Programmation Orientée Objet (POO)

Formation

La POO est très utilisée pour obtenir des codes clairs et efficaces, et c'est aussi la façon courante de coder en C++.

Dans une première partie on apprendra la POO sous Python afin d'utiliser un langage déjà bien connu. On verra ensuite la syntaxe adaptée au C++ afin de pouvoir transposer ce que l'on aura appris.

I - Petite introduction à la POO

Le principe de la POO est de définir un **objet**, auquel on peut ensuite :

- associer des valeurs que l'on appellera **attributs**.

C'est l'équivalent des caractéristiques de l'objet. Ils peuvent être définis lors de la création de l'objet, et l'on peut en rajouter ou les modifier lors de son utilisation.

- mais aussi des fonctions qui lui seront propre, appelées **méthodes**.

C'est l'équivalent des actions possibles de l'objet.

Pour créer ces objets on utilise des **classes**. Une classe et l'équivalent d'un moule pour un objet : on définit la classe en listant les attributs et les méthodes que l'on veut donner à nos objets. On pourra par la suite créer des objets à partir de ce moule, afin de les manipuler selon nos règles prédéfinies.

Notre exemple sera la création d'un personnage de jeu vidéo. Ce dernier aura divers attributs :

- Vie
- Mana
- Stamina
- ...etc...

Et divers méthodes :

- Esquiver
- Lancer un sort
- Boire une potion
- ...etc...

II - Premiers pas en POO

Un espace de travail propre

Le code que l'on va implémenter dans ce TP sera composé de plusieurs fichiers Python dans un même répertoire :

- Un fichier "main.py" dans lequel on utilisera les classes créées ailleurs.
- Des fichiers du format "nom_classe.py" pour chaque classe que l'on créera.

Définir une classe

Syntaxe : Le mot clé pour définir une classe est **class** (**def** est le mot clé pour définir une fonction)

```
# Dans le fichier "personnage.py"
class Personnage:
    pass    # "pass" dit à Python de ne rien faire
```

Nous pouvons maintenant créer des objet de la classe **Personnage** comme suit :

```
# Dans le fichier "main.py"
from personnage import *
mario = Personnage()    # mario est une instance de Personnage
```

On appelle l'objet mario une **instance** de Personnage.

Utiliser les attributs et méthodes

Pour accéder à un attribut, on utilise la syntaxe :

```
nom_objet.nom_attribut
```

Pour utiliser une méthode, on utilise la syntaxe :

```
nom_objet.nom_methode(arguments)
```

Remarque : Notre Mario n'a ni attribut, ni méthode pour l'instant...

Donner des attributs

On utilise la syntaxe **init(self)** qui est une fonction qui se lancera lors de la création de l'objet.

```
class Personnage:
    def __init__(self):
        self.vie = 100
        self.mana = 50
```

```
        self.stamina = 100
```

Exercice : Afficher les différents attributs de votre Mario.

```
# Dans le fichier "main.py"
print("Mario a :\n", mario.vie, "pv\n", mario.mana, "mana\n",
mario.stamina, "stamina")
```

Nous voulons que notre objet ait un attribut **nom** qu'on lui donne lors de l'initialisation. Pour cela, on utilise la syntaxe :

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

Lors de la création de notre instance mario, il faut donc maintenant préciser son nom :

```
mario = Personnage("Mario")
```

Créer des méthodes

Les méthodes prennent obligatoirement en argument **self**, qui est l'objet lui même. Lors de l'appel de la méthode, on ne précise pas ce qu'est **self** car c'est par défaut l'objet en question.

Pour créer une méthode, il faut définir une fonction dans la définition de votre classe. Créons la méthode **esquiver** qui :

- enregistre **True** dans un attribut **esquive** si le personnage a assez de stamina pour esquiver (disons 20).
- enregistre **False** sinon.

```
def esquiver(self):
    if self.stamina >= 20:
        self.stamina -= 20
        self.esquive = True
    else:
        self.esquive = False
```

Remarque : Pensez à rajouter l'attribut esquive dans init pour éviter d'avoir des problèmes plus tard...

Héritage, qu'est-ce que c'est ?

Nous avons défini Mario comme étant une instance de Personnage, mais au vu de ce que sait faire Mario (lancer des boules de feu, changer de forme, ...etc...), Mario est un mage ! On voudrait donc pouvoir imaginer que l'on ait plusieurs types de personnages :

- mage
- guerrier
- ...etc...

Tous ces types ont les caractéristiques de bases d'un **Personnage**, et en plus possèdent des spécificités.

La POO permet de créer des classes qui **héritent** des attributs et méthodes de classes définies auparavant. Dans notre cas, Mage est une **sous-classe** de Personnage.

