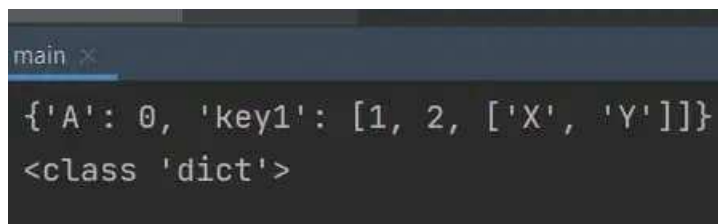


VIII

Dictionnaires 2 : écriture, manipulation, et parcours



```
main x
{'A': 0, 'key1': [1, 2, ['X', 'Y']]}
<class 'dict'>
```

FIGURE VIII.1 – Dcitionnaire en python

PLAN DU CHAPITRE

I	Principaux éléments de syntaxe des dictionnaires ou "tableau associatifs" . . .	2
I.1	Instanciation et implémentation des couples de données	2
I.2	Manipulations des dictionnaires	3
	a - Interrogation des données	3
	b - Vérification de la présence d'un couple <clé>:<valeur>.	4
	c - Longueur d'un dictionnaire	5
	d - Suppression de données : <code>.pop()</code> , <code>.popitem()</code> , <code>del()</code> , <code>.clear()</code>	5
	e - Concaténation	6
	f - Fusion de deux dictionnaires : <code>.update()</code>	7
I.3	Copie d'un dictionnaire	8
I.4	Résumé des commandes essentielles (à connaître!)	8
II	Parcours des dictionnaires	9
II.1	Parcours par les clés	9
II.2	Parcours d'un dictionnaire par ses valeurs	9
II.3	Parcours d'un dictionnaire par ses paires	10

I Principaux éléments de syntaxe des dictionnaires ou "tableau associatifs"

I.1 Instanciation et implémentation des couples de données

Un dictionnaire est un ensemble non ordonné de couples d'informations de forme :

`<clé>:<valeur>`

les délimiteurs du dictionnaire étant des accolades `{.....}`.

La recherche d'une **valeur** dans un dictionnaire se fait à l'aide de sa **clé**. Les clés peuvent être de tout type, **hormis une liste**. Les valeurs peuvent en revanche être rigoureusement de tout type.

On crée et interroge un dictionnaire de la façon suivante :

- La création d'un dictionnaire "vide" ou **instanciation** peut se faire de deux manières :

```
>>> dictioAF={ } #par les délimiteurs
>>> dictioAF=dict() #par déclaration de type
>>> type(dictioAF)
<class 'dict'>
```

- L'implémentation des données, c'est à dire des couples `<clé>:<valeur>` se fait ensuite selon la syntaxe suivante :

```
>>> dictioAF={"screen" :u"écran","table" : "table","desk" : "bureau", "watch" : "montre"}
```

Pour l'ajout d'un nouveau couple `<clé>:<valeur>` dans un dictionnaire, on procède ainsi :

```
>>> dictioAF["sofa"]="canapé"
>>> dictioAF
{'table' : 'table', 'screen' : u'écran', 'sofa' : u'canapé', 'watch' : 'montre', 'desk' : 'bureau'}
```

En outre, les dictionnaires sont des structures de données mutables (contrairement aux chaînes de caractères par exemple) ; on peut donc muter une valeur stockée sur une clé :

```
>>> dictioAF["sofa"]="divan"
>>> dictioAF
{'table' : 'table', 'screen' : u'écran', 'sofa' : 'divan', 'watch' : 'montre', 'desk' : 'bureau'}
```

Il est enfin possible d'implémenter les ensembles `<clé>:<valeur>` d'un dictionnaire par compréhension à partir de données inscrites dans un autre *conteneur*, par exemple une liste de listes (ce pourrait être des tuples également) ; le principe est le suivant :

```
>>> donnees=[["A",0],["B",1],["C",2],["D",3]]
>>> d={cle :val for cle,val in donnees}
>>> d
{'A' : 0, 'B' : 1, 'C' : 2, 'D' : 3}
```

Remarque I-1: HÉTÉROGÉNÉITÉ DES TYPES

Contrairement aux tableaux `numpy` qui, pour des raisons évidentes d'indexation mémoire, imposent l'homogénéité de type des données stockées, les dictionnaires gèrent parfaitement des données hétérogènes.

