Programmation orientée objets (Compléments)

S1 - Langages et programmation

> **Avertissement**
>
> Les compléments présentés ici sont hors programme. Ils peuvent néanmoins apporter une connaissance et une compréhension plus fine de la POO et être utiles dans le cadre du travail sur les projets.

Principes et définitions

Objet Un **objet** est une donnée manipulable par un programme : il s'agit d'un conteneur pour une *valeur* ou un *état* auquel est associé un *ensemble d'opérations*. Cet objet est associé à un **type**, défini comme l'ensemble des valeurs possibles, cette liste d'opérations, ainsi que leur codage (binaire).

Un objet est identifié dans un programme par un *nom* ou une *notation littérale*, mais peut parfois être *anonyme* (comme les variables temporaires ou les composantes d'un tableau).

Et pour une définition d'un langage orienté objet, l'idée première que l'on retrouve dans la définition de wikipédia offre un cadre intéressant : un langage objet doit permettre l'analyse et le développement logiciel fondés sur des *relations entre objets*.

Concrètement, un objet est une structure de données qui répond à un ensemble de messages. Cette structure de données définit son état tandis que l'ensemble des messages qu'il comprend décrit son comportement :

- les données, ou champs, qui décrivent sa structure interne sont appelées ses **attributs** ;
- l'ensemble des messages forme ce que l'on appelle l'interface de l'objet ; c'est seulement au travers de celle-ci que les objets interagissent entre eux. La réponse à la réception d'un message par un objet est appelée une **méthode** (méthode de mise en œuvre du message) ; elle décrit quelle réponse doit être donnée au message.

Les attributs et les méthodes constituent les **membres** d'un objet. Un objet possède un **type**.

En Python, un objet la création d'un objet se fait en utilisant une **classe** : un objet est alors une instance de sa classe. La **classe** est un *type*, un ensemble d'objets partageant les mêmes propriétés concrétisées par une liste de membres.

Langage orienté objet Un **langage orienté objet** est un langage de programmation qui comporte de manière native les éléments suivants : l'*encapsulation*, l'*héritage*, le *polymorphisme* et la *programmation générique*.

Les principes clés de la POO

L'encapsulation

Certains membres (ou plus exactement leur représentation informatique) sont cachés : c'est le principe d'**encapsulation**. Ainsi, le programme peut modifier la structure interne des objets ou leurs méthodes associées sans avoir d'impact sur les utilisateurs de l'objet. C'est un des principes fondamentaux notamment pour la robustesse du code.

En particulier, les bonnes pratiques de POO recommandent de na pas permettre un accès direct aux attributs d'un objet à l'extérieur de celui-ci. On appelle **interface** d'un objet l'ensemble de ses membres qui sont accessibles à l'extérieur de celui-ci. L'interface ne devrait donc contenir que des méthodes. Pas forcément toutes, certaines méthodes (comme `__init__`) restent privées.

Contrairement à d'autres langages, Python offre une totale liberté de modification sur les membres d'un objet. C'est au programmeur de rester vigilant. Il existe néanmoins des conventions permettant d'identifier les membres de l'interface des autres membres d'un objet.

> **!** Conventions de nommage en Python
>
> - un nom d'attribut commençant par un double underscore `__` désigne un attribut privé.
> - une méthode dont le nom est de la forme `__nom__` désigne une méthode privée.

Mais alors si les attributs doivent rester privés, comment y accéder, et comment les modifier ?

Il convient pour cela, en toute rigueur, de définir des méthodes ad-hoc : une méthode qui permet d'accéder à un attribut est un `getter`, une période qui permet de changer la valeur d'un attribut est un `setter`.

Voici par exemple une nouvelle définition de la classe "Rectangle" tenant compte des remarques précédentes.

```
class Rectangle:
    """Représente un rectangle"""

    def __init__(self, largeur=2, hauteur=3):
        self.__largeur = largeur
        self.__hauteur = hauteur

    def get_largeur(self):
        return self.__largeur

    def set_largeur(self, largeur):
        self.__largeur = largeur

    def get_hauteur(self):
        return self.__hauteur

    def set_hauteur(self, hauteur):
        self.__hauteur = hauteur

    def perimetre(self):
        """Retourne le périmètre"""
        return 2 * (self.__largeur + self.__hauteur)
```

```
    def aire(self):
        """Retourne l'aire"""
        return self.__largeur * self.__hauteur
```

Utilisation :

```
>>> rec = Rectangle(10, 5)
>>> rec.__hauteur
AttributeError: 'Rectangle' object has no attribute '__hauteur'
>>> rec.get_hauteur()
5
```

Nous voyons que l'accès direct à l'attribut n'est plus possible.

