W3-S8-C3-Complement-variables-de-boucle

December 15, 2014

1 Visibilité des variables de boucle

1.1 Complément - niveau basique

1.1.1 Une astuce

Dans ce complément, nous allons beaucoup jouer avec le fait qu'une variable soit définie ou non. Pour nous simplifier la vie, et surtout rendre les cellules plus indépendantes les unes des autres si vous devez les rejouer, nous allons utiliser la formule un peu magique suivante :

```
In []: # on détruit la variable i si elle existe
       if 'i' in locals():
           del i
```

qui repose d'une part sur l'instruction `del` que nous avons vue un peu plus tôt cette semaine, et sur la fonction *builtin* `locals` que nous verrons plus tard; cette formule a l'avantage qu'on peut l'exécuter dans n'importe quel contexte, que `i` soit définie ou non.

1.1.2 Une variable de boucle reste définie au-delà de la boucle

Cela peut paraître évident, mais explicitons-le tout de même : une variable de boucle est définie (assignée) dans la boucle et **reste *visible*** une fois la boucle terminée. Le plus simple est de le voir sur un exemple :

```
In []: # La variable 'i' n'est pas définie
       try:
           i
       except NameError as e:
           print e

In []: # on fait une boucle sur i
       for i in [0]:
           pass

       # maintenant i est définie
       i
```

On dit que la variable *fuite* (en anglais "*leak*"), dans ce sens qu'elle pollue l'environnement dans lequel est lancée la boucle.

On peut être tenté de tirer profit de ce trait, en lisant la valeur de la variable après la boucle; l'objet de ce complément est de vous inciter à la prudence, et d'attirer votre attention sur certains points possiblement source d'erreurs.

1.1.3 Attention aux boucles vides

Tout d'abord, il faut faire attention à ne pas écrire du code qui dépende de ce trait **si la boucle peut être vide**. En effet, si la boucle ne s'exécute pas du tout, eh bien la variable n'est pas affectée. Là aussi c'est évident, mais ça peut l'être moins quand on lit du code réel, comme par exemple :

```
In []: # on détruit la variable i si elle existe
       if 'i' in locals():
           del i

In []: def length(l):
           for i, x in enumerate(l):
               pass
           return i + 1

       length([1, 2, 3])
```

Ça a l'air correct, sauf que :

```
In []: length([])
```

Ce résultat mérite une explication. Nous avons déjà vu dans un complément précédent en semaine 4 l'exception `UnboundLocalError` qui se produit lorsque l'on a dans une fonction une variable locale et une variable globale de même nom. Alors, pourquoi pour l'appel `length([1, 2, 3])` il n'y a pas d'exception alors que pour l'appel 'length([])' il y a une exception. Cela est lié à la manière dont python détermine qu'une variable est locale.

Une variable est locale dans une fonction si elle est assignée dans la fonction explicitement (avec une opération d'affectation) ou implicitement (par exemple avec une boucle for comme ici), nous reviendrons sur ce point dans la dernière vidéo de cette semaine. Mais pour les fonctions, pour une raison d'efficacité, une variable est définie comme locale à la phase de pré-compilation, c'est-à-dire avant l'exécution du code. Par conséquent, le pré-compilateur ne peut pas savoir quel sera l'argument passé à la fonction, il peut simplement savoir qu'il y a une boucle `for` utilisant la variable `i`, donc, pour lui, `i` est locale pour toute la fonction.

Lors du premier appel, on passe une liste à la fonction, liste qui est parcourue par la boucle `for`. En sortie de boucle, on a bien une variale locale `i` qui vaut 3. Lors du deuxième appel, on passe une liste vide à la fonction, la boucle `for` ne peut rien parcourir, donc elle retourne immédiatement. Lorsque l'on arrive à la ligne `return i + 1` de la fonction, la variable `i` n'a pas de valeur (on doit donc chercher `i` dans le module), mais `i` a été définie par le pré-compilateur comme étant locale, on a donc dans la même fonction une variable `i` locale et une référence à une variable `i` globale, on a donc l'exception `UnboundLocalError`.

1.1.4 Comment faire alors ?

Utiliser une autre variable La première voie consiste à déclarer une variable externe à la boucle et à l'affecter à l'intérieur de la boucle, c'est-à-dire :

```
In []: candidates = [3, -12, 1, 8]

In []: # plutôt que de faire ceci
       for item in candidates:
           if (item**2 + 2 * item - 3) == 0:
               break
       print 'trouvé solution', item

In []: # il vaut mieux faire ceci
       solution = None
       for item in candidates:
           if ( item**2 + 2 * item - 3 ) == 0:
               solution = item
```

```
            break

    print 'trouvé solution',solution
```

Au minimum initialiser la variable Au minimum, si vous utilisez la variable de boucle après la boucle, il est vivement conseillé de l'**initialiser** explicitement **avant** la boucle, pour vous prémunir contre les boucles vides, comme ceci :

```
In []: def length(l):
           i = -1
           for i, x in enumerate(l):
               pass
           return i + 1

       length([])
```

1.1.5 Les compréhensions

Un phénomène analogue se produit avec les compréhensions :

```
In []: # on détruit la variable i si elle existe
       if 'i' in locals():
           del i
```

```
In []: # en python2 les variables de compréhension fuitent
       # mais ce n'est plus le cas en python3
       [i**2 for i in range(3)]

       i
```

1.1.6 python3

Pour conclure, notez bien que ce comportement a été modifié en python3 de la façon suivante : * les variables de **compréhension** ne **fuitent plus**, c'est un changement de sémantique ; * mais les variables de boucle conservent la même sémantique.

Sans doute une raison supplémentaire pour éviter autant que possible de rester indépendant de cette sémantique.