Construction de liste par compréhension

Complément - niveau basique

Ce mécanisme très pratique permet de construire simplement une liste à partir d'une autre (ou de **tout autre type iterable** en réalité, mais nous y viendrons).

Pour l'introduire en deux mots, disons que la compréhension de liste est à l'instruction `for` ce que l'expression conditionnelle est à l'instruction `if`, c'est-à-dire qu'il s'agit d'une **expression à part entière**.

Cas le plus simple

Voyons tout de suite un exemple

```
depart = [3, 7, 14, 28]
[x**2 for x in depart]
```

Comme vous l'avez sans doute deviné, le résultat de cette expression est une liste, dont les éléments sont les résultats de l'expression `x**2` pour `x` prenant toutes les valeurs de `depart`

Restriction à certains éléments

Il est possible également de ne prendre en compte que certains des éléments de la liste de départ, comme ceci

```
[x**2 for x in depart if x%2 == 0]
```

qui cette fois ne contient que les carrés des éléments pairs de `depart`

Autres types

On peut fabriquer une compréhension à partir de tout objet itérable, pas forcément une liste, mais le résultat est toujours une liste, comme on le voit sur ces quelques exemples:

```
[ord(x) for x in 'abc']
```

```
[chr(x) for x in (97, 98, 99)]
```

Nous verrons très bientôt que des mécanismes similaires sont disponibles avec les dictionnaires et les ensembles.

Complément - niveau intermédiaire

Imbrications

On peut également imbriquer plusieurs niveaux pour ne construire qu'une seule liste, comme par exemple

```
[n + p for n in [2, 4] for p in [10, 20, 30]]
```

Bien sûr on peut aussi restreindre ces compréhensions, comme par exemple

```
[n + p for n in [2, 4] for p in [10, 20, 30] if n*p >= 40]
```

Observez surtout que le résultat ci-dessus est une liste simple (de profondeur 1), à comparer avec

```
[[n + p for n in [2, 4]] for p in [10, 20, 30]]
```

qui est de profondeur 2, et où les résultats atomiques apparaissent dans un ordre différent

Un moyen mnémotechnique pour se souvenir dans quel ordre les compréhensions imbriquées produisent leur résultat, est de penser à la version "naïve" du code qui produirait le même résultat; dans ce code les clause `for` et `if` apparaissent **dans le même ordre** que dans la compréhension

```
resultat=[]
for n in [2, 4]:
    for p in [10, 20, 30]:
        if n*p >= 40:
            resultat.append(n + p)
resultat
```

Complément - niveau avancé

Note sur `map` et `filter`

Avant que les compréhensions ne soient introduites, on utilisait deux fonctions built-in intitulées `map` (nom qui provient à l'origine de Lisp) et `filter`. Leur usage est à présent déconseillé, car le code est moins lisible. On les trouve encore dans du code existant.

Pour donner un aperçu de ces fonctions, au cas où vous en rencontriez dans du code existant, voici comment on écrirait

```
[x**2 for x in depart if x%2 == 0]
```

Avec `map` et `filter` cela donnerait

```
def pair(x):
    return x%2 == 0

def carre(x):
    return x**2

map(carre, filter(pair, depart))
```

Ou encore, sur une ligne

```
map(lambda x: x**2, filter(lambda x: x%2 == 0, depart))
```

Remarque: l'instruction `lambda` permet permet de définir une fonction, que l'on appelle fonction lambda, à la volée et sans la nommer. Nous reviendrons dessus dans les semaines à venir. Mais à notre avis au moins, les fonctions lambda ont perdu beaucoup de leur intérêt depuis, précisément, l'introduction des compréhensions. Aussi nous ne recommandons pas non plus de les utiliser dans du code nouveau.

Pour en savoir plus

La section sur les compréhensions de liste (https://docs.python.org/2/tutorial/datastructures.html#list-comprehensions) dans la documentation python