Introduction à la POO

I./ La notion d'objet :

Une *classe* est une structure de données abstraite regroupant :

- Les caractéristiques de ces données, appelées les *attributs*, ou *variables d'instance*.
- Les actions que l'on peut appliquer sur les données, appelées *méthodes* de la classe.

Un objet est un élément issu d'une classe. On parle aussi d'*instance* de la classe. La création d'un objet d'une classe s'appelle une *instanciation* de cette classe.

Ces définitions étant assez abstraites, voici deux exemples pour éclairer ces concepts :

- Un avion particulier peut être vu comme un objet de la classe « Avion ». Cet objet va avoir plusieurs attributs : la masse, la longueur, l'envergure, la vitesse, l'altitude, le niveau de carburant, le cap, … Sur cet objet, on peut appliquer plusieurs méthodes, correspondant à des actions particulières : atterrir, décoller, faire le plein, embarquer les passagers, … Un Cessna 182 ou un Airbus A380 sont deux instances différentes de la classe « Avion ».
- Un chat particulier peut être vu comme un objet de la classe « Felis silvestris catus ». Cet objet va avoir plusieurs attributs : âge, taille, masse, couleur, propriétaire, … Sur cet objet, on peut appliquer plusieurs méthodes : nourrir, jouer avec, soigner, tatouer, vacciner, … Félix et Simba sont deux instances différentes de la classe « Felis silvestris catus ».
- Un ordinateur particulier peut être vu comme un objet de la classe « Ordinateur ». Cet objet va avoir plusieurs attributs : marque, modèle, type de processeur, vitesse d'horloge, quantité de RAM, taille du disque dur, … Sur cet objet, on peut appliquer plusieurs méthodes : allumer, éteindre, formater, installer un logiciel, mettre à jour, … Les PC de la salle de classe ou votre ordinateur personnel sont des instances différentes de la classe « Ordinateur ».

Dans ces trois exemples, on a donné le nom de la classe avec une lettre majuscule au début : c'est une convention en langage Python (ainsi que dans d'autres langages).

II./ Exemple de classe :

Dans le cadre d'un JdR (Jeu de Rôle), on veut définir un personnage pour chacun des joueurs. Chaque personnage aura plusieurs caractéristiques (attributs), comme le pseudo, les points de vie (PV), les points d'expérience (XP), etc…

Taper le script suivant :

```
class Personnage:
    def __init__(self, pseudo, PV, XP):
        self.pseudo = pseudo
        self.PV = PV
        self.XP = XP
        self.inventaire = []
        self.alive = True
        return None
```

La classe « **`Personnage`** » ne contient pour l'instant qu'une seule fonction, nommée **`__init__`** (attention, le mot « init » est précédé et suivi par deux caractères soulignés !). Cette fonction **`__init__`** est un *constructeur* : elle permet de créer un objet (une instance) de la classe « **`Personnage`** ». Cet objet aura plusieurs paramètres : un pseudo (sous la forme d'une chaîne de caractères), un nombre de PV (nombre entier), un nombre d'XP (nombre entier), un inventaire (sous la forme d'une liste), et une variable d'état indiquant si le personnage est vivant ou non (sous la forme d'un booléen). Le créateur est une fonction qui sera appelée dès qu'on voudra créer un objet. On voit qu'elle attend quatre paramètres : **`self`**, **`pseudo`**, **`PV`** et **`XP`**. « **`self`** » est un paramètre particulier qui fait référence à l'objet lui-même. Toutes les méthodes d'une classe auront au moins un paramètre « **`self`** ».

Une fois le script lancé, nous allons créer un objet particulier, en tapant :

```
albert = Personnage("Anakin", 100, 50)
```

Si on fait : **`print(albert)`**, on obtiendra un résultat du type :

```
<__main__.Personnage object at 0x03068B30>
```

La variable « **`albert`** » contient donc un objet de type « **`Personnage`** », cette dernière classe appartenant au programme principal (« **`__main__`** »). La dernière indication est l'adresse à laquelle est stocké l'objet, et peut donc être différente en fonction de l'utilisateur. Cet objet n'étant pas un objet d'un type natif de Python (comme par exemple les listes, les entiers, les chaînes de caractères, etc…), la fonction « **`print`** » ne peut pas l'afficher correctement. Nous verrons comment y remédier plus tard.

Comment consulter le contenu des attributs de l'objet « **`albert`** » ? Tapez : **`print(albert.pseudo)`**. Que remarque-t-on ? Affichez les autres attributs de l'objet « **`albert`** ».

