DICTIONNAIRES

À la fin de ce chapitre, je sais :

- ☞ décrire le TAD dictionnaire
- ☞ créer et utiliser un dictionnaire en Python
- ☞ expliquer l'implémentation d'un dictionnaire par une table de hachage

A Types abstraits de données (TAD)

La définition d'un TAD est donnée dans le chapitre précédent (cf. définition **??**).

■ **Définition 1 — TAD Dictionnaire.** Un dictionnaire est une extension du TAD tableau dont les éléments $\mathcal{V}$, au lieu d'être indicés par un entier sont indicés par des clefs appartenant à un ensemble $\mathcal{K}$. Soit $k \in \mathcal{K}$, une clef d'un dictionnaire `D`. Alors `D[k]` est la valeur v de $\mathcal{V}$ qui correspond à la clef k.

On dit qu'un dictionnaire est un **tableau associatif** qui associe une clef `k` à une valeur `v`.
Les opérations sur un dictionnaire sont :

1. rechercher la présence d'une clef dans le dictionnaire,
2. accéder à la valeur correspondant à une clef,
3. insérer une valeur associée à une clef dans le dictionnaire,
4. supprimer une valeur associée à une clef dans le dictionnaire.

(P) Un dictionnaire relie donc directement une clef, qui n'est pas nécessairement un entier, à une valeur : pas besoin d'index intermédiaire pour rechercher une valeur comme dans une liste ou un tableau. Par contre, cette clef est nécessairement d'un type immuable. Considérons l'exemple donnée sur l'exemple de la figure 2. 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 2. Accéder au numéro atomique de l'élément `C` s'écrit : d["C"].

(R) Un dictionnaire n'est pas une structure ordonnée, à la différence des listes ou des tableaux (cf. **??**).

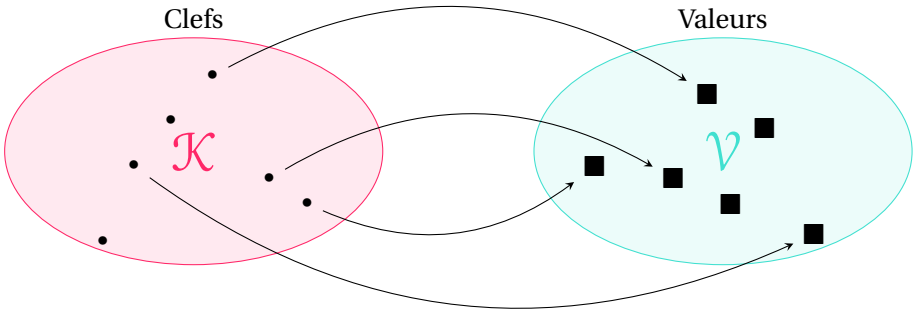

FIGURE 1 – Illustration du concept de dictionnaire

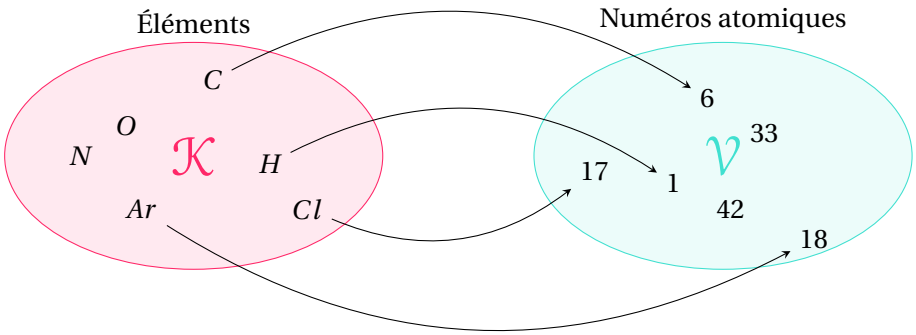

FIGURE 2 – Illustration du concept de dictionnaire, ensembles concrets

R L'intérêt majeur d'un dictionnaire est que la complexité pour accéder et recherche un élément est constante en $O(1)$.

■ **Exemple 1 — Usage des dictionnaires.** Les dictionnaires sont utiles notamment dans le cadre de la programmation dynamique 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 clef 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 (cf. tableau **??**). Si on implémente bien un dictionnaire, tester l'appartenance à un dictionnaire est de complexité constante, ce qui peut accélérer grandement l'exécution d'un algorithme.

