Héritage

Complément - niveau basique

La notion d'héritage, qui fait partie intégrante de la Programmation Orientée Objet, permet principalement de maximiser la **réutilisabilité**.

Nous avons vu dans la vidéo les mécanismes d'héritage ***in abstracto***. Pour résumer très brièvement, on recherche un attribut (souvent une méthode) à partir d'une instance en cherchant :

- d'abord dans l'instance elle-même,
- puis dans la classe de l'instance,
- puis dans les super-classes de la classe.

L'objet de ce complément est de vous donner, d'un point de vue plus appliqué, des idées de ce qu'on peut faire ou non avec ce mécanisme. Le sujet étant assez rabâché par ailleurs, nous nous concentrerons sur deux points :

- les pratiques de base utilisées pour la conception de classes, et notamment pour bien distinguer **héritage** et **composition**;
- nous verrons ensuite, sur des **exemples extraits de code réel**, comment ces mécanismes permettent en effet de contribuer à la réutilisabilité du code.

Plusieurs formes d'héritage

Une méthode héritée peut être rangée dans une de ces trois catégories :

- *implicite* : si la classe fille ne définit pas du tout la méthode,
- *redéfinie* : si on récrit la méthode entièrement,
- *modifiée* : si on récrit la méthode dans la classe fille, mais en utilisant le code de la classe mère.

Commençons par illustrer tout ceci sur un petit exemple :

```
# Une classe mère
class Fleur(object):
    def implicite(self):
        print 'Fleur.implicite'
    def redefinie(self):
        print 'Fleur.redefinie'
    def modifiee(self):
        print 'Fleur.modifiee'
```

```
# Une classe fille
class Rose(Fleur):
    # on ne definit pas implicite
    # on redefinit complement redefinie
    def redefinie(self):
        print 'Rose.redefinie'
    def modifiee(self):
        Fleur.modifiee(self)
        print 'Rose.modifiee apres Fleur'
```

Nous avons fait hériter la classe `Fleur` de la classe `object`, qui est la pratique recommandée; cela fait de toutes nos classes des ***classes new-style***, un concept que nous allons étudier dans un tout prochain complément.

On peut à présent créer une instance de Rose et appeler sur cette instance les trois méthodes.

```
# fille est une instance de Rose
fille = Rose()

fille.implicite()


fille.redefinie()
```

S'agissant des deux premières méthodes, le comportement qu'on observe est simplement la conséquence de l'algorithme de recherche d'attributs : `implicite` est trouvée dans la classe Fleur et `redefinie` est trouvée dans la classe Rose.

```
fille.modifiee()
```

Pour la troisième méthode, attardons nous un peu car on voit ici comment ***combiner*** facilement le code de la classe mère avec du code spécifique à la classe fille, en appelant explicitement la méthode de la classe mère lorsqu'on fait :

```
Fleur.modifiee(self)
```

La fonction *builtin* `super()`

Signalons à ce sujet, pour être exhaustif, l'existence de la fonction *builtin* `super()` (https://docs.python.org/2/library/functions.html#super) qui permettrait de réaliser la même chose sans nommer explicitement la classe mère, comme on le fait ici :

```
# Une version allégée de la classe fille, qui utilise super()
class Rose(Fleur):
    def modifiee(self):
        super(Rose, self).modifiee()
        print 'Rose.modifiee apres Fleur'


fille = Rose()
```

```
fille.modifiee()
```

On peut envisager d'utiliser `super()` dans du code très abstrait où on ne sait pas déterminer statiquement le nom de la classe dont il est question. Mais, c'est une question de goût évidemment, je recommande personnellement la première forme, où on qualifie la méthode avec le nom de la classe mère qu'on souhaite utiliser. En effet dans la grande majorité des cas :

- on sait déterminer le nom de la classe dont il est question,
- ou alors on veut mélanger plusieurs méthodes héritées (via l'héritage multiple, dont on va parler dans un prochain complément) et dans ce cas `super()` ne peut rien pour nous.

Héritage *vs* Composition

Dans le domaine de la conception orientée objet, on fait la différence entre deux constructions, l'héritage et la composition, qui à une analyse superficielle peuvent paraître procurer des résultats similaires, mais qu'il est important de bien distinguer.

Voyons d'abord en quoi consiste la composition et pourquoi le résultat est voisin :

```
# Une classe avec qui on n'aura pas de relation d'héritage
class Tige(object):
    def implicite(self):
        print 'Tige.implicite'
    def redefinie(self):
        print 'Tige.redefinie'
    def modifiee(self):
        print 'Tige.modifiee'

# on n'hérite pas
# mais on fait ce qu'on appelle un composition
# avec la classe Tige
class Rose(object):
    # mais pour chaque objet de la classe Rose
    # on va créer un objet de la classe Tige
    # et le conserver dans un champ
    def __init__(self):
        self.externe = Tige()

    # le reste est presque comme tout à l'heure
    # sauf qu'il faut definir implicite
    def implicite(self):
        self.externe.implicite()

    # on redefinit complement redefinie
    def redefinie(self):
        print 'Rose.redefinie'

    def modifiee(self):
        self.externe.modifiee()
        print 'Rose.modifiee apres Tige'
```

```
# on obtient ici exactement le même comportement pour les trois sortes de méthodes
fille = Rose()

fille.implicite()
fille.redefinie()
fille.modifiee()
```

Comment choisir ?

