02 - Les dictionnaires

1 Définition

Un dictionnaire est un ensemble d'éléments (comme les listes, les chaines de caractères, les tuples), dont chaque élément se compose d'une association `clé : valeur`.

Les dictionnaires en Python sont aussi appelés aussi *tableaux associatifs* ou *table de hachage*.

- Les dictionnaires sont des objets mutables : on peut ajouter, supprimer ou modifier le contenu (rappel : les listes sont mutables, les tuples et les chaines de caractères ne le sont pas)
- Les dictionnaires ne sont pas des séquences : on ne peut pas accéder à leur contenu via un indice.
- L'ordre des éléments dans un dictionnaire n'a pas d'importance.
- Les clés (keys) peuvent être de type *str, int, float, tuple de nombres, tuple de tuple de nombres*... mais **pas de type liste** (un tuple de listes n'est pas permis non plus).
- Les valeurs (values) peuvent être de n'importe quel type.
- Le type Python d'un dictionnaire est le type `dict`.

2 Création d'un dictionnaire

- Création d'un dictionnaire vide : `D`$= \{\}$
- Ajout au fur et à mesure des clés et des valeurs : `D[clé]=valeur`
- Création en une seule fois : `D`$=\{$`clé1:valeur1,clé2:valeur2,...`$\}$

Pour récupérer la value associée à une clé, la syntaxe est semblable à celle des listes (où on accède à un élément grâce à son indice) : l'instruction `v=D[clé]` permet d'obtenir la valeur du dictionnaire correspondante à la clé. Si la clé n'existe pas, une erreur arrête le programme.

Exemple 1 : a) Ecrire une instruction ou une suite d'instruction permettant de créer un dictionnaire nommé `Renseignements`, dont les clés sont les chaines de caractères `'Nom'`, `'Prénom'`, `'Age'`, et dont les valeurs correspondantes sont vos données personnelles.

b) Rajouter une entrée dans ce dictionnaire, correspondant à une clé appelée `'Classe'` (la valeur correspondante étant votre classe bien sûr...)

3 Accès à l'ensemble des clés et des valeurs, parcours d'un dictionnaire

- La méthode `keys` donne accès à l'ensemble des clés.

  Si `D` est un dictionnaire, alors `D.keys()` est l'ensemble des clés de `D`.

  Attention le résultat n'est pas une liste (mais un type `dict_keys`), cependant le parcours de cet ensemble est possible grâce à une boucle `for x in D.keys():`

- La méthode `values` donne accès à l'ensemble des valeurs.

  Si `D` est un dictionnaire, alors `D.values()` est l'ensemble des valeurs de `D`.

  Attention le résultat n'est pas une liste (mais un type `dict_values`), cependant le parcours de cet ensemble est possible grâce à une boucle `for x in D.values():`
- Enfin la méthode `items` donne accès à l'ensemble des couples (clé,valeur).

  L'instruction `for clé,valeur in D.items()` parcourt la liste des (clés, valeurs)

***Remarque* 1 :** Pour parcourir la liste des clés, on peut utiliser la syntaxe simplifiée

`for clé in D:` *(à la place de* `for clé in D.keys():` *)*

Exemple 2 : Ecrire une fonction `appartient`, ayant pour paramètres un dictionnaire et une valeur, qui retourne :

- `False` si la valeur n'est pas une valeur référencée par le dictionnaire.
- une clé correspondant à cette valeur si cette valeur est référencée dans ce dictionnaire.

4 Fonctions - opérations - méthodes

- Les instructions `in` ou `not in` permettent de tester **l'appartenance d'une clé à un dictionnaire** (mais pas d'une valeur) : `x in D` renvoie `True` si x est une clé de D, `False` sinon.
- Le contenu d'un dictionnaire peut être modifié en remplaçant la valeur associée à une clé par une autre valeur. On utilise pour cela la syntaxe d'affectation : `D[clé]=nouvelleValeur`. (Si `clé` existe, sa valeur sera modifiée en `nouvelleValeur`, sinon une nouvelle entrée est créée dans le dictionnaire.)
- La fonction `len(D)` renvoie la longueur d'un dictionnaire (i.e. le nombre de clés qu'il contient).

