Programmation objet

qkzk

2019/12/25

La programmation objet

Les objets : un moyen de séparer la conception de l'utilisation

La programmation orientée objet repose, comme son nom l'indique, sur le concept d'objet.

Un objet dans la vie courante, vous connaissez, mais en informatique, qu'est ce que c'est ? Une variable ? Une fonction ? Ni l'un ni l'autre, c'est un nouveau concept.

Imaginez un objet très complexe (par exemple un moteur de voiture) : il est évident qu'en regardant cet objet, on est frappé par sa complexité (pour un non spécialiste).

Imaginez que l'on enferme cet objet dans une caisse et que l'utilisateur de l'objet n'ait pas besoin d'en connaître son principe de fonctionnement interne pour pouvoir l'utiliser.

L'utilisateur a, à sa disposition, des boutons, des manettes et des écrans de contrôle pour faire fonctionner l'objet, ce qui rend son utilisation relativement simple. C'est ce qu'on fait quand on conduit !

La mise au point de l'objet (par des ingénieurs) a été très complexe, en revanche son utilisation est relativement simple. Programmer de manière orientée objet, c'est un peu reprendre cette idée : *utiliser des objets sans se soucier de leur complexité interne.*

Pour utiliser ces objets, nous n'avons pas à notre disposition des boutons, des manettes ou encore des écrans de contrôle, mais des *attributs* et des *méthodes* (nous aurons l'occasion de revenir longuement sur ces 2 concepts).

Les classes

Un des nombreux avantages de la programmation orientée objet (POO), est qu'il existe des milliers d'objets (on parle plutôt de classes, mais là aussi nous revien-

drons sur ce terme de classe est peu plus loin) prêts à être utilisés (vous en avez déjà utilisé de nombreuses fois sans le savoir). On peut réaliser des programmes extrêmement complexes uniquement en utilisant des classes préexistantes.

Les idées sous-tendant le paradigme objet datent des années 60. Mais il faudra attendre le début des années 70 et la mise au point du langage SmallTalk pour que le paradigme objet gagne en popularité chez les informaticiens. Aujourd'hui de nombreux langages permettent d'utiliser le paradigme objet : C++, Java, C#, Javascript, Python...

Python permet d'utiliser le paradigme impératif (ce que nous avons fait jusqu'à présent), mais il permet aussi d'utiliser le paradigme objet. Il est même possible, comme nous le verrons plus loin, d'utiliser les 2 paradigmes dans un même programme.

La création d'une classe en python commence toujours par le mot `class`. Ensuite toutes les instructions de la classe seront indentées :

```
class LeNomDeMaClasse:
  # instructions de la classe
# La définition de la classe est terminée.
```

La classe est une espèce de moule (nous reviendrons plus tard sur cette analogie qui a ses limites), à partir de ce moule nous allons créer des objets (plus exactement nous parlerons d'*instances*).

Par exemple, nous pouvons créer une classe voiture :

```
class Voiture:
  # tout le code propre aux objets Voiture
```

Puis créer différentes instances de cette classe (Peugeot407, Renault Espace,...). Pour créer une de ces instances, la procédure est relativement simple :

```
peugeot407 = Voiture()
```

Cette ligne veut tout simplement dire : "crée un objet (une instance) de la classe Voiture que l'on nommera peugeot407."

Ensuite, rien ne nous empêche de créer une deuxième instance de la classe Voiture :

```
renaultEspace = Voiture()
```

Nous rencontrons ici la limite de notre analogie avec le moule. En effet 2 objets fabriqués avec le même moule seront (définitivement) identiques, alors qu'ici nos 2 instances pourront évoluer différemment.

Premier programme

Pour développer toutes ces notions (et d'autres), nous allons écrire un premier programme :

Nous allons commencer par écrire une classe `Personnage` (qui sera dans un premier temps une coquille vide) et, à partir de cette classe créer 2 instances : `bilbo` et `gollum`.

Ensuite ils vont se taper, parce que je l'aime pas Gollum.

À faire vous-même 1

Saisissez, analysez et testez ce code

```
class Personnage:
  pass
gollum = personnage()
bilbo = personnage()
```

Attributs

Pour l'instant, notre classe ne sert à rien et nos instances d'objet ne peuvent rien faire. Comme il n'est pas possible de créer une classe totalement vide, nous avons utilisé l'instruction `pass` qui ne fait rien. Ensuite nous avons créé 2 instances de la classe Personnage : gollum et bilbo.

Comme expliqué précédemment, une instance de classe possède des attributs et des méthodes. Commençons par les attributs :

Un attribut est une variable spécifique à la classe.

Nous allons associer un attribut vie à notre classe personnage (chaque instance aura un attribut vie, quand la valeur de vie deviendra nulle, le personnage sera mort !)

