POO = Programmation orienté objet

La programmation orienté objet est une méthode de programmation qui permet de mieux structurer son code. Elle est utilisée dans quasiment tous les projets.

Le principe est de définir nous-même des nouveaux types d’objet (des nouvelles classes), afin de les réutiliser dans d’autres programmes (ou d’autres classes), comme nous l’avons fait avec la classe Turtle de l’API turtle.

# Classe et objet (instance)

* 1. ***Les classes***

En POO, on va manipuler des objets.

Les objets sont en fait des instances d’une classe.

*Exemple:*

On peut par exemple en poo définir une Classe Personnage. En fait, cela correspond à un nouveau type de donnée.

Comme les entiers les flottant, les listes... maintenant on pourra utiliser notre type Personnage.

Dans un jeu la classe personnage sera instancié plusieurs fois.

Par exemple, notre héros sera un personnage, les ennemis seront des personnages, les autres héros seront des personnages, etc…

En fait, tous les éléments que nous avons vu, les personnages que l'on va créer les entiers que l'on a utilisés, les listes, ... sont des objets.

Pour mieux structurer notre code, il est possible d’utiliser un nouveau fichier pour chaque nouvelle classe que l’on va définir, mais il arrive aussi de définir plusieurs classe dans le même fichier.

*Exemple:*

Déclaration (ou définition) d’une classe:

class Personnage :

def \_\_init\_\_(self) :

        self.\_\_hp = 100

Notre classe *Personnage* est en fait un nouveau type, et la méthode \_\_init\_\_ est la méthode qui nous permettra de créer par la suite plusieurs personnages.

La méthode \_\_init\_\_() est appelée de manière caché lors de la création d'un objet (c’est un constructeur).

self.\_\_hp = 100

« self.\_\_hp » est une manière de stocker une information dans la classe. On parle d'attribut ou de propriété d’une classe. Dans notre cas, on stocke le nombre de point de vie dans l'attribut \_\_hp.

* 1. ***Les objets ou instances de classe***

Une fois que l'on a défini nos classes (et la méthode \_\_init\_\_ pour construire des objets). On peut définir des instances de cette classe. C'est-à-dire des objets dont le type est la classe que nous venons de définir.

Exemple:

[1,2] est un objet/une instance du type liste

perso est un objet (une instance) du type personnage

Instanciation (Déclaration d’une instance ou d’un objet) en utilisant le constructeur de la classe:

perso1 = Personnage(); #appelle la méthode \_\_init\_\_

perso2 = Personnage(); #appelle la méthode \_\_init\_\_

perso3 = Personnage(); #appelle la méthode \_\_init\_\_

Un objet est une instance d'une classe. On peut créer autant d'objets que l'on désire avec une **classe**.

* 1. ***Constructeurs***

Les constructeurs sont les méthodes qui vont nous permettre de créer des objets de la classe que l’on est en train de définir.

Lorsque je définis ma classe Personnage, le but est en parti de pouvoir créer des nouveaux objets/instances de type Personnage, je dois donc définir des méthodes permettant de réaliser cette création.

En python, ces méthodes s’appellent toujours \_\_init\_\_ et peuvent prendre un ou plusieurs arguments. Les deux méthodes ci-dessous sont donc deux constructeurs de la classe Personnage.

class Personnage:

    def \_\_init\_\_(self):

        self.\_\_name = ""

        self.\_\_force = 0

        self.arme = Arme(5)

        self.\_\_hpMax = 100

        self.\_\_hp = 100

        self.\_\_lvl = 1

    def \_\_init\_\_(self, force, hpMax, hp, lvl):

        self.\_\_force = force

        self.arme = Arme(5)

        self.\_\_hpMax = hpMax

        self.\_\_hp = hp

        self.\_\_lvl = lvl

Pour créer des objets, on peut donc utiliser ces deux constructeurs comme suit :

perso1 = Personnage(); #appelle la méthode \_\_init\_\_

perso2 = Personnage(10,200,200,5); #appelle la méthode \_\_init\_\_(self, force, hpMax, hp, lvl)

* self

En programmation orienté objet, self correspond à l’objet courant. Par exemple dans un constructeur, self désigne l’objet que l’on est en train de créer.

Dans d’autres méthodes, par exemple, perso.attaquer(), self désignera l’objet sur lequel la méthode attaquer est appelé (ici, notre perso).

# Propriétés ou attributs

Lors de la définition de notre classe personnage. On va définir les attributs qu'un personnage devra avoir. Les attributs sont les propriétés que devront avoir les éléments de la classe que l’on est en train de définir. Les attributs d’une classe permettent de stocker des informations au niveau de la classe. Elles sont similaires aux variables.

