Python : Programmation orientée objet

POO et gestion de librairie en Python

Base de la programmation orientée objet et gestion de modules

Jérôme Buisine
jerome.buisine@univ-littoral.fr

22 septembre 2022

Ce document propose des rappels concernant les concepts de la programmation orientée objet dans le langage Python.

1 Base de Python

Avant tout chose, vous pouvez vous documenter sur ces différents liens proposés ci-dessous pour comprendre les éléments et concepts de base de Python (présents dans tout autre langage) :

- Les variables et leurs différents types ;
- L'ajout de commentaires simples et complets ;
- Les chaînes de caractères et les listes (accès aux éléments, ajout/supression d'éléments) ;
- Les instructions de conditions ;
- Les boucles : for et while ;
- Définir des fonctions.

2 Classe, attribut et instance

En Python il est possible de créer une classe de la manière suivante (classe présente dans le fichier `dog.py`) :

2.1 Notion d'instance

```python
class Dog:

  def __init__(self, name, age):
    self.name = name
    self.age = age

  def make_noise(self):
    print(f"{self.name} says waf")
```

Indications :

— La méthode `__init__` définit le constructeur de la classe `Dog` ;

— Chaque méthode d'instance prend en premier paramètre l'instance courante reconnue par le mot clé `self` ;

— Il est possible d'affecter ou mettre à jour les attributs de l'instance.

La classe représente la structure d'un élément que l'on souhaite représenter. À partir de cette représentation il est possible d'instancier un objet (une instance). Voici un exemple d'instanciation dans le fichier `main.py` :

```
from dog import Dog

my_dog = Dog("Milou", 2)
my_dog.make_noise()
```

Remarques :

— Le fichier `dog.py`, se nomme en réalité **module** ;

— Il est possible d'importer les éléments d'un module, comme ici la classe `Dog`.

2.2 La notion de référence

Il faut rappeler que chaque instance possède sa propre mémoire allouée. Une instance qui pourtant semble identique, n'est pas pour autant égale :

```
from dog import Dog

my_dog_1 = Dog("Milou", 2)
my_dog_2 = Dog("Milou", 2)

my_dog_1 == my_dog_2 # returns false
```

2.3 Attributs de classe

Au sein d'une classe il est possible de définir des attributs lui appartenant (appellées attributs de classe) et donc différenciés d'attributs d'instance :

```
1  class Dog:
2
3    family = "Canidae"
4
5    def __init__(self, name, age):
6      self.name = name
7      self.age = age
8
9    def make_noise(self):
10     print(f"{self.name} says waf")
```

L'accès à ce type spécifique d'attribut est réalisé de la manière suivante :

```
from dog import Dog

print(f"Dog family is {Dog.family}")
```

Remarque : par convention, les variables de classes considérées comme constante sont nommées en majuscules. Par exemple : `FAMILY_NAME`.

2.4 Méthodes de classe

Il peut être parfois nécessaire de posséder des méthodes dites de classe (ou encore méthode statique). Si l'on reprend l'exemple précédent, on aurait pu envisager procéder comme suit :

```
1  class Dog:
2
3    def __init__(self, name, age):
4      self.name = name
5      self.age = age
6
7    def make_noise(self):
8      print(f"{self.name} says waf")
9
10   @staticmethod
11   def family():
12     return "family"
```

Ce qui implique maintenant d'appeler une méthode plutôt qu'un attribut, mais toujours depuis la classe directement :

```
from dog import Dog

print(f"Dog family is {Dog.family()}")
```

Remarque : le mot clé `self` n'apparaît pas comme premier paramètre de méthode étant donné que l'on ne traite pas d'instance.

2.5 Méthode d'instance spécifique

Il existe des méthodes particulières d'instance qui peuvent être surchargée, comme par exemple la méthode d'affichage `__str__` :

```
1  class Dog:
2
3    def __init__(self, name, age):
4      self.name = name
5      self.age = age
```

```
6
7   def make_noise(self):
8     print(f"{self.name} says waf")
9
10  @staticmethod
11  def family():
12    return "family"
13
14  def __str__(self):
15    return f"{self.name} has {self.age} years old"
```

Qui permet d'afficher directement le contenu d'une instance :

```
from dog import Dog

my_dog = Dog("Milou", 2)

print(my_dog)
```

Remarque : on appelle ce genre de méthodes les « magic methods ». Pour les curieux, vous pouvez voir les différentes méthodes magiques existantes.

