Chapitre 10

POO - partie 2

« Quel est l'objet de ce chapitre ? »

À retenir

- Pour plus d'encapsulation et de modularité, on peut « cacher » les attributs des instances et, à la place, implémenter des *getters* et des *setters*.
- De nombreuses méthodes spéciales, les *dunders* peuvent être redéfinies pour un code encore plus flexible.

1 Les bonnes pratiques d'encapsulation

Considérons la classe suivante

```
class Person:
    def __init__(self, a: int, n: str, s: str):
        self.age = a
        self.name = n
        self.surname = s
```

Elle modélise une personne avec un âge, un prénom (*name*) et un nom de famille *surname*. Tel qu'écrite on s'en servira ainsi :

Python

```
>>> p = person(30,'Louise','Dupont')
>>> print(p.surname) # pour afficher le nom de la dame
Dupont
>>> p.surname = 'Durant' # madame s'est mariée
```

Cela fonctionne très bien mais va à l'encontre des règles d'encapsulation et de modularité. On va donc créer des attributs privés commençant par « _ » et pour chacun d'entre eux

- une méthode appelée *accesseur* (getter en Anglais) qui permet d'accéder à la valeur de l'attribut;
- une méthode appelée *mutateur* (setter en Anglais) pour changer la valeur de l'attribut;

```
class Person:
    def __init__(self, a: int, n: str, s: str):
        self._age = a
        self._name = n
        self._surname = s

    def get_age(self) -> int:
        return self._age

    def set_age(self, a: int):
        self._age = a

    def get_name(self) -> str:
        return self._name

    def set_name(self, n: str):
        self._name = n

    def get_surname(self) -> str:
        return self._surname

    def set_surname(self, s: str):
        self._surname = s
```

C'est mieux du point de vue de l'encapsulation : les attributs de l'objet restent cachés mais on peut les voir et les modifier *via* des méthodes.
C'est mieux du point de vue de la modularité : si on veut changer les attributs (ou autre chose, pour une raison ou une autre) dans la classe `Person`, on peut garder les *getters* et les *setters*.

Remarque

> Les *getters* et les *setters* font partie de l'interface d'une classe.

Python

```
>>> p = person(30,'Louise','Dupont')
>>> print(p.get_surname()) # pour afficher le nom de la dame
Dupont
```

```
>>> p.set_surname('Durant') # madame s'est mariée
```

2 Les autres méthodes dunder de Python

À part `__init__`, il existe une multitude de fonctions *dunder*.

2.1 La méthode `__str__`

Reprenons la classe `Person`, imaginons qu'on l'a enregistrée dans un module appelé `person.py`

Python

```
>>> print(p)
<person.Person object at 0x7fdd9d8866d0>
```

Pas très parlant, n'est-ce -pas ? Il n'y a qu'à redéfinir la méthode `__str__` dans la classe `Person`

Python

```
def __str__(self):
    return "Name : " + self._name + ", Surname : " +
     ↪  self._surname + ", Age : " + str(self._age)
```

Ce qui nous donne ensuite

Python

```
>>> print(p)
Name : Louise, Surname : Durant, Age : 30
```

Et c'est bien mieux !

2.2 La méthode `__eq__`

Ce morceau de code nous navre :

Python

```
>>> p1 = Person(10,'Tom','Dupont')
>>> p2 = Person(10,'Tom','Dupont')
```

```
>>> print(p1==p2)
False
```

C'est parce que deux instances différentes d'une même classe sont stockées à des endroits différents de la mémoire, comme on peut le voir ainsi :

Python

```
>>> print(id(p1))
    140411289495248
>>> print(id(p2))
    140411291729200
```

Mais heureusement on peut redéfinir la méthode `__eq__` de la classe `Person` :

Python

```
def __eq__(self, other):
    return self._age == other._age and self._name == other._name
     ↪  and self._surname == other._surname
```

Et ainsi

Python

```
>>> print(p1==p2)
True
```

Ouf, tout rentre dans l'ordre.

Remarque

> La méthode `__eq__` prend évidemment `self` en paramètre, et un deuxième appelé `other` qui est censé être la deuxième instance de la classe `Person` avec laquelle on veut comparer la première.

