# Programmation orientée objets

*Synthèse de cours*

Version du 14-07-2016



## Introduction

Jusque dans les années 70, les langages informatiques (comme FORTRAN) distinguaient les données de leurs manipulations (celles-ci se faisant via des fonctions, des programmes). On pouvait constater deux limitations essentielles à cette situation : d'une part, les données étaient « simples » (une variable était de type entier ou flottant ou chaîne de caractères…) même si des langages comme PASCAL ou C autorisaient la création de données composites ; d'autre part, une fonction spécifiquement écrite pour un type de données (par exemple une fonction supprimant tous les blancs (éventuels) d'une chaîne de caractères) n'était pas étroitement associée à ce type.

Avec les objets, on associe étroitement, dans le cadre du concept d'« **objet** », les données et les manipulations fondamentales susceptibles d'être effectuées dessus : c'est la notion d'« **encapsulation** ». L'objet apparaît alors comme une boîte noire communiquant avec le monde extérieur via les manipulations pouvant être effectuées dessus. Les codes correspondant à ces manipulations ne sont pas accessibles directement (ils ont internes à la boîte noire). Les autres objets ne connaissent que la façon dont une manipulation donnée doit être déclenchée.

Comme exemples de langages orientés objets célèbres, on peut citer : ADA, SmallTalk, C++, Eiffel, Java, Python…

## Définitions et vocabulaire

Un objet comporte en général des données de différents types (on parle de « données composites ») que l'on appelle « **attributs** ».

Par exemple, on peut imaginer un objet correspondant à une carte à jouer. Un tel objet comporterait vraisemblablement deux attributs : un attribut de type chaîne de caractères pour la couleur de la carte et un attribut numérique pour sa valeur.

On agira sur l'objet via des « **méthodes** » (que nous avons appelées « manipulations » dans l'introduction).

C'est exactement ce qui se produit lorsque l'on utilise une syntaxe comme `L.append('hello')`. Dans cette syntaxe on déclenche la méthode `append` sur l'objet `L` qui est du type `list`. Cette méthode est une fonction et reçoit ici comme argument la chaîne de caractères `'toto'`. Elle ajoute à la fin de la liste `L` un nouvel élément correspondant à `'toto'`.

La description générique d'un certain type d'objet se fait en définissant une « **classe** ». Une classe doit donc être vue comme un modèle de l'objet correspondant.
Dans une telle définition (voir plus loin), on trouvera principalement :

- Les définitions des différents attributs.
- Les codes des méthodes agissant sur l'objet considéré.
- D'éventuelles fonctions requises par les méthodes précédentes.

La classe `int` va ainsi décrire ce qu'est un objet de type `int` et comment il est possible d'agir dessus.

Si l'on souhaite disposer d'une description (synthétique) d'une classe donnée, on peut utiliser, dans la console, la commande `help`. Par exemple, pour la classe `int`, on saisira :

```
help(int)
```

Tout objet Python d'un type donné est alors appelé une « **instance** » de la classe correspondante et le mécanisme consistant à créer un objet d'un type donné (i.e. correspondant à une classe donnée) s'appelle l'« **instanciation** ».

Par exemple, la syntaxe `x=3.141592` conduit à la création d'une variable (`x`) de type flottant qui est donc une instance de la classe `float`.

## Classe et éléments fondamentaux

Nous allons illustrer la construction d'une classe en Python en créant un « nouvel » objet correspondant à la notion de nombre complexe (dans Python, une telle classe existe déjà et s'appelle `complex`). Nous appellerons notre classe `cplx`.

Nous choisissons, pour créer un nouveau complexe, de fournir sa partie réelle et sa partie imaginaire que nous notons classiquement `x` et `y`.

