W6-S1-C1-Complement-expressions-generatrices

December 15, 2014

1 Expressions génératrices

1.1 Complément - niveau basique

1.1.1 Comment transformer une compréhension de liste en itérateur ?

Nous venons de voir les fonctions génératrices qui sont un puissant outil pour créer facilement des itérateurs. Nous verrons prochainement comment utiliser ces fonctions génératrices pour tranformer en quelques lignes de code vos propres objets en itérateurs.

Vous savez maintenant qu'en python on favorise la notion d'itérateurs puisqu'ils se manipulent comme des objets itérables et qu'ils sont en général beaucoup plus compacts en mémoire que l'itérable correspondant.

Comme les compréhensions de listes sont fréquemment utilisées en python, mais qu'elles sont des itérables potentiellement gourmands en ressources mémoire, on souhaiterait pouvoir créer un itérateur directement à partir d'une compréhension de liste. C'est possible et très facile en python. Il suffit de remplacer les crochets par des parenthèses, regardons cela.

```
In []: # c'est une compréhension de liste
       comprehension = [x**2 for x in xrange(100) if x%17==0]
       print comprehension
```

```
In []: # c'est une expression génératrice
       generator = (x**2 for x in xrange(100) if x%17==0)
       print generator
```

Ensuite pour utiliser une expression génératrice, c'est très simple, on l'utilise comme n'importe quel itérateur.

```
In []: generator is iter(generator) # generator est bien un itérateur
```

```
In []: # affiche les 6 premiers carrés des multiples de 17
       for count, carre in enumerate(generator):
           print 'Contenu de generator après {} itérations : {}'.format(count+1, carre)
```

Avec une expression génératrice on n'est plus limité comme avec les compréhensions par le nombre d'éléments :

```
In []: # trop grand pour une compréhension,
       # mais on peut créer le générateur sans souci
       generator = (x**2 for x in xrange(10**18) if x%17==0)

       # et utiliser le début, on ne paie que ce qu'on utilise
       mon_set = set([])
       for x in generator:
           if x > 10**10:
               break
           elif str(x)[-4:] == '1316':
               mon_set.add(x)
       print len(mon_set)
```

1.2 Complément - niveau avancé

Il est important de comprendre que l'objet expression génératrice se comporte comme une fonction, notamment vis-à-vis des espaces de nommage et de la portée des variables. C'est donc un objet très différent des compréhensions. Regardons un exemple.

```
In []: class Comprehension:
           increment = 10
           liste = [a + increment for a in range(10)]
```

Il n'y a pas de difficultés particulières ici, on peut regarder l'attribut liste de l'objet classe `Comprehension`.

```
In []: print Comprehension.liste
```

Remplaçons maintenant la compréhension par une expression génératrice.

```
In []: class Generator:
           increment = 10
           liste = (a + increment for a in range(10))
```

```
In []: print Generator.liste
```

Cela fonctionne toujours. Que se passe-t-il si l'on transforme maintenant immédiatement notre expression génétratrice en liste (notons que lorsque l'expression génératrice est passée à une fonction, on peut omettre les parenthèses autour de l'expression).

```
In []: class Mixed:
           increment = 10
           liste = list(a + increment for a in range(10))
```

On obtient ici une exception qui nous dit que la variable globale `increment` n'est pas définie. Quelles sont les différences entre les classes `Comprehension`, `Generator` et `Mixed` ? Je vous rappelle que les objets classe et fonction sont créés au moment du chargement du module (ici, au moment de l'évaluation de la cellule), mais que le corps de la fonction (donc aussi des expressions génératrices) n'est évalué qu'au moment de l'appel.

Dans le cas de la classe `Comprehension`, la compréhension de liste est évaluée au moment de la création de l'objet classe `Comprehension`, lorsque l'on arrive sur la compréhension, la variable `increment` est définie et vaut 10.

Dans le cas de la classe `Generator`, on a remplacé la compréhension par une expression génératrice. Cela a deux impacts : l'expression génératrice ne sera évaluée qu'à son premier appel, l'expression génératrice étant une fonction, elle ne peut pas accéder à l'espace de nommage de la classe. Donc, l'objet classe `Generator` a bien été créé, mais l'appel de la fonction génétratrice devrait échouer parqu'elle ne peut pas avoir accès à l'attribut `increment` de la classe. Vérifions cela

```
In []: Generator.liste.next()
```

Effectivement, on obtient bien une exception. Dans le cas de `Mixed`, on a immédiatement une exception parce que dans la classe on transforme immédiatement la compréhension en liste avec le contructeur `list`. C'est donc un appel à l'expression génératrice qui force l'évaluation de son code au moment de la création de la classe.

Mais alors, comment peut-on faire marcher ce code ? On pourrait utiliser `Mixed.val` dans l'expression, regardons cela.

```
In []: class Mixed:
           increment = 10
           liste = list(a + Mixed.increment for a in range(10))
```

Ça ne fonctionne toujours pas avec une erreur étrange... Le nom global `Mixed` n'existe pas lorsque l'on fait `Mixed.increment`. En y réfléchissant, c'est tout à fait normal. L'objet classe `Mixed` ne sera créé qu'à la fin de l'évaluation de son bloc de code, or lorsque l'on appel `Mixed.increment`, on est toujours en cours d'évaluation du bloc de code de la classe.

Comment s'en sortir alors ? Il y a de nombreuses possibilités. Par exemple, on peut utiliser la construction avec une compréhension de liste qui, comme on l'a vu avec la classe `Comprehension`, fonctionne bien. On peut aussi mettre notre expression génératrice à l'intérieur d'une fonction que l'on appelera plus tard après la création de l'objet classe.

1.2.1 Pour aller plus loin

Vous pouvez regarder cette intéressante discussion de Guido van Rossum sur les compréhensions et les expressions génératrices.