Programmation orientée objet

Les paradigmes de programmation

Nous avons déjà vu qu'il existait de très nombreux langages de programmation. Ces langages diffèrent par leur syntaxe, mais aussi par rapport à certains concepts qui guident la façon dont ils seront utilisés pour résoudre tel ou tel problème. On appelle **paradigme** un ensemble de concepts de programmation.
Pour l'instant nous avons principalement utilisé la programmation **impérative**. Un programme est une succession d'instructions, essentiellement composée d'affectations de variables. Nous verrons plus tard la programmation **fonctionnelle** où un programme est vu comme un enchaînement d'appels de fonctions. La récursivité est un des concepts à la base de la programmation fonctionnelle, alors que les boucles correspondent plus à la progammation impérative.
Ces quelques exemples suffisent à montrer que les langages de programmation, comme Python, ne se limitent pas à un unique paradigme. Les paradigmes ne sont pas non plus disjoints. Ils peuvent partager certains concepts, comme la récursivité qui est commune à presque tous les langages. Même un programme n'est généralement pas limité à un seul paradigme. Bien qu'ayant privilégié la programmation impérative, nous avons principalement défini des fonctions.
Il faut plutôt voir les paradigmes comme des approches différentes, plus ou moins adaptées à certains problèmes. Il faut donc savoir reconnaître, en fonction du problème étudié, le paradigme le plus approprié afin d'appliquer la bonne stratégie de résolution.

Programmation orientée objet

La **programmation orientée objet** (POO) est un paradigme qui repose sur le concept d'abstraction. Prenons l'exemple d'un jeu dans lequel on contrôle un personnage. Ce personnage est représenté par une position, une vitesse, des points de vie… Au lieu de faire une variable pour chacune de ces valeurs, la programmation orientée objet propose de les abstraire dans un **objet** qui permettra d'y accéder et de les manipuler via une interface. Cela ne semble pas forcément très intéressant s'il y a un seul personnage, mais si on fait la même chose pour les ennemis, il est plus simple de manipuler une liste d'objets correspondant à chacun de ces ennemis que plusieurs listes ou de nombreuses variables pour chacunes des valeurs nécessaires pour gérer leur comportement. On parle d'**encapsulation**. Les valeurs sont contenues dans l'objet, sans qu'on ait besoin de savoir comment elles sont représentées.
En Python, nous avons déjà utilisé des objets. Les listes ou les dictionnaires sont des objets. Nous ne savons pas comment ils sont représentés en interne, mais nous savons que `liste.append(v)` permet de rajouter `v` à la fin de la liste.

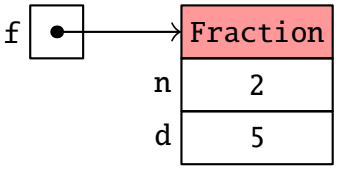
Construction des objets

Afin d'illustrer la façon dont sont définis les objets, en général, et plus particulièrement en Python, nous allons reprendre l'exemple des fractions et définir des objets qui les représenteront. Un objet est avant tout caractérisé par les valeurs qu'ils contient, qu'on appelle **attributs**. Pour les fractions, nous allons utiliser deux attributs, pour le numérateur et le dénominateur. Mais un objet contient aussi des fonctions, appelées **méthodes**. La principale s'appelle le **constructeur**. Voici un exemple en Python :

```
class Fraction:
    def __init__(self, n, d):
        self.n = n
        self.d = d
```

La première ligne permet de définir la classe `Fraction` (par convention, les noms de classes commencent par une majuscule) et la fonction **`__init__`** est le constructeur, qui se contente de stocker le numérateur et le dénominateur donnés en paramètre. Le mot **`self`** représente l'objet qui va être créé. Dans tout ce qui sera écrit dans la définition de la classe, à chaque fois que nous auront besoin d'utiliser l'objet manipulé, il faudra utiliser **`self`**. Cette fonction s'appelle ainsi :

```
>>> f = Fraction(2, 5)
>>> f
<__main__.Fraction object at 0x7fd2369d3070>
```