3 Portées et propriétés

Par défaut, un attribut est considéré comme publique et donc accessible. Si l'on reprend l'exemple précédent, il serait possible de :

```
from dog import Dog

my_dog = Dog("Milou", 2)

print(my_dog.name)
print(my_dog.age)
```

Remarque : ce comportement est généralement peu souhaité. La gestion de portée des attributs reste donc importante.

3.1 Portée privée

Le principe d'encapsulation en Python est possible bien que limité. Par convention, on définit un attribut privé en le faisant précéder d'un simple « underscore » :

```
1 class Dog:
2
3   def __init__(self, name, age):
4     self._name = name
5     self._age = age
```

```
6
7   def get_name(self):
8     return self._name
```

Il reste possible d'accéder à l'attribut, mais de manière moins « directe » :

```
from dog import Dog

my_dog = Dog("Milou", 2)
print(dog._name)

# accès aux attributs de dog
dog.__dict__ # returns: {'_name': 'Rex', '_age': 2}
```

Remarque : l'accès reste donc possible. Cette convention stipule toutefois qu'il s'agit d'un attribut qui ne doit pas être modifié depuis l'extérieur même si vous n'êtes pas techniquement empêché de le faire. En réalité, Python abandonne cette prétention à la sécurité (liée à la portée des variables) et encourage les programmeurs à être responsables.

Il est également possible de manipuler des noms d'attributs avec double « underscore » :

```
1 class Dog:
2
3   def __init__(self, name, age):
4     self.__age = age
5         self.__name = name
6
7   def get_name(self):
8     return self.__name
```

En procédant ainsi, vous ne pouvez pas accéder directement à l'attribut « ___ » depuis l'extérieur d'une classe, mais plutôt par :

```
from dog import Dog

my_dog = Dog("Milou", 2)
print(dog._Dog__name)

# accès aux attributs de dog
dog.__dict__ # returns: {'_Dog__name': 'Rex', '_Dog__age': 2}
```

Remarque : le nom de l'attribut est automatiquement modifié pour l'accès extérieur. Cette dernière convention de nommage reste moins utilisée.

3.2 Accesseurs et mutateurs

L'encapsulation consiste à regrouper des données et des méthodes dans une classe afin de cacher les informations et d'en limiter l'accès depuis l'extérieur. Dans ce principe, Python propose des méthodes spécifiques pour faciliter l'encapsulation des attributs par le biais de `propriétés` :

```python
class Dog:

  def __init__(self, name, age):
    self._age = age
        self._name = name

  @property
  def name(self):
    return self._name

  @property
  def age(self):
        return self._age

  @age.setter
  def age(self, age):
         self._age = age
```

Ainsi, il est possible d'avoir un accès à nos attributs (du moins ceux souhaités) et modifier ceux qui peuvent l'être :

```python
from dog import Dog

dog = Dog("Rex", 2)

dog.name # returns: "Rex"
dog.age = 3
dog.age # returns: 3
```

Remarques :

— La « metadata » `property` définit une méthode comme attribut à accès direct ;

— Dans cet exemple on suppose que le nom du chien ne changera jamais, mais que son âge oui ;

— Pour ajouter une méthode de mutation d'un attribut, il faut forcément que la propriété soit définie.

4 Héritage et abstraction

L'héritage en Python est réalisé de la même manière que dans les autres langages.

4.1 Quelques exemples d'héritage

Voici les différentes points importants concernant l'héritage en Python :

— La classe fille hérite des attributs de la classe mère ;

— La classe fille hérite des méthodes non privées de la classe mère (nous verrons par la suite la notion de méthode privée) ;

— Il est bien sûr possible de définir des méthodes abstraites et donc des classes abstraites (composées d'au moins une méthode abstraite). Une méthode abstraite est définie par le « metadata » `abstractmethod` de la librairie `abc` ;

— Il est possible d'accéder aux méthodes de la classe mère par le mot clé `super()`.