5 Copie d'un dictionnaire

Les comportements sont similaires à ceux rencontrés avec les listes.
Si on écrit `dico1=D`, la modification de `dico1` entrainera la modification de `D` car c'est un objet mutable comme les listes.
Si on veut créer une copie indépendante(*) il faut écrire : `dico2=D.copy()`

(*) : *pas tout à fait indépendante en réalité, car on a un problème si les valeurs sont des listes de listes par exemple, il faudra alors effectuer une copie profonde grâce à la fonction* `deepcopy` *du module* `copy`.
Alors on pourra modifier les deux dictionnaires complètement indépendamment l'un de l'autre.

Exercice 1 (*Points au Scrabble*) : On définit le dictionnaire suivant :

```
1 scrabble={"A":1,"B":3,"C":3,"D":2,"E":1,"F":4,"G":2,"H":4,"I":1,"J":8,"K":10,"L":1,"M":2,"N":1,"O":1,"P":3,"Q":8,"R":1,"S
      ":1,"T":1,"U":1,"V":4,"W":10,"X":10,"Y":10,"Z":10}
```

Ce dictionnaire définit les points attribués à chaque lettre lorsque l'on joue au Scrabble.
On suppose que la variable `scrabble` est une variable **globale**.
Ecrire une fonction `points(mot)` qui prend en paramètre une chaîne de caractère composée de lettres majuscules, et qui renvoie un entier égal au nombre de points que vaut ce mot au Scrabble.

Exercice 2 (*Nombres d'occurrences*) : Écrire une fonction `occurrences(L)` qui prend en paramètre une liste, et qui renvoie un dictionnaire dont les clés sont les éléments de la liste et les valeurs le nombre de fois que l'élément apparait. *Exemple* : `occurrences([7,3,20,1,1,7,20,1])` peut retourner le dictionnaire $\{7:2, 3:1, 20:2, 1:3\}$.

6 Les clés d'un dictionnaire doivent être hachables

Le langage Python impose une restriction sur les clés d'un dictionnaire : certains types de données ne sont pas autorisés. Il n'est par exemple pas possible d'utiliser le type list :

```
In [12]: D={[1495,1598]:'Renaissance'}
Traceback (most recent call last):

  File "<ipython-input-12-efed0f95b809>", line 1, in <module>
    D={[1495,1598]:'Renaissance'}

TypeError: unhashable type: 'list'
```

mais il est en revanche possible d'utiliser le type tuple :

```
In [13]: D={(1495,1598):'Renaissance'}

In [14]: print(D)
{(1495, 1598): 'Renaissance'}
```

Pourquoi cette limitation à certains types autorisés ?

Pour que la manipulation d'un dictionnaire soit efficace, il faut accéder rapidement à la valeur associée à une clé.
Lors de l'évaluation de `D[clé]`, Python ne fait pas un parcours systématique de toutes les clés du dictionnaire `D` jusqu'à trouver celle correspondant à la valeur de `D[clé]`. En effet une telle méthode serait de complexité linéaire (i.e. en $O(n)$, où n est le nombre de clés du dictionnaire). Or il suffirait que cet appel soit à l'intérieur d'une boucle, et l'on obtiendrait une complexité quadratique, ce qui n'est pas très efficace.

