PROGRAMMATION ORIENTEE OBJET

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

Un objet dans la vie de tous les jours, 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**.

Les idées sous-tendant le **paradigme de programmation 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, Python, …

Pour nous initier à la programmation orientée objet nous allons bien entendu utiliser un langage que vous connaissez bien : Python.

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.

La mise au point de l'objet (par des ingénieurs) a été très complexe, en revanche son utilisation est relativement simple.

1. Classes et instances

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 de boutons, de manettes ou encore d'écrans de contrôle, mais des **attributs** et des **méthodes**.

Un des nombreux avantages de la programmation orientée objet (POO), est qu'il existe des milliers d'objets (on parle plutôt de **classes**, 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**.

Une classe définit et nomme une structure de données qui vient s'ajouter aux structures de bases du langage de programmation

Python permet d'utiliser le **paradigme impératif** (qui utilise des instructions qui apportent un changement d'état du programme), mais il permet aussi d'utiliser le **paradigme objet**. Il est même possible d'utiliser les 2 paradigmes dans un même programme.

La création d'une classe en python commence toujours par le mot clé « **class** », suivi du nom choisi pour la classe (le nom doit commencer obligatoirement par une **Majuscule**), et terminé par le symbole « **:** ». Ensuite toutes les instructions de la classe seront indentées (placées en retrait).

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

La **classe** est une espèce de moule dans lequel nous allons créer des **objets** (plus exactement nous parlerons d'**instances**).

Par exemple, nous pouvons créer une **classe Voiture**, puis créer différentes **instances** de cette classe (Peugeot407, RenaultEspace, ...). Pour créer une de ces **instances**, la procédure est relativement simple.

On écrit le nom choisi pour l'instance suivi du symbole « = » puis du nom de la classe suivi de la double parenthèses « ( ) ».

```
class Voitures:
    #instructions de la classe
Peugeot407=Voitures()
```

Cette ligne veut tout simplement dire : "crée un objet (une instance) de la classe Voiture que nommé « peugeot407 »."

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

```
class Voitures:
    #instructions de la classe
Peugeot407=Voitures()
RenaultEspace=Voitures()
```

Exemple-1 : Commencez par écrire une classe « Personnage » et, à partir de cette classe créez 2 instances : « Bilbo » et « Gollum ».

```
class Personnage:
    #instructions de la classe
Bilbo=Personnage()
Gollum=Personnage()
```

Pour l'instant, notre **classe** ne sert à rien et nos **instances** (objets) de classe ne peuvent rien faire.

Comme expliqué précédemment, une **instance** (objet) de **classe** possède des **attributs** et des **méthodes**.

2. Attributs et Méthodes

Un **attribut** possède une **valeur** (un peu comme une variable). 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, pour les définir, on écrit le nom de l'instance puis le symbole « **.** » suivi du nom de l'attribut.

```
Bilbo.vie
```

De la même façon l'**attribut** « vie » de l'**instance** « Gollum » sera noté

```
Gollum.vie
```

Exemple-2 : Saisissez et analysez ce code.

```
class Personnage:
    pass
Bilbo=Personnage()
Bilbo.vie=20
Gollum=Personnage()
Gollum.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 une bonne pratique, en effet, nous ne respectons pas un principe de base de la **POO** (Programmation Orientée Objet : **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" !

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

Une méthode est une fonction définie dans une classe d'initialisation des attributs.

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. Le mot 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.

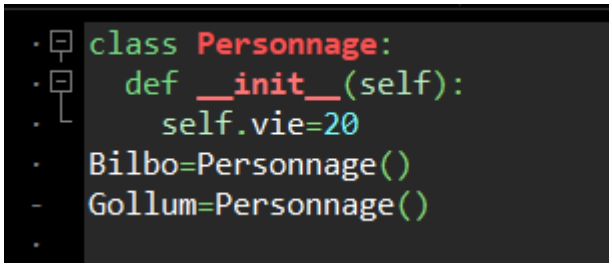
Dans le code source, nous allons avoir :

```
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.

Exemple-3 : Saisissez et analysez ce code.



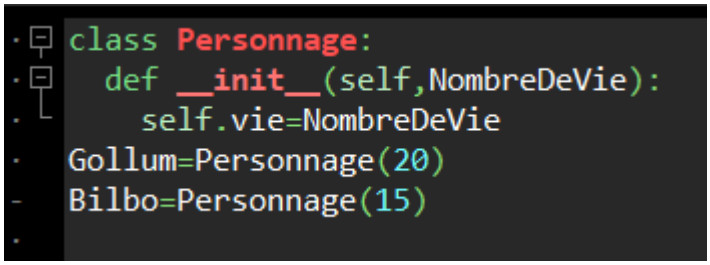
```
class Personnage:
  def __init__(self):
    self.vie=20
Bilbo=Personnage()
Gollum=Personnage()
```

Le résultat est identique au résultat de l'exemple-2, mais cette fois nous n'avons pas défini les attributs « Gollum.vie=20 » et « Bilbo.vie=20 » en dehors de la classe, nous avons utilisé la **méthode** « __init__ » ».