On peut aussi modifier les attributs d'un objet : essayez les lignes suivantes dans la console Python :

```
>>> print(albert.PV)
>>> albert.PV = 500
>>> print(albert.PV)
```

A ce stade, notre objet est uniquement constitué de ses attributs. Son utilité est donc limitée. Nous allons donc rajouter des méthodes pour agir sur les attributs. Tapez le script suivant :

```
class Personnage:
    def __init__(self, pseudo, PV, XP):
        self.pseudo = pseudo
        self.PV = PV
        self.XP = XP
        self.inventaire = []
        self.alive = True
        return None

    def prendre_objet(self, objet):
        self.inventaire.append(objet)
        return None

    def poser_objet(self, objet):
        if objet in self.inventaire:
            self.inventaire.remove(objet)
            print("Vous venez de poser : " + objet)
        else:
            print("Vous ne possédez pas : " + objet)
        return None

    def donner_inventaire(self):
        if self.inventaire == []:
            print("Vous ne portez rien !")
        else:
            print("Vous portez :")
            for objet in self.inventaire:
                print("* " + objet)
            return None
```

Une fois ce script lancé, tapez les lignes suivantes, **dans cet ordre**, dans la console Python :

```
>>> albert = Personnage("Anakin", 100, 50)
>>> albert.donner_inventaire()
>>> albert.prendre_objet("Sabre laser")
>>> albert.prendre_objet("Holocron")
>>> albert.prendre_objet("Potion")
>>> albert.donner_inventaire()
>>> albert.poser_objet("Livre")
>>> albert.poser_objet("Holocron")
>>> albert.donner_inventaire()
```

L'objet « **albert** » peut donc voir ses attributs modifiés par ses méthodes.

Créez un deuxième objet, (en faisant par exemple : **bob = Personnage("Yoda", 50, 250)**) et vérifiez que cet objet est totalement indépendant de l'objet « **albert** ».

Les trois méthodes que nous avons implémentées servent à gérer l'inventaire de notre personnage, c'est-à-dire la liste des objets qu'il porte.

Programmation Python : rajoutez des méthodes pour gérer la santé du personnage. Lors d'un combat, par exemple, notre personnage peut être blessé : on pourra créer une fonction **blessure(self, degats)** qui diminue le nombre de PV du personnage d'un nombre égal au paramètre « **degats** », nombre entier. Si le nombre de PV atteint 0, l'attribut « **alive** » doit devenir égal à « **False** ». Inversement, on peut imaginer que certains items permettent de regagner des PV, comme des potions par exemple : créez une fonction **soin(self, points)** qui augmente le nombre de PV du personnage d'un nombre égal au paramètre « **points** ». Lorsque le personnage est mort (**alive == False**), il ne peut logiquement rien faire. Modifiez les autres fonctions de façon à ce que le message « Désolé, votre personnage est mort ! » s'affiche lorsque c'est effectivement le cas. Comme il s'agit d'un jeu dans un univers fictif, la notion de mort peut être parfois relative : ajoutez la fonction **respawn(self)** qui ressuscite le personnage, en mettant l'attribut **alive** à **True**, et en lui donnant quelques PV.

III./ L'encapsulation :

Nous avons vu que lorsque nous avons créé un objet, nous pouvons directement, de l'extérieur, avoir accès à tous ses attributs en lecture et en écriture. Ce n'est pas toujours souhaitable. L'usage est de ne laisser l'accès direct à l'utilisateur qu'à certaines méthodes et certaines données, qui sont dites *publiques*. Les autres données et méthodes seront cachées à l'utilisateur, et seront dites *privées*. Dans la plupart des langages orientés objet, comme Java, il existe des mots-clés permettant de définir de manière stricte le caractère public ou privé d'un attribut ou d'une méthode. En Python cette distinction est faite en utilisant une *convention de nommage*.

Dans la méthode **__init__** de la classe « **Personnage** », modifiez la ligne **self.__XP = XP** en **self.__XP = XP** (attention, il y a deux caractères « soulignés »). Après avoir relancé le script, créez un nouveau personnage :

```
caroline = Personnage("Leïa", 100, 25)
```

Vérifiez que l'on peut avoir accès à certains attributs : **print(caroline.pseudo)** ; **print(caroline.PV)**, … Essayez ensuite : **print(caroline.__XP)**.

On obtient le message suivant : « **AttributeError: 'Personnage' object has no attribute '__XP'** ». L'attribut « **__XP** » est maintenant privé : on ne peut pas le lire ou le modifier directement de l'extérieur.

Le fait de cacher certains attributs et certaines méthodes à l'utilisateur s'appelle l'*encapsulation*. Cette technique permet d'éviter que le programmeur modifie par erreur certains attributs d'un objet. Dans notre exemple, cela empêche la « triche » : comme par exemple augmenter de façon illégale le nombre de PV de son personnage.

Dans certains cas, il faut avoir accès au contenu de certains attributs privés, voire de les modifier. Comment faire puisque l'accès direct est maintenant interdit ? En créant des méthodes spéciales, appelées *accesseurs* (ou *getters* en anglais) et *mutateurs* (ou *setters* en anglais).

