Chap. 3 Tableaux Thème 2

Retour sur les tableaux

Sommaire du chapitre 3

On rappelle qu'un tableau est une collection ordonnée d'éléments de même type auxquels on peut accéder par leur indice. Supposons que l'on veuille lister la température à midi de chaque jour de l'année. On pourrait définir 366 variables : `temp1 = 14.3`, `temp2 = 25.1`, ..., `temp366 = 23.9` mais ce ne serait pas pratique. On va donc plutôt regrouper ces variables dans un tableau `temperatures = [14.3, 25,1, ..., 23.9]`.

I Construire et parcourir un tableau

I.a Construction

Nous avons vu au chapitre précédent trois manières de construire un tableau :

```
1 t1 = [1, 4, 3, 7] # lister explicitement les valeurs
2 t2 = [0] * 3      # [0, 0, 0] : répéter un tableau
3 t3 = t1 + t2      # [1, 4, 3, 7, 0, 0, 0] : concaténer des tableaux
```

Remarques

On rappelle aussi qu'un tableau est une valeur muable : on peut modifier son contenu :

```
1 t1 = [0, 0]
2 t2 = t1
3 t1[0] = 1
4 # t1 = [1, 0]
5 # t2 = [1, 0]
6 t1 = [2, 4]
7 # t1 = [2, 4]
8 # t2 = [1, 0]
```

- Ligne 3 : on *modifie* `t1`, donc aussi `t2` puisque les deux référencent le même tableau.
- Ligne 6 : on *ne modifie pas* le tableau mais affecte une nouvelle valeur à `t1`. Ceci redirige la référence `t1` vers un nouveau tableau, mais `t2`, lui, référence toujours le tableau de départ et donc ne change pas.

I.b Parcours

Nous avons vu aussi qu'il existe deux approches pour parcourir/énumérer les éléments d'un tableau :

```
1 for x in t1 :
2     print(x)   # affiche 1, puis 4, puis 3, puis 7
3 for i in range(len(t1)) :
4     t1[i] = t1[i] + 2   # modifie t1 qui vaut [3, 6, 5, 9]
```

Remarques

Dans le second cas, on accède aux éléments du tableau par leur indice. Dans le premier cas, on parcourt directement les éléments du tableau, mais alors on n'a pas accès à leur indice et il n'est pas facile de modifier le tableau (`x` est une référence). Enfin, Python autorise des indices négatifs pour faciliter le parcours d'un tableau en partant de la fin : `t[-1]` renvoie ainsi le dernier élément du tableau, `t[-2]` l'avant dernier, etc. C'est parfois pratique mais cela augmente le risque d'erreur sur les indices puisque l'on peut en Python utiliser sur un tableau de taille n tous les indices i tels que $-n \leq i < n$. Seuls les indices hors de cet intervalle renvoient une erreur `IndexError : list index out of range`.

```
1 t = ["érable", "chêne", "charme", "hêtre"]
2 print(t[-1])   # hêtre
```

II Parcourir des valeurs itérables : list, string, ...

De nombreux types de données peuvent être parcourus avec la syntaxe `for x in t :`, par exemple les tableaux, les chaînes de caractères, les tuples, les dictionnaires, et même les fichers ainsi que la suite de valeurs retournée par `range()`. On dit alors que la valeur `t` est **itérable**.

```
t = ["érable", "chêne", "charme", "hêtre"]
s = 'Aux âmes bien nées'
tup = ('Corneille', 1606, 1684)

for x in t : ...      # x vaut successivement : 'érable', 'chêne', ...
for x in s : ...      # x vaut successivement : 'A', 'u', ...
for x in tup : ...    # x vaut successivement : 'Corneille', 1606, 1684
```

Exemples

Cette construction qui itère sur (parcourt) les éléments d'un tableau peut être utilisée pour calculer sa moyenne :

```
def moyenne(t) :
    """ Renvoie la moyenne d'un tableau de nombres t non vide """
    total = 0
    for x in t :
        total = total + x
    return total/len(t)   # provoque une erreur si t = []
```

Itérer sur les valeurs du tableau avec `for x in t` comme ci-dessus ne donne pas accès aux indices des éléments parcourus. Or il est parfois nécessaire de manipuler ces indices, par exemple pour modifier une valeur du tableau, pour manipuler simultanément plusieurs valeurs du tableau ou encore pour accéder à l'indice correspondant dans un second tableau. Lorsque l'on souhaite connaître l'indice des éléments parcourus, on va plutôt parcourir le tableau avec `for i in range(len(t))` ou utiliser la fonction `enumerate`.
Ceci s'applique non seulement aux tableaux, mais aussi aux chaînes de caractères et aux tuples, qui peuvent aussi être parcourus par indice. En fait, les chaînes de caractères et les tuples sont implémentés par Python comme des tableaux un peu particuliers.

```
s = 'aviva'
for i in range(len(s)) :
    if s[i] != s[len(s)-1-i] :
        print(s, "n'est pas un palindrome")
```