Voici ce que fait Python pour retrouver rapidement la valeur associée à une clé d'un dictionnaire : il utilise d'abord une *fonction de hachage*, `hash`. Cette fonction associe à un objet Python un entier de type int, qui est appelé "signature" ou "empreinte" de l'objet de départ. L'objet est alors dit *hachable* :

```
In [8]: hash(1495) # pour un entier assez petit, hash est la fonction identité
Out[8]: 1495

In [9]: hash ([1495,1598]) # incorrect (le type list n'est pas hachable)
Traceback (most recent call last):

  File "<ipython-input-9-2a2203e26423>", line 1, in <module>
    hash ([1495,1598]) # incorrect (le type list n'est pas hachable)

TypeError: unhashable type: 'list'
```

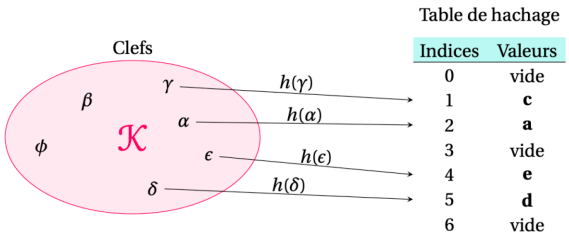
```
In [10]: hash('bonjour') #la fonction hash est aussi définie pour des chaines
Out[10]: 3597498019684976123

In [11]: hash(2/3) # ou des flottants
Out[11]: 1537228672809129216
```

Les types hachables que nous serons amenés à manipuler sont les types simples (int, float, bool, str, ..) , et les tuples formés d'éléments de type eux-mêmes hachables.

Plus généralement, les types hachables sont les types non mutables dont tous les constituants (pour les types composés) sont également hachables.
Ainsi, une liste n'est pas hachable (car mutable), mais un tuple formé d'entiers est hachable.

L'intérêt de la fonction de hachage est qu'elle permet d'associer à chaque clef un entier, qui aura le même rôle que l'indice dans une liste. On pourra accéder à la valeur correspondante à une clé, donc à un indice, en temps constant (donc sans parcourir tout le dictionnaire).



A retenir : , **La recherche d'une clé dans un dictionnaire se fait en temps constant** (c'est-à-dire avec une complexité $O(1)$, indépendante de la taille du dictionnaire).

Remarque : Il n'y a en revanche aucune contrainte sur les valeurs d'un dictionnaire, qui peuvent être de type quelconque.

7 Mémoïsation

Une application très utile des dictionnaires est la mémoïsation d'une fonction.

Définition 1
La mémoïsation d'une fonction consiste à mettre en place (en général à l'aide d'un dictionnaire) un système qui mémorise les arguments d'entrée (clé) et le résultat de la fonction (valeur) lors de chaque appel de la fonction, et évite ainsi de refaire un même calcul lorsque la fonction est appelée plusieurs fois avec les mêmes arguments.

Comme nous allons le voir, la mémoïsation est particulièrement efficace pour les fonctions récursives

---

Exercice 3 : Considérons la suite de Fibonacci $(u_n)_{n\in\mathbb{N}}$ définie par $u_0 = 0, u_1 = 1$ et la relation de récurrence $\forall n \geq 2, \quad u_n = u_{n-1} + u_{n-2}$.
Implémenter le calcul de u_n à l'aide d'une fonction récursive "naïve", que vous appellerez `fib`.

Avec cette implémentation, l'exécution de `fib(50)` prend plus d'une heure !
La raison est simple : l'appel `fib(50)` provoque un appel à `fib(49)` et `fib(48)`, mais `fib(49)` produit de son côté un appel indépendant à `fib(48)` et `fib(47)`, de sorte que `fib(48)` est calculé 2 fois. Et la situation empire pour les arguments inférieurs : `fib(47)` est appelé 3 fois, `fib(46)` 5 fois, .. et `fib(1)` est appelé plus de 12 milliards de fois !

