Le langage Python

Éric Jacoboni

16 octobre 2021

jacoboni@univ-tlse2.fr

Sommaire

Classes et POO

Définition d'une classe

- Une classe simple est introduite par le mot-clé *class*, suivi du nom de la classe :

```
class NomClasse:
  ... corps de la classe : définition de méthodes, affectation de variables.
```

- Attention, par défaut les classes Python sont plutôt permissives (tout est public, on peut rajouter des champs à un objet, etc.) :

```
class Cercle:
  pass

un_cercle = Cercle()   # Crée un objet Cercle
un_cercle.rayon = 5    # On peut lui rajouter un attribut à la volée !
del un_cercle.rayon    # Ou en supprimer...
```

- Généralement, on ajoute la méthode spéciale *__init__* pour initialiser une instance lors de sa création (celle-ci joue donc le rôle des « constructeurs » de C++ ou Java) :

```
class Cercle:
  def __init__(self, rayon):   # Toutes les méthodes doivent avoir self comme premier paramètre
    self.rayon = rayon         # self.rayon est une variable d'instance, rayon est le paramètre

un_cercle = Cercle(5)   # Appel implicite de Cercle.__init__(un_cercle, 5)
```

Définition

Quelques remarques importantes à savoir lorsque l'on a pratiqué d'autres langages de POO :

- Les méthodes d'instance doivent avoir un premier paramètre *self* qui désignera l'instance au moment de l'appel. Ce paramètre est passé implicitement à l'appel de la méthode.
- Les variables d'instance doivent obligatoirement être préfixées par *self* afin de les distinguer des variables locales et des paramètres. Python étant un langage dynamique, les variables d'instances sont créées lors de leur première affectation.
- Par défaut, les méthodes et les variables sont publiques et les classes sont « ouvertes » (on peut leur rajouter/ôter des variables et des méthodes depuis l'extérieur).
- Mais le programmeur peut créer des méthodes et des variables privées, empêcher la modification d'une classe, etc.

Variables d'instance et variables de classe

- Dans le corps de la classe, les variables d'instances sont préfixées par *self* et sont créées lors de leur première affectation (généralement, dans *__init__*).
- Toute variable initialisée en dehors de toute méthode est considérée comme une *variable de classe* : elle existe même si aucune instance n'a été créée et sa valeur est partagée par toutes les instances de la classe.
- Pour accéder à une variable de classe, il faut préfixer son nom du nom de la classe.
- Attention à certains problèmes (voir l'exemple).

```
class Cercle:
  pi = 3.14159                # Variable de classe
  def __init__(self, rayon):  # Toutes les méthodes doivent avoir self comme premier paramètre
    self.rayon = rayon        # self.rayon est une variable d'instance, rayon est le paramètre

print(Cercle.pi)              # Affiche 3.14159
c = Cercle(2)                 # Crée un cercle de rayon 2
print(c.rayon)                # Affiche 2
print(c.pi)                   # Affiche 3.14159
c.pi = 12                     # Ouch : on vient de créer une variable d'instance pi pour l'objet c
print(c.pi)                   # Affiche 12...
print(Cercle.pi)              # Affiche 3.14159
```

Variables d'instance et variables de classe

- Si l'on veut vraiment empêcher la création dynamique d'attributs d'instance, on peut utiliser la méthode spéciale *__setattr__* :

```
class Cercle:
  pi = 3.14159                # Variable de classe
  def __init__(self, rayon):  # Toutes les méthodes doivent avoir self comme premier paramètre
    self.rayon = rayon        # self.rayon est une variable d'instance, rayon est le paramètre

  def __setattr__(self, nom, val):  # Appelée à chaque affectation d'un attribut
    if nom not in ['rayon']:
       raise AttributeError(f"{nom} n'est pas défini")
    else:
       self.__dict__[nom] = val

c = Cercle(2)
c.pi = 12                     # AttributeError: pi n'est pas défini
c.pouet = 24                  # AttributeError: pouet n'est pas défini
c.rayon = 42                  # Ok
```

- Mais ce n'est que rarement nécessaire...

Méthodes d'instance et de classe

- Alors que les méthodes d'instance s'appliquent à une instance particulière de la classe, les méthodes de classe s'appliquent à *toutes* les instances d'une classe.
- Python permet de créer à la fois des méthodes *de classe* et des méthodes *statiques*. Leur principale différence est la syntaxe de leur définition.
- Une méthode de classe ou une méthode statique peut être appelé sur le nom de la classe où sur celui d'une instance.
- Les décorateurs *@classmethod* et *@staticmethod* permettent, respectivement, de définir une méthode de classe et une méthode statique.
- Évidemment, une méthode statique ou de classe qui manipulerait une variable d'instance n'aurait aucun sens (alors qu'une méthode d'instance peut manipuler une variable de classe).

