Pascal ORTIZ

Les listes en compréhension

Version du 26 novembre 2022

Table des matières

Listes en compréhension

Étant donné une liste d'entiers telle que `t=[5, 2, 0, 3, 7, 10]`, on cherche à construire la liste `L` dont les éléments sont 10 fois les éléments de `t`, c'est-à-dire `[50, 20, 0, 30, 70, 100]`.

Voici un code Python répondant au problème :

```
1 t = [5, 2, 0, 3, 7, 10]
2 print(t)
3 L = [10 * x for x in t]
4 print(L)
```

```
5 [5, 2, 0, 3, 7, 10]
6 [50, 20, 0, 30, 70, 100]
```

A la ligne 3, est définie une liste `L` dite *liste en compréhension*.

Une liste en compréhension `L` a la syntaxe minimale suivante

```
[expr for x in t]
```

où

- la paire de crochets, les mots-clefs `for` et `in` sont obligatoires
- `t` est, par exemple, une liste (voir plus bas pour d'autres possibilités)
- `x` est l'élément courant qui parcourt la liste `t` ; `x` est appelé *variable de contrôle* de la liste en compréhension
- `expr` est une expression qui dépend en général de `x` et dont la valeur est placée dans `L`

Si `t` est un conteneur (une liste, une chaîne, etc), la liste en compréhension `L` avec la syntaxe ci-dessus a toujours même nombre d'éléments que le conteneur `t`.

Intérêt d'une liste en compréhension : générer une liste en une seule *expression* et non en une ou plusieurs *instructions*. L'intérêt des listes en compréhension est avant tout leur compacité d'édition dans le code et leur bonne lisibilité.

Vocabulaire : la documentation officielle en français a utilisé jusqu'à la version 3.9 de Python le terme assez incompréhensible de « compréhension de liste » pour finalement adopter le terme de « liste en compréhension ». On rencontre aussi parfois le terme de *liste en intension* censé s'opposer à l'expression *liste en extension* ce dernier signifiant, grosso modo, liste littérale, voir cette discussion.

Liste en compréhension et boucle `for`

Une liste en compréhension admet un équivalent créé avec une boucle `for` mais nécessitant un code plus long.

Soit la liste en compréhension `L` suivante :

```
t = [5, 2, 0, 3, 7, 10]
L = [10 * x for x in t]
print(L)
```

```
[50, 20, 0, 30, 70, 100]
```

Elle pourrait être obtenue avec une boucle `for` :

```
t = [5, 2, 0, 3, 7, 10]
L = []
for x in t:
    L.append(10*x)
print(L)
```

```
[50, 20, 0, 30, 70, 100]
```

— La liste L est obtenue en trois lignes de code au lieu d'une.

Listes en compréhension à partir d'un itérable

Une liste en compréhension est souvent construite sur la base d'une liste mais on peut construire une liste en compréhension sur la base de n'importe quel itérable.

Par exemple, on peut construire une liste en compréhension à partir d'une chaîne

```
w = "SERPENT"
L = [2 * c for c in w]
print(L)
```

```
['SS', 'EE', 'RR', 'PP', 'EE', 'NN', 'TT']
```

Listes en compréhension et la clause `if`

Il existe une variante dans la syntaxe des listes de compréhension utilisant une clause `if`.

Par exemple, étant donné une liste L d'entiers, soit à construire la liste `t` obtenue en ne gardant que les entiers x de L tel que $x > 42$:

```
t = [65, 31, 9, 32, 81, 82, 46, 12]
L = [x for x in t if x >= 42]
print(L)
```

```
[65, 81, 82, 46]
```

— x varie dans `t` et est inséré dans la liste L en construction si $x \geq 42$.

On pourrait même écrire tout le code en une seule ligne.

L'équivalent avec une boucle `for` serait le suivant :

```
1 t = [65, 31, 9, 32, 81, 82, 46, 12]
2 L= []
3 for x in t:
4     if x >= 42:
5         L.append(x)
6 print(L)
```

Listes en compréhension imbriquées

Des listes en compréhension peuvent être imbriquées. Néanmoins la liste n'est pas forcément créée dans l'ordre où on s'y attendrait.

Soit à créer une liste formée de tous « mots » commençant par une lettre A ou B et suivie d'un chiffre parmi 1, 2 ou 3. On peut utiliser une liste en compréhension :

```
1 L = [x+y for x in "AB" for y in "123"]
2 print(L)
```

```
3 ['A1', 'A2', 'A3', 'B1', 'B2', 'B3']
```

Le résultat ligne 3 montre que l'on fixe d'abord la lettre, c'est-à-dire les éléments du `for` lexicalement le plus interne (ici `for x`) puis que y varie. Ce n'est pas forcément très intuitif.

