dico.keys()
>>> cles
dict_keys (['H', 'N', 'O', 'P'])
>>> couples=dico.items()
>>> couples
dict_items([('H':1), ('N':7), ('O':8), ('P':15)])
```

On obtient à chaque fois un objet de type `dict_keys` et `dict_items` respectivement, qui se comporteront de manière presque similaire à un objet de type `list` (on peut le parcourir, mais toutefois pas accéder à un élément par son indice).

2. Copies

Si une valeur est d'un type mutable, on peut alors la modifier sans réaffectation. Tous les exemples précédents montrent bien que les dictionnaires sont des objets mutables en Python. Se poseront donc les mêmes problèmes d'alias et de copie que pour les listes.
On pourra utiliser la méthode `copy()` pour réaliser une copie superficielle d'un dictionnaire, ou bien la fonction `deepcopy` de la bibliothèque `copy` pour réaliser une copie profonde de ce dictionnaire.

Exercice - Commenter les scripts suivants.

```
>>> d={0:['a','b']}
>>> e=d
>>> d[1]=['c','d']
>>> d
{0: ['a', 'b'], 1: ['c', 'd']}
>>> e
{0: ['a', 'b'], 1: ['c', 'd']}
```

```
>>> d={0:['a','b']}
>>> e=d.copy()
>>> d[1]=['c','d']
>>> d[0].append(['e','f'])
>>> d
{0: ['a', 'b', ['e', 'f']], 1: ['c', 'd']}
>>> e
{0: ['a', 'b', ['e', 'f']]}
```

