Programmation Objet - 2e partie

Chapitre 10

NSI2

30 novembre 2020

Les bonnes pratiques d’encapsulation

Définition de la classe

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

Cette classe modélise une personne avec un âge, un prénom (*name*) et un nom de famille *surname*.

Utilisation de la classe

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

Problème

Cela fonctionne très bien mais va à l'encontre des règles d'encapsulation et de modularité : on ne devrait pas pouvoir modifier directement l'attribut `surname` d'une instance.

Solution

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.

Code Python

```python
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
```

Pourquoi 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.

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.

Utilisation

```
>>> 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
```

Les autres méthodes dunder de Python

La méthode `__str__`

On a déjà vu cette méthode, il y en a d'autres.

La méthode __equ__

Ce morceau de code nous navre :

```
>>> 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 :

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

solution

On peut redéfinir la méthode `__eq__` de la classe `Person` :

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

Et ainsi

```
>>> 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.

Dunders de conteneurs

```python
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])
```

Explications

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.

Utilisation

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

D'autres dunders 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)`	//

Exemple et surcharge

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

Définition
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.