Gestion de la mémoire : Comparaison entre Java et Python

Ces exercices ont pour but de mieux comprendre le fonctionnement de la gestion automatique de la mémoire en Java et en Python. Pour cela, nous allons chaque fois créer de grandes quantités d'objets non référencés et voir comment le langage se débrouille pour récupérer la mémoire.

1 Python, sans cycles

La méthode spéciale `__del__` est appelée lors de la destruction d'objets. Ceci permet de voir facilement ce qui se passe :

```
class Garbage:
    def __init__(self,i):
        self.i = i
        print "creating", self.i

    def __del__(self):
        print "** deleting", self.i

for i in xrange(100000):
    Garbage(i)
```

Lancez ce programme et étudiez à quel moment la mémoire des objets inutiles est récupérée.

2 Java, sans cycles

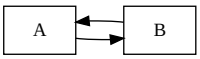
À part quelques détails syntaxiques et la transformation de `__del__` en `finalize`, on fait en gros la même chose :

```
public class Garbage {
    int id;

    public Garbage(int i) {
        id = i;
        System.out.println("creating " + id);
    }

    public void finalize() {
        System.out.println("** deleting " + id);
    }

    public static void main(String args[]) {
        for(int i=0;i<100000;i++) new Garbage(i);
    }
}
```

En exécutant ce programme, que constatez-vous ? quand les objets sont-ils détruits ? que pouvez-vous en déduire sur les stratégies de libération de la mémoire de Java ?

3 Java, avec cycles

Modifiez votre programme Java pour qu'il produise à chaque itération, non pas un objet seul, mais un cycle de deux objets :



Constatez-vous une grosse différence dans la libération de la mémoire ?

4 Python, avec cycles

Là aussi, construisez un cycle "isolé" à chaque itération.

Malheureusement, la volonté d'observer ce qui se passe influe sur ce qui se passe vraiment : le détecteur de cycles de python refuse de détruire des cycles d'objets qui implémentent la méthode `__del__`, car il ne peut pas savoir dans quel ordre il doit les appeler.

Il y a donc deux solutions :

1. Supprimer la méthode `__del__`, tout marchera parfaitement, mais on ne le saura pas... (c'est la meilleure solution, sauf dans le cadre de cet exercice !)
2. Heureusement, le *garbage collector* de python stocke dans une variable les objets non référencés qu'il ne peut pas détruire. On va donc l'aider de la manière suivante : à chaque itération, on va consulter cette variable, casser les cycles et détruire les objets "à la main".

   Après avoir importé `gc`, on fera donc à chaque itération quelque chose comme :

```
for g in gc.garbage: # pour chaque objet non-récupérable...
    g.next = None # on casse le cycle...
del gc.garbage[:] # puis on détruit le tout!
```

Qu'observez-vous ? y a-t-il une grosse différence avec la version sans cycle ? que pouvez-vous en déduire sur la gestion de la mémoire de python ?