I.2 Manipulations des dictionnaires

a - Interrogation des données

On interroge une valeur présente dans un dictionnaire en invoquant simplement sa **clé**, sans connaissance préalable d'un indice de localisation mémoire contrairement aux tableaux ou listes par exemple : c'est précisément la **fonction de hachage** qui se charge de retrouver l'adresse mémoire à laquelle se trouve la valeur attachée à la clé, et c'est justement toute la puissance de cette structure de données !

Remarque I-2: RECHERCHE \equiv LECTURE

Le principe de construction autour d'une table de hachage assure donc que les deux primitives `RECHERCHER` et `LIRE` reviennent finalement à faire la même chose, avec une complexité temporelle $\mathcal{O}(1)$.

La syntaxe la plus simple est l'emploi des séparateurs `[]` exactement comme pour l'ajout d'un nouveau couple `<clé>: <valeur>` :

```
>>> dictioAF["desk"]
bureau
```

Cette méthode renvoie en revanche une erreur (on parle de `em` "levée d'une exception") dans l'hypothèse où l'on tente d'interroger une association `<clé>: <valeur>` non présente dans le dictionnaire :

```
>>> dictioAF["gate"]
Traceback (most recent call last) :
.....
KeyError : 'gate'
```

Pour éviter ce problème qui engendrerait par exemple l'arrêt d'un programme en cas d'absence de gestion d'erreur, on peut employer la méthode `<dict>.get(<clé>)` qui renvoie `None` en cas d'absence dans le dictionnaires de la clé invoquée, à condition de demander une impression écran avec `print()` :

```
>>> dictioAF.get("desk")
'bureau'
>>> dictioAF.get("gate")
>>> print(dictioAF.get("gate"))
None
```

Pour prévenir l'absence d'une clé lors de l'interrogation d'un dictionnaire, il est possible d'ajouter une valeur alternative :

```
>>> print(dictioAF.get("computer","ordinateur")) #si clé non présente, on peut renvoyer une valeur al-
ternative
ordinateur >>> print(dictioAF.get("desk","ordinateur")) #si clé présente, la valeur alternative est ignorée
bureau
```

On peut enfin renvoyer l'ensemble des couples <clé>:<valeur>, ou bien les clés, ou enfin les valeurs contenus dans un dictionnaire :

```
>>> dictioAF.items()
dict_items([('table', 'table'), ('screen', 'écran'), ('sofa', 'canapé'), ('watch', 'montre'), ('desk', 'bureau')])
>>> dictioAF.keys()
dict_keys(['table', 'screen', 'sofa', 'watch', 'desk'])
>>> dictioAF.values()
dict_values(['table', 'écran', 'canapé', 'montre', 'bureau'])
```

Exercice de cours: (I.2) - n° 1. Construction d'un dictionnaire par lecture d'un autre dictionnaire

1. Définir par compréhension un dictionnaire `dict1` dont les clés sont la suite des entiers naturels de 0 à 25 et les valeurs les 26 lettres majuscules de l'alphabet. On rappelle les commandes `chr(n)` qui renvoie le caractère correspondant à la valeur `n` dans le code ASCII `car` et `ord(car)` qui renvoie le code ASCII du caractère `car`.
2. A partir du dictionnaire `dict1`, construire le dictionnaire `dict2` dont les clés sont cette fois les 26 lettres de l'alphabet et les valeurs les entiers naturels de 0 à 25.