Le résultat est plus facilement compréhensible si la liste en compréhension est interprétée par deux boucles `for` imbriquées **exactement** dans l'ordre où on lit les apparitions des `for` dans la liste en compréhension, ce qui donne ici :

```
L = []
for x in "AB":
    for y in "123":
        L.append(x+y)
print(L)
```

```
['A1', 'A2', 'A3', 'B1', 'B2', 'B3']
```

— la boucle `for x` apparaît avant la boucle `for y`, dans le même ordre que la liste en compréhension `[x+y for x in "AB" for y in "123"]`.

Création d'un tableau 2D initialisé

Une liste en compréhension est un moyen **très simple** de créer un conteneur pour un tableau ayant n lignes et p colonnes. Le code ci-dessous crée un tableau à 2 lignes et 3 colonnes initialisé avec des 0 :

```
L = [[0 for j in range(3)] for i in range(2)]
print(L)
L[1][2]=42
print(L)
```

```
[[0, 0, 0], [0, 0, 0]]
[[0, 0, 0], [0, 0, 42]]
```

— On crée une liste de 2 lignes, chaque ligne étant une liste de 3 éléments. L'indice `i` parcourt les lignes et l'indice `j` parcourt chaque colonne de chaque ligne.

Au passage, comme l'élément le plus interne dans les listes est immuable (il s'agit de l'entier 0), on peut même simplifier la syntaxe :

```
L = [[0]*3 for i in range(2)]
print(L)
L[1][2]=42
```

```
print(L)
```

```
[[0, 0, 0], [0, 0, 0]]
[[0, 0, 0], [0, 0, 42]]
```

Confusion possible

Attention toutefois que le code suivant lui ne donne pas le résultat attendu :

```
L = [[0]*3]*2
print(L)
L[1][2]=42
print(L)
```

```
[[0, 0, 0], [0, 0, 0]]
[[0, 0, 42], [0, 0, 42]]
```

Code équivalent

Voici l'équivalent du code correct donné en début de section **sans utiliser** de liste en compréhension et en utilisant deux boucles for imbriquées. On notera que le code est plus long :

```
L=[]
for i in range(2):
    lig = []
    for j in range(3):
        lig.append(0)
    L.append(lig)
print(L)
```

```
[[0, 0, 0], [0, 0, 0]]
```

Listes en compréhension imbriquées et clause `if`

Soit à placer dans une liste `L` tous les couples d'entiers (x, y) tel que

$$0 \leq x, y \leq 3 \text{ et } x + y \leq 3$$

Voici un code utilisant une liste en compréhension :

```
a = 3
L =[[x,y] for x in range(0,a+1) for y in range(0,a+1) if x+y <= a]
print(L)
```

```
[[0, 0], [0, 1], [0, 2], [0, 3], [1, 0],
[1, 1], [1, 2], [2, 0], [2, 1], [3, 0]]
```

La syntaxe de la liste en compréhension a été prévue pour que la **traduction** depuis des boucles for imbriquées soit **immédiate** : on place les `for` et `if` dans l'ordre de gauche à droite où ils apparaissent dans la liste en compréhension :

```
a = 3
L =[]
for x in range(0,a+1):
    for y in range(0,a+1):
        if x+y <= a:
            L.append([x,y])
print(L)
```

```
[[0, 0], [0, 1], [0, 2], [0, 3], [1, 0],
[1, 1], [1, 2], [2, 0], [2, 1], [3, 0]]
```

Efficacité des listes en compréhension

Les listes en compréhension seraient plus rapides que l'équivalent avec une boucle `for`. Voici un exemple de comparaison de performances :

```
from time import perf_counter

N=10**7

begin_perf = perf_counter()

L=[]
for i in range(N):
    L.append(i**2)

delta = perf_counter() - begin_perf

print(f"Boucle for : {delta:.2f}s")

begin_perf = perf_counter()
L=[i**2 for i in range(N)]
delta = perf_counter() - begin_perf

print(f"Compréhension : {delta:.2f}s")
```

```
Boucle for : 2.70s
Compréhension : 2.14s
```

On constate une meilleure performance de 20

```
from time import perf_counter

N=10**7

begin_perf = perf_counter()

L=[]
for i in range(N):
```

```
    L.append(i**2)

delta = perf_counter() - begin_perf

print(f"Boucle for : {delta:.2f}s")


begin_perf = perf_counter()

L=[]
M=L.append
for i in range(N):
    M(i**2)

delta = perf_counter() - begin_perf

print(f"Boucle for var : {delta:.2f}s")

begin_perf = perf_counter()
L=[i**2 for i in range(N)]
delta = perf_counter() - begin_perf

print(f"Compréhension : {delta:.2f}s")
```

```
Boucle for : 2.68s
Boucle for var : 2.53s
Compréhension : 2.07s
```

Toutefois, la meilleure performance est à relativiser, cf. List Comprehensions instead of loop

Portée des variables de contrôle d'une liste en compréhension

La variable de contrôle d'une boucle `for` est visible après la fin de la boucle `for` :

```
1 L=[]
2
3 for i in range(5):
4     L.append(5*i)
5 print(i)
```

```
6 4
```

— Ligne 5 : `i` est accessible après la fin de l'exécution de la boucle `for`.