Par exemple:

Force, magie, résistance, Vie, RM, Mana, …

Vous pouvez à tout moment créer un attribut pour votre objet:

mon\_perso.force = 25

Et le lire ainsi:

mon\_perso.force #vaut 25

1. **Statique versus Dynamique**
2. **Statique (static)**

Une propriété statique est une propriété qui s’applique sur toute la classe, par exemple, comme le niveau maximum des personnages sera le même pour tous les personnages. On ne va pas déclarer une nouvelle propriété niveauMax pour chacun de ces personnages mais plutôt utiliser une propriété qui soit partagé par tous les personnages. C’est ce qu’on appelle un attribut/une propriété statique de notre classe.

Cela se défini comme suit :

class Personnage :

niveauMax = 99

1. **Dynamique.**

Par contre, les noms de nos personnages seront différents d’une instance à l’autre de personnage. On va donc pour cela utiliser des propriétés dynamiques (c’est à dire propre à nos instances).

Cela se défini comme suit :

class Personnage :

   def \_\_init\_\_(self) :

      self.nom = "Heros"

1. Visibilité
2. Publique (public)

Une variable publique est accessible sur une instance de la classe même lorsque l’on travaille à l’extérieur de cette classe.

1. Privé (private)

Un attribut est privé quant à il n’est accessible qu’à l’intérieur de cette classe (donc lorsque l’on définit la classe), mais ne pourra pas être réutilisé directement par les autres classes. Par contre il est possible que des méthodes publiques viennent modifié ou utilisé la valeur de cette propriété privé.

En python, les attributs privés sont différenciés des attributs public par l’utilisation des doubles underscore en début de nom self.\_\_nomAttribut 🡪 signifie que l’attribut est privé

    def \_\_init\_\_(self):

        self.\_\_name = ""

        self.\_\_force = 0

        self.arme = Arme(5)

        self.\_\_hpMax = 100

        self.\_\_hp = 100

        self.\_\_lvl = 1

Ici, arme est le seul attribut public.

# Méthodes

Un personnage doit aussi pouvoir exécuter plusieurs actions. Ces actions vont être définies dans des méthodes (fonctions) spécifiques à nos personnages.

Les méthodes sont en fait des fonctions que l’on définies et qui sont propres à cette classe. Créons une nouvelle méthode dans notre classe Personnage.

Exemple:

attaquer

esquiver

estVivant

...

class Personnage :

    def \_\_init\_\_(self) :

      self.nom = "heros"

    def estVivant() :

        return self.hp>0

## 3.1 Constructeur

Les constructeurs sont les méthodes qui servent à créer des instances du type de la classe qu’on est en train de créer.

En python, les constructeurs sont toujours nommés \_\_init\_\_.

Attention : Il ne peut y avoir qu’un seul constructeur par classe en Python. Si on veut avoir des constructeurs avec des paramètres différents, il faut passer des valeurs par défaut aux arguments de la méthode \_\_init\_\_.

Ici Personnage est une méthode de ma classe Personnage qui sert à créer des nouveaux personnages.

class Personnage :

    def \_\_init\_\_(self) :

      self.nom = "heros"

#Peut être remplacer par :

    def \_\_init\_\_(self, force=10, hpMax=100, hp=100, lvl=1):

        self.\_\_force = force

        self.arme = Arme(5)

        self.\_\_hpMax = hpMax

        self.\_\_hp = hp

        self.\_\_lvl = lvl

Ce constructeur peut ensuite être appelé en utilisant tous les arguments possible, ou en utilisant les arguments par défaut.

1. **Statique**

Une méthode statique sera donc appelé directement sur la classe, on peut donc les utiliser comme nous le faision pour les fonctions python avant de découvrir la Programmation orienté objet.

    def getNiveauMax() :

      return lvlMax

Pour appeler cette méthode, on peut faire directement :

getNiveauMax()

#ou

Personnage.getNiveauMax() #selon l’import de que l’on a fait.

1. **Dynamique**

Une méthode dynamique, sera appelée sur un objet en particulier de cette classe.

Pour savoir quel objet appelle cette méthode, lors de la définition, on va utiliser le paramètre self.

    def attaquer (self, enemy):

        enemy.\_\_hp -= self.\_\_force \* self.arme.\_\_degats

Lorsqu’on appelle cette méthode, on va l’appeler directement sur l’objet que l’on veut :

    self.estVivant()

enemy.estVivant()

      heros.attaquer(enemy)

Ici, arme est le seul attribut public.