```
>>> from copy import deepcopy
>>> d={0:['a','b']}
>>> e=deepcopy(d)
>>> d[1]=['c','d']
>>> d[0].append(['e','f'])
>>> d
{0: ['a', 'b', ['e', 'f']], 1: ['c', 'd']}
>>> e
{0: ['a', 'b']}
```

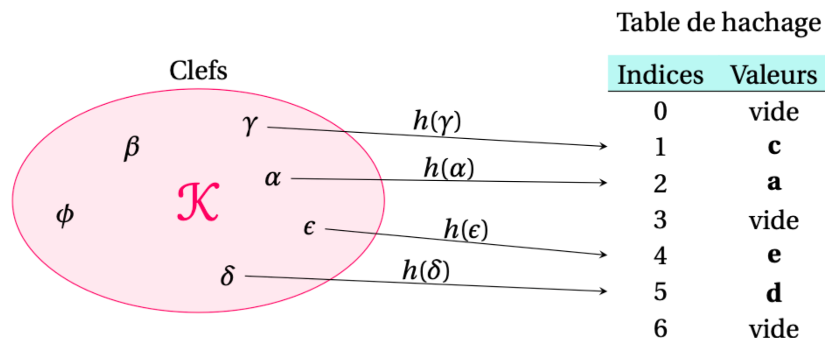
3. Hachage

L'objectif de cette partie est de comprendre comment les dictionnaires sont implémentés en utilisant le tables de hachage afin de garantir, notamment, un accès, une insertion ou modification en temps constant (indépendant de la taille du dictionnaire).

3.1. Principe et définition

Une **fonction de hachage** est une fonction qui à une clé associe un entier appelée **valeur de hachage de la clé**.
Une fonction de hachage n'est pas toujours injective. Deux objets ayant même valeur de hachage créent une **collision**.

Une **table de hachage** est constituée d'un tableau t et d'une fonction de hachage h. Elle implémente un dictionnaire sur un ensemble de clés $\mathscr{K}$ et un ensemble de valeurs $\mathscr{V}$.
Pour tout élément c de $\mathscr{K}$, $h(c)$ est l'indice de la case du tableau t auquel on stocke la valeur v.



La fonction de hachage h permet de calculer les indices du tableau. La valeur a associée à α se trouve à la case $h(\alpha)$ du tableau.

Toutes les clés n'ont pas forcément de valeur associée à un moment donné de l'algorithme. Dans ce cas, à l'indice associé à cette clé, le tableau est vide.
Soit k le cardinal de $\mathscr{K}$, c'est à dire le nombre de clés possibles. On pourrait représenter un dictionnaire à l'aide d'un tableau de dimension k et une fonction bijective $h : \mathscr{K} \longrightarrow [\![0, k-1]\!]$. L'accès aux éléments et l'ajout d'un élément seraient en $O(1)$, le temps de calculer la valeur de la fonction bijective pour une clé donnée.
Néanmoins, d'un point de vue complexité mémoire, cette solution n'est pas réalisable : le nombre de clés possibles est souvent immense alors que les clés effectivement utilisées sont moins nombreuses. Ce qui nous amènerait à réserver un espace mémoire bien supérieur aux besoins réels.

On adopte donc l'hypothèse réaliste suivante : la taille m du tableau qu'on utilise est petite devant le nombre de clés possibles, c'est à dire $m \ll k$.
En procédant ainsi, on renonce à l'injectivité de la fonction de hachage et donc à sa bijectivité, car on engendre des collisions : il pourra exister des clés différentes pour lesquelles le code calculé par la fonction de hachage sera le même. Ce problème devra être géré.

3.2. Fonctions de hachage

Le choix d'une fonction de hachage n'est pas évident. Ces fonctions doivent permettre de générer un index dont la taille est inférieure à celle du tableau, tout en distinguant au mieux les clés, c'est-à-dire en évitant le

plus possible les collisions.

On recherche donc des fonctions de hachage qui possèdent les caractéristiques suivantes :

- → son calcul doit être rapide,
- → pour une même clé, on obtient un même code (cohérence),
- → pour des clés différentes, on obtient des codes différents (injectivité). Dans le cas contraire, on obtient un collision qu'on cherchera à minimiser.
- → les codes doivent présenter une distribution uniformément répartie sur l'espace des indices, ce qui permet de minimiser les collisions.

Une fonction de hachage peut procéder en deux étapes :

- — Une fonction h_e qui encode la clé d'entrée,
- — et une fonction h_c qui compresse le code précédent obtenu dans l'ensemble des indexes possibles (c'est-à-dire entre 0 et $m-1$, si on note m la taille du tableau).

Cela donne donc : $h = h_e \circ h_c$

Exercice de cours - Pour encoder les clés qui sont des chaînes de caractères $c_0 c_1 \dots c_{p-1}$, on peut utiliser le code ASCII associé à chaque caractère, ascii (c_i) (s'obtient avec la fonction `ord` de Python), puis calculer $\sum_{i=0}^{p-1} \text{ascii}(c_i) \times 2^{8+i}$.

Des chaînes de caractères similaires auront des codes différents. Et deux clés identiques auront le même code. C'est ce que l'on recherche.

Une fois les clés encodées, on cherche à compresser la valeur encodée dans l'intervalle des index possibles $[\![0, m-1]\!]$, si m est la taille de la table de hachage.
Pour cela on peut utiliser une division en calculant la valeur encodée modulo m.

1. Écrire la fonction `he(chaine)` qui prend en argument une chaine de caractère et qui renvoie la valeur encodée de la chaine de caractère selon la somme ci-dessus : $h_e(c) = \sum_{i=0}^{p-1} \text{ascii}(c_i) \times 2^{8+i}$.

```
def he(chaine):
        S=0
        p=len(chaine)
        for i in range(p):
                S=S+ord(chaine[i])*2**(8+i)
        return S
>>> he('physique')
7023616
>>> he('chimie')
1666816
```

2. Écrire la fonction `hc(chaine,m)` qui compresse la valeur encodée dans l'intervalle des index possibles dans la table de taille `m`.

```
def hc(chaine ,m):
        return he(chaine)%m
```

3. Écrire l'instruction pour obtenir la valeur renvoie par la fonction de hachage d'une chaine de caractère donnée.

```
>>> hc('physique', 45305)
1341
>>> hc('chimie', 45305)
35836
```

En Python, la fonction `hash` code les clés. Elle prend en argument un objet non mutable (entiers de type `int`, flottants de type `float`, chaînes de caractères de type `str` et des n-uplets constitués des éléments précédents de type `tuple`).

Puis il faut réduire chaque entier pour obtenir un entier appartenant aux index possibles, c'est-à-dire dans $[\![0, m-1]\!]$, si m est la taille de la table de hachage. Pour cela, on pourra utiliser l'opérateur modulo : `hash(cle)%m`.

4. Exercices

4.1. Utilisation des dictionnaires

4.1.1. Températures

On dispose des températures à Montélimar à 8h00 dans un dictionnaire :