B Constructeurs de dictionnaires

Cette section s'intéresse à la manière dont on peut créer des dictionnaires en Python.

a Accolades

À tout seigneur tout honneur, les accolades sont la voie royale pour créer un dictionnaire.

```
d = {}  # Empty dict
print(d, type(d))  # {} <class 'dict'>
d = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
# keys are strings, values integers
print(d)  # {'Ar': 18, 'Na': 11, 'Cl': 17, 'H': 1}

d = {("Alix", 14): True, ("Guillaume", 7): False, ("Hannah", 24): 17}
print(d, type(d))  # keys are tuples, values booleans
# {('Alix', 14): True, ('Guillaume', 7): False, ('Hannah', 24): 17} <class 'dict'>


d = {13: [1, 3], 219: [2, 1, 9], 42: [4, 2]}
print(d, type(d))  # keys are integers, values lists
# {13: [1, 3], 219: [2, 1, 9], 42: [4, 2]} <class 'dict'>
```

L'exemple précédent montre que :

- Les clefs d'un dictionnaire sont nécessairement constituées de types **immuables**, c'est à dire ni une liste ni un dictionnaire par exemple. Si la clef était muable, alors le code calculé par la fonction de hachage serait variable pour une même clef : on ne saurait donc plus identifier la valeur associée ou bien celle-ci changerait.
- Les valeurs peuvent être de n'importe quel type de données.

b Constructeur `dict`

La fonction `dict()` permet également de créer un dictionnaire à partir de n'importe quel objet itérable. Elle s'utilise le plus souvent pour convertir un liste de tuples en dictionnaire.

```
d = dict()  # Empty dict
print(d, type(d))  # {} <class 'dict'>
d = dict([("Ar", 18), ("Na", 11), ("Cl", 17), ("H", 1)])
# keys are strings, values integers
print(d)  # {'Ar': 18, 'Na': 11, 'Cl': 17, 'H': 1}
d = dict(zip(["Alix", "Guillaume", "Hannah"], [14, 7, 24]))
# zip is really useful here !
print(d)  # {'Alix': 14, 'Guillaume': 7, 'Hannah': 24}
```

---

c Définir un dictionnaire en compréhension

Tout comme les ensembles en mathématiques, les dictionnaires peuvent être construits à partir d'une description compréhensible de ses éléments. Cette méthode de création de dictionnaires est à rapprocher du paradigme fonctionnel.

```
pairs = [("Alix", 14), ("Guillaume", 7), ("Hannah", 24)]
d = {name: age for name, age in pairs} # comprehension dictionnary
print(d) # {'Alix': 14, 'Guillaume': 7, 'Hannah': 24}
```

P Cette méthode de création de dictionnaire est puissante mais est très délicate à manipuler. C'est pourquoi il est préférable de ne l'utiliser que si on est vraiment sûr de soi, sinon c'est une perte de points assurée au concours. On peut l'éviter avec une boucle `for` et ainsi assurer des points.

Le paragraphe précédent est important pour l'épreuve d'informatique!

C Opérations sur un dictionnaire

a Nombes d'éléments d'un dictionnaire

La fonction `len` renvoie le nombre de clefs d'un dictionnaire.

```
d = dict([("Ar", 18), ("Na", 11), ("Cl", 17), ("H", 1)])
print(len(d)) # 4
```

P En Python, on peut tester si un dictionnaire est vide avec la fonction `len`.

```
empty_dict = {}
if not empty_dict:    # direct
  print("Dictionary is empty!")
else:
```

```
    print("Dictionary is not empty!")

if len(empty_dict) == 0:  # nb of elements
  print("Dictionary is empty!")
else:
  print("Dictionary is not empty!")

if empty_dict == {}:  # compare to an empty dict
  print('Dictionary is empty!')
else:
  print('Dictionary is not empty!')
```

b Appartenance à un dictionnaire

Les mots-clefs `in` et `not in` permettent de tester l'appartenance à un dictionnaire et renvoient les booléens correspondants.

```
d = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
if  "Ar" in d:
  print("Ar", d["Ar"]) # Ar 18
if "O" not in d:
  print("O is not in d")  # O is not in d
```

