W5-S8-C1-Complement-espaces-de-nommage

December 14, 2014

1 Espaces de nommage

1.1 Complément - niveau basique

Nous venons de voir les règles pour l'affectation (ou l'assignation) et le référencement des variables et des attributs ; en particulier, on doit faire une distinction entre les attributs et les variables.

- Les attributs sont résolus de manière **dynamique**, c'est-à-dire au moment de l'exécution (à *run-time*),
- alors que la liaison des variables est par contre **statique** (à compile-time) et **lexicale**, en ce sens qu'elle se base uniquement sur les imbrications de code.

Vous voyez donc que la différence entre attributs et variables est fondamentale. Dans ce complément, nous allons reprendre et résumer les différentes règles qui régissent l'affectation et le référencement des attributs et des variables.

Attributs Un attribut est un symbole `x` utilisé dans la notation `obj.x` où `obj` est l'objet qui définit l'espace de nommage sur lequel `x` existe.

L'**affectation** (explicite ou implicite) d'un attribut `x` sur un objet `obj` va créer un symbole `x` dans l'espace de nommage de `obj`, symbole qui va référencer l'objet affecté, par exemple, l'objet à droite du signe `=` lors d'une affectation explicite.

```
In []: class MaClasse:
           pass
       MaClasse.x = 10 # affectation explicite
       #le symbole x est défini dans l'espace de nommage de MaClasse
       print MaClasse.__dict__
```

Le **référencement** d'un attribut va chercher cet attribut de long de l'arbre d'héritage en commençant par l'instance, puis la classe qui a créé l'instance, puis les super classes et suivant la MRO (voir le complément sur l'héritage multiple).

Variables Une variable est un symbole qui n'est pas précédé de la notation `obj.` et l'affectation d'une variable rend cette variable locale au bloc de code dans lequel elle est définie, un bloc de code pouvant être : * une fonction, dans ce cas la variable est locale à la fonction ; * une classe, dans ce cas la variable est locale à la classe ; * un module, dans ce cas la variable est locale au module, on dit également que la variable est globale.

Une variable référencée est toujours cherchée suivant la règle LEG : * localement au bloc de code dans lequel elle est référencée ; * puis dans les blocs de code des **fonctions ou méthodes** englobantes, s'il y en a, de la plus proche à la plus eloignée ; * puis dans le bloc de code du module.

Si la variable n'est toujours pas trouvée, elle est cherchée dans le module `__builtin__` et si elle n'est toujours pas trouvée, il y a une exception.

Dans la vidéo, pour bien montrer les différents cas, on a défini trois règles : la règle G pour les modules, la règles LEG pour les fonctions et la règles LG pour les classes. Mais il s'agit en fait toujours d'une dérivation de la règle LEG. En effet, * la règle G pour les modules correspond en fait à la règle LEG sauf que dans

ce cas, le seul bloc de code pertinent est celui du module ; * la règle LG pour les classes correspond aussi à la règle LEG sauf que dans l'exemple de la vidéo on n'a que des classes englobantes, pas de fonctions englobantes.

Évidement, même pour les classes, si on a des fonctions (ou méthodes) englobantes, la règle LEG s'applique. Par exemple

```
In []: var = 'dans le module'
       class A:
           var = 'dans la classe A'
           def f(self):
               var = 'dans la fonction f'
               class B:
                   print var
               B()
       A().f()
```

En résumé Dans la vidéo et dans ce complément basique, on a couvert tous les cas standards, et même si python est un langage plutôt mieux fait, avec moins de cas particuliers, que d'autres langages, il a également ses cas étranges entre raisons historiques et bugs qui ne seront jamais corrigés (parce que ça casserait plus de choses que ça n'en réparerait). Pour éviter de tomber dans ces cas spéciaux, c'est simple vous n'avez qu'à suivre deux règles : * ne jamais affecter dans un bloc de code local une variable de même nom qu'une variable globale ; * toujours mettre la directive `global` comme première instruction du bloc de code où elle s'applique (mais nous vous rappelons qu'il faut éviter d'utiliser cette directive dans le code que vous écrivez).

Si vous ne suivez pas ces règles, vous risquez de tomber dans un cas particulier que nous détaillons ci-dessous dans la partie avancée.

