Introduction aux classes

Complément - niveau basique

On définit une classe lorsqu'on a besoin de créer un type spécifique au contexte de l'application. Il faut donc voir une classe au même niveau qu'un type ***builtin*** comme `list` ou `dict`.

Un exemple simpliste

Par exemple, imaginons qu'on a besoin de manipuler des matrices 2×2

$A = \left(\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array} \right)$

Et en guise d'illustration, nous allons utiliser le déterminant; c'est juste un prétexte pour implémenter une méthode sur cette classe, ne vous inquiétez pas si le terme ne vous dit rien, ou vous rappelle de mauvais souvenirs. Tout ce qu'on a besoin de savoir c'est que, sur une matrice de ce type, le déterminant vaut

$ det (A) = a_{11}.a_{22} - a_{12}.a_{21}$

Dans la pratique, on utiliserait la classe `matrix` de `numpy` (http://www.numpy.org/) qui est une librairie de calcul scientifique très populaire et largement utilisée. Mais comme premier exemple de classe, nous allons écrire **notre propre classe `Matrix2`** pour voir les mécanismes de base des classes de python en action. Naturellement, il s'agit d'une implémentation jouet.

```
class Matrix2:
    "Une implémentation sommaire de matrice carrée 2x2"

    def __init__(self, a11, a12, a21, a22):
        "construit une matrice à partir des 4 coefficients"
        self.a11 = a11
        self.a12 = a12
        self.a21 = a21
        self.a22 = a22

    def determinant(self):
        return self.a11 * self.a22 - self.a12 * self.a21
```

La première version de `Matrix2`

Une classe peut avoir un *docstring*

Pour commencer, vous remarquez qu'on peut attacher à cette classe un *docstring* comme pour

les fonctions

```
help(Matrix2)
```

La classe définit donc deux méthodes, nommées `__init__` et `determinant`.

La méthode `__init__`

La méthode **`__init__`**, comme toutes celles qui ont un nom en *__nom__*, est une **méthode spéciale**. En l'occurrence, il s'agit de ce qu'on appelle le **constructeur** de la classe, c'est-à-dire le code qui va être appelé lorsqu'on crée une instance. Voyons cela tout de suite sur un exemple.

```
matrice = Matrix2(1, 2, 2, 1)
print matrice
```

Vous remarquez tout d'abord que `__init__` s'attend à recevoir ***5 arguments***, mais que nous appelons `Matrix2` avec seulement ***4 arguments***.

L'argument surnuméraire, le **premier** de ceux qui sont déclarés dans la méthode, correspond à l'**instance qui vient d'être créée** et qui est automatiquement passée par l'interpréteur python à la méthode `__init__`. En ce sens, le terme constructeur est impropre puisque la méthode `__init__` ne crée pas l'instance, elle ne fait que l'initialiser. Nous reviendrons sur le processus de création des objets lorsque nous parlerons des métaclasses en dernière semaine.

La **convention** est de nommer le premier argument de ce constructeur **`self`**, nous y reviendrons un peu plus loin.

On voit également que le constructeur se contente de mémoriser, à l'intérieur de l'instance, les arguments qu'on lui passe, sous la forme d'**attributs** de l'**instance** `self`.

C'est un cas extrêmement fréquent; de manière générale, il est recommandé d'écrire des constructeurs passifs de ce genre; dit autrement, on évite de faire trop de traitements dans le constructeur.

La méthode `determinant`

La classe définit aussi la méthode `determinant`, qu'on utiliserait comme ceci

```
print "Determinant =", matrice.determinant()
```

Vous voyez que la **syntaxe** pour appeler une méthode sur un objet est **identique** à celle que nous avons utilisée jusqu'ici avec **les types de base**. Nous verrons dans quelques séquences comment on peut pousser beaucoup plus loin la similitude, pour pouvoir par exemple calculer la **somme** de deux objets de la classe `Matrix2` avec l'opérateur `+`, mais n'anticipons pas.

Vous voyez aussi que, ici encore, la méthode définie dans la classe attend ***1 argument*** **`self`**, alors qu'apparemment nous ne lui en passons **aucun**. Comme tout à l'heure avec le constructeur, le premier argument passé automatiquement par l'interpréteur python à `determinant` est l'objet `matrice` lui-même.

En fait on aurait pu aussi bien écrire, de manière parfaitement équivalente

```
print "Determinant =", Matrix2.determinant(matrice)
```

qui n'est presque jamais utilisé en pratique, mais qui illustre bien ce qui se passe lorsqu'on invoque une méthode sur un objet. En réalité, lorsque l'on écrit `matrice.determinant()` l'interpréteur python va essentiellement convertir cette expression en `Matrix2.determinant(matrice)`.

Complément - niveau intermédiaire

À quoi ça sert ?

Ce cours n'est pas consacré à la Programmation Orientée Objet (OOP) en tant que telle. Voici toutefois quelques-uns des avantages qui sont généralement mis en avant.

- Encapsulation
- Résolution dynamique de méthode
- Héritage

Encapsulation

L'idée de la notion d'encapsulation consiste à ce que

- une classe définit son **interface**, c'est-à-dire les méthodes par lesquelles on peut utiliser ce code,
- mais reste tout à fait libre de modifier son **implémentation**, et tant que cela n'impacte pas l'interface, **aucun changement** n'est requis dans les **codes utilisateurs**.

Nous verrons plus bas une deuxième implémentation de `Matrix2` qui est plus générale que notre première version, mais qui utilise la même interface, donc qui fonctionne exactement de la même manière pour le code utilisateur.

