Programmation orientée objets (Cours)

S1 - Langages et programmation

Objectifs

- Connaître le vocabulaire de la POO : classes, attributs, méthodes objets.
- Écrire la définition d'une classe.
- Accéder aux méthodes et attributs d'une classe.

Des compléments sur la POO sont proposés à la suite de ce cours.

Introduction

La Programmation Orientée Objets (**POO**) pour les intimes (que vous allez devenir !) est un **paradigme** (c'est-à-dire une manière de faire) de programmation

Pour un bref aperçu historique de l'idée d'objet en programmation, lire l'encadré ci-dessous, en grande partie tiré de l'article Object-oriented programming de Wikipedia en anglais.

Un peu d'histoire

Les termes **objets orienté** au sens moderne de la programmation orientée objet ont fait leur première apparition au MIT à la fin des années 1950 et au début des années 1960. Dans l'environnement du groupe d'intelligence artificielle, dès 1960, "objet" pouvait désigner des éléments identifiés avec des propriétés (attributs).
C'est l'informaticien Alan Kay (1940-) qui est considéré comme l'un de pères de la programmation orientée objets.

> Je pensais que les objets étaient comme des cellules biologiques et / ou des ordinateurs individuels sur un réseau, uniquement capables de communiquer avec des messages.
>
> Alan Kay

Le langage Simula dans les années 1960, puis Smalltalk dans les années 1970 posent les bases toujours actuelles de ce paradigme.
Au début et au milieu des années 1990, la programmation orientée objet s'est développée comme le paradigme de programmation dominant lorsque les langages de programmation prenant en charge ces techniques sont devenus largement disponibles. Ceux-ci incluent par exemple C++ ou Delphi. Sa domination a été renforcée par la popularité croissante des interfaces utilisateur graphiques, qui reposent fortement sur des techniques de programmation orientées objet.
Des fonctionnalités orientées objet ont été ajoutées à de nombreux langages existants, notamment Ada, BASIC, Fortran, Pascal et COBOL.
Plus récemment, un certain nombre de langages ont émergé qui sont principalement orientés objet, mais qui sont également compatibles avec la méthodologie procédurale. Deux de ces langages sont Python et Ruby. Les langages orientés objet récents les plus importants sur le plan commercial sont probablement Java, développé par Sun Microsystems, ainsi que C# et Visual Basic.NET (VB.NET), tous deux conçus pour la plate-forme.NET de Microsoft.
*[MIT] : Massachusetts Institute of Technology

Alors de quoi s'agit-il ? Une approche intuitive consiste à dire que cette méthode de programmation nous permet de définir des nouveaux types de données, de nouveaux objets, correspondant à un objectif précis. Ces nouveaux types sont appelés **classes**. En définissant une classe, nous pouvons également définir ses **attributs**, c'est-à-dire les variables qui lui sont associées et ses **méthodes**, c'est-à-dire les fonctions sui lui sont propres.

Voici ce que Gérard SWINNEN écrit dans son livre *Apprendre à programmer avec Python 3* :

> Les classes sont les principaux outils de la programmation orientée objet (Object Oriented Programming ou OOP). Ce type de programmation permet de structurer les logiciels complexes en les organisant comme des ensembles d'objets qui interagissent, entre eux et avec le monde extérieur.
>
> Le premier bénéfice de cette approche de la programmation réside dans le fait que les différents objets utilisés peuvent être construits indépendamment les uns des autres (par exemple par des programmeurs différents) sans qu'il n'y ait de risque d'interférence. Ce résultat est obtenu grâce au concept d'encapsulation : la fonctionnalité interne de l'objet et les variables qu'il utilise pour effectuer son travail, sont en quelque sorte « enfermées » dans l'objet. Les autres objets et le monde extérieur ne peuvent y avoir accès qu'à travers des procédures bien définies : l'interface de l'objet.
>
> En particulier, l'utilisation de classes dans vos programmes va vous permettre – entre autres avantages – d'éviter au maximum l'emploi de variables globales. Vous devez savoir en effet que l'utilisation de variables globales comporte des risques, d'autant plus importants que les programmes sont volumineux, parce qu'il est toujours possible que de telles variables soient modifiées, ou même redéfinies, n'importe où dans le corps du programme (ce risque s'aggrave particulièrement si plusieurs programmeurs différents travaillent sur un même logiciel).

En utilisant Python, nous avons déjà fréquenté des classes d'objets : il est usuel de lire qu'en Python "tout est objet", même si Python ne permet pas vraiment de faire de la POO dans toute sa rigueur.

Par exemple, définissons une chaîne de caractères et demandons à Python quel est son type :

```
>>> a = "Coucou !"
>>> type(a)
<class 'str'>
```

Le type 'str' bien connu est en fait une classe d'objet prédéfinie. On dit que `a` est une **instance** de l'objet `str`.

Un exemple de **méthode** rattachée à la classe str est la fonction `capitalize` qui met le premier caractère en majuscule. Cette méthode est appelée par la notation pointée déjà rencontrée.

```
>>> a.capitalize()
'Coucou !'
```

La commande `help(str)` affiche toutes les méthodes prédéfinies pour les objets de la classe `str`.

Classe, initialiseur, attributs

En Python, la définition d'une classe se fait avec le mot-clef `class` suivi du nom de la classe.

Supposons par exemple que nous voulons définir une classe pour représenter des rectangles. Nous avons besoin, pour chaque rectangle, de connaître sa largeur et se hauteur, ce qui nous permettra de faire quelques calculs.

