W5-S5-C3-Complement-heritage-multiple

December 15, 2014

1 Héritage multiple

1.1 Complément - niveau avancé

1.1.1 Avertissement - toutes les classes concernées sont *new-style*

Toutes les classes considérées dans ce complément sont supposées *new-style*. Pour les classes classiques, le mécanisme d'héritage multiple est plus rudimentaire, vous en trouverez le détail dans la référence au blog de Guido van Rossum (voir dernière section).

Il est, à nouveau, très fortement **conseillé d'utiliser des classes *new-style*** si vous écrivez du nouveau code, car c'est la seule sorte de classe qui reste disponible en python3.

1.1.2 Rappels

L'héritage en python consiste principalement en l'algorithme de recherche d'un attribut d'une instance; celui-ci regarde : 1. d'abord dans l'instance, 1. ensuite dans la classe, 1. ensuite dans les super-classes.

1.1.3 Le problème

Les deux premiers points ne posent pas de problème de définition, puisque l'objet lui-même et sa classe sont **définis de manière unique**.

Par contre, lorsqu'on utilise l'héritage multiple, on peut imaginer trouver l'attribut recherché dans **plusieurs superclasses**. Il est donc important de préciser, dans le cas où plusieurs superclasses possèdent l'attribut recherché, quel est celui qui doit être retenu.

1.1.4 Ordre sur les superclasses

Le problème revient donc à définir un **ordre** sur l'ensemble des **super-classes**. On parle bien, naturellement, de **toutes** les super-classes, pas seulement celles dont on hérite directement - en termes savants on dirait qu'on s'intéresse à la fermeture transitive de la relation d'héritage ou à l'arbre de recouvrement du graphe d'héritage.

Après quelques errements (décrits dans la première référence de la dernière section, "Pour en savoir plus"), l'algorithme qui a été choisi pour déterminer l'ordre des superclasses, qui est en service depuis la version 2.3, est connu sous le nom de **linéarisation C3**. Cet algorithme n'est pas propre à python, comme vous pourrez le lire dans les références citées dans la dernière section.

Nous ne décrirons pas ici l'algorithme lui-même dans le détail ; par contre nous allons : * dans un premier temps résumer **les raisons** qui ont guidé ce choix, en décrivant les bonnes propriétés que l'on attend, ainsi que les **limitations** qui en découlent; * puis voir l'ordre obtenu sur quelques exemples concrets de hiérarchies de classes.

Vous trouverez dans les références (voir ci-dessous la dernière section, "Pour en savoir plus") des liens vers des documents plus techniques si vous souhaitez creuser le sujet.

1.1.5 Les bonnes propriétés attendues

Il y a un certain nombre de bonnes propriétés que l'on attend de cet algorithme.

Priorité au spécifique Lorsqu'une classe A hérite d'une classe B, on s'attend à ce que les méthodes définies sur A, qui sont en principe plus spécifiques, soient utilisées de préférence à celles définies sur B.

Priorité à gauche Lorsqu'on utilise l'héritage multiple, on mentionne les classes mères dans un certain ordre, qui n'est pas anodin. Les classes mentionnées en premier sont bien entendu celles desquelles on veut hériter en priorité.

De manière un peu plus formelle Pour reformuler les deux points ci-dessus d'une manière un peu plus formelle : * on se donne un objet o de la classe O * on considère l'ensemble $\mathcal{S}$ des superclasses de O (qui contient O), * et on cherche à construire la MRO (Method Resolution Order), c'est-à-dire la liste $\mathcal{MRO}$ ordonnée des éléments de $\mathcal{S}$ dans lesquelles chercher les attributs de o.

Les deux règles ci-dessus s'écrivent alors comme ceci :

- $\forall A, B \in \mathcal{S}$, A hérite de B $\Rightarrow$ A est avant B dans $\mathcal{MRO}$
- $\forall B, C, D \in \mathcal{S}$, B hérite de C puis de D $\Rightarrow$ B est avant C qui est avant D dans $\mathcal{MRO}$

La formule "B hérite de C puis de D" est à prendre ici au sens large; B peut hériter de plus de deux classes, tout ce qui est demandé est que C apparaisse avant D dans la liste des classes mères.

Limitations : toutes les hiérarchies ne peuvent pas être traitées L'algorithme C3 permet de calculer un ordre sur $\mathcal{S}$ qui respecte toutes ces contraintes, lorsqu'il en existe un. En effet, dans certains cas on ne peut pas trouver un tel ordre. Nous en verrons un exemple vers la fin de ce document ; comme vous le verrez on peut exhiber de tels exemples qui restent relativement simples. Cependant, dans la pratique, il est relativement rare de tomber sur de tels cas pathologiques ; et lorsque cela se produit c'est en général le signe d'erreurs de conception plus profondes.