Pour créer une sous-classe héritant d'une autre classe on écrit :

```
class Mage(Personnage):
   pass    # "pass" dit à python de ne rien faire
```

Vous pouvez définir votre mario comme une instance de **Mage()** maintenant. Les commandes réalisées précédemment sont alors toujours fonctionnelles ! Vous pouvez essayer.



Priorité sur l'héritage : Si vous définissez un attribut ou une méthode dans la sous-classe qui ait le même nom qu'un attribut ou une méthode de la classe dont elle hérite, c'est la définition de la sous-classe qui l'emporte.

III - Un peu de pratique

Enoncé

Il est temps d'étoffer votre code. Réalisez les tâches suivantes :

- Personnage() :
  1. Un personnage apparaît avec 2 potions.
  2. Afficher le statut d'un personnage sur une ligne (vie, mana, stamina, potions)
  3. Boire une potion lui régénère 70 PV.
  4. Il peut mourir dans un cri effroyable.
  5. Il peut prendre des *dégâts*, mais l'esquive permet de passer outre.
  6. Il peut se concentrer pour récupérer 30 mana.
- Mage(Personnage):
  1. Peut lancer un sort qui fait 35 dégâts à un *Personnage*, et coûte 20 mana.
- Guerrier(Personnage) :
  1. Peut frapper un *Personnage*, ce qui fait 15 dégâts et coûte 5 mana.
  2. Peut frapper violemment, ce qui lui fait 15 dégâts, et 50 dégâts à un *Personnage*.
- Faire combattre Mario (un mage) contre Luigi (un Guerrier):
  1. Utiliser toutes les méthodes possible.
  2. Finir sur la mort d'au moins un des deux.

Aide

- Les mots soulignés sont les noms des variables ou méthodes.
- Les mots en italiques sont les arguments pour les méthodes.

Solutions

```
"""
Ici nous définirons la classe 'Personnage'
"""


class Personnage:
    def __init__(self, nom):
        self.nom = nom
        self.vie = 100
        self.mana = 50
        self.stamina = 100
        self.esquive = False
        self.potions = 2

    def __str__(self):
        return self.nom+" : PV = "+str(self.vie)+" , PA = "+str(self.mana)+"
, PM = "+str(self.stamina)

    def esquiver(self):
        if self.stamina >= 20:
            self.stamina -= 20
            self.esquive = True
            print(self.nom+" se prépare à esquiver")
        else:
            self.esquive = False
            print(self.nom+" échoue de manière ridicule")

    def boire(self):
        if self.potions > 0:
            self.vie += 70
            print(self.nom+" se requinque")
            if self.vie > 100:
                self.vie = 100

    def mourir(self):
        print(self.nom+" : AAAH *!/•@ stpgh")
        print(self.nom+" est mort...")

    def prendre_des_degats(self, degats):
        if self.esquive:
            self.esquive = False
            print(self.nom+" esquive l'attaque")
```

```
        else:
            self.vie -= degats
            if self.vie <= 0:
                self.mourir()

    def se_concentrer(self):
        self.mana += 40
        print(self.nom+" se concentre")
        if self.mana > 50:
            self.mana = 50
```

```
"""
Ici nous définirons la classe 'mage', sous-classe de 'Personnage'
"""
from personnage import *


class Mage(Personnage):

    def lancer_un_sort(self, cible):
        if self.mana >= 20:
            self.mana -= 20
            cible.prendre_des_degats(35)
            print(self.nom + " lance un sort sur " + cible.nom)
        else :
            print(self.nom+" échoue de manière ridicule")
```

```
"""
Ici nous définirons la classe 'guerrier', sous-classe de 'Personnage'
"""
from personnage import *


class Guerrier(Personnage):
    def frapper(self, cible):
        if self.mana >= 5:
            self.mana -= 5
            print(self.nom + " frappe " + cible.nom)
            cible.prendre_des_degats(15)
        else :
            print(self.nom+" échoue de manière ridicule")

    def frapper_violemment(self, cible):
        print(self.nom + " frappe violemment " + cible.nom)
        self.prendre_des_degats(15)
        cible.prendre_des_degats(50)
```

```
from mage import *
```

```
from guerrier import *
mario = Mage("Mario")
luigi = Guerrier("Luigi")


def statuts():
   print(mario)
   print(luigi)


statuts()
luigi.frapper(mario)
statuts()
mario.lancer_un_sort(luigi)
statuts()
luigi.frapper_violemment(mario)
statuts()
mario.lancer_un_sort(luigi)
statuts()
luigi.boire()
statuts()
mario.esquiver()
statuts()
luigi.frapper_violemment(mario)
statuts()
mario.se_concentrer()
statuts()
luigi.frapper_violemment(mario)
statuts()
```

IV - Syntaxe C++

La syntaxe en C++ est bien différente, et certains principes changent. Étant moins à l'aise en C++, je reconnais m'être complètement inspiré du site suivant. N'oubliez pas, pour coder vous pouvez tout trouver sur internet !

https://fr.wikibooks.org/wiki/Programmation_C%2B%2B/Les_classes

Et si vous n'avez pas d'IDE actuellement, je vous invite à en utiliser un en ligne, comme le suivant par exemple :

https://www.onlinegdb.com/online_c_compiler

Nous allons donc créer une **classe point**, qui contient des coordonnées et différentes fonctions de calcul avec d'autres points. Commençons !