c Ajouter et supprimer un élément sur un dictionnaire

L'opérateur `[]` est nécessaire pour ajouter une valeur d'après sa clef. On peut ajouter plusieurs éléments avec `update`. Enfin, les fonctions qui retirent un élément modifient le dictionnaire en place.

```
d = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
d["O"] = 8
print(d)  # {'Ar': 18, 'Na': 11, 'Cl': 17, 'H': 1, 'O': 8}
del d["Ar"]
print(d)  # {'Na': 11, 'Cl': 17, 'H': 1, 'O': 8}
d.pop("Na")
print(d)  # {'Cl': 17, 'H': 1, 'O': 8}
d.update({"F": 9, "Br": 35})
print(d)  # {'Cl': 17, 'H': 1, 'O': 8, 'F': 9, 'Br': 35}
```

D Fusionner des dictionnaires

Pour fusionner deux dictionnaires, on peut soit utiliser la méthode `update` qui modifie le dictionnaire en place, soit utiliser un syntaxe dépendante des versions de Python.

```
d1 = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
d2 = {"F": 9, "Br": 35}
d1.update(d2)  # simple, in place
print(d1) #{'Cl': 17, 'H': 1, 'O': 8, 'F': 9, 'Br': 35}
```

```python
d1 = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
d2 = {"F": 9, "Br": 35}
d = {**d1, **d2}  # Python 3.5
print(d) #{'Ar': 18, 'Na': 11, 'Cl': 17, 'H': 1, 'F': 9, 'Br': 35}


d1 = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
d2 = {"F": 9, "Br": 35}
d = d1 | d2  # Python 3.9
print(d)  #{'Ar': 18, 'Na': 11, 'Cl': 17, 'H': 1, 'F': 9, 'Br': 35}
```

E Des dictionnaires itérables

Un dictionnaire Python est itérable, c'est à dire qu'il peut être l'objet d'une itération via une boucle `for`.

```python
d = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}

for atom in d:
  print(atom, d[atom])

for atom, number in d.items():
  print(atom, number)

# Same result :
# Ar 18
# Na 11
# Cl 17
# H 1
# F 9
# Br 35
```

F Implémentation d'un TAD dictionnaire

On peut implémenter efficacement un TAD dictionnaire à l'aide

1. des tables de hachage,
2. des arbres rouges et noirs,
3. d'arbres binaires de recherche.

Il existe de nombreuses implémentations possibles du TAD dictionnaire. On peut, par exemple, les implémenter avec des listes, mais l'efficacité n'est pas au rendez-vous. C'est pourquoi, dans la suite ce chapitre, on s'intéresse aux tables de hachage pour implémenter un TAD dictionnaire.

Opération	Table de hachage	Arbre de recherche équilibré
Ajout (pire des cas)	$O(n)$	$O(\log n)$
Accès (pire des cas)	$O(n)$	$O(\log n)$
Ajout (en moyenne)	$O(1)$	$O(\log n)$
Accès (en moyenne)	$O(1)$	$O(\log n)$

TABLE 1 – Complexité des opérations associées à l'utilisation des tables de hachage ou des arbres pour implémenter un TAD dictionnaire. Les coûts indiqués sont dans le pire des cas ou en moyenne.

G Tables de hachage

a Principe

■ **Définition 2 — Table de hachage.** 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 clefs $\mathcal{K}$ et un ensemble de valeurs $\mathcal{V}$.

Pour tout élément k de $\mathcal{K}$, $h(k)$ est l'indice de la case du tableau t auquel on stocke la valeur v, ce qui peut s'exprimer ainsi :

$$\forall (k, v) \in (\mathcal{K} \times \mathcal{V}), t[h(k)] \leftarrow v \tag{1}$$

La figure 3 illustre l'implémentation d'un dictionnaire par une table de hachage.