Rajoutez les deux méthodes suivantes dans la classe « **Personnage** », puis relancez le script :

```
    def get_XP(self):
        return self.__XP

    def set_XP(self, valeur):
        self.__XP = valeur
        return None
```

Tapez dans la console Python :

```
>>> daniel = Personnage("Obi-Wan", 100, 75)
>>> daniel.get_XP()
>>> daniel.set_XP(80)
>>> daniel.get_XP()
```

Vérifiez à nouveau que vous ne pouvez pas accéder directement à l'attribut « **__XP** ».

Modifiez le script Python de façon à ce que tous les attributs soient privés et créez les méthodes nécessaires pour les consulter : **get_pseudo**, **get_PV**, **set_pseudo**, **set_PV**, …

Point de vocabulaire : l'ensemble des attributs et des méthodes publiques (et donc accessibles directement à l'utilisateur) constituent *l'interface de la classe*.

IV./ Prolongements :

1./ Nous avons vu que la fonction **print** ne pouvait pas afficher correctement un objet de la classe « **Personnage** ». Il existe une méthode particulière, nommée « **__str__** » qui permet de combler cette lacune. Rajoutez la méthode suivante à la classe « **Personnage** » :

```
    def __str__(self):
        chaine = "Votre personnage s'appelle " + self.__pseudo + ". "
        if not(self.__alive):
            chaine += "Ce personnage est décédé."
        return chaine
```

Créez un personnage (par exemple « **robert** »), puis faites : **print(robert)**. A l'aide de la méthode blessure, infligez suffisamment de dégâts à votre personnage pour que ce dernier meure. Refaites : **print(robert)**.

Attention : la méthode « **__str__** » doit renvoyer obligatoirement une chaîne de caractères.

Il existe aussi une méthode « **__repr__** » dont le fonctionnement est un peu similaire à la méthode « **__str__** ». Elle permet de faire en sorte que le contenu de l'objet soit affiché quand on appelle l'objet directement dans la console. Son fonctionnement est identique à celui de la méthode « **__str__** ».

Si on utilise les deux méthodes dans la même classe, il y aura donc un comportement (et donc un affichage) différent selon que l'on appelle directement un objet dans la console ou si on utilise la fonction **print**.

Si on souhaite que le comportement soit le même dans les deux cas, on peut juste utiliser la méthode « **__repr__** ».

2./ Rajoutez une méthode « **fiche_perso(self)** » qui affiche les différentes caractéristiques du personnage.

Exemple :

```
>>> ernest.fiche_perso()
Votre personnage s'appelle : Han Solo
PV : 120
XP : 90
Statut : en vie
Inventaire : ['Pistolet laser', 'Fiole de whisky', 'Couteau suisse']
```

3./ Rajoutez une méthode « **renommage(self, nouveau_pseudo)** » permettant de changer le pseudo du personnage.

Exemple :

```
>>> joueur1.get_pseudo()
'Arnold'
>>> joueur1 = Personnage("Yoda", 500, 75)
>>> joueur1.get_pseudo()
'Yoda'
>>> joueur1.renommage("Arnold")
>>> joueur1.get_pseudo()
'Arnold'
```

4./ A partir de la classe « **Personnage** », créer un jeu de combat très simple : deux personnages sont créés au départ, avec le même nombre de PV. Un nombre au hasard est tiré (voir le module « **random** », dont la documentation peut être trouvée ici : https://docs.python.org/fr/3/library/random.html) et en fonction de sa valeur, on attribue la victoire à l'un des deux personnages. On tire ensuite au hasard le nombre de PV qui seront enlevés au perdant. On recommence jusqu'à ce qu'un des deux joueurs soit mort. Le programme doit indiquer à chaque tour le nombre de PV de chacun des deux personnages, et indique le vainqueur à la fin.

5./ Créer une classe « **Pokemon** » permettant de décrire les créatures éponymes. Les attributs d'un Pokémon sont son nom, son type (ou ses types), son niveau, ses statistiques d'attaque et de défenses, spéciales ou non, sa vitesse, son nombre de PV, ses attaques, son dresseur. On peut en rajouter d'autres, comme sa taille, son poids, etc… (plus d'infos sur : https://www.pokepedia.fr/Portail:Accueil). Rendez tous les attributs privés et créez les différents accesseurs et mutateurs nécessaires. Créez une méthode **pokedex(self)** permettant d'afficher toutes les informations sur le Pokémon. En vous inspirant de la classe « **Personnage** » et des questions précédentes, réalisez un jeu de combat entre deux Pokémons.