Observons le code suivant :

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

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

À l'intérieur de la classe `Rectangle`, la **méthode** `__init__` est l'**initialiseur** : elle est toujours exécutée lorsqu'une **instance** de l'objet `Rectangle` est créée (on parle aussi de **constructeur**). Cette *fonction* accepte des paramètres qui seront les valeurs à donner aux attributs à la création de l'objet (ici on a aussi donné des valeurs par défaut à ces paramètres) ; le premier paramètre est particulier : ce sera **toujours** le mot-clé `self` : ce mot-clef désigne l'instance qui est en train d'être définie au moment où cette fonction s'exécute.

> Remarque
>
> Il est d'usage, et recommandé, de nommer une classe par un nom commençant par une majuscule.

Observons les lignes suivantes dans la console Python :

```
>>> rec1 = Rectangle()
>>> rec2=Rectangle(15, 25)
>>> rec1.largeur
2
>>> rec2.largeur
15
```

`rec1` est une instance de l'objet `Rectangle`. Aucun paramètre n'étant donné lors de sa création, les valeurs par défaut ont été appliquées. `rec2` est une autre instance de l'objet `Rectangle` pour laquelle on a défini les **attributs d'instance** `largeur` et `hauteur`. On accède à ces attributs par la notation pointée.

Méthodes

Une **méthode**, c'est ce qui permet à une instance de réaliser des actions. Techniquement se sont des fonctions définies **dans le corps de la classe**, et qui prennent toujours au moins un premier paramètre qui est `self`.

Certaines méthodes sont particulières, leurs noms commencent par un double *undescore* (le caractère `_`). On a vu une première de ces méthodes : l'initialiseur `__init__`.

Définissons ici une méthode permettant de calculer le périmètre d'un rectangle et une autre pour l'aire :

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

    def __init__(self, largeur=2, hauteur=3):
        self.largeur = largeur
        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.perimetre()
30
>>> rec.aire()
50
```

Une méthode peut modifier la valeur d'un attribut. Définissons par exemple une méthode permettant de doubler la largeur d'un rectangle.

```
def double_largeur(self):
    """Double la largeur du rectangle)"""
    self.largeur *= 2
```

```
>>> rec = Rectangle(10, 5)
>>> rec.double_largeur()
>>> rec.largeur
20
```

La POO encourage à n'exposer que des méthodes vers l'extérieur (on parle d'**interface**) en masquant les attributs. Il s'agit d'un des quatre éléments constitutifs de l'orienté objet : l'**encapsulation**.

> **Remarque**
>
> Nous avons présenté ci-dessus les **attributs d'instance**, mais il est aussi possible de définir des **attributs de classe** qui seront donc les mêmes pour toutes les instances d'une même classe créées dans un programme. Ces attributs sont définis à l'intérieur de la classe, sans le préfixe `self` (puisque celui-ci fait référence à l'instance en train d'être créée).
> Par exemple, dans la définition de la classe `Atome` ci-dessous, la liste `table` est un attribut de classe, alors que l'entier `np` est un attribut d'instance. Dans la méthode `__init__`, on accède à l'attribut de classe en le préfixant par le nom de la classe `Atome.table` et on accède à l'attribut d'instance en le préfixant par le nom de l'instance `self.np`.

```
class Atome:
"""atomes simplifiés, choisis parmi les 10 premiers éléments du TP"""
table = [None, ('hydrogène', 0), ('hélium', 2), ('lithium', 4), ('béryllium', 5),
            ('bore', 6), ('carbone', 6), ('azote', 7), ('oxygène', 8), ('fluor', 10), ('né

def __init__(self, nat):
    "le n° atomique détermine le n. de protons, d'électrons et de neutrons"
    self.np, self.ne = nat, nat  # nat = numéro atomique
    self.nn = Atome.table[nat][1]

def affiche(self):
    print()
    print("Nom de l'élément :", Atome.table[self.np][0])
    print(f"{self.np} protons, {self.ne} électrons, {self.nn} neutrons")
```

De la même façon, il est possible de définir des **méthodes de classe**.

Représentation simplifiée d'une classe

Lorsqu'un module comportant plusieurs définitions de classe est développé, on peut réaliser une représentation graphique, appelée **diagramme de classes** qui permet de visualiser le nom de chaque classe, son interface (c'est-à-dire l'ensemble de ses attributs et méthodes publiques) et les relations éventuelles entre les différentes classes.

Une présentation détaillée de ce type de diagramme n'est pas au programme, mais de manière simplifiée, le diagramme correspondant à la classe `Rectangle` pourrait ressembler à ce qui suit :

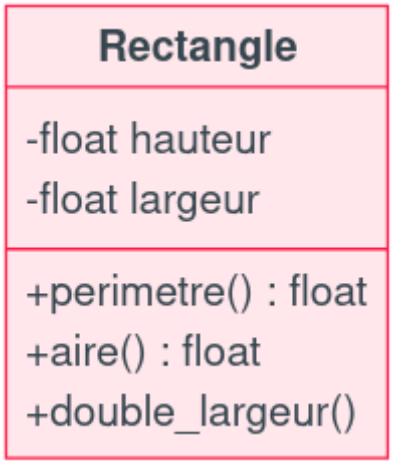

Figure 1: Diagramme de classe

Dans des cases séparées, on place d'abord le nom de la classe, puis ses attributs et enfin ses méthodes. Un codage spécifique permet de préciser le type des différents membres, leur caractère public ou privé (voir les compléments à ce sujet) ou même leur état.

Un attribut de classe ou une méthode de classe seront soulignés dans un tel diagramme.

Pour en savoir un peu plus sur les diagrammes de classes, je vous conseille ce document ou encore ce cours plus difficile.

Des compléments sur la POO sont proposés à la suite de ce cours.