Imaginons que nos 2 personnages n'aient pas au départ les mêmes points de vie ! Pour l'instant, impossible d'introduire cette contrainte puisque « self.vie=20 » pour toutes les instances.

Une **méthode**, comme une fonction, peut prendre des **paramètres**. Le passage de paramètres se fait au moment de la création de l'instance.

Exemple-4 : Saisissez et analysez ce code.



```
class Personnage:
  def __init__(self,NombreDeVie):
    self.vie=NombreDeVie
Gollum=Personnage(20)
Bilbo=Personnage(15)
```

Vérifiez 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 « NombreDeVie »

(NombreDeVie n'est pas tout à fait le premier argument de la méthode __init__ puisque devant il y a « self », mais comme, « **self** » est obligatoire, nous pouvons dire que « NombreDeVie » est bien le premier argument non obligatoire).

On parle bien de variable pour « NombreDeVie » (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).

3. Création de 2 nouvelles méthodes :

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

Exemple-5 : Saisissez et analysez ce code.

```
class Personnage:
  def __init__(self, NombreDeVie):
    self.vie=NombreDeVie
  def donneEtat (self):
    return self.vie
  def perdVie (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.donneEtat()
- Bilbo.donneEtat()
- Gollum.perdVie()
- Gollum.donneEtat()
- Bilbo.perdVie()
- Bilbo.donneEtat()

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").

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.perdVie() permet de manipuler l'attribut self.vie)

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.

Exemple-6 : Saisissez et analysez ce code.

```
class Personnage:
  def __init__(self, NombreDeVie):
    self.vie=NombreDeVie
  def donneEtat (self):
    return self.vie
  def perdVie (self):
    self.vie=self.vie-1

Bilbo = Personnage(15)
Bilbo.perdVie()
Point=Bilbo.donneEtat()
```

Évaluez la variable « Point » à l'aide de la console.

Exemple-7 : Une potion pour gagner une vie

Nos personnages peuvent boire une potion qui leur ajoute un point de vie. Modifiez le programme de l'exemple-5 en ajoutant une méthode « BoirePotion ». Testez ensuite cette modification à l'aide de la console.

Exemple-8 : Type d'attaque et « NombreDeVie » perdues

Selon le type d'attaque subit, le personnage peut perdre plus ou moins de points de vie. Pour tenir compte de cet élément, il est possible d'ajouter un paramètre à la méthode perdVie.

Saisissez et analysez ce code

```
class Personnage:
  def __init__(self, NombreDeVie):
    self.vie=NombreDeVie
  def donneEtat (self):
    return self.vie
  def perdVie (self,NombrePoint):
    self.vie=self.vie-NombrePoint
Bilbo = Personnage(15)
Bilbo.perdVie(2)
Point=Bilbo.donneEtat()
```

Évaluez la variable « Point » à l'aide de la console.

Exemple-9 : Méthode « Aléatoire »

Il est possible d'ajouter une part d'aléatoire dans la méthode perdVie :

Saisissez, analysez et testez ce code

Saisissez et analysez ce code

```
import random
class Personnage:
  def __init__(self, NombreDeVie):
    self.vie=NombreDeVie
  def donneEtat (self):
    return self.vie
  def perdVie (self):
    if random.random()>0.5:
      NombrePoint = 1
    else :
      NombrePoint = 2
    self.vie=self.vie-NombrePoint
Bilbo = Personnage(15)
Bilbo.perdVie()
Bilbo.perdVie()
Bilbo.perdVie()
Point=Bilbo.donneEtat()
```

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

Remarque : « random.random() » renvoie une valeur aléatoire comprise entre 0 et 1

Expliquez le fonctionnement de la méthode perdVie

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**.

Exemple-10 : Combat Virtuel

Nous allons maintenant organiser un combat virtuel entre nos 2 personnages :

Saisissez et analysez ce code

```python
import random
class Personnage:
  def __init__(self, NombreDeVie):
    self.vie=NombreDeVie
  def donneEtat (self):
    return self.vie
  def perdVie (self):
    if random.random()>0.5:
      NombrePoint = 1
    else :
      NombrePoint = 2
    self.vie=self.vie-NombrePoint
def game():
  Bilbo = Personnage(20)
  Gollum = Personnage(20)
  while Bilbo.donneEtat()>0 and Gollum.donneEtat()>0 :
    Bilbo.perdVie()
    Gollum.perdVie()
  if Bilbo.donneEtat()<=0 and Gollum.donneEtat()>0:
    message = f"Gollum est vainqueur, il lui reste encore {Gollum.donneEtat()} points alors que Bilbo est mort"
  elif Gollum.donneEtat()<=0 and Bilbo.donneEtat()>0:
    message = f"Bilbo est vainqueur, il lui reste encore {Bilbo.donneEtat()} points alors que Gollum est mort"
  else :
    message = "Les deux combattants sont morts en même temps"
  return message
```

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.