Ces attributs s'utilisent comme des variables, l'attribut vie pour `bilbo` sera noté

```
bilbo.vie
```

de la même façon l'attribut vie de l'instance `gollum` sera noté

```
gollum.vie
```

À faire vous-même 2

Saisissez, analysez et testez ce code

```
class Personnage:
  pass
gollum=Personnage()
gollum.vie=20
bilbo=Personnage()
bilbo.vie=20
```

Comme pour une variable il est possible d'utiliser la console Python pour afficher la valeur référencée par un attribut. Il suffit de taper dans la console

`gollum.vie` ou `bilbo.vie` (sans bien sûr avoir oublié d'exécuter le programme au préalable...)

Cette façon de faire n'est pas très "propre" et n'est pas une bonne pratique

En effet, nous ne respectons pas un principe de base de la POO : l'**encapsulation**

Il ne faut pas oublier que notre classe doit être "enfermée dans une caisse" pour que l'utilisateur puisse l'utiliser facilement sans se préoccuper de ce qui se passe à l'intérieur, or, ici, ce n'est pas vraiment le cas.

En effet, les attributs (gollum.vie et bilbo.vie), font partie de la classe et devraient donc être enfermés dans la "caisse" !

Méthode

Pour résoudre ce problème, nous allons définir les attributs, dans la classe, à l'aide d'une **méthode** d'initialisation des attributs.

définition : **une méthode est une fonction définie dans une classe**

Cette méthode est définie dans le code source par la ligne :

```
def __init__ (self):
```

La méthode `__init__` est automatiquement exécutée au moment de la création d'une instance. La variable `self` est obligatoirement le premier argument d'une méthode (nous reviendrons ci-dessous sur ce mot `self`)

Nous retrouvons ce mot self lors de la définition des attributs. La définition des attributs sera de la forme :

```
self.vie=20
```

Le mot self représente **l'instance**. Quand vous définissez une instance de classe (`bilbo` ou `gollum`) le nom de votre instance va remplacer le mot self.

Dans le code source, nous allons avoir :

```
class Personnage:
  def __init__ (self):
    self.vie=20
```

Ensuite lors de la création de l'instance `gollum`, python va automatiquement remplacer `self` par `gollum` et ainsi créer un attribut gollum.vie qui aura pour valeur de départ la valeur donnée à self.vie dans la méthode `__init__`

Il se passera exactement la même chose au moment de la création de l'instance `bilbo`, on aura automatiquement la création de l'attribut `bilbo.vie`.

À faire vous-même 3

Saisissez, analysez et testez ce code

```
class Personnage:
  def __init__(self):
    self.vie = 20
bilbo = Personnage()
gollum = Personnage()
```

Utilisez la console Python comme dans le "À faire vous-même 2"

Le résultat du "À faire vous-même 3" est identique au résultat de l'exemple du "À faire vous-même 2". Mais cette fois nous n'avons pas défini l'attribut `gollum.vie=20` et `bilbo.vie=20` en dehors de la classe, nous avons utilisé une méthode `__init__`.

C'est une meilleure pratique.

Passer des paramètres à une instance de classe

Imaginons que nos 2 personnages n'aient pas au départ les mêmes points de vie ! Pour l'instant, impossible d'introduire cette contrainte (`self.vie=20`)

Une méthode, comme une fonction, peut prendre des paramètres.

Le passage de paramètres se fait au moment de la création de l'instance :

À faire vous-même 4

Saisissez, analysez et testez ce code

```
class Personnage:
  def __init__(self, point_de_vie):
    self.vie = point_de_vie
gollum = Personnage(20)
bilbo = Personnage(15)
```

Utilisez la console Python pour vérifier que `gollum.vie` est égal à 20 et `bilbo.vie` est égal à 15

Au moment de la création de l'instance gollum, on passe comme argument le nombre de vies (`gollum=Personnage (20)`).

Ce nombre de vies est attribué au premier argument de la méthode `__init__`, la variable `point_de_vie` (`point_de_vie` n'est pas tout à fait le premier argument de la méthode `__init__` puisque devant il y a `self`, mais bon, `self` étant

obligatoire, nous pouvons dire que `point_de_vie` est le premier argument non obligatoire).

N.B. Je parle bien de variable pour `point_de_vie` (car ce n'est pas un attribut de la classe personnage puisqu'elle ne commence pas par self).

Nous pouvons passer plusieurs arguments à la méthode `__init__` (comme pour n'importe quelle fonction).

