

- Un dictionnaire PYTHON est une collection d'objets « étiquetés ». On accède aux éléments de la collection *via* leurs étiquettes, qu'on appelle **clefs**, qui peuvent être n'importe quel objet PYTHON *non-mutable* (typiquement des nombres entiers ou décimaux — des nombres en virgule flottante, de type `float` — des chaînes de caractères, des tuples — *mais pas des listes* qui, elles, sont mutables!). Attention, il ne peut y avoir deux fois la même clef : **les diverses clefs doivent être toutes différentes!**

Exemple : pour créer une variable de type `dict` en déclarant son contenu, on fera quelque chose comme `dic = {"a": 97, (1, 0): ["tuple", 2, "éléments"], 3.14: "pi"}`.

Remarque : en bon français, on peut parler d'objets muables ou mutables, mais on ne dira que immuables et non immutables (comme souvent en sciences, l'anglais prend le pas et dans cette langue on dit `mutable object` ou `immutable object`).

- On accède à un élément d'un dictionnaire en spécifiant la clef correspondante (c'est semblable aux listes).
Exemple : avec la variable `dic` précédemment déclarée, `dic["a"]` renverra 97, `dic[(1, 0)]` renverra `["tuple", 2, "éléments"]`.
- On obtient la **longueur d'un dictionnaire** avec l'instruction `len(...)` : elle correspond à son nombre de clefs (qui doivent être toutes distinctes).

Exemple : avec la variable `dic` précédemment définie, `len(dic)` renverra la valeur 3.

- À une clef correspond une et une seule valeur. Mais, en fonction du type choisi pour telle ou telle valeur, plusieurs données peuvent être rassemblées en son sein. Ainsi, une liste PYTHON peut être utilisée comme valeur, bien qu'elle ne puisse servir de clef (un `tuple` peut être utilisé comme clef, lui).


Exemple : dans le dictionnaire `dic`, la clef `(1, 0)` est associée à la valeur `["tuple", 2, "éléments"]` (et cette valeur est elle-même une structure de données composée de trois éléments).

- Vérifier qu'un élément appartient à un dictionnaire implique le mot-clef `in`. La vérification d'appartenance porte sur les clefs!

Exemple : `'a' in dic` donnera `True`, car `'a'` est bien une clef de `dic`. Mais les instructions `'b' in dic`, `'A' in dic` ou `97 in dic` renverront toutes `False`.

- Un dictionnaire est également un objet **modifiable** (ou *mutable*, ou *muable*).

Exemple : après l'instruction `dic[(1, 0)] = ["tuple", "deux", "éléments"]`, la variable `dic` désignera le dictionnaire `{'a': 97, (1, 0): ['tuple', 'deux', 'éléments'], 3.14: 'pi'}`.

 Cela implique que lorsqu'une fonction a un dictionnaire comme paramètre, **il est possible de modifier le dictionnaire directement depuis le code de la fonction**. Si ça n'est pas ce que l'on souhaite, il faut alors **commencer par recopier le contenu du dictionnaire** (voir plus loin) dans une variable locale, déclarée dans le corps de la fonction.

- La création d'un dictionnaire se fait souvent par **accumulation** : on part d'un dictionnaire vide `dic = {}`, et on lui ajoute des éléments. Il y a deux possibilités :

- la méthode directe, par création d'une nouvelle clef **obligatoirement associée à sa valeur**. Après l'exécution de l'instruction `dic["b"] = 98`, la variable `dic` désignera :

```
{'a': 97, (1, 0): ['tuple', 'deux', 'éléments'], 3.14: 'pi', 'b': 98};
```

- l'actualisation du dictionnaire à partir d'un **autre dictionnaire**. Après exécution de l'instruction : `dic.update({3.14: "~pi", "A": 65})`, la variable `dic` désignera :

```
{'a': 97, (1, 0): ['tuple', 'deux', 'éléments'], 3.14: '~pi', 'b': 98, 'A': 65}.
```



Remarque : les clefs pré-existantes voient leurs valeurs associées se faire écraser lors du processus, alors que les paires « nouvelle clef – valeur » sont ajoutées au dictionnaire.

- Depuis PYTHON 3.7, l'ordre d'insertion des couples « clef : valeur » dans un dictionnaire est mémorisé et conservé. Mais la structure de données elle-même n'est pas ordonnée.