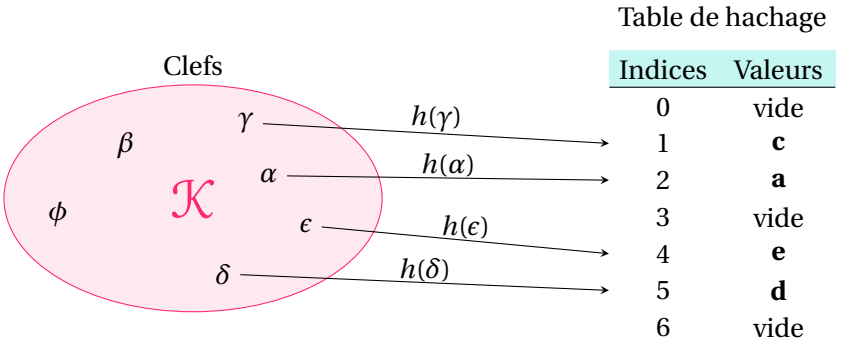

FIGURE 3 – Illustration de l'implémentation d'un dictionnaire par table de hachage. La fonction de hachage h permet de calculer les indices du tableau. **La valeur** a **associée à** α **se trouve à la case** $h(\alpha)$ **du tableau.** Toutes les clefs n'ont pas forcément de valeur associée à un moment donnée de l'algorithme. Dans ce cas, à l'indice associé à cette clef, le tableau est vide.

■ **Définition 3 — Fonction de hachage.** Un fonction de hachage prend une clef en entrée et génère un index entier associé à cette clef.

Plus formellement, si $\mathcal{K}$ est l'un ensemble des clefs (non nécessairement numériques) et $\mathcal{V}$ l'ensemble des valeurs associées aux clefs de cardinal n, on peut définir une fonction de hachage :

$$h : \mathcal{K} \longrightarrow [\![0, n-1]\!] \tag{2}$$

$$k \longmapsto i \tag{3}$$

Soit k le cardinal de $\mathcal{K}$, c'est à dire le nombre de clefs possibles. On pourrait représenter un TAD dictionnaire à l'aide d'un tableau de dimension k et une fonction bijective $h : \mathcal{K} \longrightarrow [\![0, n-1]\!]$. 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 clef donnée. Néanmoins, d'un point de vue complexité mémoire, cette solution n'est pas réalisable : le nombre de clefs possibles est souvent immense alors que les clefs effectivement utilisées sont moins nombreuses. Ce qui nous amèneraient 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 clefs possibles, c'est à dire $m << c$. 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 clefs différentes pour lesquelles le code calculé par la fonction de hachage sera le même. Tout dépend du nombre de clefs utilisées et de la fonction de hachage.

b Choix d'une fonction 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 clefs, tout en évitant le plus possible les collisions. On recherche donc des fonctions de hachage qui possèdent les caractéristiques suivantes :

1. son calcul doit être rapide,
2. pour une même clef, on obtient un même code (**cohérence**),
3. pour des clefs différentes, on obtient des codes différents (**injectivité**). Dans le cas contraire, on obtient un **collision** qu'on cherchera à minimiser.
4. pour des clefs qui se ressemblent, les codes obtenus doivent être très différents. D'une manière générale, les codes doivent présenter une distribution uniformément répartie sur l'espace des indices (**répartition uniforme**).

Pour minimiser les collisions, c'est à dire lorsque la fonction de hachage appliquée à deux clefs différentes c_1 et c_2 produit le même résultat $h(c_1) = h(c_2)$, on cherche à répartir uniformément les clefs dans les différentes cases du tableau. Ceci revient à faire en sorte que la probabilité qu'une valeur v associée à une clef c occupe la case i du tableau devrait être proche de $1/m$, si la taille de ce tableau est m. En faisant cette hypothèse, si on suppose qu'on utilise g clefs, alors la probabilité p d'obtenir des codes différents lors du calcul des g clefs par la fonction de hachage vaut :

$$p = \frac{m(m-1)\dots(m-g+1)}{m^g} = \frac{m!}{(m-g)!m^g} \tag{4}$$

On en déduit la probabilité de collision p_c :

$$p_c = 1 - \frac{m(m-1)\dots(m-g+1)}{m^g} = \frac{m!}{(m-g)!m^g} \tag{5}$$

g/m	5000	10000	100000	1000000
100	0,63	0,39	0,05	0,005
500	0,99	0,99	0,71	0,12
1000	1	1	0,99	0,39
1500	1	1	0,99	0,68
2000	1	1	0,99	0,86
2500	1	1	0,99	0,96

TABLE 2 – Probabilité de collision p_c dans l'hypothèse d'une répartition uniforme des valeurs dans le tableau d'une table de hachage. Même dans le cas d'un tableau à un million d'éléments, la probabilité de collision est quasi-certaine dès que la taille des clefs utilisées est supérieure à 2500. C'est le paradoxe des anniversaires.

