Programmation Orientée Objet

1 Introduction

2 Classe

Les classes sont un moyen de définir de nouveaux types modifiables de variables. Peu de programmes ne les utilisent pas. Une classe est un ensemble d'attributs (ou variables) et de méthodes (ou fonctions). Un programme utilisant les classes est orienté objet. Il est possible de faire les mêmes choses avec ou sans classes mais leur utilisation rend d'ordinaire les grands programmes plus facile à comprendre et à construire.

2.1 Déclaration d'une classe

Pour créer une nouvelle classe d'objets en Python, on utilise le mot clé **class** qui permet de déclarer un nouveau type en Python.
Exemple :

```
class ma_classe :
    def __init__ (self, att1, att2) :
        self.att1 = att1
        self.att2 = att2
        self.att3 = att1 + att2
        self.att4 = self.calculer()

    def calculer (self) :
        return self.att1 * self.att2
```

3 Objet

3.1 Création d'une objet (instance)

Syntaxe : `var = Classe(args)`

Exemple : Pour déclarer une variable (instance) de type *ma_classe*

```
a = ma_classe(1,2)
print(a.att1) # affiche 1
```

Lors de la déclaration de la variable *a*, le langage Python exécute la méthode `__init__` aussi appelée **constructeur**.

Elle permet de définir les attributs de la classe directement à partir des paramètres ou comme le résultat d'un calcul ou d'une fonction.

Le constructeur comme toutes les autres méthodes possède comme premier paramètre **self**.
self : désigne l'objet non encore instancié, de type *ma_classe*, qui permet d'accéder aux attributs et aux méthodes de la classe.
A la suite de création d'objet (à la fin de l'instanciation `ma_classe(1,2)`), Python retourne la *référence* de l'objet créé. Si cette référence est non sauvegardée, l'objet récemment créé sera automatiquement supprimé par Python à travers le mécanisme nommé *Garbage collector*.
Dans l'exemple précédent, la variable `a` enregistre la référence (l'adresse) de l'objet créé.

Pour connaître la classe (type) d'un objet ou savoir si un objet appartient à une classe, Python propose deux fonctions `type(a)` et `isinstance(a,ma_classe)`

3.2 Copie d'instances

Les instances de classes sont des objets modifiables, comme pour les listes, une simple affectation ne signifie pas une copie mais un second nom pour désigner le même objet.

```
class ma_classe :
    def __init__ (self, att1, att2) :
        self.att1 = att1
        self.att2 = att2
a = ma_classe (1,2)
b = a
b.att1 = -16
print (a.att1) # affiche -16
print (b.att1) # affiche -16
```

Il faut donc copier explicitement l'instance pour obtenir le résultat souhaité.

```
a = ma_classe (1,2)
import copy
b = copy.copy (a)
b.att1 = -16
print (a.att1) # affiche 1
print (b.att1) # affiche -16
```

Lorsque une classe inclut des attributs de type modifiable, il faut utiliser la fonction `deepcopy` et non `copy`.

4 Attributs

Les attributs sont déclarés le plus souvent à l'*intérieur du constructeur*, plus généralement à l'***intérieur de toute méthode***, voire à l'*extérieure de la classe*. Pour y faire référence

- à l'intérieur d'une méthode on fait précéder le nom de l'attribut par « **self.** »
- à l'extérieur de la classe, c'est le nom de l'instance suivi d'un point « . » qui précède le nom de l'attribut comme le montre le précédent exemple.

Les attributs d'un objet peuvent être accessibles et modifiables après la construction de l'objet.
Lorsque vous accédez à un attribut d'un objet qui n'existe pas, Python créera simplement un nouvel attribut associé uniquement à cet objet.
Donc en Python, il est toujours possible après la définition d'une classe, d'ajouter des attributs pour une instance particulière de cette classe.

Remarque : En Python un attribut **privé** est un attribut accessible seulement dans la classe où il est défini.
Syntaxe : `self.__attribut`

4.1 Attributs de classes

5 Méthodes

Une méthode est déclarée à l'*intérieur de la classe* et qui utilise ou/et modifie les attributs de l'objet de cette classe qui lui fait appel.
Elle accepte invariablement au moins un paramètre qui est **self** qui sera remplacé lors de l'appel de la fonction par l'objet lui-même (l'objet sur lequel on effectue le traitement).
Les règles d'accès sont les mêmes que pour les attributs. Elles acceptent également la récursivité et les paramètres par défaut à l'exception du premier. Chaque instance de classe est également munie d'un dictionnaire **__dict__** qui recense tous les attributs.