```
temp={'J1':-10, 'J2':-9, 'J3':-4, 'J4':0, 'J5':-1, 'J6':4, 'J7':-5, 'J8':1, 'J9'
    :-2}
```

1. Écrire une fonction `moyenne` qui prend en argument un dictionnaire `d` du type de celui défini ci-dessus et qui renvoie la valeur moyenne des températures.

2. Écrire une fonction `froid(d,T0)` qui prend en argument un dictionnaire d du type de celui défini ci-dessus et qui renvoie la liste des jours et le nombre de jours où la température a été inférieure à une certaine température `TO`.

4.1.2. Moyennes

Étant donné un dictionnaire python dont les clés sont les noms des élèves et les valeurs sont les listes des notes et qui renvoie un autre dictionnaire dont les clés sont le nom des élèves et les valeurs la moyenne de leur note. Par exemple : `nom_moy({'Louise':[12,15,12], 'Gabriel':[15,17,16], 'Léon':[8,18,7]})` renverra `{'Louise': 13, 'Gabriel': 16, 'Léon': 11$}`.

Indication : On pourra commencer par programmer une fonction intermédiaire.

4.1.3. Occurrences dans une chaîne de caractère

Écrire une fonction `occurrences(texte)` qui prend en argument une chaîne de caractère texte et renvoie un dictionnaire dont les clés sont les lettres qui apparaissent dans le texte et les valeurs le nombre d'occurrences de ces lettres.

Par exemple : `occurrences('ACCTAGCCCTA')` renverra `'A':3, 'C':5, 'T':2, 'G':1}`.

4.1.4. Min-Max dans un dictionnaire

Écrire une fonction `min_max` qui prend en argument une liste de nombres non vide et renvoie un dictionnaire dont les clés sont les chaînes `"min"` et `"max"` avec pour valeurs respectives le minimum et le maximum des nombres de la liste.

Par exemple : `min_max([8, 5, 9, 3, 1, 7]` renverra `{"min":1, "max": 9}`.

4.1.5. Matrice

1. Comment peut-on manipuler des matrices en Python en utilisant un dictionnaire ?

On s'intéresse ici aux matrices parcimonieuses, c'est-à-dire dont la plupart des coefficients sont nuls. Une telle matrice M de dimensions (n, p) pourra être codée par un dictionnaire ayant pour couples clefs/valeurs :

- → `'dim'` : (n, p)
- → (i, j) : M_{ij} pour chaque couple (i, j) tel que $M_{i,j} \neq 0$.

2. Donner le dictionnaire qui code la matrice : $\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \end{pmatrix}$

3. Proposer une fonction d'addition de deux matrices (de même dimension).

4. Si la première matrice contient c coefficients non nuls, et la seconde c', quelle est la complexité temporelle de cet algorithme ?

4.1.6. Quelle heure est-il ?

Un site de voyage permet de calculer les temps de parcours entre deux villes sous forme d'un dictionnaire qui contient 4 clefs (`'jours'`,`'heures'`,`'minutes'` et `'secondes'`) dont les valeurs associées représentent respectivement les durées en jours, heures, minutes et secondes pour le voyage, le tout de manière unique, c'est-à-dire que 27 heures vaut en fait 1 jour et 3 heures. Néanmoins, si vous faites plusieurs escales, vous voudriez bien connaître la durée totale que vous aurez passée dans les transports.

On peut décomposer ce problème en plusieurs sous-problèmes plus simples à résoudre.

1. Écrire une fonction `decomposition(duree)` qui prend en argument une durée exprimée en secondes et renvoie un dictionnaire à valeurs entières dont les clefs sont `'jours'`, `'heures'`, `'minutes'` et `'secondes'`.
2. Écrire la fonction inverse `secondes(dico)` qui prend en argument un dictionnaire du type précédent pour renvoyer la valeur correspondante en secondes.
3. En utilisant les deux fonctions précédentes, écrire la fonction `addition(dico1, dico2)` qui va additionner correctement deux dictionnaire `dico1` et `dico2` correspondant à deux voyages successifs et renvoyer le dictionnaire du même type correspondant à la durée totale du voyage.
4. Écrire une fonction `affichage(dico)` qui affiche (à l'aide de `print`) le temps total passé en transport de manière un peu plus lisible pour le commun des mortels.

   Par exemple, l'appel à la fonction renvoie :

