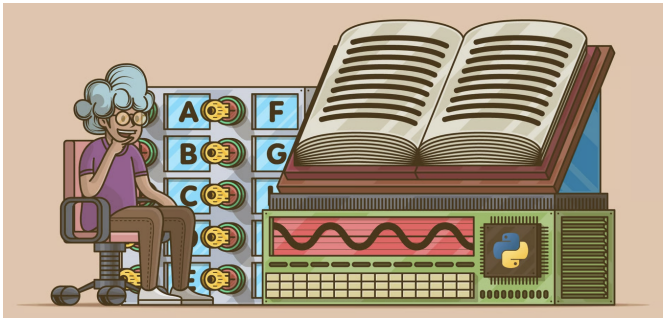
TP INFO 7 | Dictionnaires

Cette séance est dédiée à l'étude des dictionnaires de Python, structure permettant des associations clés-valeurs plus générales que celles des listes.



1. Introduction à la structure de dictionnaire

Commençons par un exemple. Pour déterminer les occurrences des différents caractères apparaissant dans une chaîne, il suffit de la parcourir une seule fois et d'alimenter « un tableau » de compteurs. Pour "anticonstitutionnellement", on obtient :

"a"	"c"	"e"	"i"	"l"	"m"	"n"	"o"	"s"	"t"	"u"
1	1	3	3	2	1	5	2	1	5	1

On voit ici qu'il est commode de remplacer des indices abstraits par les caractères « a », « c », etc. La structure de dictionnaire permet celle généralisation des listes : les indices entiers sont remplacés par des valeurs d'un type arbitraire (mais immuable).

Définition 7.0. Dictionnaire

Un dictionnaire (aussi appelé tableau associatif ou encore table d'association) associe à un ensemble de clés un ensemble correspondant de valeurs. Chaque clé est associée à une valeur : un tableau associatif correspond donc à une application de l'ensemble des clés dans l'ensemble des valeurs.

L'implémentation des dictionnaires sous Python sera étudiée en seconde année (utilisation de tables de hachage). Ils sont utilisés dans les systèmes de fichiers ou pour gérer la table des symboles des compilateurs durant l'analyse lexicale.

Ils s'écrivent en Python sous la forme d'une énumération d'associations `cle:valeur` délimitée par des accolades. Le dictionnaire associé à notre exemple ci-dessus s'écrit :

```
{"a":1, "c":1, "e":3, "i":3, "l":2, "m":1, "n":5, "o":2, "s":1, "t":5, "u":1}
```

On retiendra les opérations suivantes :

Action	Code
Création du dictionnaire vide	`dico={}`
Ajout ou modification d'une association clef-valeur	`dico[clef]=valeur`
Suppression d'une association clef-valeur	`del dico[cle]`
Existence d'une clef	`clef in dico` (renvoie un booléen)
Longueur du dictionnaire (nombres de clefs)	`len(dico)`
Copie d'un dictionaire	`d=dico.copy()`

```
>>> dico={}
>>> dico["open"]="ouvert"
>>> dico["close"]="fermé"
```

```
>>> dico
{'close': 'fermé', 'open': 'ouvert'}
>>> 'open' in dico
True
```

Clés d'un dictionnaire

Les clés peuvent être de type hétérogène mais immuable (i.e. non modifiable) : on ne peut utiliser ni listes, ni dictionnaires.

```
>>> d={}
>>> d["a"]=1
>>> d[11]=-2
```

```
>>> d["abc"]=3
>>> d
"a":1, 11:-2, "abc:3"
```

Dans le cas où l'on souhaite utiliser des clés de la forme $(x_1,\dots,x_n)$ où les x_i sont de type numérique, on optera pour la structure de tuple qui est analogue à celle de liste mais non modifiable.

Les tuples sont définis comme des listes mais sans crochet. On peut les délimiter au moyen de parenthèses mais celles-ci sont optionnelles. Cette structure est utile pour représenter les indices dans une matrice ou encore les coordonnées dans le plan.

```
>>> d={}
>>> d[1,2]=1
>>> d[2,1]=-1
>>> d
(1,2):1,(2,1):-1
>>> d[(1,2)]=4
>>> d
(1,2):4 ,(2,1):-1
```

On peut itérer sur les clefs, les valeurs, les deux à la fois ou le dictionnaire :

```
>>> for c in dico.keys():print(c)
close
open
>>> for val in dico.values():print(v)
fermé
ouvert
```

```
>>> for c,v in dico.items():print(c,v)
close, fermé
open, ouvert
>>> for x in dico:print(x)
close
open
```

Comme dans le cas des listes, on utilise une syntaxe spécifique pour la copie d'un dictionnaire :

```
>>> d={}
>>> d["a"]=1
>>> d["b"]=-2
>>> dic=d
```

```
>>> dic["a"]=3
>>> d
"a":3, "b":-2
>>> dic=d.copy()
```

```
>>> dic["a"]=5
>>> d
"a":3, "b":-2
```

Comme pour les listes, la présence de valeurs mutables nécessitera l'utilisation de `deepcopy`.