Exemple de méthode statique

Fichier cercle.py

```
class Cercle:
  # Variables de classe
  tous = []
  pi   = 3.14159

  def __init__(self, rayon):
     self.rayon = rayon
     Cercle.tous.append(self)   # On ajoute le cercle à la liste de tous les cercles
                                # Ou : self.__class__.tous.append(self)
  def surface(self):
     return Classe.pi * self.rayon**2

  @staticmethod
  def surface_totale():
     total = 0
     for c in Cercle.tous:
        total += c.surface()
     return total
```

Exemple d'utilisation :

```
import cercle
c = cercle.Cercle(4)
c2 = cercle.Cercle(3)
print(cercle.Cercle.surface_totale())        # 78.53975
```

Exemple de méthode de classe

Fichier cercle.py

```
class Cercle:
  ... idem précédemment ...

  @classmethod
  def surface_totale(cls):
     total = 0
     for c in cls.tous:
        total += c.surface()
     return total
```

- L'avantage d'utiliser une méthode de classe est que cette méthode connaît la classe via le paramètre qui lui a été passé (*cls*, ici). On peut donc écrire simplement *cls.tous* dans le corps de la méthode.
- Une méthode statique n'a pas ce paramètre et doit donc soit coder « en dur » le nom de la classe (*Cercle.tous*, dans notre exemple), soit passer par la variable spéciale *__class__*.
- Une méthode de classe peut également être redéfinie dans une sous-classe, pas une méthode statique... Mon conseil : utilisez plutôt des méthodes de classe.

Variables et méthodes privées

- Par défaut, tous les membres d'une classe sont *publics* : les variables d'instance notamment, peuvent être modifiées depuis l'extérieur, ce qui nuit au principe d'encapsulation des données.
- En Python, il n'existe pas de mot-clé *private* comme en Java, C++ ou C# : pour créer des variables et des méthodes privées, il suffit de préfixer leur nom par deux blancs soulignés.
- En réalité, l'attribut (ou la méthode) ne sont pas « privés » : leur nom est simplement modifié par Python pour compliquer leur accès depuis l'extérieur de la classe.
- Attention : un nom de membre privé ne doit pas se terminer par deux blancs soulignés (ce format est réservé aux noms spéciaux de Python, comme *__init__* ou *__doc__*).
- L'avantage de cette convention est qu'elle facilite l'identification des membres privés.

Exemple

Fichier point.py

```
import math

class Point:
  "Classe définissant un point du plan"

  def __init__(self, x, y):
    "Crée le point d'abscisse x et d'ordonnée y"
    self.__x, self.__y = x, y                    # __x et __y sont "privées"

  def getX(self): return self.__x
  def getY(self): return self.__y

  def __distance_origine(self):                  # __distance_origine est "privée"
     return math.hypot(self.__x, self.__y)

  def getDistance(self):
     "Renvoie la distance du point par rapport à l'origine (0,0)"
     return self.__distance_origine()
```

Exemple d'utilisation :

```
from point import Point
p = Point(3, 2)
p.__x                   # AttributeError: 'Point' object has no attribute '__x'
p.getX()                # 3
p.__distance_origine()  # AttributeError: 'Point' object has no attribute '__distance_origine'
p.getDistance()         # 3.605551275463989
```

Propriétés

- Au lieu d'utiliser des accesseurs Java-esque comme *getX()* ou *setX(valeur)* pour lire ou modifier des variables d'instances privées, il est préférable d'utiliser des *propriétés*.
- L'avantage des propriétés est qu'elles allègent le code utilisateur et qu'elles assurent le principe d'accès uniforme : l'utilisateur d'une classe n'a pas besoin de savoir si une information lui est fournie par une méthode ou par une variable.
- Ce mécanisme est également utilisé par d'autres langages, comme C# ou Ruby.
- Une propriété de lecture est introduite par le décorateur *@property*.
- Si l'on veut lui ajouter une propriété d'écriture, on ajoute à cette propriété un décorateur « setter » (voir exemple).

Exemple

```
import math

class Point:
  """Classe définissant un point du plan"""

  def __init__(self, x, y):
    """Crée le point d'abscisse x et d'ordonnée y"""
    self.__x, self.__y = x, y

  @property
  def x(self): return self.__x
  @x.setter
  def x(self, new_x): self.__x = new_x

  @property
  def y(self): return self.__y
  @y.setter
  def y(self, new_y): self.__y = new_y

  @property
  def distance(self): return math.hypot(self.__x, self.__y)
```

Exemple d'utilisation :

```
from point import Point
p = Point(3, 2)
p.x                    # 3 (utilisation de la propriété en lecture x)
p.y = 4                # utilisation de la propriété en écriture y
p.distance             # 5.0 (utilisation de la propriété en lecture distance)
```

Héritage

- Une classe Python peut hériter d'une ou plusieurs classes. En réalité, les classes précédentes héritaient de *object*, la classe racine de la hiérarchie des classes Python.
- Contrairement à Java, Ruby ou C#, Python autorise l'héritage multiple : une classe peut hériter de plusieurs classes.
- Une classe hérite de tous les membres non privés de ses super-classes et elle peut redéfinir certaines méthodes.
- Un appel à *super().une_methode()* permet d'appeler la méthode *une_methode()* définie dans une super-classe (mais son utilisation est critiquée par certains, **notamment par moi**...).