Les dunder de conteneurs

Ils permettent d'utiliser `len`, les notations avec crochets, et d'itérer sur un objet :

```
class MyList:

    def __init__(self):
        self.content = [None] * 10
```

```
        self.changes = 0

    def __setitem__(self, key, value):  # permet de changer un
    ↪ élément
        self.content[key] = value
        self.changes += 1

    def __delitem__(self, key):
        self.content[key] = None

    def __getitem__(self, key):  # accès à l'élément
        return self.content[key]

    def __len__(self):  # pour utiliser len
        return len([x for x in self.content if x is not None])

    def __iter__(self):  # pour itérer sur les éléments
        return iter([x for x in self.content if x is not None])


    def __in__(self, item): # pour utiliser in
        return item in self.content
```

L'exemple précédent permet de simuler une mémoire à 10 cases, vides au départ mais que l'on peut remplir comme on veut. Ce qui est intéressant c'est que l'objet garde en mémoire le nombre de fois où une case a été changée. `len` donne le nombre de cases *non vides* et quand on itère sur l'objet on n'itère que sur ses cases non-vides.

Python

```
>>> a = MyList()
>>> len(a)
0
>>> a[1]=2
>>> a[3]=4
>>> len(a)
2
>>> print(10 in a)
False
>>> for x in a:
>>> ...     print(x)
>>> ...
```

```
2
4
```

2.3 D'autres dunders pour d'autres opérateurs

Voici un récapitulatif des dunders les plus utiles

Dunder	opérateur
`__lt__(self, other)`	`<`
`__le__(self, other)`	`<=`
`__ne__(self, other)`	`!=`
`__gt__(self, other)`	`>`
`__ge__(self, other)`	`>=`
`__add__(self, other)`	`+`
`__sub__(self, other)`	`-`
`__mul__(self, other)`	`*`
`__truediv__(self, other)`	`/`
`__floordiv__(self, other)`	`//`
`__contains__(self, item)`	`in`

Par exemple, lorsqu'on implémente `__add__` pour une classe, alors on peut écrire `c = a + b`, pour `a` et `b` instances de cette classe.

Définition : surcharge

Lorsqu'on attribue un sens supplémentaire à un opérateur pour une nouvelle classe, on dit qu'on *surcharge* cet opérateur. Les dunders précédents servent donc à surcharger.

3 Exercices

Exercice 1 : fractions

Reprendre le module `fractions_custom` des chapitres précédents et le reprogrammer « objet » :

- la classe `Fraction` implémente les fractions.
- on implémentera les dunders suivants :

- `__eq__` (deux fractions sont égales ssi les produits en croix sont égaux);
- `__lt__` et `__le__`;
- `__add__`, `__sub__`, `__mul__` et `__truediv__`.

On se concentrera sur la programmation objet en priorité, on peut laisser la programmation défensive de côté.

On devra pouvoir écrire :

```
>>> f1 = Fraction(2, 7)
>>> f2 = Fraction(3, 5)
>>> f1 == f2
False
>>> f1 < f2
True
>>> (f1+f2)/(f1*(f1-f2))
-217/22
```

Exercice 2

1. Créer une classe `Rectangle` dont chaque instance possède (au moins) des attributs `top`, `left`, `width` et `height`.
2. Implémenter `__str__` pour afficher « proprement » un objet de la classe.
3. Implémenter `__contains__` pour pouvoir tester si un point (`tuple` composé de deux `float`) se situe dans un rectangle ou non.
4. Implémenter `__add__` pour que la somme de 2 rectangles soit le plus petit rectangle contenant les 2 rectangles.
5. Implémenter `__mul__` pour que la somme de 2 rectangles soit le plus grand rectangle contenu dans les 2 rectangles (on fera attention au cas ou les rectangles sont disjoints).

On devra pouvoir écrire :

```
>>> r1 = Rectangle(10,20,100,200)
>>> (3,4) in r1
False
>>> (12,100) in r1
True
>>> r2 = Rectangle(30,40,50,300)
>>> r1 + r2
Rectangle : (10,20,100,320)
```

```
>>> r1 * r2
Rectangle (30,40,50,180)
```