1.2 Complément - niveau avancé

UnboundLocalError Nous avons déjà vu cette erreur en semaine 4, dans le deuxième complément de la vidéo 6. Cette erreur se produit justement lorsque vous ne respectez pas la première des règles ci-dessus.

```
In []: x = 1
       def f():
           print x
           x = 10 + x
           return x
       f()
```

Ce code n'a que des défauts. Il est ambigu : on mélange x global et x local et il est impossible de savoir ce que ça doit faire, il ne respecte pas la première règle car x est une variable globale et on utilise le même nom pour une variable locale à la fonction. Comment faire alors ? Respecter la première règle. Oui, mais comment être sûr qu'on n'utilise pas localement un nom défini globalement ? Il faut utiliser des noms de variables qui ont un sens. Regardons un exemple

```
In []: increment_par_defaut = 1
       def shift_10():
           #on print pour débugger ici
           print increment_par_defaut

           total_shift = 10 + increment_par_defaut
           return total_shift
       shift_10()
```

Ma fonction n'est sans doute toujours pas révolutionnaire, mais elle est parfaitement claire et suit la la première règle.

La documentation officielle est fausse Oui, vous avez bien lu, la documentation officielle est fausse sur un point subtil. Regardons le modèle d'exécution, on trouve la phrase suivante "If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block." qui est fausse, il faut lire "If a name binding operation occurs anywhere within a code block **of a function**, all uses of the name within the block are treated as references to the current block."

En effet, les classes se comportent différemment des fonctions

```
In []: x = "x du module"
       class A():
           print "dans classe A: " + x
           x = "x dans A"
           print "dans classe A: " + x
           del x
           print "dans classe A: " + x
```

Alors pourquoi si c'est une mauvaise idée de mélanger variables globales et locales de même nom dans une fonction, c'est possible dans une classe ?

Cela vient de la manière dont sont implémentés les espaces de nommage. Normalement, un objet a pour espace de nommage un dictionnaire qui s'appelle `__dict__`. D'un côté un dictionnaire est un objet python qui offre beaucoup de flexibilité, mais d'un autre côté, il induit un petit surcoût pour chaque recherche d'éléments. Comme les fonctions sont des objets qui par définition peuvent être appelés très souvent, il a été décidé de mettre toutes les variables locales à la fonction dans un objet écrit en C qui n'est pas dynamique (on ne peut pas ajouter des éléments à l'exécution), mais qui est un peu plus rapide qu'un dictionnaire lors de l'accès aux variables. Mais pour faire cela, il faut déterminer la portée de la variable dans la phase de précompilation. Donc si le précompilateur trouve une affectation (explicite ou implicite) dans une fonction, il considère la variable comme locale pour tout le bloc de code. Donc si on référence une variable définie comme étant locale avant une affectation dans la fonction, on ne va pas la chercher globalement, on a une erreur `UnboundLocalError`.

Cette optimisation n'a pas été faite pour les classes, parce que dans l'évaluation du compromis souplesse contre efficacité pour les classes, c'est la souplesse, donc le dictionnaire qui a gagné.

Le bug 532860 Ce bug n'apparait que si vous ne suivez pas la première règle. On ne détaillera pas plus ce bug ici, bug qui ne sera jamais corrigé, mais si vous êtes intéressés, vous pouvez lire le bug report et une intéressante discussion liée à ce bug.

Directive `global` au début du bloc de code La deuxième règle est que la directive `global` doit toujours être au début du bloc de code. Regardons ce qu'il se passe si ça n'est pas le cas

```
In []: x = 'x dans le module'

       def f():
           x = 'x dans f, première affectation'
           global x

       f()
       print x
```

Alors que `global` est après l'affectation, on voit que la variable globale `x` a bien été modifiée. En fait, `global` n'est pas une instruction, mais une directive au précompilateur. De même que le précompilateur détermine qu'une variable affectée dans une fonction est locale, s'il trouve la directive `global`, il déterminera que cette variable est globale pour tout le bloc de code.

C'est source de confusion et d'erreurs, pour cette raison, dans les dernières versions de python 2, si une directive `global` arrive après une affectation locale de la variable, il y a une exception `SyntaxWarning`.

Pour finir, rappelons que l'utilisation de la directive `global` est fortement déconseillée puisqu'elle rend implicite les communications entre espaces de nommage. Il faut à la place toujours favoriser les passages d'arguments et les retours de fonctions. Souvenons-nous du Zen de python...

```
In []: import this
```

Et en particulier

```
In []: zens = ("".join([this.d.get(x, x) for x in this.s])).split('\n')
       print zens[3]
       print zens[-1]
```

Et pour ceux qui se demandent pourquoi il faut une expression compliquée pour sortir deux phrases du Zen de python, c'est parce que le zen de python se mérite :)