Les fichiers

En C++, on garde les bonnes habitudes et on présente bien. Il nous faudra donc un fichier 'header' du

nom de point.h, c'est le fichier d'entête dans lequel seront listés les attributs et méthodes de l'objet. Un second fichier du nom de point.cpp, c'est le fichier source dans lequel seront contenues les méthodes. Et tant qu'à faire, un fichier où tester tout ça, main.cpp.

Principes de bases

La POO en C++ va s'appuyer sur des principes utilisable en Python, mais très fortement conseillé en C++.

Ainsi toute méthode ou tout attribut pourra être définie comme :

- *public* : dans ce cas on peut y accéder en dehors de la classe en les appelants.
- *private* : dans ce cas les appels ne fonctionnent qu'au sein de la classe.
- *protected* : comme private, mais les classes qui en hérite y on accès.

L'objectif est de protéger les attributs pour qu'ils ne soient pas modifier à tort. On doit alors créer des '**getter**' et des '**setter**' pour lire et modifier ces attributs. Ce sont des méthodes appelables en dehors de la définition de l'objet (donc publiques), qui pourront accéder aux attributs 'private' depuis l'intérieur de l'objet.

Tout comme la méthode _init()_ en Python, on peut (doit) définir un ou plusieurs **constructeurs** selon si l'on utilise des arguments.

```
#ifndef POINT_H
#define POINT_H

#include <iostream>
using namespace std;

class Point
{
public:
    // Constructeurs
    Point();
    Point(double x, double y);

    //Accesseurs et mutateurs
    void setX(double x);
    void setY(double y);
    double getX();
    double getY();

    // Autres méthodes
    void saisir();
    void afficher();

private:
    // Attributs privés
    double x,y;
```

```
};

#endif
```

```
#include "point.h"
#include <cmath> // pour 'sqrt', racine carrée
#include <iostream>
using namespace std;

Point::Point() : x(0), y(0)
{}

Point::Point(double x, double y) : x(x), y(y)
{}

void Point::setX(double x) // setter de l'attribut x
{
    this->x = x;
}

void Point::setY(double y) // setter de l'attribut y
{
    this->y = y;
}

double Point::getX() // getter de l'attribut x
{
    return this->x;
}

double Point::getY() // getter de l'attribut y
{
    return this->y;
}

void Point::saisir()
{
    cout << "Tapez l'abscisse : ";  cin >> this->x;
    cout << "Tapez l'ordonnée : ";  cin >> this->y;
}

void Point::afficher()
{
    cout << "L'abscisse vaut " << this->x << endl;
    cout << "L'ordonnée vaut " << this->y << endl;
}
```

```
#include "point.h"
#include <stdio.h>
```

```
int main()
{
    Point p1 = Point();
    p1.afficher();
    printf("Changement de x et y avec les setters :\n");
    p1.setX(4);
    p1.setY(7);
    p1.afficher();
}
```

À vous

Pour finir, après avoir pris connaissance de la syntaxe utilisé dans l'exemple, vous allez implémenter deux méthodes :

- **distance** : qui renvoie la distance du point considéré à un autre point.
- **milieu** : qui renvoie le point milieu entre le point considéré et un autre point.

'distance' renvoie un réel de type **'double'**, et prend en argument un autre point de type **'Point'**

'milieu' renvoie un point de type **'Point'**, et prend en argument un autre point de type **'Point'**

Il faut mettre à jour l'entête de la classe, avec dans les “autres méthodes” :

```
  // Autres méthodes
  double distance(const Point &P);
  Point milieu(const Point &P);
```

Puis on implémente les deux nouvelles méthodes dans le fichier point.cpp :

```
double Point::distance(Point &P)
{
    double dx = this->x - P.x;
    double dy = this->y - P.y;
    return sqrt(dx*dx + dy*dy);
}

Point Point::milieu(Point &P)
{
    Point result;
    result.x = (P.x + this->x) / 2;
    result.y = (P.y + this->y) / 2;
    return result;
}
```

Documents téléchargeables

Le dossier solutions Python



tp_corrige_forma_poo_egab.zip

From:
https://wiki.centrale-med.fr/egab/ - **E-Gab**

Permanent link:
https://wiki.centrale-med.fr/egab/formation:poo

Last update: **07/10/2020 23:23**