W5-S4-C4-Complement-attributs-classe-instance

December 14, 2014

1 Attributs de classe et attributs d'instance

1.1 Complément - niveau intermédiaire

Nous avons vu jusqu'à présent que l'on peut ajouter des attributs à toutes sortes d'objets en python, et notamment à * un module, * une fonction, * une classe, * une instance de classe.

Nous avons vu également, si vous vous souvenez de la classe `Matrix2`, qu'**en règle générale** : * une *méthode* est un *attribut de la classe*, * et que les *données* qui décrivent l'objet sont rangées dans des *attributs de l'instance*.

1.1.1 Propos de ce complément

Dans ce complément, nous allons approfondir les notions d'attributs de classe et d'attributs d'instance.

Le **premier point** que nous voulons illustrer ici est que le langage **ne fait pas la différence** entre un attribut dont la valeur est du code (une fonction ou une méthode), et un attribut contenant des données.

Le **second point** est qu'un attribut est cherché **en premier dans l'instance puis dans la classe**.

Pour illustrer tout ceci nous allons voir que le langage permet également : * d'attacher des *données à une classe* - pour définir par exemple une valeur par défaut valable pour toutes les instances de la classe, ou de * de *définir une méthode sur une instance* - pour spécialiser un comportement pour un seul objet, et ainsi éviter de définir une nouvelle classe pour un seul objet.

Nous démontrons ce second usage, rare en pratique, dans la dernière partie de ce complément qui est de niveau avancée.

1.1.2 Un attribut de donnée défini sur la classe

Voyons, pour commencer, un exemple de classe avec un attribut de données qui est en fait un attribut qui référence un objet `builtin` contenant des données, comme par exemple `int`, `list` ou `str`.

```
In []: class Spam:
           attribut = "attribut de classe"
```

Naturellement, on aurait pu aussi définir des méthodes dans cette classe, mais nous avons choisi de montrer un exemple très simple.

La classe `Spam` possède donc maintenant l'attribut `attribut` qui vaut

```
In []: Spam.attribut
```

Créons à présent une instance de cette classe :

```
In []: # une instance spéciale de Spam - on va lui attacher un attribut
       special = Spam()
```

On peut naturellement attacher à cette instance un attribut `attribut`, comme on l'a déjà vu :

```
In []: # on affecte l'attribut de l'instance
       special.attribut = "attribut de l'instance"

       # naturellement on retrouve cette valeur quand
       # on cherche l'attribut à partir de l'instance
       print 'special:', special.attribut
```

Le point important de ce complément, c'est qu'une **instance** à laquelle on n'a pourtant pas attaché d'attribut `attribut` peut tout de même **référencer** cet attribut et **trouver celui de la classe** comme si c'était le sien:

```
In []: # une autre instance de Spam
       defaut = Spam()

       # par contre on ne lui affecte *pas* d'attribut d'instance
       # mais on peut *tout de même* chercher l'attribut
       print 'defaut:', defaut.attribut
```

1.1.3 Discussion

En fait, on a déjà vu ce mécanisme en action; c'est exactement la même chose qui se passe lorsqu'on a : * une classe qui définit la méthode `foo` * et une instance `obj` de la classe sur laquelle on appelle la méthode en faisant

```
obj.foo()
```

Le mécanisme de recherche d'un attribut sur une instance est le même, que cet attribut représente une méthode ou une donnée. En effet, en python tout est un objet et un attribut peut référencer n'importer quel objet. Ça n'est que lorsque l'interpréteur python accède à l'objet qu'il peut finalement connaître son type.

1.1.4 Conclusion

Comme on l'a vu, le mécanisme d'attributs, bien qu'extrêmement simple, est très souple et très puissant. Le langage ne fait pas de différence entre attributs de données et de méthodes puisqu'un attribut référence n'importe quel objet et qu'en python tout est en objet. On peut attacher, au choix, **à une instance ou à une classe**, des attributs représentant **n'importe quel objet**, et la recherche de ces attributs se fait dans l'ordre **instance** puis **classe**.

Nous verrons dans la prochaine vidéo que les mécanismes d'héritage ne font que prolonger ce mécanisme de recherche d'attributs.

