Python

Orienté Objet

By B. Raynal

Terminologie

Objet : Structure de données contenant des données et des algorithmes

Classe : prototype défini par l'utilisateur qui décrit l'ensemble des **attributs** qui caractérisent tout objet de cette classe.

Instance: un objet d'une classe A est une instance de A.

Terminologie

Attributs:

- **variable d'instance :** variable propre à une instance (un objet) en particulier
- **variable de classe :** variable partagée entre toutes les instances de la classe
- **méthode :** fonction définie dans la classe, dont le fonctionnement est relatif au contenu de l'instance

Définir une classe : exemple

```
class Employee:
  'Common base class for all employees'
  emp_count = 0

  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.emp_count  += 1

  def display(self):
    print ("Name : ", self.name,  ", Salary: ", self.salary)
```

Définir une classe : exemple

```
class Employee:
  'Common base class for all employees'
  emp_count = 0

  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.emp_count  += 1

  def display(self):
    print ("Name : ", self.name,  ", Salary: ", self.salary)
```

nom de la classe

Définir une classe : exemple

```
class Employee:
  'Common base class for all employees'
  emp_count = 0

  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.emp_count  += 1

  def display(self):
    print ("Name : ", self.name,  ", Salary: ", self.salary)
```

nom de la classe

doc string

Définir une classe : exemple

```
class Employee:
  'Common base class for all employees'
  emp_count = 0

  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.emp_count  += 1

  def display(self):
    print ("Name : ", self.name,  ", Salary: ", self.salary)
```

nom de la classe

doc string

variable de classe

Définir une classe : exemple

```
class Employee:
  'Common base class for all employees'
  emp_count = 0

  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.emp_count  += 1

  def display(self):
    print ("Name : ", self.name,  ", Salary: ", self.salary)
```

nom de la classe

doc string

variable de classe

méthode d'initialisation

Définir une classe : exemple

```
class Employee:
  'Common base class for all employees'
  emp_count = 0

  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.emp_count  += 1

  def display(self):
    print ("Name : ", self.name,  ", Salary: ", self.salary)
```

nom de la classe

doc string

variable de classe

méthode d'initialisation

variables d'instance

Définir une classe : méthodes

Les méthodes sont les fonctions propres à une classe.

Elles sont définie dans le corps de celle-ci, et leur premier argument est toujours **self**.

self n'a pas besoin d'être mis dans les paramètres lors de l'appel aux méthodes, il est ajouté automatiquement.

self est une référence à l'objet sur lequel on appelle les méthodes, et permet d'accéder aux variables d'instances.

Définir une classe : initialiseur

La méthode __init__ est un **initialiseur**.

Elle sert à initialiser les instances.

C’est dans cette méthode que l’on défini les variables d’instances.

Creer une instance

Pour créer une instance d'une classe, on appelle la classe en passant son nom et en fournissant tout paramètre demandé par **__init__**.

```
emp1 = Employee("Tom", 2000)
emp2 = Employee("Bob", 5000)
```

Accès aux attributs

Pour accéder aux attributs, on utilise l'opérateur point .

```
emp1.display()
emp2.display()
print ("Total Employee %d" % Employee.emp_count)
```

Accès aux attributs

On peut facilement ajouter, supprimer ou modifier des variables d'instance :

```
emp1.age = 7  # Ajoute un attribut 'age'.
emp1.age = 8  # Modifie l'attribut 'age'.
del emp1.age  # Supprime l'attribut 'age'.
```

Accès aux attributs

Pas d’encapsulation….

Accès aux attributs

On peut mettre un attribut privé en le faisant commencer par un double _

Pour accéder à l'attribut x, on devrait utiliser des accesseurs : **get_x** et **set_x**

Propriétés

Cependant, on aimerait quand même accéder aux attributs de façon simple :

my_obj.my_attr = x

print(my_obj.my_attr)

Mais en ayant derrière un contrôle sur ce qui se passe.

Propriétés

Pour cela, Python propose les propriétés :

- on défini un getter et un setter : _get_x et _set_x
  - Ils manipulent **self._x** et non **self.x**
- on défini x comme étant une propriété :
  - x = property(_get_x, _set_x)

On utilise ensuite directement obj.x

Propriétés

```
class Employee:
   'Common base class for all employees'
    def _get_name(self):
        return self._name
   def _set_name(self, name):
       self._name = name
   name = property(_get_name, _set_name)
```

Surcharge opérateurs

On peut surcharger les opérateurs en python :

Soit une classe X, avec deux instances x1 et x2.

Si X contient une méthode __add__(self, x)

alors on peut écrire x1+x2

Surcharge opérateurs

La liste de tous les opérateurs est disponible dans la documentation.

https://docs.python.org/3/reference/datamodel.html#special-method-names

Les plus souvent surchargés sont :

- __str__ : pour la conversion en string
- __repr__ : pour l'affichage dans l'interpreteur

Héritage

L’héritage permet d’ajouter des fonctionnalités à une classe existante, en héritant de toutes les fonctionnalités de celle-ci.

Héritage

```
class ITGuy(Employee):
 def __init__(self, name, salary, computer):

    Employee.__init__(self, name, salary)
    self.computer = computer



Bob = ITGuy("Bob", 2000, "ASUS")
```

Heritage multiple

Une alternative est de considérer qu'un ITGuy est à la fois un Employee et un Geek (qui a un nom et un ordinateur)

Héritage

```
class Geek:
 def __init__(self, name, computer):

    ....
class ITGuy(Employee,Geek):
 def __init__(self, name, salary, computer):

    Employee.__init__(self, name, salary)
    Geek.__init__(self, name, computer)


Bob = ITGuy("Bob", 2000, "ASUS")
```