Cela n'est pas très pratique et change nos habitudes : nous aimerions en effet pouvoir accéder à la valeur d'un attribut en utilisant la notation pointée. Deux remarques à ces objections. D'une part, ces règles de programmation ne sont pas là pour nous embêter ! Il s'agit de sécuriser notre code : la définition d'un `setter` par exemple, peut permettre de vérifier la validité des arguments entrés et afficher un message d'erreur si besoin (par exemple si on appelle `set_hauteur(-10)`. Deuxième remarque : Python propose une fonctionnalité avancée, appelée **décorateurs** et qui permet de retrouver, en apparence, l'accès direct aux attributs. Voici une nouvelle version de la classe "Rectangle" avec l'utilisation du décorateur `@property` et la redéfinition des `getter` et `setter` (qui doivent maintenant porter le même nom que le pseudo argument). On a introduit dans les `setter` des tests de validité des données.

```
class Rectangle:
    """Représente un rectangle"""

    def __init__(self, largeur=2, hauteur=3):
        self.__largeur = largeur
        self.__hauteur = hauteur

    @property
    def largeur(self):
        return self.__largeur

    @largeur.setter
    def largeur(self, largeur):
        if isinstance(largeur, (int, float)) and largeur >= 0:
            self.__largeur = largeur
        else:
            print("Argument invalide, largeur inchangée !")

    @property
    def hauteur(self):
        return self.__hauteur

    @hauteur.setter
    def hauteur(self, hauteur):
        if isinstance(hauteur, (int, float)) and hauteur >= 0:
            self.__hauteur = hauteur
        else:
            print("Argument invalide, hauteur inchangée !")
```

```
    def perimetre(self):
        """Retourne le périmètre"""
        return 2 * (self.__largeur + self.__hauteur)

    def aire(self):
        """Retourne l'aire"""
        return self.__largeur * self.__hauteur


rec = Rectangle(10, 25)
print(rec.largeur)
rec.largeur = -15
print(rec.largeur)
```

Sortie en console :

```
10
Argument invalide, largeur inchangée !
10
```

L'héritage

L'**héritage** est une relation asymétrique entre deux classes : l'une est la **classe mère** (aussi nommée classe parente, superclasse, classe de base), l'autre la **classe-fille**. L'héritage permet une économie d'écriture par la réutilisation automatique, lors de la définition de la classe-fille, de tous les membres et autres éléments définis dans la classe mère. Ainsi, les objets de la classe-fille *héritent de toutes les propriétés* de leur classe mère.

Par exemple, nous pouvons définir une classe `carre`, fille de la classe `Rectangle`. Les attributs et les méthodes définis pour la classe `Rectangle` existent alors automatiquement aussi pour la classe `carre`.

Voici la syntaxe Python pour définir une classe fille :

```
class Carre(Rectangle):

    def __init__(self, cote=2):
        Rectangle.__init__(self, cote, cote)
```

Utilisation :

```
>>> car = Carre(5)
>>> car.perimetre()
20
```

La méthode `perimetre` est héritée de la classe mère `Rectangle`.

> ⚠ Héritage et initialiseur
>
> La méthode initialiseur de la classe `Carre` fait appel à la méthode initialiseur de sa classe parente par la commande `Rectangle.__init__(self, cote, cote)`. Cet appel est nécessaire afin que les membres de la classe `Carre` soient définis de la même manière que les membres de la classe `Rectangle`. La méthode `__init__` est un initialiseur d'instance : elle n'est pas invoquée

automatiquement lorsqu'on instancie des objets d'une classe fille.

Le polymorphisme et la redéfinition

La **redéfinition des méthodes** permet à un objet de raffiner une méthode définie avec la même en-tête dans la classe mère. Une même méthode pourra ainsi avoir un comportement différent selon qu'elle s'applique à la classe mère ou à la classe fille : on parle de **polymorphisme d'héritage**.

Par exemple, nous pouvons redéfinir la méthode `aire` de la classe `Carre` comme ci-dessous : appliquée à un objet `Carre`, la nouvelle définition sera utilisée à la place de la méthode héritée.

```
class Carre(Rectangle):

    def __init__(self, cote=2):
        self.__largeur = cote
        self.__hauteur = cote

    def aire(self):
        """Retourne l'aire"""
        return self.__largeur ** 2
```

Les méthodes spéciales

Un bon exemple de polymorphisme est fourni par la redéfinition des méthodes spéciales.

Nous savons que la fonction `dir()` renvoie tous les membres d'un objet.

Appliquons cette commande à notre objet `rec`, instance de la classe `Rectangle` :

```
>>> dir(rec)
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
```

```
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 '_hauteur',
 '_largeur',
 'aire',
 'hauteur',
 'largeur',
 'perimetre']
```

Nous reconnaissons en fin de liste les attributs et méthodes que nous avons définis, mais nous découvrons l'existence d'un grand nombre de **méthodes spéciales** privées (puisque leur nom est entouré de `__`) qui sont en fait **héritées** d'une classe `Object` parente de toutes les classes. Parmi celles-ci, nous avons déjà rencontré `__init__`, la méthode initialiseur.

Les curieux pourront rechercher le rôle de chacune de ces méthodes spéciales. Le voici pour certaines d'entre elles :

Méthode spéciale	Usage
add	+
mul	*
sub	-
eq	==
ne	!=
lt	<
ge	<=
gt	>
ge	>=
repr	affichage dans la console »> obj
str	str(obj), print(obj)

La redéfinition de la méthode `__add__` permettrait par exemple de donner un sens à l'utilisation du symbole `+` entre deux objets (instruction du type `rec1 + rec 2`).

Dans notre exemple, nous allons redéfinir la méthode `__str__` pour spécifier ce qui doit s'afficher quand l'instruction `print(rec)` est exécutée.

Pour l'instant, on obtient :

```
>>> print(rec)
<__main__.Rectangle object at 0x000002386735C730>
```

Ajoutons la méthode ci-dessous **dans la classe `Rectangle`** :

```
def __str__(self):
    return f"Rectangle de largeur {self.__largeur} et de hauteur {self.__hauteur}."
```

On obtient maintenant :

```
>>> print(rec)
Rectangle de largeur 10 et de hauteur 25.
```