Alors, quand faut-il utiliser l'héritage et quand faut-il utiliser la composition ? On arrive ici à la limite de notre cours, il s'agit plus de conception que de codage à proprement parler, mais pour donner une réponse très courte à cette question :

- on utilise l'héritage lorsqu'un objet de la sous-classe **est aussi un** objet de la super-classe (une rose est aussi une fleur), et
- on utilise la composition lorsqu'un objet de la sous-classe **a une relation avec** un objet de la super-classe (une rose possède une tige, mais c'est un autre objet).

Complément - niveau intermédiaire

Des exemples de code

Sans transition, dans cette section un peu plus prospective, nous vous avons signalé quelques morceaux de code de la librairie standard qui utilisent l'héritage. Sans aller nécessairement jusqu'à la lecture de ces codes, il nous a semblé intéressant de commenter spécifiquement l'usage qui est fait de l'héritage dans ces bibliothèques.

Le module `calendar`

On trouve dans la librairie standard le module `calendar` (https://docs.python.org/2/library/calendar.html) . Ce module expose deux classes `TextCalendar` et `HTMLCalendar`. Sans entrer du tout dans le détail, ces deux classes permettent d'imprimer dans des formats différents le même type d'informations du type rendez-vous.

Le point ici est que les concepteurs ont choisi un graphe d'héritage comme ceci :

```
Calendar
|-- TextCalendar
|-- HTMLCalendar
```

qui permet de grouper le code concernant la logique dans la classe `Calendar`, puis dans les deux sous-classes d'implémenter le type de sortie qui va bien.

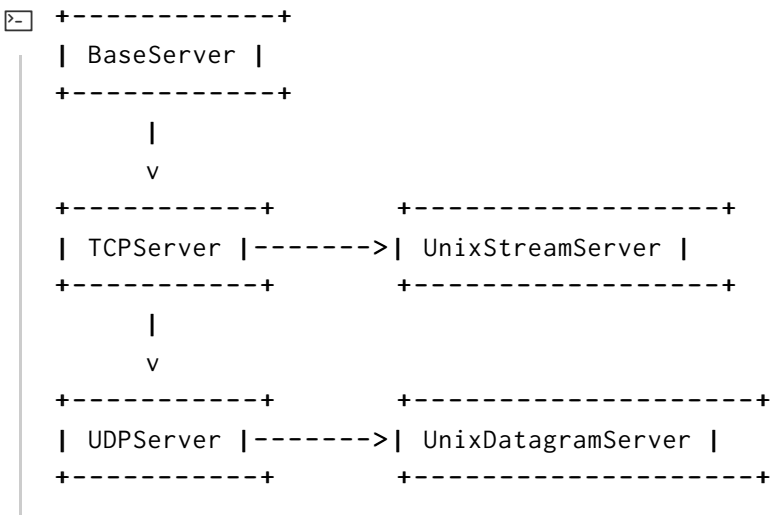
De cette manière, le maximum de code est partagé entre les deux classes; et de plus si vous avez besoin d'une sortie au format, disons PDF, vous pouvez envisager d'hériter de `Calendar` et n'implémenter que la partie spécifique au format PDF.

C'est un peu le niveau élémentaire de l'héritage.

Le module `SocketServer`

Toujours dans la librairie standard, le module `SocketServer` (https://docs.python.org/2/library/socketserver.html) - qui, incidemment est écrit en C - fait un usage beaucoup plus sophistiqué de l'héritage.

Le module propose une hiérarchie de classes comme ceci:



Ici encore notre propos n'est pas d'entrer dans les détails, mais d'observer le fait que les différents ***niveaux d'intelligence*** sont ajoutés les uns aux les autres au fur et à mesure que l'on descend le graphe d'héritage.

Cette hiérarchie est par ailleurs étendue par le module `BaseHTTPServer` (https://docs.python.org/2/library/basehttpserver.html) et notamment au travers de la classe `HTTPServer` (https://docs.python.org/2/libra ry/basehttpserver.html#BaseHTTPServer.HTTPServer) qui hérite de `TCPServer`.

Pour revenir au module `SocketServer`, j'attire votre attention dans la page d'exemples (https://docs.python.org/2/library/socketserver.html#examples) sur cet exemple en particuler (https://docs.python.org/2/library/socketserver.html#asynchronous-mixins) , où on crée une classe de serveurs multi-threads (la classe `ThreadedTCPServer`) par simple héritage multiple entre `Threading Mixin` et `TCPServer`. La notion de `Mixin` est décrite dans cette page wikipédia (http://en.wikipedia.org/wiki/Mixin) dans laquelle vous pouvez accéder directement à la section consacrée à python (http://en.wikipedia.org/wiki/Mixin#In_Python) .