La classe `cplx` est alors créée via la syntaxe suivante :

```
class cplx(self,x,y):
```

Ainsi, pour créer le complexe $z1 = 3 - 4i$, on utilisera simplement l'appel :

```
z1 = cplx(3,-4)
```

Mentionnons le fait que la syntaxe ci-dessus est celle d'une fonction mais que nous utilisons le nom de la classe elle-même pour construire (voir plus loin) un nouvel objet de ce type.

Cette syntaxe fait apparaître deux mots-clés : `class` et `self`.
Le premier est suffisamment explicite pour que nous ne revenions pas dessus.

Le second en revanche, appelle quelques commentaires.
Dans tout le contenu de la classe, le mot-clé `self` désignera de façon générique un objet correspondant à cette classe. Ainsi, la définition des attributs (voir ci-après) ainsi que toutes les méthodes et fonctions définies dans la classe pourront faire référence à l'objet générique correspondant via le mot-clé `self`.
Ce mot-clé est réservé, il fait partie intégrante du langage Python.

Les attributs de l'objet sont définis grâce à une méthode spéciale appelée « **constructeur** » et systématiquement notée `__init__`.
Par exemple, pour la classe `cplx`, on pourrait avoir :

```
def __init__(self,x,y):
    self.re = x
    self.im = y
```

Nous avons choisi des noms d'attributs simples correspondant aux notations mathématiques mais bien d'autres choix étaient possibles !

Avec l'exemple ci-dessus, on obtiendra les parties réelle et imaginaire de `z1` en utilisant respectivement les syntaxes : `z1.re` et `z1.im`.

Il convient désormais de noter un point important : si, au-delà du constructeur, une méthode ou une fonction doit faire usage de tout ou partie des attributs de l'objets, elle utilisera bien sûr les noms des attributs et non les noms des arguments fournis après le mot-clé `self` dans la syntaxe `class(self,…)`.

Se pose alors le « problème » de la représentation de nos objets !

Quand, par exemple, on tape puis valide avec la touche `Entrée` le nom d'une variable dans la console, celle-ci nous renvoie la valeur de la variable avec un affichage adapté (pensez aux listes). Lorsque l'on procède de la sorte, on déclenche en fait l'exécution d'une autre méthode spéciale notée `__repr__` et recevant comme seul argument `self`. On pourrait par exemple avoir :

```
def __repr__(self):
     return(str(self.re)+'+'+str(self.im)+'i'))
```

Telle que proposée, cette méthode est basique et ne tient pas compte des cas suivants :

- Partie réelle et/ou imaginaire nulle.
- Partie imaginaire négative.

A vous de la parfaire !

Une méthode permettant d'obtenir le module d'un complexe pourrait ressembler à :

```
def module(self):
     return((self.re**2+self.im**2)**0.5)
```

D'où, finalement, l'ébauche de notre classe `cplx` :

```
class cplx(self,x,y):

     def __init__(self,x,y):
          self.re = x
          self.im = y

     def __repr__(self):
          return(str(self.re)+'+'+str(self.im)+'i'))

     def module(self):
          return((self.re**2+self.im**2)**0.5)
```

## Surcharge d'opérateur

Supposons que l'on ait créé deux variables `z1` et `z2` de type `cplx`.

On peut avoir besoin de calculer la somme de ces deux complexes. On souhaiterait que la syntaxe d'un tel calcul soit aussi simple que : `z1+z2`.

Le symbole « + » est certes un symbole très général (utilisable avec les entiers, les flottants, les listes, les chaînes de caractères, …) MAIS Python n'a AUCUNE IDEE à priori de ce à quoi la syntaxe précédente peut bien correspondre lorsqu'il s'agit d'un nouvel objet ! Il va falloir le lui préciser dans la définition de la classe `cplx`.

Dans le cadre de l'addition, la méthode spéciale à utiliser est `__add__`.
En tant que fonction, elle recevra comme argument `self` et, disons, `z` (mais on peut parfaitement choisir un autre nom pour le deuxième argument).
En redéfinissant une telle méthode dans notre classe, on dit que l'on « **surcharge** l'opérateur + ».