1.2 Complément - niveau avancé

1.2.1 Un attribut de méthode défini sur une instance

Définir un attribut de données dans une classe, comme on vient de le voir, présente un intérêt pratique; il est parfois commode de définir une constante, ou une valeur par défaut, au niveau de la classe, qui s'applique alors à tous les objets.

Nous allons à présent illustrer la possibilité de définir une méthode au niveau d'une instance. Comme ceci est rare en pratique, il s'agit ici de bien comprendre les mécanismes du langage plutôt que de découvrir une technique de programmation. Pour cela nous prenons à nouveau une classe jouet.

```
In []: class Eggs:
           def __init__(self, name):
               self.name = name
```

Et nous créons à nouveau deux instances, une qui va se comporter comme la classe, et une qui aura un comportement spécial

```
In []: defaut = Eggs('defaut')
       special = Eggs('special')

       print 'defaut', defaut, 'special', special
```

On va maintenant attacher à la classe une méthode `__repr__` pour redéfinir comment imprimer les objets de la classe `Eggs`

```
In []: # on redéfinit la méthode __repr__ sur la classe 'Eggs'
       def repr_eggs(self):
           return "<<Eggs instance, name={}>>".format(self.name)
       Eggs.__repr__ = repr_eggs

       # à présent les deux instances s'impriment selon ce format
       print 'defaut', defaut, 'special', special
```

Jusqu'ici rien de nouveau.

Nous allons maintenant attacher une méthode `__repr__` **à une instance** en propre. C'est ici que les choses se compliquent un tout petit peu. Nous avons deux choix pour cela.

Première technique La première technique consiste à définir une fonction **sans argument**. C'est surprenant car nous avons vu jusqu'ici que `__repr__` s'attend à recevoir **un argument**.

Voyons d'abord le code

```
In []: # on définit une fonction, sans argument donc
       def repr_special_sans_argument():
           return "<<SPECIAL (technique sans argument)>>"

       # et on l'attache à l'instance
       special.__repr__ = repr_special_sans_argument

       # à présent on a changé le format d'impression de special
       print 'defaut', defaut, 'special', special
```

Avant d'aller plus loin, souvenez-vous qu'avant de parler de classes on a parlé de fonctions; on pourrait très bien avoir envie de ranger dans un attribut une vraie fonction et de pouvoir l'appeler comme une fonction, mais pas comme une méthode, c'est-à-dire sans mettre en œuvre la *magie* qui consiste à mettre l'instance elle-même comme premier argument.

Il faut donc pouvoir différencier entre le cas d'une fonction simple et le cas d'une méthode à laquelle l'interpréteur passe automatiquement une référence de l'instance comme premier argument.

C'est exactement ce que l'on fait ici : la fonction que nous avons attachée à l'instance n'est pas de même nature que celle attachée à la classe; voyons ces deux objets :

```
In []: print "sur la classe", Eggs.__repr__
       print "sur l'instance", special.__repr__
```

On voit que sur la classe, l'attribut `__repr__` est un objet de type `unbound method`, alors que sur l'instance c'est un objet de type `function` tout bête.

Et c'est sur cette différence que se base l'interpréteur pour ajouter, ou non, l'objet en premier argument de l'appel.

Deuxième technique Si on veut se mettre dans un cadre plus conforme à ce qui est habituel, il faut dire explicitement que la fonction est en fait une méthode de l'instance. On commence par écrire une fonction qui prend bien **un argument** :

```
In []: # on définit la métode d'impression sur l'instance 'special'
       def repr_special_avec_argument(self):
           return "<<SPECIAL {} (technique avec argument)".format(self.name)
```

Mais au lieu de l'attacher telle quelle on la convertit en méthode en définissant un nouvel objet de type `MethodType` :

```
In []: import types
       special.__repr__ = types.MethodType(repr_special_avec_argument, special)
```

après quoi on verra cet objet s'afficher différemment des autres

```
In []: special
```

Et la valeur de l'attribut `__repr__` est bien maintenant vue comme une méthode et non comme une fonction :

```
In []: print "sur l'instance", special.__repr__
```

Pour en savoir plus Ce sujet est discuté plus en profondeur dans cet article de stackoverflow