Une rapide évaluation de cette probabilité est donnée sur le tableau 2. On en conclut que, quelle que soit la taille du tableau, les collisions existeront. Il faut donc trouver un moyen de les gérer.

c Fonctions de hachages possibles

Une fonction de hachage h peut procéder en deux étapes :

1. une fonction h_e qui encode la clef d'entrée,
2. et une fonction h_c qui compresse le code dans l'ensemble des indexes.

Plus formellement, on la fonction de hachage comme une fonction composée :

$$h_e : \mathcal{K} \longrightarrow \mathbb{N} \tag{6}$$

$$h_c : \mathbb{N} \longrightarrow [\![0, n-1]\!] \tag{7}$$

et

$$h = h_e \circ h_c \tag{8}$$

■ **Exemple 2 — Fonctions d'encodage des clefs.** Pour compresser une clef, il faut d'abord l'encoder, c'est à dire la convertir en un nombre entier. Dans ce but, on peut par exemple, si la clef est une chaîne de caractères $c_0 c_1 \dots c_{k-1}$, utiliser le code ASCII associé à un caractère $\text{ascii}(c_i)$ pour calculer $\sum_{i=0}^{k-1} \text{ascii}(c_i) 2^{8+k}$. On verra en TP que faire juste la somme des valeurs

ASCII des caractères n'est pas forcément une bonne idée. Prendre l'adresse en mémoire non plus.

L'important est que l'encodage génère un même code pour une même clef et un code vraiment différent si les clefs sont différentes.

■ **Exemple 3 — Hachage par modulo simple.** Si la taille du tableau de la table de hachage est m, on peut choisir la fonction $h : c \longrightarrow c \mod m$. Le choix de m mérite néanmoins quelques points d'attention : on évitera de prendre des nombres du type 2^q ou 2^{q-1}. On choisit généralement un nombre premier éloigné pas trop proche d'une puissance de 2 pour garantir une bonne répartition des clefs.

■ **Exemple 4 — Hachage par multiplication et modulo .** On peut également choisir la fonction $h : c \longrightarrow \lfloor (\phi c \mod 1) \times m \rfloor$. Si ϕ est le nombre d'or, cette méthode garantit une bonne répartition des clefs sans restreindre les valeurs de m.

d Gestion des collisions

Deux grandes méthodes permettent de gérer les collisions :

1. **par chainage** : stocker valeurs associées aux collisions dans une même case sous la forme d'une liste comme l'illustre la figure 4. Cette solution induit une augmentation de la complexité car en cas de collision, on n'accède plus à l'élément directement : il faut le chercher dans une liste. C'est ce qui explique le $O(n)$ dans le pire des cas sur le tableau 1.
2. **par adressage ouvert** : chercher une place vide dans le tableau en le sondant et placer la valeur dedans. Cela induit que le nombre de clefs utilisées est inférieur à la taille du tableau et ralentit l'accès à un élément. Lorsque le tableau est trop petit, on peut en changer pour un plus grand : cela a un coût également.

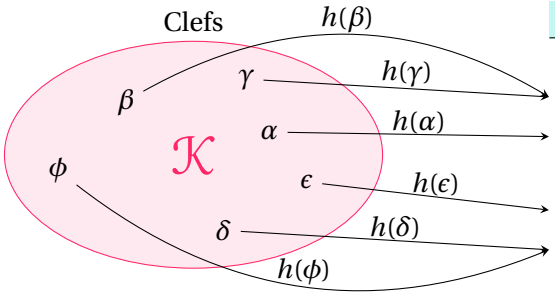

Table de hachage

Indices	Valeurs
0	vide
1	$[(\gamma, c), (\beta, b)]$
2	a
3	vide
4	e
5	$[(\delta, d), (\phi, f)]$
6	vide

FIGURE 4 – Illustration de l'implémentation d'un dictionnaire par table de hachage avec chainage. Les clefs β et ϕ, engendrent des collisions. On stocke donc les valeurs possibles pour un même code (indice) dans une liste.

e Implémentation des `dict` en Python

L'implémentation du TAD dictionnaire en Python fait l'objet de travaux et d'évolutions en permanence et c'est un sujet complexe. En résumé, pour implémenter les `dict`, Python utilise des tables de hachage en adressage ouvert dont la taille évolue dynamiquement (comme le type `list`).