Héritage

- L'ordre de recherche des attributs part de la classe, puis remonte dans la première super-classe de la liste, puis remonte dans les super-classes de celle-ci. La recherche se poursuit avec la seconde super-classe, etc. On a donc une recherche de gauche à droite, en profondeur d'abord.
- La fonction *isinstance(obj, cls)* teste si un *obj* est une instance de *cls* ou de l'une de ses sous-classes.
- La fonction *issubclass(cls1, cls2)* teste si la classe *cls1* est une sous-classe de *cls2*.

Exemple

```
class PointNomme(Point):
  """Classe définissant un point étiqueté"""

  def __init__(self, x, y, nom):
     Point.__init__(self, x, y)            # Appel du constructeur de Point
     # Ou : super().__init__(x, y)
     self.__nom = nom

  @property
  def nom(self): return self.__nom
```

Exemple d'utilisation :

```
import point2

p = point2.PointNomme(3, 2, "Point 1")
p.nom                                  # 'Point 1'
p.x                                    # 3
p.distance                             # 3.605551275463989
p.y = 3
p.distance                             # 4.242640687119285
p.__class__                            # <class 'point2.PointNomme'>
```

Polymorphisme

Toutes les méthodes d'une classe Python sont virtuelles, ce qui signifie qu'elles peuvent être redéfinies dans les classes filles :

```
# Module polymorph.py

class ClasseBase:

  def __init__(self, x, y):
    self.x, self.y = x, y

  def deplace(self, delta_x, delta_y):
     self.efface()
     self.x += delta_x
     self.y += delta_y
     self.affiche()

  def efface(self):   # on ne sait pas encore le faire...
    pass              # Ou  ...  (en Python 3)

  def affiche(self):  # on ne sait pas encore le faire...
    pass

class Fille(ClasseBase):

   def __init__(self, x, y, nom):
      ClasseBase.__init__(self, x, y)      # Ou super().__init__(x, y)
      self.nom = nom

   def efface(self):
      print(f"{self.nom} s'efface...")

   def affiche(self):
      print(f"{self.nom} s'affiche...")
```

Polymorphisme

Exemple d'utilisation :

```
import polymorph

f = polymorph.Fille(10, 12, "fille")
f.x                                     # 10
f.y                                     # 12
f.deplace(10, 10)                       # deplace() est définie dans ClasseBase
                                        # Affiche 'fille s'efface...'
                                        # Affiche 'fille s'affiche...'
f.x                                     # 20
f.y                                     # 22
```

- L'appel *f.deplace(...)* a appelé la méthode *deplace()* de la super-classe.
- Comme la classe *Fille* redéfinit les méthodes *efface()* et *affiche()*, la méthode *deplace()* les appelle car ce sont elles qui sont « les plus proches » de l'objet *f*.
- On a donc un « double dispatch » : comme *Fille* ne définit pas *deplace()*, on remonte vers sa super-classe pour trouver *deplace()*, puis on redescend vers la classe de *f* pour trouver les méthodes *efface()* et *affiche()* les plus spécialisées.

Représentation textuelle

- Les deux méthodes spéciales *__repr__* et *__str__* permettent de gérer la représentation textuelle d'un objet. Ces deux méthodes sont appelées automatiquement par les fonctions *repr()* et *str()*.
- Par défaut, toutes les deux produisent une chaîne décrivant l'objet : *<point3.Point object at 0x7f5a2f857910>*, par exemple.
- La méthode *__repr__* est censée produire une représentation textuelle de l'objet. Cette représentation n'est pas destinée à être lue par un humain, mais doit permettre de recréer l'objet via un appel à *eval()* (cf. exemple). Elle est appelée automatiquement par la fonction *repr(obj)* (et par idle ou l'interpréteur intéractif).
- La méthode *__str__* est censée produire une représentation lisible de l'objet. Elle est appelée automatiquement par les instructions comme *print* et par la fonction de conversion *str(obj)*.

Exemple

```
# Module point4.py

class Point:
  (...)
  def __repr__(self):
    return f"Point({self.x}, {self.y})"

  def __str__(self):
    return f"({self.x}, {self.y})"

class PointNomme(Point):
  (...)
  def __repr__(self):
    return f"PointNomme({self.x}, {self.y}, {self.nom})"

  def __str__(self):
    return f"({self.x}, {self.y}, {self.nom})"
```

Exemple d'utilisation :

```
from point4 import Point

p = Point(3, 2)
repr(p)                                 # Point(3, 2)
q = eval(repr(p))                       # exécute donc q = Point(3, 2)...
r = eval(p.__module__ + '.' + repr(p))  # Si on avait simplement fait 'import point4'
q == p                                  # True
print(p)                                # Affiche (3, 2)  (appel de str(p))
```