Dictionnaires Versus listes

Nous verrons dans la suite du cours d'informatique que l'implémentation des dictionnaires rend plus rapide certaines opérations par rapport aux listes : par exemple, l'expression booléenne `x in t` nécessite un temps d'exécution plus long que `cle in dico` pour une liste `t` et un dictionnaire `dico` de même longueur.

2. Exemples d'utilisation des dictionnaires

Nous ouvrirons ce TP avec deux applications très classiques de la structure de dictionnaire.

2.1. Calcul des fréquences dans une liste ou une chaîne

Nous revenons à présent à l'exemple introductif de ce TP.

Nous allons écrire une fonction `frequences` prenant en argument une chaîne de caractères `s` et renvoyant sous la forme d'un dictionnaire le nombre d'occurrence(s) des caractères de `s` (les clés étant les caractères). Le principe est clair : on initialise le dictionnaire au vide puis on parcourt linéairement `s` en le mettant à jour à chaque itération (si la clé existe déjà, on ajoute 1 à la valeur correspondante et sinon on crée la clé initialisée à la valeur 1).

```
def frequences(s):
    dico={}
    for e in s:
        if e in dico:
            dico[e]=dico[e]+1
        else:
            dico[e]=1
    return dico
```

2.2. Mémoïsation d'une fonction récursive

Afin d'éviter d'appeler une fonction récursive sur des valeurs déjà calculées, une stratégie consiste à conserver en mémoire celles-ci au moyen d'un dictionnaire. Par exemple, dans la cas de la suite de Fibonacci $(\phi_n)_{n\in\mathbb{N}}$, on choisira un dictionnaire dont les clés sont des entiers naturels et les valeurs les ϕ_n. Le principe est le suivant : on utilise un dictionnaire comme variable en plus de n et les appels récursifs ne sont faits que si la clé n'existe pas encore dans le dictionnaire.

On en déduit une fonction récursive `fibomem` prenant en argument un entier `n` et un dictionnaire `f` tel que l'appel `fibomem(n,{0:0,1:1})` renvoie ϕ_n :

```
def fibomem(n,f):
    if n in f:
        return f[n]
    else:
        f[n]=fibomem(n-2,f)+fibomem(n-1,f)
        return f[n]
```

On commence par tester si ϕ_n a déjà été calculé au moyen du dictionnaire. En voici une variante :

```
def fibomem(n,f):
    if n not in f:
        f[n]=fibomem(n-2,f)+fibomem(n-1,f)
    return f[n]
```

3. Exercices

1 — *Représentation des polynômes creux*

On peut représenter un polynôme au moyen de la liste de ses coefficients. Par exemple, le polynôme

$$P = \sum_{k=0}^{n} a_k X^k \quad \text{peut être codé par } [a_0, a_1, \ldots, a_n]$$

Dans le cas de polynômes ayant de nombreux coefficients nuls, cette représentation n'est pas judicieuse et on préfère utiliser un dictionnaire aux entrées de la forme *degré : coefficient*.

Ainsi $1+2X+3X^{100}$ sera représenté avantageusement par le dictionnaire `{0:1,1:2,100:3}`. Le polynôme nul est représenté par le dictionnaire vide `{}`.

1. Écrire une fonction `somme` prenant en argument des polynômes P et Q et renvoyant P + Q.
2. Écrire une fonction `prodElem` prenant en argument un polynôme P, un entier naturel k, un réel a et renvoyant le polynôme $aX^kP(X)$.
3. Écrire une fonction `prod(P,Q)` prenant en argument des polynômes P et Q et renvoyant PQ.

2 — *Doublons* f

Écrire une fonction `doublons` prenant en entrée une liste d'entiers `t` et renvoyant une liste contenant sans répétition tous les éléments de `t` apparaissant au moins deux fois.

3 — *Un exemple de mémoïsation* f

Écrire une fonction récursive `binome` prenant en argument des entiers naturels k, n et un dictionnaire de mémoïsation `d` qui renvoie $\binom{n}{k}$ en utilisant la relation de Pascal.

Les clés du dictionnaire seront les tuples `(n,k)` et l'appel `binome(n,k,{})` doit renvoyer $\binom{n}{k}$.

4 — *Le code de César* f

Le code de César est le plus rudimentaire que l'on puisse imaginer. Il a été utilisé par Jules César pour certaines de ses correspondances. Afin de simplifier le traitement, on ne considérera que des messages uniquement constitués de lettres majuscules non accentuées et sans espace, stockés dans une chaîne de caractères.

Le principe est de décaler *circulairement* les lettres de l'alphabet vers la droite d'un certain nombre d de lettres appartenant à $[\![0,25]\!]$, appelé clé du code. Par exemple, en choisissant $d = 1$, le caractère `A` se transforme en `B`, le `B` en `C`, ..., et le `Z` en `A`. Le message `"LOUISLEGRAND"` est donc codé par `"MPVJTMFHSBOE"`.