Le nom `f` est associé à un pointeur vers l'objet de classe Fraction qui vient d'être créé. Cet objet a deux attributs qui valent 2 et 5. Il est possible d'y accéder directement, et même de les modifier :

```
>>> f.n
2
>>> f.d
5
>>> f.n = 17
```

Néanmoins, pour bien respecter le concept d'abstraction, on peut fournir des fonctions permettant d'obtenir la valeur des attributs et d'éviter d'y accéder directement :

```
class Fraction:
    def __init__(self, n, d):
        self.n = n
        self.d = d

    def numerateur(self):
        return self.n

    def denominateur(self):
        return self.d
```

Pour appeler une méthode, on l'écrit après le nom de l'objet suivi d'un point. On ne fait pas apparaître **`self`**, qui est implicite :

```
>>> f.numerateur()
17
>>> f.denominateur()
5
```

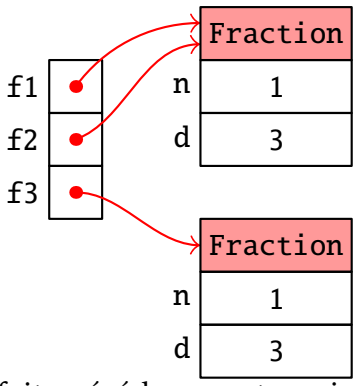
Il est également possible de définir des méthodes telles que `change_numerateur` qui permettraient de changer les valeurs des attributs.
En ne donnant pas directement accès aux attributs, on permet ainsi à celui qui programme la classe de choisir les représentations de son choix. Par exemple, une méthode `decimal` permettra d'obtenir la valeur décimale de la fraction, qu'elle soit stockée comme un attribut ou qu'elle soit calculée à chaque fois.

Certains langages, tels que Java ou C++, permettent de définir explicitement les attributs qui sont accessibles depuis l'extérieur de l'objet (on dit qu'ils sont **publics**) et ceux qui ne le sont pas (ils sont **privés**). Python ne permet pas une telle distinction et tous les attributs sont publics. Il est également possible de rajouter ou supprimer des attributs après la création d'un objet, ce qui est totalement interdit dans la plupart des autres langages. Nous éviterons donc d'utiliser ces fonctionnalités.

Méthodes

En plus des méthodes permettant d'obtenir ou de modifier les attributs, il est possible de définir dans la classe toutes les méthodes permettant de manipuler ou d'exploiter un objet. Dans le cas d'un personnage de jeu, on peut avoir besoin de méthodes permettant de modifier son nombre de points de vie ou de le déplacer. Pour les fractions, on peut rajouter une méthode permettant de les simplifier et lors de la construction de l'objet, cette méthode sera utilisée pour garantir que la fraction obtenue est déjà sous forme réduite.
Mais les méthodes ne servent pas uniquement à manipuler un seul objet. On peut vouloir déterminer si deux objets sont égaux ou pas. On pourrait se dire qu'il suffit de regarder si les attributs sont les mêmes. Mais ce n'est pas forcément suffisant. Pour les fractions, si elles ne sont pas réduites, les attributs ne seront pas égaux. D'autres objets peuvent contenir des listes et on peut considérer qu'ils sont égaux si les deux listes contiennent les mêmes valeurs, même si ce n'est pas dans le même ordre. A contrario, deux ennemis seront toujours différents, même s'ils ont les mêmes attributs. L'égalité de deux objets doit donc être définie pout chaque classe. Par défaut, Python considère que deux objets sont égaux s'ils pointent vers la même structure :

```
>>> f1 = Fraction(1, 3)
>>> f2 = f1
>>> f3 = Fraction(1, 3)
>>> f1 == f2
True
>>> f1 == f3
False
```



