Chapitre 8

POO - partie 1

« La POO c'est la classe ! »

À retenir

- Pour représenter des entités qui ont des caractéristiques et des fonctionnalités communes, on fabrique une *classe* qui décrit le modèle général que suit une entité ;
- chaque entité qu'on crée suivant la classe s'appelle *une instance* de cette classe, c'est un *objet* qui a ses propres *membres* :
  - ses variables, appelées *attributs* ;
  - ses propres fonctions, appelées *méthodes*.
- la classe elle même peut avoir ses propres attributs et méthodes ;
- les objets peuvent interagir entre eux, avec la classe et « avec l'extérieur ».

La *programmation orientée objet* (ou programmation objet, ou POO) est un *paradigme de programmation* qui pousse un peu plus loin la notion de modularité et d'encapsulation que nous avons déjà vue.

1 Un exemple simple et complet

1.1 Pourquoi concevoir un objet ?

On imagine qu'on veut représenter une liste de rectangles, chacun ayant ses propres dimensions. On peut les représenter par une liste de listes :

Python

```
rectangles=[[3, 4], [5, 6], [7, 8]]
```

Ce n'est pas très parlant. Supposons maintenant qu'on veuille calculer la surface d'un rectangle. Avec des listes on écrira

Python

```
def area(rect : list)-> float :
    return rect[0] * rect[1]
```

Ça marche, mais comme écrit précédemment, ce n'est pas très élégant et si on utilise cette méthode pour des structures plus complexes qu'un simple rectangle dans des programmes longs, on risque fort de finir par se tromper.

On peut alors se dire qu'on va utiliser une liste de dictionnaires :

Python

```
rectangles=[{'width' : 3, 'height' : 4}, {'width' : 5, 'height' :
 ↪  6},
            {'width' : 7, 'height' : 8}]
```

C'est encore assez encombrant même si c'est mieux. Autant passer le cap et définir un objet.

1.2 Créons la classe

Python

```
class Rectangle:

def __init__(self, w: float, h: float):
    self.width = w
    self.height = h
```

On a donc défini une *classe* `Rectangle` (les noms de classe commencent par des majuscules) à l'intérieur de laquelle on a défini la *méthode* `__init__`.

Définition : classe, membres, methodes, attributs

Une *classe* est un ensemble de membres.
Un *membre* peut être une méthode ou un attribut.
Une *méthode* est une fonction. Un *attribut* est une variable ou une constante.

Cette méthode `__init__` est spéciale, comme sa forme le laisse penser. En PYTHON les méthodes spéciales sont entourées de 2 "_" de part et d'autre. Ces « doubles tirets bas » se disent *double underscore* en Anglais et on abrège souvent en *dunder*. `__init__` peut donc se lire/dire *dunder*

init.
Cette méthode s'appelle le constructeur.

Définition : constructeur

> Un constructeur est une méthode qui crée et renvoie un *objet*. On dit que cet objet est *une instance de la classe*.

Dans le code de `__init__`, on remarque que le premier paramètre s'appelle `self` : il fait référence à l'objet que la méthode crée.
À partir de 2 `float`, ce constructeur instancie un objet de la classe `Rectangle`. Cet objet possède 2 attributs : `width` et `height`. Voici comment on s'en sert :

Python

```
>>> r1 = Rectangle(3,4)
>>> r1.width
3
>>> r1.height
4
>>>  r2 = Rectangle(5,6)
>>> r2.width
5
>>> r2.height
6
```

On a crée 2 objets différents, chacun d'eux possède *les mêmes attributs* mais avec *des valeurs différentes*.

1.3 Créons des méthodes

On veut pouvoir calculer le périmètre et l'aire d'un rectangle, donc, toujours à l'intérieur de la classe, on définit les méthodes suivantes :

Python

```
def perimeter(self):
    return (self.width + self.height) * 2

def area(self):
    return self.width * self.height
```

On peut maintenant les utiliser

Python

```
>>> r1.perimeter()
14
>>> r1.area()
>>> 12
```

Remarque

Il faut bien noter qu'on écrit `r.perimeter()` *avec des parenthèses*, pour évaluer le résultat de cette méthode . Sans parenthèses, on fait référence à la méthode elle-même.

D'ailleurs certaines méthodes requièrent un ou plusieurs paramètres, comme par exemple celle-ci :

Python

```
def rescale_by_factor(self, f: float):
    self.width *= f
    self.height *= f
```

Cette méthode sert à redimensionner le rectangle. Il faut remarquer que quand une méthode s'applique à une instance de la classe, alors son premier paramètre *doit impérativement* être `self`.

Python

```
>>> r1.rescale_by_factor(10)
>>> r1.width
30
>>> r1.height
40
```

1.4 Différence entre classe et instance

Définitions

- Un *attribut d'instance* est une variable attachée à chaque instance de la classe. Deux instances différentes peuvent donc très bien présenter des valeurs différentes pour cet attribut.
  Un *attribut de classe* est une variable qui n'est pas directement liée aux instances de

la classe mais à la classe elle-même.

- De la même manière on distingue les *méthodes d'instances* des *méthodes de classe* : on peut dire qu'une méthode d'instance prend cette instance en paramètre (c'est ce fameux `self`) alors qu'une méthode de classe non.

Redéfinissons la classe `Rectangle` (on garde les mêmes méthodes d'instance `area` et `perimeter`) pour qu'elle garde une trace des objets créés.

Python

```python
class Rectangle:
    rectangle_list = []

    @staticmethod
    def count() -> int:
        return len(Rectangle.rectangle_list)

    def __init__(self, w: float, h: float):
        self.width = w
        self.height = h
        Rectangle.rectangle_list.append(self)
```

- On a ajouté un attribut de classe `rectangle_list`. Ce nom d'attribut est valable à l'intérieur de définition la classe, mais partout ailleurs, y compris dans les méthodes d'instances, on devra l'appeler `Rectangle.rectangle_list`.
- On a ajouté une méthode statique, comme on peut le voir à l'aide du *décorateur* `@staticmethod` (un décorateur c'est une notion assez compliquée que l'on verra peut-être plus tard).
  Elle est dite *statique* car ne prend pas d'instance en entrée (pas de `self`). Elle est appelée avec `Rectangle.count()` et renvoie simplement la longueur de la liste `Rectangle.rectangle_list`.
- Enfin à l'intérieur du constructeur `__init__` on rajoute l'objet créé `self` à la liste `Rectangle.rectangle_list`. Pour ce dernier point on peut se demander comment on peut ajouter un objet en train d'être créé, et qui va sans doute être modifié plus tard, à la liste. La réponse est que `self` est une référence, pas une valeur, tout comme pour les listes.