Programmation Orientée Objet - Pokemon !

GCC ! – Prologin

2020

Introduction

Imaginons que vous soyez en train de créer votre jeu vidéo ; comme nous ne vivons pas dans un monde de bisounours, il y a des ennemis et pas qu'un seul. Comment gérer la vie et les positions des différents ennemis ?

Vous pourriez définir une liste pour gérer tous les points de vie, une autre pour les positions horizontales, et une autre pour les positions verticales… Mais ce n'est vraiment pas le plus facile !

À la place, vous pourriez créer une liste d'ennemis, dans laquelle à chaque ennemi serait associé ses coordonnées, sa vie, etc…

En Python :

```
1 class Ennemi:
2     def __init__(self):
3         pass
4
5 boss_niveau1 = Ennemi()
```

Dans l'exemple précédent, `boss_niveau1` est un **objet** de la **classe** `Ennemi`. `boss_niveau1` est donc un cas particulier d'`Ennemi`.

Pour initialiser un objet, on utilise un constructeur (`__init__(self)` dans la fonction plus haut). Un objet naît à l'appel de son constructeur. Un constructeur porte toujours le nom de `__init__`. Il a obligatoirement l'argument `self` qui correspond à l'objet manipulé. Il peut également avoir d'autres arguments comme une fonction classique si l'on veut pouvoir préciser des informations lors de la construction de l'objet.

Les attributs décrivent les informations contenues dans l'objet et les méthodes décrivent la manière d'interagir avec ces informations.

```python
class Ennemi:
    def __init__(self, vie, x, y): # un constructeur a plusieurs arguments
        self.vie = vie
        self.coord_x = x
        self.coord_y = y

boss = Ennemi(200, 0, 0)
# on cree un objet boss de type Ennemi avec un attribut 'vie' qui vaut 200
minion = Ennemi(50, 10, 10)
```

Cela crée un ennemi `boss` qui a 200 points de vie, et se situe aux coordonnées (0,0) et un ennemi "classique" qui possède 50 points de vie et se situe en (10, 10).

On peut donc accéder à la vie du boss avec `boss.vie` et ses coordonnées avec `boss.coord_x` et `boss.coord_y`.

À chaque coup que prend le boss, on peut donc lui enlever de la vie avec

```python
boss.vie = boss.vie - 1
```

Par exemple, pour créer une liste d'ennemis, on peut faire :

```python
liste_ennemis = []

for i in range(nombre_ennemis):
    liste_ennemis.append(Ennemi(50, 10, 10))
```

Les classes servent à regrouper et lier des informations - ou **attributs** - et comportements - **méthodes**. On peut considérer que les attributs sont l'équivalent des variables dans une classe, et les méthodes l'équivalent des fonctions.

Bien évidemment, il est possible de modifier les attributs d'un objet depuis une méthode. Concrètement, une méthode est une fonction définie à l'intérieur d'une classe qui a toujours comme premier argument `self`.

FIGURE 1 – Pika !

```python
class Humain:
    def __init__(self, nom, age):
        self.age = age # Ceci est un attribut
        self.nom = nom # Ceci est un autre attribut

    def info(self): # Ceci est une méthode
        print("Mon nom est " + self.nom + " et j'ai " + str(self.age) + " ans.")

    def anniversaire(self): # Ceci est une autre méthode
        self.age += 1

lea = Humain("Lea", 14)
```

`lea.info()` affichera `Mon nom est Lea et j'ai 14 ans.` `lea.anniversaire()` augmente l'âge de Léa de 1. Si on fait `lea.anniversaire()` puis `lea.info()` on obtient :

`Mon nom est Lea et j'ai 15 ans.`

Attrapez les tous !

Exercice 1

Créer une classe *Pokemon* qui a les attributs suivants :

- Nom
- Vie
- Dégâts

— Niveau

Exercice 2

Créer une méthode `statut` qui affiche le nom, la vie, les dégâts et le niveau du Pokémon.

Exercice 3

Créer une méthode `attaque` qui prend en argument un autre pokémon. Le Pokémon adverse devra perdre de la vie en fonction des dégâts de notre Pokémon.