### Encapsulation

L’encapsulation est un principe de la POO, qui permet de protéger les propriétés d’une classe.

Pour cela, on met l’attribut de la classe en private.

### 4.1 Accesseurs et mutateurs (getters et setters)

Quel que soit le langage, pour la **programmation orientée objet** il est de préférable de passer par des propriétés pour changer les valeurs des attributs. Alors bien que cela ne soit pas obligatoire, il existe une convention de passer par des **getter** (ou **accesseur** en francais) et des **setter** (**mutateurs**) pour changer la valeur d'un attribut. Cela permet de garder une cohérence pour le programmeur, si je change un attribut souvent cela peut également impacter d'autres attributs et les mutateurs permettent de faire cette modification une fois pour toute.

class Personnage :

    def \_\_init\_\_(self) :

        self.\_\_force = 0

    def getForce(self):

        return self.\_\_force

    def setForce(self, v):

        self.\_\_force  =  v

# Héritage

Maintenant, si on a besoin d’étendre la classe personnage en ajoutant de nouvelles méthodes ou de nouveaux attributs.

On peut créer une nouvelle classe (Guerrier, Mage) qui hérite des attributs de personnage avec des méthodes ou des attributs en plus.

On peut par exemple ajouter de nouveaux attributs ou de nouvelles méthodes à notre nouvelle classe.

class Personnage:

    def \_\_init\_\_(self):

        self.\_\_force = "A déterminer"

class Mage(Personnage):

    def \_\_init\_\_(self):

  super().\_\_init\_\_()

        self.magie = 10

On remarque tout d'abord que l'attribut force a bien été hérité. Ensuite on remarque que la méthode \_\_init\_\_ a écrasé la méthode de la classe Personnage. On parle alors de surcharge de méthode.

Lorsque l’on rédéfini des méthodes, notamment les constructeurs, on ne

    def lvlUp(self):

        super().lvlUp()

        self.\_\_magie+=2

## 5.1 Redéfinition

Par défaut, les attributs et les méthodes que l’on avait définies dans notre classe Personnage seront toutes disponibles dans la classe Mage.

Certaines implémentations de méthodes de la classe personnage ne sont plus pertinentes pour un mage.

Par exemple, on peut penser qu’un Mage n’aura pas la même façon d’attaquer que les autres personnages, son attaque utilisera la magie plutôt que la force par exemple.

Dans ce cas, il faudra redéfinir cette méthode attaquer et coder une implémentation spécifique à notre classe Mage (comme nous l’avions fait pour \_\_init\_\_).

C’est ce qu’on appelle la redéfinition (overriding).

class Personnage:

    force = 10

    def \_\_init\_\_(self):

        self.nom = "A déterminer"

class Mage(Personnage):

    def \_\_init\_\_(self):

        self.nom = "Gandalf"

        self.magie = 10

On dit aussi que la méthode *attaquer* de personnage est masqué par la méthode attaquer du Mage.

class Personnage:

    force = 10

    def attaquer(self):

        self.nom = "A déterminer"

class Mage(Personnage):

    def attaquer(self):

        self.nom = "Gandalf"

        self.magie = 10

## 5.2 Surcharge

Il est possible qu’une fonction ne doive pas avoir le même comportement selon le type d’élément que l’on met en paramètre.

Par exemple, pour la fonction \_\_init\_\_ définit tout à l’heure :

On pourrait aussi avoir une méthode *attaquer* qui n’ai pas le même comportement

class Personnage:

    force = 10

    def attaquer(self):

        self.nom = "A déterminer"

class Mage(Personnage):

    def attaquer(self, arme):

        self.nom = "Gandalf"

        self.magie = 10

Admettons qu’on veuille quelque chose du genre:

Dans ce cas, selon le paramètre (la cible) que l’on va attaquer, on utilisera la deuxième si c’est un Mage et la première si c’est une autre sorte de personnage.

## 5.3 Méthode et classe abstraite

Le concept de méthode abstraite (abstract method) ou une classe abstraite (abstract class) ne sont pas clairement définis en **Python (contrairement à d’**autres langages). Mais il est tout de même possible d’en implémenter.

Une classe est appelée abstraite (abstract) définit des méthodes abstraites et la sous classe doit outrepasser (override) ces méthodes si vous voulez les utilier. Les méthodes abstraites jètent toujours l'exception **NotImplementedError**.

  def getClassName(self):

      raise NotImplementedError("Subclass must implement abstract method")