b - Vérification de la présence d'un couple <clé>:<valeur>

Il est possible de vérifier si une association <clé>:<valeur> est présente ou non dans un dictionnaire à l'aide de l'opérateur `in` :

```
>>> dictioAF
'screen' : 'écran', 'sofa' : 'canapé', 'watch' : 'montre', 'table' : 'table'
>>> "screen" in dictioAF
True
>>> "jardin" in dictioAF
False
>>> "screen" in dictioAF.keys() # méthode alternative en énumérant toutes les clés
True
```

c - Longueur d'un dictionnaire

Pour renvoyer la longueur d'un dictionnaire, on peut exploiter à l'instar des listes, la commande `len()` :

```
>>> dictioAF
{'screen' : 'écran', 'sofa' : 'canapé', 'watch' : 'montre', 'table' : 'table'}
>>> len(dictioAF)
4
```

d - Suppression de données : `.pop()`, `.popitem()`, `del()`, `.clear()`

Un dictionnaire étant une structure de données mutable, on peut à loisir retirer un (ou plusieurs) couple(s) `<clé>: <valeur>`. Plusieurs possibilités existent et dépendent de votre souhait de récupérer ou non la (les) valeur(s) supprimée(s).

- Pour supprimer un couple `<clé>: <valeur>`, mais également renvoyer la valeur associée à la clé, on peut utiliser la méthode `.pop` selon la syntaxe :

```
>>> dictioAF
{'table' : 'table', 'screen' : 'écran', 'sofa' : 'canapé', 'watch' : 'montre', 'desk' : 'bureau'}
>>> dictioAF.pop("desk")
bureau
>>> dictioAF
{'table' : 'table', 'screen' : 'écran', 'sofa' : 'canapé', 'watch' : 'montre'}
```

- Pour supprimer le dernier couple `<clé>: <valeur>` avec renvoi de celui-ci sous forme d'un tuple, on utilise la méthode `.popitem()` :

```
>>> dictioAF
{'table' : 'table', 'screen' : 'écran', 'sofa' : 'canapé', 'watch' : 'montre', 'desk' : 'bureau'}
>>> dictioAF.popitem()
('desk', 'bureau')
>>> dictioAF
{'table' : 'table', 'screen' : 'écran', 'sofa' : 'canapé', 'watch' : 'montre'}
```

- Pour supprimer un couple `<clé>: <valeur>` sans renvoi de celle-ci, on peut utiliser la fonction `del` avec la syntaxe suivante :

```
>>> dictioAF
{'table' : 'table', 'screen' : 'écran', 'sofa' : 'canapé', 'watch' : 'montre', 'desk' : 'bureau'}
>>> del dictioAF["watch"]
>>> dictioAF
{'table' : 'table', 'screen' : 'écran', 'sofa' : 'canapé'}
```

- On peut enfin supprimer la totalité des couples `<clé>: <valeur>` d'un dictionnaire avec la commande `clear` :

```
>>> dictioAF
{'table' : 'table', 'screen' : 'écran', 'sofa' : 'canapé'}
>>> dictioAF.clear()
>>> dictioAF
{}
```

ATTENTION : si l'on tente de supprimer dans le dictionnaire un couple dont la clé n'existe pas, Python renverra une erreur, quelque soit la méthode employée :

```
>>> dictioAF
{'table' : 'table', 'screen' : 'écran', 'sofa' : 'canapé'}
>>> dictioAF.pop("jardin") Traceback (most recent call last) :
....
KeyError : 'jardin'
>>> del dictioAF["jardin"]
Traceback (most recent call last) :
....
KeyError : 'jardin'
```

e - Concaténation

ATTENTION : contrairement aux listes, les dictionnaires ne supportent pas l'opérateur de concaténation "+". Une tentative renvoie un message d'erreur :

```
>>> dictioAF=dictioAF+{"screen" :u"écran","table" : "table","desk" : "bureau", "watch" : "montre"}
Traceback (most recent call last) :
.....
TypeError : unsupported operand type(s) for + : 'dict' and 'dict'
```