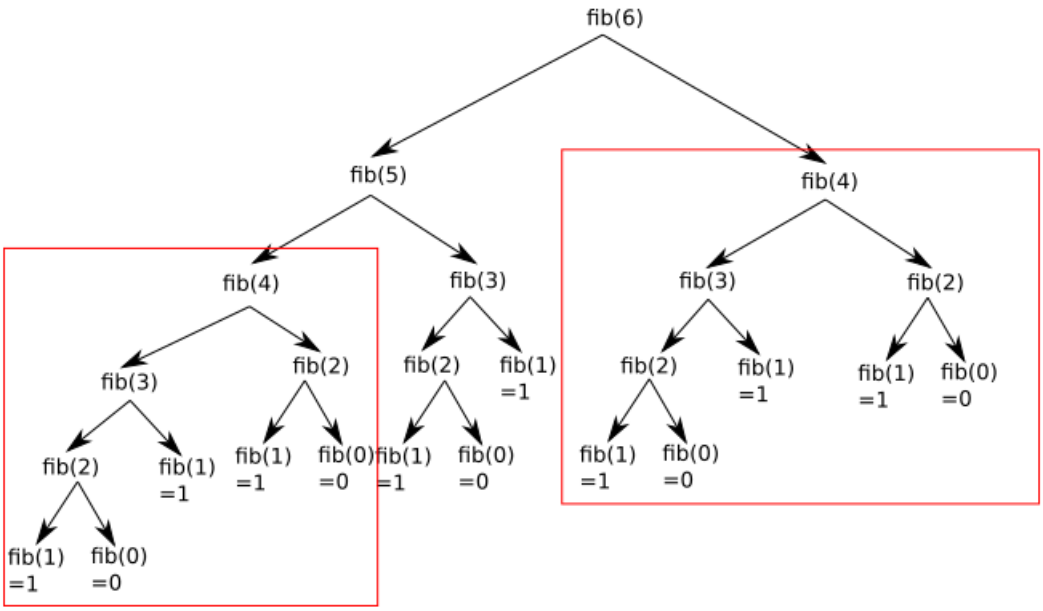


FIGURE 1 – Illustration de l'appel de fib(6)

Évidemment, l'implémentation que nous avons choisie est particulièrement inefficace, à cause du double appel récursif, et on pourrait facilement éviter cet écueil en utilisant une fonction non récursive.

Mais comme nous allons le voir, on peut également conserver l'idée initiale d'une fonction récursive, et ajouter simplement une étape de mémoïsation pour éviter les recalculs inutiles :

```
D_fib={0:0,1:1}  #le dictionnaire qui nous servira à stocker les valeurs de la suite

def fib_mem(n):
    if n in D_fib: #si n est déjà une clé, on a sa valeur
        return D_fib[n]  #ce cas inclut les cas d'arrêts de la ré cursivit é

    else:  #si n n'est pas déjà une clé du dictionnaire
        D_fib[n]=fib_mem(n-1)+fib_mem(n-2) #on calcule le terme de la suite
                           # et on stocke la valeur dans le dictionnaire
        return D_fib[n]
```

Avec cette implémentation, l'évaluation de `fib(50)` est instantanée.

Le principe de la mémoïsation apparaît clairement sur cet exemple : on créée (en variable globale) un dictionnaire `D_fib` qui va stocker, au fur et à mesure des appels récursifs, les valeurs déjà calculées de la fonction.

La mémoïsation permet de diminuer la complexité en temps d'un algorithme. Dans l'exemple précédent, on peut montrer que l'on passe d'une complexité $O(\varphi^n)$ à $O(n)$, où $\varphi = \dfrac{1+\sqrt{5}}{2}$ est le nombre d'or, ce qui est un gain considérable (on passe d'une complexité exponentielle à une complexité linéaire).

Il faut cependant noter que ce gain de complexité temporelle se fait généralement au prix d'une complexité en mémoire supérieure, en raison du stockage des valeurs calculées dans un dictionnaire.