La notion d'encapsulation peut paraître à première vue banale; il ne faut pas s'y fier, c'est de cette manière qu'on peut efficacement découper un gros logiciel en petits morceaux indépendants, et réellement découplés les uns des autres, et ainsi gérer la complexité.

La programmation objet est une des techniques permettant d'atteindre cette bonne propriété. Il faut reconnaître que certains langages comme Java et C++ ont des mécanismes plus sophistiqués, mais aussi plus complexes, pour garantir une bonne étanchéité entre l'interface publique et les détails d'implémentation. Les choix faits en la matière en python reviennent, une fois encore, à privilégier la simplicité.

Aussi, il n'existe pas en python l'équivalent des notions d'interface `public`, `private` et `protected` qu'on trouve en C++ et en Java. Il existe tout au plus une convention, selon laquelle les attributs commençant par un underscore (le tiret bas _) sont privés et ne ***devraient*** pas être utilisés par un code tiers, mais le langage ne fait rien pour garantir le bon usage de cette convention.

Si vous désirez creuser ce point nous vous conseillons de lire

- [*Reserved classes of identifiers*] (https://docs.python.org/2/reference/lexical_analysis.html (https://docs.python.org/2/reference/lexical_analysis.html)

reserved-classes-of-identifiers) où l'on décrit également les noms privés à une

classe (les noms de variables en *__nom*),

- *Private Variables and Class-local References* (https://docs.python.org/2/tutorial/classes.html#tut-private) , qui en donne une illustration.

Malgré cette simplicité revendiquée, les classes de python permettent d'implémenter en pratique une encapsulation tout à fait acceptable, on peut en juger rien que par le nombre de librairies tierces existantes dans l'écosystème python.

Résolution dynamique de méthode

Le deuxième atout de OOP, c'est le fait que l'envoi de méthode est résolu à "run-time" et non pas à "compile-time". Ceci signifie qu'on peut écrire du code générique, qui pourra fonctionner avec des objets non connus ***a priori***. Nous allons en voir un exemple tout de suite, en redéfinissant le comportement de `print` sur nos objets de type `Matrix2`, dans la deuxième implémentation de `Matrix2`.

Héritage

L'héritage est le concept qui permet de :

- dupliquer une classe presque à l'identique, mais en redéfinissant une ou quelques méthodes seulement (héritage simple);
- composer plusieurs classes en une seule, pour réaliser en quelque sorte l'union des propriétés de ces classes (héritage multiple).

Illustration

Nous revenons sur l'héritage dans la prochaine vidéo. Dans l'immédiat, nous allons voir une seconde implémentation de la classe `Matrix2`, qui illustre l'encapsulation et l'envoi dynamique de méthodes.

```
class Matrix2:
    """Une deuxième implémentation, tout aussi
    sommaire, mais différente, de matrice carrée 2x2"""

    def __init__(self, a11, a12, a21, a22):
        "construit une matrice à partir des 4 coefficients"
        # on décide d'utiliser un tuple plutôt qu'une liste
```

```
        self.a = (a11, a12, a21, a22)

    def determinant(self):
        return self.a[0] * self.a[3] - self.a[1] * self.a[2]

    def __repr__(self):
        return "<<mat-2x2 {}>>".format(self.a)
```

Pour une raison ou pour une autre, disons que l'on décide de remplacer les 4 attributs nommés `self.a11`, `self.a12`, etc., qui n'étaient pas très extensibles, par un seul attribut `a` qui regroupe tous les coefficients de la matrice dans un seul tuple.

Grâce à l'**encapsulation**, on peut continuer à utiliser la classe exactement de la même manière

```
matrice = Matrix2(1, 2, 2, 1)
print "Determinant =", matrice.determinant()
```

Et en prime, grâce à la **résolution dynamique de méthode**, et parce que dans cette seconde implémentation on a défini une autre méthode spéciale `__repr__`, nous avons maintenant une impression beaucoup plus lisible de l'objet `matrice`

```
print matrice
```

Ce format d'impression reste d'ailleurs valable dans l'impression d'objets plus compliqués, comme par exemple

```
composite = [matrice, None, Matrix2(1, 0, 0, 1)]
print 'composite=', composite
```

Cela est possible parce que le code de `print` envoie la méthode `__repr__` sur les objets qu'elle parcourt. Le langage fournit une façon de faire par défaut, comme on l'a vu plus haut pour l'impression d'une `Matrix2` avec la première implémentation; et en définissant notre propre méthode `__repr__` nous pouvons surcharger ce comportement, et définir notre format d'impression.

Nous reviendrons sur les notions de surcharge et d'héritage dans les prochaines séquences vidéos.

La convention d'utiliser `self`

Avant de conclure, revenons rapidement sur le nom `self` qui est utilisé comme nom pour le premier argument des méthodes habituelles (nous verrons en semaine 7 d'autres sortes de méthodes, les méthodes statiques et de classe, qui ne reçoivent pas l'instance comme premier argument).

Comme nous l'avons dit plus haut, le premier argument d'une méthode s'appelle **`self`** par ***convention***. Cette pratique est particulièrement bien suivie, mais ce n'est qu'une convention, en ce sens qu'on aurait pu utiliser n'importe quel identificateur; pour le langage `self` n'a aucun sens particulier, ce n'est pas un mot clé.

Ceci est à mettre en contraste avec le choix fait dans d'autres langages, comme par exemple en C++ où l'instance peut dans certains cas être référencée par le mot-clé `this`, qui n'est pas mentionné dans la signature de la méthode. En python, selon le manifeste, ***explicit is better than implicit***, c'est pourquoi on mentionne l'instance dans la signature, sous le nom `self`.