En voici un premier exemple, où la classe `Dog` hérite des comportements de la classe `Animal` :

```
class Animal:

  def __init__(self, name, age):
    self._age = age
    self._name = name

  @property
  def name(self):
    return self._name

class Dog(Animal):

  def __init__(self, name, age):
    super().__init__(name, age)
```

De la même manière il est possible d'instancier un chien :

```
from dog import Dog

dog = Dog("Rex", 2)
dog.name # returns: "Rex"
```

4.2 Méthodes privées

On ne souhaite pas forcément qu'une méthode soit accessible au sein de la classe fille. Il est possible pour cela de préfixer le nom de la méthode par deux « underscore » :

```
class Animal:

  def __init__(self, name, age):
    self._age = age
    self._name = name

  def __private_method(self):
    print("This method is private!")

class Dog(Animal):

  def __init__(self, name, age):
    super().__init__(name, age)
```

Si un appel à cette méthode privée est réalisée depuis une classe fille, alors cela provoque une erreur :

```python
from dog import Dog

dog = Dog("Rex", 2)
dog.__private_method() # Error
```

4.3 Méthodes abstraites

Les méthodes abstraites en Python sont définies de la manière suivante :

```python
from abc import ABC, abstractmethod

class Animal(ABC):

  def __init__(self, name, age):
    self._age = age
    self._name = name

  @abstractmethod
  def make_noise(self):
    pass

class Dog(Animal):

  def __init__(self, name, age):
    super().__init__(name, age)

  def make_noise(self):
    print(f"{self._name} says waf")
```

Remarques :

— La spécificité de la méthode `make_noise` est définie par la classe `Dog`. Ce qui permet d'instancier des objets de la classe `Dog` ;

— La classe abstraite doit hériter de `ABC` pour être considérée comme telle. Sinon il reste possible d'instancier ;

— De ce fait et en l'état, il n'est aucunement possible de réaliser une instance de la classe `Animal`.

Un cas spécifique concerne celui où la méthode abstraite définie un premier comportement commun aux classes filles :

```python
from abc import ABC, abstractmethod

class Animal(ABC):

  def __init__(self, name, age):
    self._age = age
    self._name = name

```

```
  @abstractmethod
  def make_noise(self):
    print("This animal wants to say something")

class Dog(Animal):

  def __init__(self, name, age):
    super().__init__(name, age)

  def make_noise(self):
    super().make_noise()
    print(f'After few minutes... {self._name} says waf')
```

Il est ainsi possible d'appeler la méthode de la classe mère avant d'exécuter le contenu spécifique. Ce qui dans notre cas donne :

```
>>> from dog import Dog
>>> dog = Dog("Rex", 2)
>>> dog.make_noise()

This animal wants to say something
After few minutes... Rex says waf
```

4.4 Propriétés abstraites

On peut également définir qu'une propriété est abstraite, un attribut devra donc être détaillé dans la classe fille :

```
from abc import ABC, abstractmethod

class Animal(ABC):

  def __init__(self, name, age):
    self._age = age
    self._name = name

  @property
  @abstractmethod
  def human_age(self):
    pass

class Dog(Animal):

  def __init__(self, name, age):
    super().__init__(name, age)

  @property
  def human_age(self):
    return self._age * 7
```

Remarque : dans cet exemple on détaille que la classe chien possède une manière spécifique de calculer l'équivalent de l'âge humain relativement à l'âge de l'instance courante. On pourrait imaginer des comportements beaucoup plus complets, mais toujours considérés comme propriétés.

5 Simplification des imports

Au sein d'une librairie, il peut exister plusieurs modules. Généralement, on définit un seul fichier Python (module) pour une seule classe.

Remarque : le fichier `__init__.py` permet de reconnaître le dossier dans lequel il est présent comme module et donnant accès aux sous-modules.

Prenons la structure de librairie suivante :

```
kennel
├── species
│   ├── dog.py
│   ├── cat.py
│   └── __init__.py
├── animal.py
└── __init__.py
main.py
```

En l'état, il est possible d'importer les classes filles dans notre `main.py` comme dans cet exemple :

```python
from kennel.species.dog import Dog
from kennel.species.cat import Cat
```

On remarque une certaine redondance dans l'import, puisque l'on spécifie le module `dog` pour importer la classe `Dog` et de même pour `Cat`.

Pour améliorer les imports au sein de librairie, il est généralement proposé d'ajouter les imports directement en contenu du fichier « `__init__.py` ». En utilisant des imports relatifs, nous modifions le fichier « `kennel/species/__init__.py` » comme suit :

```python
"""Species module
"""
from .dog import Dog
from .cat import Cat
```

Ainsi, lors de l'utilisation de la librairie depuis l'extérieur (par exemple depuis notre `main.py`) les imports peuvent être réalisés plus proprement :

```python
from kennel.species import Dog
from kennel.species import Cat

# ou encore
from kennel.species import Dog, Cat
```