Nous allons créer 2 nouvelles méthodes :

- Une méthode qui enlèvera un point de vie au personnage blessé
- Une méthode qui renverra le nombre de vies restantes

À faire vous-même 5

Saisissez, analysez et testez ce code

```
class Personnage:
  def __init__(self, point_de_vie):
    self.vie = point_de_vie

  def get_etat(self):
    return self.vie

  def perd_vie(self):
    self.vie = self.vie-1

gollum = Personnage(20)
bilbo = Personnage(15)
```

Pour tester ce programme, dans la console, tapez successivement les instructions suivantes :

```
>>> gollum.get_etat()
>>> bilbo.get_etat()
>>> gollum.perd_vie()
>>> gollum.get_etat()
>>> bilbo.perd_vie()
>>> bilbo.get_etat()
```

Vous avez sans doute remarqué que lors de "l'utilisation" des instances biblo et gollum, nous avons uniquement utilisé des méthodes et nous n'avons plus directement utilisé des attributs (plus de "gollum.vie").

Encapsulation et interface

Il est important de savoir qu'en dehors de la classe l'utilisation des attributs est une mauvaise pratique en programmation orientée objet : les **attributs**

doivent rester "à l'intérieur" de la classe, l'utilisateur de la classe ne doit pas les utiliser directement.

Il peut les manipuler, mais uniquement par l'intermédiaire d'une méthode (la méthode `self.perd_vie` permet de manipuler l'attribut `self.vie`).

Les méthodes constituent "l'interface" de l'objet.

Pour l'instant nous avons utilisé les méthodes uniquement en tapant des instructions dans la console, il est évidemment possible d'utiliser ces méthodes directement dans votre programme :

À faire vous-même 6

Saisissez, analysez et testez ce code

```
class Personnage:

  def __init__(self, point_de_vie):
    self.vie = point_de_vie

  def get_etat (self):
    return self.vie

  def perd_vie (self):
    self.vie = self.vie - 1

bilbo = Personnage(15)
bilbo.perd_vie()
point = bilbo.get_etat()
```

Évaluez la variable point à l'aide de la console.

À faire vous-même 7

Nos personnages peuvent boire une potion qui leur ajoute un point de vie. Modifiez le programme du "À faire vous-même 5" en ajoutant une méthode `boirePotion`. Testez ensuite cette modification à l'aide de la console.

À faire vous-même 8

Saisissez, analysez et testez ce code

```
class Personnage:
  def __init__(self, point_de_vie):
    self.vie = point_de_vie

  def get_etat (self):
```

```
    return self.vie

  def perd_vie (self, nbPoint):
    self.vie = self.vie - nbPoint

bilbo = Personnage(15)
bilbo.perd_vie(2)
point = bilbo.get_etat()
```

Évaluez la variable point à l'aide de la console.

À faire vous-même 9

Saisissez, analysez et testez ce code

```
import random


class Personnage:

  def __init__(self, point_de_vie):
    self.vie = point_de_vie

  def get_etat (self):
    return self.vie

  def perd_vie (self):
    if random.random() > 0.5:
      nbPoint = 1
    else:
      nbPoint = 2
    self.vie = self.vie - nbPoint

bilbo = Personnage(15)
bilbo.perd_vie()
bilbo.perd_vie()
bilbo.perd_vie()
point = bilbo.get_etat()
```

Évaluez la variable point à l'aide de la console.

N.B : `random.random()` renvoie une valeur aléatoire comprise entre 0 et 1

Expliquez le fonctionnement de la méthode perd_vie

Comme vous l'avez remarqué, il est possible d'utiliser une instruction conditionnelle (`if / else`) dans une méthode. Il est donc possible d'utiliser dans le même programme le *paradigme objet* et le *paradigme impératif*.

Nous allons maintenant organiser un combat virtuel entre nos personnages :

À faire vous-même 10

Saisissez, analysez ce code

```
import random
class Personnage:

  def __init__(self, point_de_vie):
    self.vie = point_de_vie

  def get_etat(self):
    return self.vie

  def perd_vie(self):
    if random.random() < 0.5:
      nbPoint = 1
    else:
      nbPoint = 2

    self.vie = self.vie - nbPoint

def game():
  bilbo = Personnage(20)
  gollum = Personnage(20)

  while bilbo.get_etat() > 0 and gollum.get_etat() > 0:
    bilbo.perd_vie()
    gollum.perd_vie()

  if bilbo.get_etat() <= 0 and gollum.get_etat() > 0:
    msg = f'''Gollum est vainqueur, il lui reste encore {gollum.get_etat()} points
    alors que Bilbo est mort'''

  elif gollum.get_etat() <= 0 and bilbo.get_etat() > 0:
    msg = f'''Bilbo est vainqueur, il lui reste encore {bilbo.get_etat()} points
     alors que Gollum est mort'''

  else:
    msg = "Les deux combattants sont morts en même temps"

  return msg
```

Pour tester le programme, exécutez la fonction game dans une console. Vérifiez que l'on peut obtenir des résultats différents en exécutant plusieurs fois la fonction `game`.

