W4-S4-C1-comprehensions-imbriquees

December 15, 2014

1 Rappels - compréhensions imbriquées

1.1 Compléments - niveau intermédiaire

1.1.1 Ordre d'évaluation de `[[ .. for .. ] .. for .. ]`

Pour rappel, on peut imbriquer des compréhensions de compréhensions. Commençons par poser

```
In []: n = 4
```

On peut alors créer une liste de listes comme ceci:

```
In []: [[(i, k) for i in range(1, k + 1)] \
                for k in range(1, n + 1)]
```

Et dans ce cas, très logiquement, l'évaluation se fait **en commençant par la fin**, ou si on veut **"par l'extérieur"**, c'est-à-dire que le code ci-dessus est équivalent à:

```
In []: # en version bavarde, pour illustrer l'ordre des "for"
       resultat_exterieur = []
       for k in range(1, n + 1):
           resultat_interieur = []
           for i in range(1, k + 1):
               resultat_interieur.append((i, k))
           resultat_exterieur.append(resultat_interieur)
       resultat_exterieur
```

1.1.2 Avec `if`

Lorsqu'on assortit les compréhensions imbriquées de cette manière de clauses `if`, l'ordre d'évaluation est tout aussi logique. Par exemple, si on voulait se limiter - arbitrairement - aux lignes correspondant à `k` pair, et aux diagonales où `i+k` est pair, on écrirait:

```
In []: [[(i, k) for i in range(1, k + 1) if (i + k)%2 == 0] \
                for k in range(1, n + 1) if k % 2 == 0]
```

ce qui est équivalent à

```
In []: # en version bavarde à nouveau
       resultat_exterieur = []
       for k in range(1,n + 1):
           if k % 2 == 0:
               resultat_interieur = []
               for i in range(1, k + 1):
                   if (i + k) % 2 == 0:
                       resultat_interieur.append((i, k))
               resultat_exterieur.append(resultat_interieur)
       resultat_exterieur
```

Le point important ici est que l'**ordre** dans lequel il faut lire le code est **naturel**, et dicté par l'imbrication des `[ .. ]`.

1.2 Compléments - niveau avancé

1.2.1 Un défaut de python-2

Faisons tout d'abord une première remarque. L'implémentation des `for` en python-2 laisse à désirer, dans ce sens qu'une variable interne à un `for` **reste active** à la sortie de la boucle. Par exemple à ce stade nous avons toujours accès aux variables

```
In []: i, k
```

Ce comportement est discutable - et cela a été d'ailleurs amélioré en python-3.

C'est pourquoi, afin de comparer les deux formes de compréhension imbriquées nous allons explicitement retirer les variables `i` et `k` de l'environnement

```
In []: del i, k
```

1.2.2 Ordre d'évaluation de `[ .. for .. for .. ]`

Toujours pour rappel, on peut également construire une compréhension imbriquée mais **à un seul niveau**. Dans une forme simple cela donne

```
In []: [(x, y) for x in [1, 2] for y in [1, 2]]
```

Avertissement méfiez-vous toutefois, car il est facile de ne pas voir du premier coup d'oeil qu'ici on évalue les deux clauses `for` **dans un ordre différent**.

Pour mieux le voir, essayons de reprendre la logique de notre tout premier exemple, mais avec une forme de double compréhension *à plat*.

```
In []: [i**2 for i in range(1, k + 1) for k in range(1, n + 1)]
```

On obtient une erreur, l'interpréteur se plaint à propos de la variable `k` (c'est pourquoi nous l'avons effacée de l'environnement au préalable).

Ce qui se passe ici, c'est que, comme nous l'avons déjà mentionné en semaine 3, le code que nous avons écrit est en fait équivalent à

```
In []: # la version bavarde de cette imbrication à plat, à nouveau
       # [i**2 for i in range(1, k + 1) for k in range(1, n + 1)]
       # serait
       resultat = []
       for i in range(1, k + 1):
           for k in range(1, n + 1):
               resultat.append((i, k))
```

Et dans cette version * dépliée* on voit bien qu'en effet on utilise `k` avant qu'il ne soit défini.

1.2.3 Conclusion

La possibilité d'imbriquer des compréhensions avec plusieurs niveaux de `for` dans la même compréhension est un trait qui peut rendre service, car c'est une manière de simplifier la structure des entrées (on passe essentiellement d'une liste de profondeur 2 à une liste de profondeur 1).

Mais il faut savoir ne pas en abuser, et rester conscient de la confusion qui peut en résulter, et en particulier être prudent et prendre le temps de bien se relire. N'oublions pas non plus ces deux phrases du zen de python : "Flat is better than nested" et surtout "Readability counts".