On commence par définir l'alphabet sous la forme d'une chaîne de caractères : `alpha="ABC ... Z"` (il faudra l'initialiser en entier avant de traiter les questions suivantes).

1. Créer un dictionnaire `num` dont les clés sont les 26 majuscules de l'alphabet et les valeurs leur position dans l'ordre alphabétique en commençant à 0. Par exemple, on aura les entrées `"A":0` et `"Z":25`.
2. Écrire une fonction `codage` prenant en argument une chaîne de caractères `s` et un décalage d, et qui renvoie le message `M` décalé de d lettres.
3. Donner le codage de "ONLYGODFORGIVES" pour un décalage de 12 lettres.
4. Écrire une procédure `decodage` prenant les mêmes arguments et mais qui réalise le décodage d'un message `s`.

La lettre E est en moyenne la plus fréquente dans un texte « *suffisamment long* » en français. Pour décoder un message crypté `s`, il suffit donc de déterminer la lettre la plus fréquente dans `s`. Cette approche ne sera bien-sûr valable qu'en moyenne.

5. Écrire une fonction `cle` qui prend en argument un message codé `s` et qui renvoie la valeur de la clé utilisée en supposant que la lettre la plus fréquente est le E.
6. Écrire une fonction `cassage` qui prend en argument un message `s` et qui renvoie le message décodé selon la même hypothèse.
7. Décoder `SLYVTHUZWSLUKLBYZLATPZLYLZKLZJVBYAPZHULZLZASHZBPALKLZPSSBZPVUZWLYKBLZ`.

5 — *Plus longue sous-chaîne commune* ***ff***

Soit A $= a_0 \cdots a_n$ une chaîne de caractères. On définit une sous-chaîne de A comme étant une chaîne de la forme $a_{i_1} a_{i_2} \cdots a_{i_k}$ avec $0 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$ (non nécessairement consécutifs). On considère deux mots (sur un alphabet quelconque, non nécessairement fini) A $= a_0 \cdots a_n$ et B $= b_0 \cdots b_m$. On recherche alors les sous-chaînes communes à A et B, et plus particulièrement la (ou l'une des) plus longue. Par exemple, la plus longue sous-chaîne commune à A $= aabaababaa$ et B $= ababaaabb$ est $ababaaa$. On généralise le problème à tous les préfixes de A et de B, et on note $p(i,j)$ la longueur de la plus longue sous-chaîne commune (PLSCC) à $a_0 \cdots a_i$ et $b_0 \cdots b_j$.

1. Démontrer que, pour i et j dans un ensemble que l'on précisera :

$$p(i,j) = \begin{cases} 1 + p(i-1, j-1) & \text{si } a_i = b_j \\ \max\big(p(i, j-1), p(i-1, j)\big) & \text{sinon} \end{cases}$$

2. En déduire une fonction `lgM` prenant en argument des chaînes de caractères A et B ainsi que deux indices valides i et j et qui renvoie $p(i,j)$. On pourra ajouter aux quatre arguments un dictionnaire de mémoïsation.

6 *Les nombres chanceux* ***fff***

Les nombres chanceux sont générés à partir d'un crible analogue à celui d'Eratosthène. On considère la suite des entiers naturels non nuls

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Puis on enlève un nombre sur deux, ce qui ne laisse que les entiers impairs :

1 3 5 7 9 11 13 15 17 19 21 23 25

Le deuxième terme de la suite est désormais 3. Ensuite, on enlève un nombre sur trois parmi ceux qui restent dans la liste :

1 3 7 9 13 15 19 21 25

Le troisième nombre survivant est 7. On enlève alors un nombre sur sept parmi ceux qui restent dans la liste :

1 3 7 9 13 15 21 25

Les nombres chanceux sont ceux qui restent en appliquant à l'infini ce procédé :

1, 3, 7, 9, 13, 15, 21, 25, 31, 33, 37, 43, 49, 51, 63, 67, 69, ⋯

Écrire une fonction `chanceux` prenant en argument un entier naturel n non nul et renvoyant la liste des nombres chanceux inférieur à n.

4. Indications

1

Pour calculer P + Q, on peut partir d'une variable S initialisée à P et, pour chaque monôme de Q, on teste si un monôme de même degré apparaît dans S et on modifie en conséquence S en ajoutant ce monôme.

2

On pourra, par exemple, utiliser un dictionnaire pour compter les occurrences d'un entier.

3

On ajoute aux cas d'initialisation déjà vus dans le TP 4 (sur les fonctions récursives) le cas où la clé `(n,k)` existe déjà dans `d`.

4

Décoder revient à coder en sens inverse.

5

Un cas de base naturel est $i = 0$ ou $j = 0$.

6

Attention : si on supprime des termes dans une liste, les indices se décalent...