Python offre la possibilité d'appeler une méthode de deux manières :

- `nom_objet.nom_methode( )`
- `nom_classe.nom_methode(nom_objet)`

```
class ma_classe :
    def __init__ (self, att1, att2) :
        self.att1 = att1
        self.att2 = att2
        self.att3 = att1 + att2
        self.att4 = self.calculer(att2)

    def calculer (self,x) :
        return self.att1 * x

a = ma_classe (1,2)
print (a.att1) # affiche 1
print (a.__dict__["att1"]) # affiche aussi
 ↪  1, ligne équivalente à la précédente
print (a.calculer(2)) # appel d'une méthode
```

Remarque : En Python une méthode **privée** est une méthode accessible seulement dans la classe où elle est définie.
Syntaxe : `def __methode(self)`

5.1 Méthodes statiques

Les méthodes statiques sont comme des fonctions : elle ne nécessite pas d'instance d'une classe pour être appelée.

```
class ma_classe :
    def __init__ (self, att1, att2, att3) :
        # ...

    def calcul_static (x,y) : # méthode statique
        return x * y

print (ma_classe.calcul_static(2,3)) # appel d'une méthode statique
```

6 Héritage

L'héritage est l'intérêt majeur des classes et de la programmation orientée objet. Lorsqu'une classe hérite d'une autre, elle hérite de ses attributs et de ses méthodes. Le simple fait d'hériter crée donc une classe équivalente.

```
class ma_classe :
    def __init__ (self, att1, att2) :
        self.att1 = att1
        self.att2 = att2

class ma_classe2 (ma_classe) :  # héritage simple
    pass                        # pass : pour dire que la classe est vide
```

Mais hériter permet de faire deux choses :

1. Ajouter des attributs et des méthodes.
2. Modifier le comportement d'une méthode existante.

```
class ma_classe :
    def __init__ (self, att1) :
        self.att1 = att1
        self.att2 = self.calculer ()
    def calculer (self) :
        return self.att1 ** 2
```

Dans la sous-classe suivante, on ajoute un attribut d'instance et on change le comportement de la méthode calculer tout en se servant de celui de la classe mère.

```
class ma_classe2 (ma_classe) :  # classe fille
    def __init__(self,att1):
        ma_classe.__init__(self,att1) # appel au constructeur de la classe mère
        self.att3 = 0 # Ajouter un attribut à la sous-classe

    # modifier le comportement de la méthode calculer
    def calculer (self) :
        return ma_classe.calculer(self) * self.att1

    def afficher(self):
        print(self.att1, self.att2, self.att3)
```

```
a = ma_classe (2)
b = ma_classe2 (2)
print (a.att2) # affiche 4 = 2 * 2
print (b.att2) # affiche 8 = (2*2) * 2
```

6.1 Méthodes spéciales

Une méthodes spéciale (magique) en Python est une fonction déclarée à l'intérieur d'une classe dont le nom doit être impérativement délimité par deux traits de soulignement « _ » et n'est pas invoquée directement par son propre nom, mais elle invoquée implicitement :

- soit via une autre fonction.

  Exemples :

  `__str__` retourne une chaîne de caractères, et appelée par `str()`.

  `__len__` est appelée implicitement lorsqu'on utilise la fonction len().

- soit via l'utilisation d'un opérateur arithmétique ou logique.

  Exemples :

  `__add__` est appelée implicitement lorsqu'on utilise l'opérateur « + ».

  `__cmp__` retourne un entier permettant de comparer des instances d'une classe.

6.1.1 Surcharge des opérateurs

Les opérateurs sont des méthodes spéciales qui permettent une manipulation plus simple des objets.
Il existe un opérateur spécifique pour chaque opération, cet opérateur permet de donner un sens à une addition, une soustraction, ..., de deux instances d'une classe.
Leur nom est fixé par convention par le langage Python, ils commencent et terminent par `__` (deux underscores '_').
Exemples : `__add__`, `__sub__`, `__mul__`, `__floordiv__` (division entière //) , `__eq__`

```
class ma_classe :
    def __init__ (self, att1, att2) :
        self.att1 = att1
        self.att2 = att2

    def __add__ (self, other) :
        return ma_classe (self.att1 + other.att1, self.att2 + other.att2)

a = ma_classe (1,2)
b = ma_classe (4,5)
c = a + b # n'a de sens que si l'opérateur __add__ a été redéfini
```