6./ Créer une classe « **Point** » décrivant des points sur un plan à deux dimensions. Chaque objet point aura deux attributs : sa coordonnée X et sa coordonnée Y. Rendez ces deux attributs privés. Créez les accesseurs **get_X(self)** et **get_Y(self)** pour pouvoir accéder aux coordonnées X et Y du point. Créez les mutateurs **set_X(self, newX)** et **set_Y(self, newY)** pour pouvoir modifier les coordonnées X et Y du point. Rajoutez une méthode **move(self, dX, dY)** qui change la coordonnée X d'une valeur dX, et la coordonnée Y d'une valeur dY. Créez finalement une méthode **distance(self, autre_point)** qui calculera la distance entre le point en question et un deuxième point donné en argument.

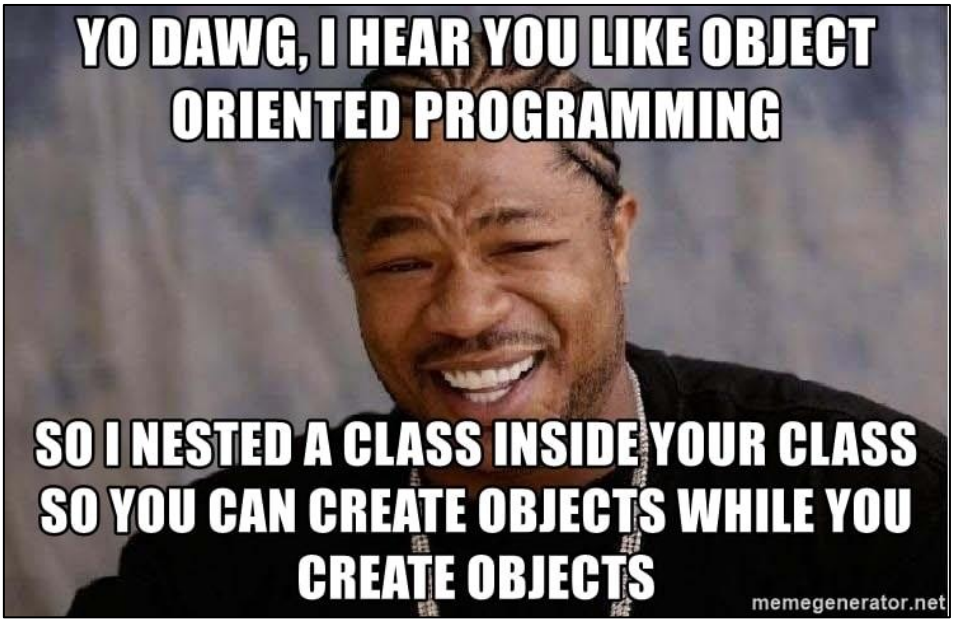
Quelques informations et conseils avant de finir :

* Il est conseillé d'enregistrer chacune de vos classes dans un fichier unique. Cela permet de réutiliser vos classes plus tard, dans un autre projet (cf. Chapitre sur la modularité). Par exemple on peut enregistrer la classe **`Point`**, définie précédemment dans un fichier nommé « **`Point.py`** ». On pourra alors l'importer dans un autre programme à l'aide de l'instruction : « **`import Point`** ». Cela permet de faciliter la réutilisation du code (sans avoir besoin de faire du copier-coller) ainsi que la maintenabilité du code (car on a alors des codes sources avec beaucoup moins de lignes de code).

* Un autre conseil est de largement documenter les différentes méthodes de classe à l'aide de docstrings (cf. Chapitre sur la modularité). Cela rendra plus facile l'utilisation de la classe par d'autres programmeurs (utilisation de la fonction « **`help`** »). En règle générale, il est très important de bien commenter son code (*documentation interne*).

* Nous avons vu plusieurs méthodes particulières dont le nom commence et finit par deux caractères « soulignés » : **`__init__`**, **`__str__`**, **`__repr__`**, … Ces méthodes sont appelées *méthodes spéciales* en français, ou *magic methods*, ou *dunder* (comme *double underscores*) en anglais. Il en existe bien d'autres : voir par exemple le lien suivant : https://gayerie.dev/docs/python/python3/dunder.html. On peut par exemple, à l'aide de la méthode « **`__add__`** » faire en sorte de pouvoir utiliser l'opérateur « **`+`** » de Python entre deux objets : c'est ce que l'on appelle la surcharge des opérateurs. Nous détaillerons cela plus en détail avec l'étude de la classe « **`Fraction`** ».

Ceci ne constitue qu'une introduction basique à la POO (OOP pour *Object-oriented programming* en anglais). Nous n'avons notamment pas abordé les notions de *polymorphisme*, d'*abstraction*, ou d'*héritage*. Pour aller plus loin : https://www.youtube.com/watch?v=yncatRjlEa0.



2025
Bruno PEYROT