Il n'en est pas de même d'une variable de contrôle d'une liste en compréhension :

```
L = [10 * i for i in range(5)]
print(i)
```

```
NameError: name 'i' is not defined
```

— Le nom `i` a une portée limitée à la liste en compréhension : à la ligne suivante, `i` est inconnu.

Liste en compréhension pour répéter un effet de bord

On peut invoquer une liste en compréhension non pour générer une liste mais pour exploiter l'action de répétition associée à la liste en compréhension.

Par exemple, pour retirer les 5 derniers éléments d'une liste L, il suffit de répéter 5 fois l'action L.pop(), d'où le code suivant :

```
1 L=[65, 31, 9, 32, 81, 82, 46, 12]
2 print(L)
3 [L.pop() for i in range(5)]
4 print(L)
```

```
5 [65, 31, 9, 32, 81, 82, 46, 12]
6 [65, 31, 9]
```

— Ligne 3 : L.pop() retire l'élément final de la liste.

— Ligne 3 : Une liste en compréhension est bien créée mais son contenu est ignoré.

— Lignes 5-6 : La liste L a bien perdu ses 5 derniers éléments.

L'intérêt est néanmoins réduit : cela évite juste d'écrire une boucle `for`, c'est intéressant si ça vous amuse d'écrire des one-liner !

Expressions génératrices

Une expression génératrice est une construction syntaxique **proche** des listes en compréhension et qui renvoie un itérateur.

Le code suivant illustre la notion d'expression génératrice :

```
1 g=(10**i for i in range(5))
2 print(g)
3 for x in g:
4     print(x)
```

```
5 <generator object <genexpr> at 0xb726e25c>
6 1
7 10
8 100
9 1000
10 10000
```

— Ligne 1 : le membre de droite est une expression génératrice.

— Ligne 1 : l'objet évalué par une expression génératrice s'appelle un *générateur* mais en fait c'est un itérateur. Il permet de générer à la volée une suite d'éléments, ici la suite des 5 premières puissances de 10 sans avoir à stocker simultanément tous les éléments générés.

— Ligne 3 : l'usage le plus courant d'une expression génératrice est de la placer dans l'en-tête d'une boucle for et de générer les éléments du générateur.

La syntaxe minimale d'une expression génératrice g est la suivante :

```
(expr for i in iter)
```

ou `expr` est une expression qui en général dépend de `i` et où `iter` est un itérable, par exemple une liste. Quand le générateur `g` est exécuté, `i` parcourt l'itérable `iter` puis l'élément correspondant `expr` est généré par `g`, jusqu'à ce que l'itérable `iter` soit épuisé. En particulier, les éléments générés par l'expression `expr` ne sont pas stockés.

Les justifications d'utilisation d'une expression génératrice sont à peu près les mêmes que pour un itérateur : itérer sans stocker. Un usage courant d'une expression génératrice est de la placer dans l'en-tête d'une boucle for :

```
for z in (expr for i in iter):
```

Confusion possible

Ne pas confondre une expression génératrice avec une liste ou un tuple ; en particulier, si `g` est une expression génératrice, la syntaxe `g[i]` est invalide :

```
g=(10**i for i in range(5))
print(g[2])
```

```
TypeError: 'generator' object is not subscriptable
```

La seule différence de syntaxe entre les expressions génératrices et les listes en compréhension, est la nature des délimiteurs qui sont des parenthèses au lieu de crochets. Cependant, il ne faudrait pas croire qu'une expression génératrice serait une sorte de « tuple en compréhension ».

Listes en compréhension et itérables

Soit `L` une liste d'entiers ; on cherche à calculer la somme `S` des carrés des éléments de `L`. L'usage d'une liste en compréhension se prête bien à la recherche de `S` :

```
L = [10, 16, 100, 9, 5]
M = [x**2 for x in L]
S = sum(M)
print(S)
```

```
10462
```

Avant de calculer `S`, on a stocké la liste des carrés des éléments de `L`. Mais en fait le calcul de `S` ne nécessite pas de stocker les éléments de `M` mais juste d'itérer sur ces éléments.

Il suffit donc de créer un itérable qui permette de parcourir les éléments de `M` sans avoir à stocker en permanence ses éléments. Un moyen simple est d'utiliser un *générateur* sous forme d'expression génératrice dont la syntaxe est proche des listes en compréhension :

```
L = [10, 16, 100, 9, 5]
it = (x**2 for x in L)
print(it)
S = sum(it)
print(S)
```

```
<generator object <genexpr> at 0xb728125c>
10462
```

— `it` est un itérateur, il permet d'itérer sur des éléments sans les stocker

— sum accepte tout itérable et donc en particulier un itérateur comme `it`.

Une liste en compréhension est d'abord une liste et donc une structure de données supposant un stockage en mémoire. En fonction de l'opération que l'on souhaite faire, il se peut que le stockage ne soit pas utile et se pose alors la question de la pertinence du choix d'une structure de liste.