La fonction Python `enumerate(t)` permet d'énumerer les paires (indice, valeur) d'un tableau `t` plus élégamment que `for i in range(len(t)): ...`. Elle renvoie la suite des tuples (indice, valeur) obtenus en parcourant le tableau. Cette fonction s'utilise souvent avec un tableau mais peut s'appliquer à n'importe quelle valeur itérable.

```
for k, v in enumerate(t) :
    # (k, v) = (0, 'érable'), (1, 'chêne'), (2, 'charme'), (3, 'hêtre
    ')
    if v == "chêne" :
        print(k)
```

Remarques

Comme `enumerate` renvoie un tuple, `k` récupère ici le premier élément du tuple et `v` le second. On peut aussi écrire :

```
for x in enumerate(t) :
    k, v = x
    ...
```

A l'inverse, un tableau peut être vu comme un dictionnaire un peu particulier dans lequel les clés sont des entiers consécutifs. Les méthodes ci-dessous permettent de parcourir un dictionnaire Python par clé, valeur ou paire (clé, valeur) :

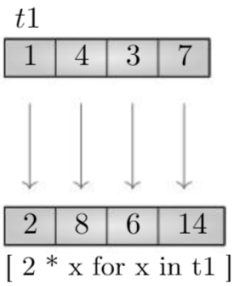
```
d = {'roussette' : 'Chondrichthyes', 'salamandre' : 'Amphibia'}
for k in d.keys() : ...        # k = 'roussette', 'salamandre'
for v in d.values() : ...      # v = 'Chondrichthyes', 'Amphibia'
for k, v in d.items() : ...  # (k,v) = ('roussette', 'Chondrichthyes')
    , ('salamandre', 'Amphibia')
```

III Construire un tableau par compréhension

Il est possible et élégant de construire un **tableau par compréhension** avec le langage Python. C'est très pratique pour créer, transformer ou filtrer un tableau.

Pour créer un **tableau par compréhension** avec Python, on va parcourir un tableau existant en utilisant la syntaxe `for i in range()` et éventuellement une condition de filtrage :

```
# Constructions de tableaux par compréhension
t4 = [ 2*x for x in t1 ]          # t4 = [2, 8, 6, 14], t1 ne change pas
t5 = [ x for x in t1 if x < 4 ] # t5 = [1, 3]
t6 = [ (2.5, 'a'+str(x)) for x in t1 if (x%2 == 1)] # t6 = [(2.5, 'a1
    '), (2.5, 'a3'), (2.5, 'a7') ]
```




```
t7 = [i**2 for i in range(5) if i**2 % 2 == 0]
```

Valeur Itérable Condition de filtrage

Remarques

Un **itérable** est un objet dont les valeurs sont accessibles grâce à une boucle `for`, par exemple. Les types `str`, `tuple`, `list` et `dict` sont des itérables fréquents en Python.

Ajouter une **condition de filtrage** permet de ne sélectionner que certains éléments du tableau, grâce à une expression booléenne ne pouvant prendre que deux états ; `True` ou `False`. Si la condition est vrai (`True`), les éléments seront ajoutés à la liste.

La syntaxe `[expr(x) for x in t if condition]` construit un *nouveau* tableau « à la volée » en parcourant les éléments du tableau `t`. Pour chaque élément `x` de `t` qui satisfait la condition (on ignore les autres), Python calcule l'expression `expr(x)` et ajoute le résultat à ce nouveau tableau.

Remarques

Cette construction est donc une façon élégante d'écrire la boucle suivante :

```
res = []
for x in t :
    if condition :
        res = res + expr(x) # ou bien : res.append(expr(x))
```

On peut construire un tableau par compréhension à partir de n'importe quelle valeur itérable, et pas seulement d'un tableau :

```
d = {'roussette' : 'Chondrichthyes', 'salamandre' : 'Amphibia'}

t7 = [ pow(2,x) for x in range(1,4) ] # [2, 4, 8]
t8 = [ s[0] for s in d.keys() ]       # ['r', 's']
t9 = [ x.upper() for x in 'aBe' ]     # ['A', 'B', 'E']
```

Lorsque l'on veut uniquement recopier tel quel le contenu d'une valeur itérable `v` dans un tableau, il n'est pas nécessaire d'utiliser une compréhension, on peut utiliser directement la fonction Python, `list(v)` :

```
s = 'abc'
tup = (0, 1, 'a')

list(s)    # ['a', 'b', 'c']
list(tup)  # [0, 1, 'a'] (tableau pas homogène)
```

La syntaxe par compréhension peut même s'appliquer à des boucles imbriquées. On voit que dans ce cas l'ordre des boucles est identique à celui qu'on utiliserait hors d'une compréhension :

```
t = ['ab', 'dce']; r = []
for s in t :
    for caractere in s :
        r.append(caractere)
```

```
5 # r = ['a', 'b', 'd', 'c', 'e']
6 r2 = [ car for s in t for car in s ] # r2 = r
7 # on peut même ajouter des conditions après chaque 'for'
```