```
>>> affichage({'jours': 3, 'heures': 22, 'minutes': 10, 'secondes': 54})
Vous allez voyager un total de 3 jours, 22 heures, 10 minutes et 54 secondes
```

5. Enfin, écrire une fonction `temps_total(liste_de_durees)` qui prend en argument une liste (de taille arbitraire) de durées sous la forme des dictionnaires précédents et renvoie le temps total de parcours à l'aide de la fonction `affichage(dico)` précédente.

4.2. Dictionnaire et table de hachage

4.2.1. Double hachage

Le double hachage est l'une des meilleurs méthodes connues pour l'adressage ouvert. Il utilise une fonction de hachage de la forme :

$$h: \begin{array}{ccc} N \times N & \longrightarrow & [\![0, m-1]\!] \\ (k, i) & \longmapsto & (h_1(k) + i h_2(k)) \bmod m \end{array}$$

où h_1 et h_2 sont des fonctions de hachages. i prend les valeurs suivantes :

- → Par défaut, $i = 0$.
- → S'il y a collision, on incrémente i de 1 , jusqu'à ne plus avoir de collision pour la clé considérée.
- → Il reprend la valeur 0 pour la clé suivantes ...

1. Insérer les clés : 5,28,19,15,20,33,12,17,10 dans une table de taille $m = 13$ avec $h_1(k) = k \bmod 13$ et $h_2(k) = 1 + (k \bmod 12)$.
2. Proposer une fonction en Python qui prend en argument une clé `c` (entier) et la taille `m` de la table, et renvoie la valeur de `h(c)`.

4.2.2. Polynôme

On considère des polynômes non nuls à coefficients entiers de degré quelconque mais qui ne contiennent pas plus de cinq monômes. On utilise un tableau de longueur $16 = 8 \times 2$ pour stocker les couples (degré, coefficient) dans lequel on pourrait stocker au maximum huit couples. Les places non occupées contiennent la valeur -1 .

La fonction de hachage h est la fonction identité : pour tout $n \in N, h(n) = n$. Donc à un degré qui vaut 10 , on associe le nombre 10, soit $h(10) = 10$. Ensuite, avec `10%8`, on obtient l'indice 2 et à cet indice on écrit le degré, (la clé), suivi du coefficient, (la valeur).

Par exemple, le polynôme $8 + 3x^{10} - 5x^{12}$ est stocké dans un tableau de la forme :

indice 0	0	8
indice 1	-1	-1
indice 2	10	3
indice 3	-1	-1
indice 4	12	-5
indice ...	...	...

1. Donner le tableau correspondant au stockage du polynôme $2x^5 - 3x^{34} + 4x^{105}$.
2. Quel est le problème avec par exemple le polynôme $8 - 5x^2 + 3x^{10}$?
3. En cas de collision, on décide d'utiliser la première place libre suivante. Les monômes sont entrés dans le tableau suivant l'ordre de lecture. Donner un exemple de polynôme de degré minimum qui génère une collision pour chaque monôme excepté le premier.

4. On envisage une autre possibilité de stockage avec deux tableaux, un tableau pour les couples (degré, coefficient) et un tableau pour les indices, les deux tableaux ayant pour capacité 8 . Avec le polynôme $4x^3 - 2x^5 + 4x^9$, on obtient les deux tableaux de la manière suivante :

- → dans le premier tableau, on écrit chaque degré avec le coefficient correspondant suivant l'ordre des degrés et on complète le tableau avec des 0 ;
- → dans le second tableau, on calcule $d\%8$ où d est un degré et on place à l'indice trouvé l'indice où on trouve le couple (degré,coefficient) dans le premier tableau. On complète le tableau avec des -1 . Extraits des tableaux :

indice 0	3	4
indice 1	5	-2
indice 2	9	4
indice 3	0	0
indice …	…	…

indice 0	-1
indice 1	2
indice 2	-1
indice 3	0
indice 4	-1
indice 5	1

Donner les deux tableaux correspondant au stockage du polynôme $3x^5 - x^{18} + 7x^{20}$.