Nous avons encore ici la démonstration qu'il est possible d'utiliser le paradigme objet et le paradigme impératif dans un même programme.

À faire vous-même 11

Améliorez le programme développé au "À faire vous-même 10" en modifiant les méthodes et en implémentant les méthodes suivantes.

1. Dans l'initialisation, on ne peut donner de nom au personnage !
   - créez un attribut `nom` qu'on doit donner en premier paramètre : on crée une instance de `Personnage` comme ceci :

     ```
     gollum = Personnage("Gollum", 20)
     ```

   - créez une méthode `get_nom` qui renvoie le nom du personnage.
2. Modifiez la fonction `Game` pour qu'elle tienne compte du nom du personnage. On doit pouvoir créer d'autres personnages et les messages doivent tenir compte des noms de ceux-ci.

   ```
   >>> frodon = Personnage("Frodon", 20)
   >>> araignee = Personnage("Araignée", 10)
   >>> game()
   Frodon est vainqueur, il lui reste encore 3 points
   alors que Araignée est mort
   ```

   Remarquez bien que la *signature* de la fonction `game` est différente !

   On doit créer les personnages AVANT de l'appeler. Il faut changer plusieurs éléments.
3. Améliorez encore la fonction `game` pour qu'elle affiche un journal détaillé du combat :

   ```
   >>> aragorn = Personnage(10)
   >>> orc = Personnage(10)
   >>> game(aragorn, orc)
   Aragorn perd un point de vie
   Orc perd deux points de vie
   ...
   Aragorn perd deux points de vie
   Orc perd deux points de vie
   Orc est vainqueur, il lui reste encore 3 points alors que Aragorn est mort
   ```

4. Revenons à `Personnage`
   On dispose maintenant des méthodes suivantes :

```
class Personnage:
|
|  get_etat
|      --> int
|      renvoie le nombre de pts de vie
|
|  perd_vie
|      enleve un ou deux points de vie
|
|  get_nom
|      --> str
|      renvoie le nom du personnage
```

On veut créer un attribut `chance` lors de l'instanciation du personnage. C'est un entier entre 0 et 4.

L'effet de la chance est le suivant :

- dans la méthode `perd_vie`, on tire toujours un nombre aléatoire entre 0 et 1.
  - Si ce nombre multiplié par 10 dépasse la chance du personnage, il perd un point de vie.
  - Sinon il ne perd pas de vie et on affiche "Aragorn a de la chance !"

  Par exemple, Aragorn a 2 de chance.
  - dans `perd_vie`, on tire 0.3,
    `10 * 0.3 = 3` et `3 > 2` : il perd un point de vie.
  - dans `perd_vie`, on tire 0.12345
    `10 * 0.12345 = 1.2345` et `1.2345 < 2` : il ne perd pas de vie.
- Il faut aussi changer la méthode `__init__` pour pouvoir créer nos personnages ainsi :

  ```
  aragorn = Personnage("Aragorn", 30, 2)
  ```

Implémentez la chance et faites quelques essais.

Attention, si vous donnez une chance trop élevée, le personnage ne perdra jamais de vie et la boucle de la fonction `game` sera infinie !

5. Maintenant qu'on peut donner un attribut chance, il faut *protéger* le programme.

   Une valeur de chance trop élevée peut conduire à un programme qui ne termine jamais, il suffit de donner une chance de 10 pour qu'un personnage soit invicible !

Nous allons créer une méthode interne `__limiter_chance` qui empêche la chance d'être supérieure à 4.

Si le paramètre `chance` est inférieure ou égale à 4, il est inchangé, S'il dépasse 4, il est ramené à 4.

Cette méthode interne ne sera pas appelée par les éléments extérieurs au programme, seulement par le programme lui même !

On utilise cette méthode interne dans `__init__`, il faut penser à l'appeler.

Et c'est la fin de cette très longue partie !

A faire 12

Vous pouvez continuer ce jeu en mode texte avec vos propres méthodes.

L'étape suivante est, selon moi, de créer une nouvelle classe `Combat`

Celle-ci prendrai en paramètres deux personnages et détaillerait le combat.

On pourrait aussi créer des méthodes comme `taper` dans personnage, qui dépendrait de la chance et d'un attribut `force` à définir...

`taper` pourrait renvoyer un nombre aléatoire entre 1 et `force`, par exemple.

Et c'est ce nombre qui définirait le nombre de points perdus par le `Personnage`...

Ce ne sont que des idées, je vous laisse libre de choisir une amélioration.