Nous devons avoir pleinement conscience du fait que cette méthode nous permet d'additionner un complexe (`self`) et un autre objet (`z`) dans cet ordre. L'objet `z` pourra donc être de type entier, flottant ou complexe (notre méthode nous permettra de calculer `z1+32` ou `z1+3.141592`), tout autre type étant à priori à exclure. Par ailleurs, cette méthode ne permet pas d'additionner un complexe à un autre objet (voir plus loin la méthode `__radd__`) : elle ne peut nous permettre de calculer `2+z1` !

Voici un exemple de code pour la méthode `__add__` de notre classe `cplx` :

```
def __add__(self,z):
      if type(z) != cplx and type(z) != int and type(z) != float:
            return ('L\'addition est impossible !')
      else:
            if type(z) == cplx:
                  return cplx(self.re + z.re,self.im + z.im)
            else:
                  return cplx(self.re + z,self.im)
```

L'interpréteur Python lit les instructions de la gauche vers la droite. S'il rencontre un calcul de la forme `a+b` et que la variable `a` et du type `cplx`, il va y avoir exécution de la méthode `__add__` proposée ci-dessus, dans la classe `cplx`.

Mais que se passe-t-il si c'est la variable `b` qui est du type `cplx` et que la variable `a` est, par exemple, du type flottant ? L'interpréteur rencontre d'abord la première variable et la méthode `__add__` des flottants ne permet pas de lui ajouter un objet de type nouveau (un complexe en l'occurrence), il va alors rechercher dans la classe `cplx` la méthode `__radd__`. C'est cette méthode qui permet donc d'effectuer des additions entre des objets de type entier ou flottant et des objets de type complexe.

Voici un exemple de code pour la méthode `__radd__` de notre classe `cplx` :

```
def __radd__(self,z):
      if type(z) != cplx and type(z) != int and type(z) != float:
            return ('L\'addition est impossible !')
      else:
            return(self+z)
```

De façon équivalente, on écrira les méthodes :

- `__sub__` et `__rsub__` pour la soustraction.
- `__mul__` et `__rmul__` pour la multiplication.
- `__truediv__` et `__rtruediv__` pour la division.

## Héritage

Imaginons que nous ayons créé une classe `Poly` correspondant à un polynôme à coefficients réels.

On peut imaginer de nombreuses méthodes générales dans cette classe.

On peut également en imaginer un certain nombre spécifiques aux polynômes de degré 2 (calcul de la valeur du discriminant, calcul du nombre de racines réelles, calcul des éventuelles racines réelles, …). Plutôt que les inclure dans la classe `Poly`, nous pouvons créer une nouvelle classe dérivée de la classe `Poly`. Nommons-la par exemple `Poly2d`.

La syntaxe correspondante pour créer cette nouvelle classe sera :

```
class Poly2d(Poly):
```

A partir de là, lorsque l'on créera un objet du type `Poly2d`, tous les attributs et toutes les méthodes de la classe `Poly` seront disponibles pour cet objet : on dit que « la classe `Poly2d` **hérite** des attributs et des méthodes de la classe `Poly` » (c'est le mécanisme d'« **héritage** »). La classe Poly2d est une « **sous-classe** » de la classe `Poly` (appelée elle-même « **classe parente** ») de la classe `Poly2d`.

Mais bien sûr (et c'est tout l'intérêt de la notion de sous-classe), on va pouvoir, dans la sous-classe :

- Fixer des valeurs à certains attributs.
- Ajouter de nouveaux attributs.
- Redéfinir les attributs.
- Redéfinir certaines méthodes (y compris des méthodes spéciales : on parle encore de « **surcharge** »).
- Ajouter de nouvelles méthodes.

Pour terminer, précisons que l'on peut définir, à partir d'une classe donnée, plusieurs sous-classes.