Exercice 4

Créer une classe `Dresseur` qui a les attributs suivants :

— Nom
— Liste de Pokémons

Le nom devra être passé en argument au constructeur, et la liste sera initialisée à vide.

Exercice 5

Ajouter une méthode `capture`, permettant au dresseur d'ajouter un Pokémon à sa liste de Pokémons. Le nouveau Pokémon sera passé en argument à la méthode.

C'est l'heure du du-du-du-duel !

Exercice 6

Créer une fonction `combat` qui prend en argument deux Pokémons et affiche le vainqueur. Cette fonction devra faire combattre les Pokémons l'un contre l'autre. Le dernier Pokémon en vie sera gagnant.

Exercice 7

Créer une fonction `duel` qui prend en argument deux dresseurs et affiche le vainqueur. Cette fonction devra faire affronter les Pokémons des dresseurs selon leur ordre dans la liste. Dès que la vie d'un Pokémon passe à 0 ou moins, le Pokémon d'après est envoyé combattre. N'hésitez pas à réutiliser votre fonction `duel` !

Exercice 8

Créer une fonction `generer` qui génère des Pokémons aux caractéristiques aléatoires avec le module `random`.

Pour ce faire, importe d'abord le module avec

```
import random
```

Tu peux maintenant utiliser `random.choice(caracteristiques)` pour obtenir un élément aléatoire de la liste 'caracteristiques'. Il faut penser à la créer auparavant bien évidemment !

Si tu as des questions à propos du module `random` n'hésite pas à demander à un organisateur ou à chercher sur internet par toi-même.

Bonus : l'héritage

L'héritage est l'un des points primordiaux de la programmation objet. Cela permet de ne pas dupliquer de code mais surtout de partager des origines communes entre classes. La classe qui hérite est appelée *classe enfant* ou *sous-classe* et celle dont elle hérite est appelée *classe parent* ou *super-classe*. Par exemple, le schéma ci-dessous représente une classe `Archer` qui hérite de la classe `Ennemi`. Effectivement, un archer possède toutes les caractéristiques d'un ennemi. De plus, `Boss Archer` hérite de la classe `Archer`. Le `Boss Archer` est donc un archer particulier, par exemple avec des caractéristiques (attributs) supplémentaires.

```
Ennemi
|-- Archer
|   |- Boss Archer
|   + ...
+-- Guerrier
    |- Boss Guerrier
    + ...
```

Mais l'héritage ne se limite pas à une simple généalogie. Un second point important est que toute classe enfant peut être utilisée à la place de sa classe parent.

Voici un exemple :

```python
class Animal:
    def __init__(self, jambes, age):
        self.nombreJambes = jambes
        self.age = age

    def manger(self):
        print(self.nom + "mange !")

class Chien(Animal):
    def __init__(self, age, nom):
        Animal.__init__(self, 4, age) # Un chien a 4 pattes
        self.nom = nom # Un chien a un nom

    def aboie(self): # Un chien peut aboyer
        print(self.nom + " Wouaf")

class Poisson(Animal):
    def __init__(self, age, color):
        Animal.__init__(self, 0, age) # Un poisson a 0 patte
        self.color = color # Un poisson a une couleur

    def nager(self): # Un poisson peut nager
        print(">=o Je nage, je nage !")

chien = Chien(2, "Rantanplan")
chien.manger()
chien.aboie()
```

Comme vous pouvez le voir, toute classe enfant peut accéder aux attributs et méthodes de sa classe parent directement, à l'aide de `self`.

Exercice bonus

Voici une liste de fonctionnalités que vous pouvez ajouter :

- Ajouter différents types de Pokémons en utilisant l'héritage, comme feu, eau, plante, vol, ...
- Ajouter des attaques spéciales aux Pokémons avec des méthodes dans les sous-classes comme par exemple "Lance-Flamme" pour que les pokémons de type feu infligent plus de dégâts ou "Abri" pour que les pokémons de type sol se protègent, ...

Conclusion

De manière générale, ce sujet vous permettra d'acquérir les compétences nécessaires en programmation orientée objet, afin de faciliter votre travail sur les sujets Micro :bit et jeu vidéo !