Deux syntaxes sont possibles pour la concaténation :

- une astuce basée sur l'opérateur ** qui permet normalement de passer plusieurs valeurs à une fonction dont les mots clés (les clés des dictionnaires ici) n'ont pas nécessairement été précisées :

```
>>> dictio1={"val1" :8, "val2" :16, "val3" :32, "val4" :64}
>>> dictio2={"val5" :128,"val6" :256}
>>> dictio3={**dictio1,**dictio2}
>>> dictio3
{'val1' : 8, 'val2' : 16, 'val3' : 32, 'val4' : 64, 'val5' : 128, 'val6' : 256}
```

On voit ci-dessus que cette méthode nécessite de définir un "conteneur" de type dictionnaire dans lequel on va implémenter tous les couples des deux dictionnaires à concaténer. Ce n'est donc pas vraiment une commande au sens strict.

- l'opérateur "officiel en python 3" (à partir de python 3.9) qui est `|` ; la syntaxe est :

```
>>> dictio4=dict1 | dict2
>>> dictio4
{'val1' : 8, 'val2' : 16, 'val3' : 32, 'val4' : 64, 'val5' : 128, 'val6' : 256}
```

Remarque I-3: MISE À JOUR DE VALEUR PAR CLÉS COMMUNES

Si les deux dictionnaires que l'on concatène possèdent des clés communes avec cependant des valeurs associées différentes, la mise à jour se fait sur le second dictionnaire itéré :

```
>>> dictio1={"val1" :8, "val2" :16, "val3" :32, "val4" :64}
>>> dictio2={"val4" :128,"val6" :256}
>>> dictio3={**dictio1,**dictio2}
>>> dictio3
{'val1' : 8, 'val2' : 16, 'val3' : 32, 'val4' : 128, 'val6' : 256} >>> dictio3={**dictio2,**dictio1}
>>> dictio3
{'val4' : 64, 'val6' : 256, 'val1' : 8, 'val2' : 16, 'val3' : 32}
```

f - Fusion de deux dictionnaires : `.update()`

Il est également possible de fusionner deux dictionnaires ; la syntaxe de cette commande est :

```
<dict1>.update(<dict2>)
```

Deux cas de figure peuvent se présenter :

- si les dictionnaires possèdent des clés toutes distinctes, alors l'opération de **fusion** va correspondre à une **concaténation** mais sans conservation du dictionnaire `<dict1>` qui est mis à jour, contrairement au dictionnaire `<dict2>` qui est conservé :

```
>>> d1={"A" :0,"B" :1}
>>> d2={"C" :2,"D" :3}
>>> d1.update(d2) >>> d1
{'A' : 0, 'B' : 1, 'C' : 2, 'D' : 3}
>>> d2
{'C' : 2, 'D' : 3}
```

- si les dictionnaires possèdent des clés en commun, alors l'opération de fusion va correspondre à une mise à jour de `<dict1>` (d'où le nom de la méthode "update") ; comme pour la concaténation, les valeurs des couples ayant des clés en commun sont mises à jour avec celle de `<dict2>` :

```
>>> d1={"A" :0,"B" :1}
>>> d2={"B" :2,"C" :3}
>>> d1.update(d2)
>>> d1
{'A' : 0, 'B' : 2, 'C' : 3}
```

I.3 Copie d'un dictionnaire

Supposons un dictionnaire `dict1` instancié et ses couples `<clé> : <valeur>` implémentés et tentons d'en réaliser une copie `dict2` par une simple commande d'affectation :

```
>>> dict1={0 : "A", 1 : "B", 2 : "C"}
>>> dict2=dict1
>>> dict2
{0 : "A", 1 : "B", 2 : "C"}
>>> dict1[3]="D"
>>> dict2
{0 : "A", 1 : "B", 2 : "C", 3 : "D"}
```

On constate que l'affectation ne crée pas, à l'instar des listes, une véritable copie du dictionnaire `dict1` sous la variable `dict2`, mais fait pointer la variable `dict2` vers la même table de hachage que `dict1`; il n'y a donc qu'un seul dictionnaire implémenté en mémoire.