Exemple : les dictionnaires `d1 = {5: 'e', 10: 'j'}` et `d2 = {10: 'j', 5: 'e'}` sont égaux (la comparaison `d1 == d2` renverra `True`). Ce n'est pas le cas avec une liste : `[5, 10] == [10, 5]` renverra `False`.

- Il y a deux approches principales pour réaliser un parcours de dictionnaire PYTHON :

- le **parcours par clefs**, où l'on énumère l'ensemble des *clefs* à l'aide d'une boucle `for key in dic`. On obtient alors la *valeur* associée à la clef `key` avec l'instruction `valeur = dic[key]` ;
- le **parcours par valeurs**, où l'on énumère l'ensemble des *valeurs*. La boucle obtenue ressemble alors à `for val in dic.values()` : (on ne peut pas facilement retrouver la clef à partir d'une valeur).
- on peut parcourir un dictionnaire en ayant connaissance à la fois de la clef et de la valeur associée : `for key, val in dic.items()` : `print(key, "->", val)` (par exemple).

- ▶  **Copier un dictionnaire** est facilité par la méthode `copy()` des dictionnaires : `dico = dic.copy()`.
- ▶  **Cette technique n'est pas adaptée à la copie de dictionnaires dont les valeurs sont *mutables*** (un dictionnaire dont *au moins* une valeur serait une liste, par exemple). Ainsi, avec `dico` copié dans l'exemple précédent à partir de `dic`, on constate que `dic` est *aussi* modifié après l'instruction `dico[(1, 0)][0] = None`. Pour éviter cette situation, il faut (comme pour les listes de listes) utiliser la commande `deepcopy` du module `copy` : `import copy ; dico = copy.deepcopy(dic)`.
- ▶ **Quelques autres fonctionnalités des dictionnaires PYTHON**, qu'il peut être utile de connaître.
 - L'instruction `del dic[key]` supprime du dictionnaire la clef `key` **et** supprime donc aussi la valeur associée (clef et valeur sont définitivement perdues, car elles *ne* sont *pas* renvoyées).
 - L'instruction `dic.pop(key)` supprime du dictionnaire la clef `key` **mais** renvoie la valeur associée. La clef passée en paramètre est *indispensable* (car, contrairement à une liste, un dictionnaire n'est pas ordonné).
 - L'instruction `dic.popitem()` (*sans* paramètre) supprime du dictionnaire **et renvoie** sous forme de `tuple` le couple clef – valeur, en commençant par le *dernier* (en suivant l'ordre d'insertion des éléments dans le dictionnaire).
 - On obtient l'**ensemble des clefs** d'un dictionnaire avec l'instruction `dic.keys()`. Il peut être nécessaire de *transtyper* le résultat (par exemple `list(dic.keys())`). Plutôt que d'utiliser cette méthode et de devoir *transtyper*, on peut lui préférer une compréhension de liste : `[x for x in dic]`.
 - On obtient l'**ensemble des valeurs** d'un dictionnaire avec l'instruction `dic.values()` (cf. parcours d'un dictionnaire). Il peut être nécessaire de *transtyper* le résultat (exemple : `tuple(dic.values())`). Plutôt que d'utiliser cette méthode et de devoir *transtyper*, on peut lui préférer une compréhension de liste : `[dic[x] for x in dic]`.
 - PYTHON supporte la création de dictionnaires « *en compréhension* »... mais cela emmène vraiment assez loin, et il est peu probable que vous en ayez besoin!
- ▶ **Une caractéristique importante des dictionnaires PYTHON**, qu'il est utile de connaître : l'accès, la modification ou la suppression d'un élément d'un dictionnaire, *quel qu'il soit*, s'effectue *en temps constant*. Cela signifie que, quelle que soit la taille d'un dictionnaire (qu'elle soit de plusieurs centaines ou de plusieurs centaines de milliers d'entrées), récupérer ou modifier n'importe laquelle de ses valeurs (à partir de la clef correspondante) prendra quasiment le même temps (très court). On dit que la complexité en temps est en $O(1)$.

Le parcours d'un dictionnaire, lui, s'effectue en $O(n)$, ce qui signifie que le temps de parcours est proportionnel à n , qui désigne par convention la taille du dictionnaire (c'est-à-dire son nombre de clefs).

Le principe à l'œuvre en coulisses des dictionnaires PYTHON, est celui des tables de hachage (ou hashtable en anglais). Chaque clé est transformée par une fonction de hachage en une valeur qui « repère » les valeurs associées. La fonction de hachage ne peut être appliquée qu'à des objets *immuables*.