Variante : Une autre façon de faire est d'utiliser le dictionnaire de stockage comme une variable locale à la fonction :

```
def fib_memv2(n,D_local={0:0,1:1}): #D_local est un paramètre optionnel,
                                    #s'il n'est pas specifie il est initialise à {0:0,1:1} lors du premier appel de la fonction
    if n in D_local: #si n est déjà une clé, on a déjà sa valeur
        return D_local[n]  #ce cas inclut les cas d'arrêts de la recursivite

    else:  #si n n'est pas déjà une clé du dictionnaire
        D_local[n]=fib_memv2(n-1)+fib_memv2(n-2) #on calcule le terme de la suite
                           # et on stocke la valeur dans le dictionnaire
        return D_local[n]

print(fib_memv2(50))
    12586269025
```

Remarque : Il n'y a pas de grandes différences entre l'utilisation d'un dictionnaire en variable globale ou en variable locale. Faites comme vous préférez (parfois le sujet ne vous laisse pas le choix, cf Mines-Ponts 2023).

A retenir : Schéma général d'une fonction récursive avec mémoïsation :

```
D={les clés et valeurs initiales}        #le dictionnaire pour stocker les résultats de la fonction
def mafonction(arg):
    if arg in D:                         #si arg est une clé d'une dictionnaire, on connaît déjà sa valeur associée
        return D[arg]                    #ce qui inclut le cas initial (cas d'arrêt)
    else:
        ... #le code la fonction résursive produisant un résultat res
        D[arg]=res                       #ATTENTION avant de retourner le résultat on le stocke dans le dictionnaire
        return res
```

Remarque : Rien n'empêche d'utiliser la mémoïsation pour une fonction à plusieurs paramètres, tant que ceux-ci restent de type hachable. Les clés du dictionnaire de mémoïsation sont alors des tuples qui rassemblent les paramètres de la fonction. L'exercice 4 ci-dessous illustre ce principe :

Exercice 4 : Ecrire une fonction récursive, utilisant le principe de mémoïsation, qui permet le calcul du coefficient binomial $\binom{n}{p}$ par la formule de Pascal.

Exercice 5 : Pour tout entier naturel non nul p, on définit la suite de Syracuse associée à p par

$$u_0 = p \text{ et } \forall n \in \mathbb{N}, u_{n+1} = \begin{cases} \frac{u_n}{2} \text{ si } u_n \text{ est pair} \\ 3u_n + 1 \text{ sinon} \end{cases}$$

1. Calculer à la main les premiers termes de la suite pour $p = 1$, puis $p = 2$, puis $p = 3$, puis $p = 4$. Que constatez-vous ?

On observe que quelle que soit la valeur de p, la suite associée finit par boucler sur $1, 4, 2$. Cette conjecture, non démontrée à ce jour (un prix de 1 million de dollars vous sera offert par une société japonaise si vous la démontrez !), a été vérifiée jusqu'à des valeurs de $p \approx 6.10^{18}$.

Pour chaque valeur de p on appelle *temps de vol de la suite associée* la plus petite valeur de n pour laquelle $u_n = 1$.

2. Quel est le temps de vol pour $p = 1$? Pour $p = 2$? Pour $p = 3$? Pour $p = 4$?
3. Programmer une fonction itérative "simple" `vol(p)` qui détermine le temps de vol de la suite associée à p.
4. Programmer une fonction récursive "simple" `vol_rec(p)` qui détermine le temps de vol de la suite associée à p.
5. Quel est le plus grand temps de vol pour $1 \leq p \leq 1000$?

Si on cherche à déterminer le plus grand temps de vol pour $1 \leq p \leq 10^7$, le programme ne s'arrête plus. Ceci est dû au fait qu'énormément de calculs identiques sont fait plusieurs fois (si en calculant les valeurs successives de la suite, on tombe sur un nombre dont on connaît le temps de vol, pas la peine de le recalculer !)
Nous allons donc améliorer la version récursive de la question 2, en créant un dictionnaire global permettant de stocker les temps de vol associés aux différentes valeurs de p :

6. Programmer une fonction récursive avec memoïsation `vol_mem(p)` qui détermine le temps de vol de la suite associée à p.
7. A l'aide de cette fonction, déterminer le plus grand temps de vol pour $1 \leq p \leq 10^7$,
8. Combien d'entiers $p \leq 10^6$ vérifient $vol(p) \leq 30$?