Pour réaliser une copie *stricto sensu* d'un dictionnaire, il est nécessaire d'employer la méthode `<dict>.copy()` :

```
>>> dict1={0 : "A", 1 : "B", 2 : "C"}
>>> dict2=dict1.copy()
>>> dict1[3]="D"
>>> dict1
{0 : 'A', 1 : 'B', 2 : 'C', 3 : 'D'}
>>> dict2
{0 : 'A', 1 : 'B', 2 : 'C'}
```

I.4 Résumé des commandes essentielles (à connaître !)

Le tableau ci-dessous regroupe les principales commandes et méthodes applicables aux dictionnaires à connaître :

Commande	Rôle
<code><dict>.get(<clé>)</code>	Renvoie la valeur correspondant à la clé <code><clé></code>
<code><dict>.items()</code>	Renvoie l'ensemble des couples <code><clé>: <valeur></code> d'un dictionnaire
<code><dict>.keys()</code>	Renvoie l'ensemble des clés d'un dictionnaire
<code><dict>.values()</code>	Renvoie l'ensemble des valeurs d'un dictionnaire
<code><dict>.pop(<clé>)</code>	supprime le couple <code><clé>: <valeur></code> d'un dictionnaire et renvoie la valeur
<code>del <dict>[<clé>]</code>	même action que <code>.pop(<clé>)</code> sans renvoi de la valeur correspondant à <code><clé></code>
<code><dict>.clear()</code>	permet de vider le dictionnaire
<code><dict>.update(<dict2>)</code>	permet la mise à jour d'un dictionnaire/fusion de deux dictionnaires
<code><dict2>=<dict1>.copy()</code>	réalise une copie conforme du dictionnaire <code>dict1</code>

II Parcours des dictionnaires

II.1 Parcours par les clés

Comme pour les listes ou les tableaux, les dictionnaires sont des structures **itérables** ; on utilise pour les parcourir une instruction de boucle `for`. Mais comme un dictionnaire stocke des paires `{clé:valeur}`, plusieurs variantes sont possibles pour son parcours.

Par défaut, l'instruction `for` parcourt un dictionnaire par ses clés, et non pas ses valeurs :

```
>>> dictioAF={"screen" :u"écran","table" : "table","desk" : "bureau", "watch" : "montre"}
>>> for cle in dictioAF :
...     print(cle)
...
screen
table
desk
watch
```

On peut également spécifier explicitement que l'on souhaite parcourir le dictionnaire par ses clés à l'aide de la méthode `.keys()` :

```
>>> dictioAF={"screen" :u"écran","table" : "table","desk" : "bureau", "watch" : "montre"}
>>> for cle in dictioAF.keys() :
...     print(cle)
...
screen
table
desk
watch
```

II.2 Parcours d'un dictionnaire par ses valeurs

On peut également parcourir un dictionnaire en itérant sur ses valeurs. On utilise pour cela la méthode `.values()` :

```
>>> dictioAF={"screen" :u"écran","table" : "table","desk" : "bureau", "watch" : "montre"}
>>> for cle in dictioAF.values() :
... print(cle)
...
écran
table
bureau
montre
```

II.3 Parcours d'un dictionnaire par ses paires

On peut aussi parcourir un dictionnaire en itérant sur ses paires { clé:valeur} avec la méthode `.items()`. Cette dernière renvoie une collection de tuples, chacun constitué d'une clé et de sa valeur associée :

```
>>> dictioAF={"screen" :u"écran","table" : "table","desk" : "bureau", "watch" : "montre"}
>>> for paire in dictioAF.items() :
... print(paire)
...
('screen','écran')
('table','table')
('desk','bureau')
('watch','montre')
```