On pourrait définir une méthode `egalite`, comme nous l'avons fait précédemment, mais Python permet de faire une méhode `__eq__(self, other)` qui permettra ensuite d'utiliser l'opérateur == sur les fractions. En général, lorsqu'on manipule un autre objet que l'objet courant, on l'appelle `other`.

```
class Fraction:
    ...

    def __eq__(self, other):
        n1 = self.numerateur()
        d1 = self.denominateur()
        n2 = other.numerateur()
        d2 = other.denominateur()
        return n1*d2 == n2*d1
```

```
>>> Fraction(1, 3) == Fraction(1, 3)
True
```

Afin de comparer des objets avec les opérateurs de comparaison, Python permet de définir un certain nombre de méthodes, qui sont présentées ci-contre. Il suffit de définir `__eq__`, `__lt__` et `__le__` pour que Python puisse automatiquement en déduire les autres. Mais elles peuvent également être définies individuellement.

Méthode	Opérateur
`__eq__`	==
`__ne__`	!=
`__lt__`	<
`__le__`	<=
`__gt__`	>
`__ge__`	>=

Le constructeur n'est pas la seule méthode qui peut renvoyer un nouvel objet. Dans le cas d'une fraction, on peut définir l'inverse de la manière ci-contre. Cela permet d'écrire `f2 = f1.inverse()`.

```
class Fraction:
    ...

    def inverse(self):
        n = self.numerateur()
        d = self.denominateur()
        return Fraction(d, n)
```

Comme pour les opérateurs précédents, il est possible de définir les opérateurs arithmétiques à l'aide de méthodes spéciales. Cela permet ensuite de pouvoir écrire des choses comme `f1+f2`. Il existe de nombreuses autres méthodes spécifiques qui peuvent permettre de faciliter l'utilisation des objets. L'une des plus importante est `__str__` qui permet ensuite d'utiliser un objet en argument de `print`.

Méthode	Opérateur
`__add__`	+
`__sub__`	−
`__mul__`	*
`__truediv__`	/

Objection !

Même si la POO est très populaire, ce n'est pas non plus une solution miracle. Elle ne permet pas de faire plus que sans utiliser des objets. Son principal avantage réside dans le fait que le code produit est souvent plus lisible, puisque les différents objets manipulés sont clairement identifiés, et modulaire puisque l'encapsulation assure la séparation entre l'interface et l'implémentation.
La POO reste donc plus adaptée pour de "gros" projets ou lorsqu'on souhaite faire des bibliothèques pouvant être utilisées plus tard.

Pour aller plus loin

Le paradigme objet repose sur d'autres concepts, comme l'**héritage**, qui n'est pas au programme de NSI. L'idée c'est qu'il est possible de définir une sous-classe d'une classe. Ainsi, si on définit une classe `Polygone`, on peut définir une sous-classe `Parallelogramme`. L'intérêt, c'est d'hériter automatiquement de toutes les méthodes de la classe parente. Il suffit alors de redéfinir celles qui le nécessitent et d'en rajouter si on le veut. Le calcul de l'aire d'un polygone n'est pas très simple. Pour un parallélogramme, au contraire, c'est relativement aisé. Il semblerait naturel de redéfinir la méthode `aire` dans la classe `Parallelogramme` pour simplifier les calculs.
De la même manière, il est possible de définir les sous-classes `Rectangle` et `Losange` qui hériteraient de `Parallelogramme`. Et enfin, la classe `Carre` pourrait hériter de ces deux sous-classes, l'héritage n'étant pas forcément unique.
Lorsqu'on manipule un objet, il n'est pas non plus nécessaire de savoir à quelle classe il appartient. Lorsqu'on écrit `fig.aire()`, on ne sait pas forcément quelle sera la méthode utilisée. On parle alors de **polymorphisme**, qui est un autre concept fondamental dans la programmation orientée objet.