1.1.6 Un exemple très simple

On se donne la hiérarchie suivante.

```
In []: class LeftTop(object):
           def attribut(self):
               return "attribut(LeftTop)"

       class LeftMiddle(LeftTop):
           pass

       class Left(LeftMiddle):
           pass

       class Middle(object):
           pass

       class Right(object):
           def attribut(self):
               return "attribut(Right)"

       class Class(Left, Middle, Right):
           pass

       instance = Class()
```

qui donne en version dessinée, avec les deux points représentant les deux définitions de la méthode `attribut`:

Les deux règles, telles que nous les avons énoncées en premier lieu (priorité à gauche, priorité au spécifique) sont un peu contradictoires ici. En fait, c'est la méthode de `LeftTop` qui est héritée dans `Class`, comme on le voit ici :

```
In []: print instance.attribut() == "attribut(LeftTop)"
```

Exercice Remarquez qu'ici `Right` a elle-même un héritage très simple. À titre d'exercice, modifiez le code ci-dessus pour remplacer dans l'héritage de `Right` la classe `object` par `LeftMiddle`; de quelle classe d'après vous est-ce que `Class` hérite `attribut` dans cette configuration ?

Si cela ne vous convient pas C'est une évidence, mais cela va peut-être mieux en le rappelant : si la méthode que vous obtenez "gratuitement" avec l'héritage n'est pas celle qui vous convient, vous avez naturellement toujours la possibilité de la redéfinir, et ainsi d'en **choisir** une autre. Ainsi dans notre exemple si on préfère la méthode implémentée dans `Right`, on définira plutôt la classe `Class` comme ceci :

```
In []: class Class(Left, Middle, Right):
           def attribut(*args, **kwds):
               return Right.attribut(*args, **kwds)

       instance2 = Class()
       instance2.attribut()
```

Ou encore bien entendu, si dans votre contexte vous devez appelez **les deux** méthodes dont vous pourriez hériter et les combiner, vous pouvez le faire aussi, par exemple comme ceci :

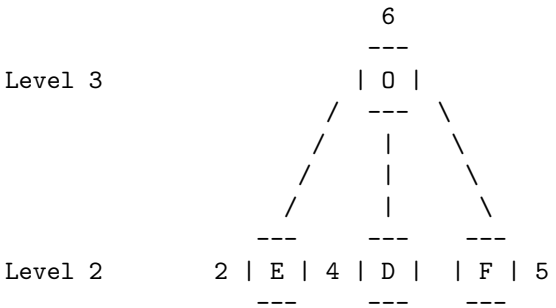
```
In []: class Class(Left, Middle, Right):
           def attribut(*args, **kwds):
               return LeftTop.attribut(*args, **kwds) + " ** " + Right.attribut(*args, **kwds)

       instance3 = Class()
       instance3.attribut()
```

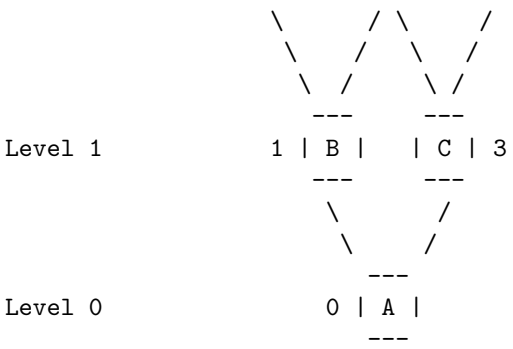
1.1.7 Un exemple un peu plus compliqué

Voici un exemple tiré de la deuxième référence (voir ci-dessous la dernière section, "Pour en savoir plus").

```
In []: O = object
       class F(O): pass
       class E(O): pass
       class D(O): pass
       class C(D, F): pass
       class B(E, D): pass
       class A(B, C): pass
```

Cette hiérarchie nous donne, en partant de A, l'ordre suivant





Que l'on peut calculer, sous l'interpréteur python, avec la méthode `mro` sur la classe de départ :

```
In []: A.mro()
```

1.1.8 Un exemple qui ne peut pas être traité

Voici enfin un exemple de hiérarchie pour laquelle on ne **peut pas trouver d'ordre** qui respecte les bonnes propriétés que l'on a vues tout à l'heure, et qui pour cette raison sera **rejetée par l'interpréteur python**. D'abord en version dessinée

```
In []: # en version code
       class X(object): pass
       class Y(object): pass
       class XY(X, Y): pass
       class YX(Y, X): pass

       # on essaie de créer une sous-classe de XY et YX
       try:
           class Class(XY,YX): pass
       # mais ce n'est pas possible
       except Exception as e:
           print "On ne peut pas créer Class"
           print e
```

1.1.9 Pour en savoir plus

1. Un blog de Guido Van Rossum qui retrace l'historique des différents essais qui ont été faits avant de converger sur le modèle actuel.
2. Un article technique qui décrit le fonctionnement de l'algorithme de calcul de la MRO, et donne des exemples.
3. L'article de wikipedia sur l'algorithme C3.