Chap 04. Programmation Orientée Objet

Livre p 19 – Chap 2 Modularité
Livre p 45 – Chap 3 Programmation objet

1. P.O.O.

a. Vocabulaire :

La Programmation Orientée Objet (P.O.O.) permet de créer des objets numériques ayant une structure plus ou moins complexe. Ces objets sont manipulables, modifiables et peuvent interagir entre eux.

- **Classes** : Une classe regroupe les fonctions (méthodes) et les caractéristiques (attributs) communs à tous les objets d'une même catégorie.
  Exemple : Une classe Vecteur en géométrie, une classe Personnage dans un jeu…
- **Attributs** : Les classes et les objets peuvent posséder des caractéristiques, appelées attributs, dont le format peut être défini lors de la création de la classe (ou plus tard mais c'est déconseillé).
  Remarque : Un attribut de classe a la même valeur pour tous les objets de la classe.
  Exemple : La classe Personnage pourrait avoir comme attribut la vie maximale (hp_max), chaque personnage ayant comme attributs : ses coordonnées (x, y) et sa vie (hp).
- **Méthodes** : Les objets crées peuvent posséder des fonctions spécifiques, appelées méthodes, dont le fonctionnement est défini lors de la création de la classe.
  Exemple : Les vecteurs en géométrie pourraient avoir comme méthode leur norme.

Remarque : Les attributs et les méthodes sont généralement définis à l'intérieur de la création de la classe, on appelle ce procédé, l'encapsulation.

b. Définition d'une classe en Python

Pour créer une classe d'objets en Python on utilise le mot clef class suivi du nom de la classe et d'un « : », puis les attributs et méthodes seront dans un bloc indenté.
Remarque : Habituellement, le nom (ou identifiant) des classes s'écrit avec une majuscule suivie de lettres minuscules
L'intérêt d'utiliser des objets est de pouvoir leurs attribuer des attributs et des méthodes spécifiques…
Le mieux est de définir ces attributs lors de la création de la classe avec la fonction **__init__** et le mot clef **self** qui fait référence à l'objet lui-même, ainsi que les méthodes qui seront applicables à ces objets.
Exemple :

```
class Personnage :
    hp_max = 100
    def __init__(self, abscisse, ordonnee, vie) :
        self.x = abscisse
        self.y = ordonnee
        self.hp = vie
    def vivant(self) :
        return (self.hp > 0)
```

Ainsi tous les personnages ont un attribut commun : hp_max.
À la création d'un objet, il faut donner :

- ses coordonnées (x, y)
- ses points de vie hp.

Un personnage est « vivant » tant que ses points de vie sont supérieurs à 0.

Si on crée l'objet `joueur = Personnage(2, 0, 80)` , alors :

`joueur.hp_max` renvoie la valeur 100
`joueur.x` renvoie la valeur 2
`joueur.y` renvoie la valeur 0
`joueur.hp` renvoie la valeur 80
`joueur.vivant()` renvoie la valeur True

c. Méthodes spéciales en Python

Si l'on veut afficher le contenu d'un objet, on ne va voir que son adresse mémoire, pour un affichage plus lisible et plus complet, on utilise la fonction **__repr__** lors de la création de la classe.

```
class Personnage :
   …
   def __repr__(self) :
      return "({},{}) hp:{}".format(self.x, self.y, self.hp)
```

Ainsi l'affichage de l'objet joueur sera : '(2,0) hp:80'

Attention : Un test d'égalité entre deux objets ne teste pas si leurs attributs sont les mêmes mais s'il s'agit exactement des mêmes objets enregistrés à la même adresse mémoire !
On peut modifier la définition du test d'égalité avec la méthode **__eq__** et même modifier d'autres fonctions basiques du langage Python pour les adapter aux nouveaux objets créés…

- __eq__(Obj_1, Obj_2) : (equal) Redéfinition de Obj_1 == Obj_2
- __ne__(Obj_1, Obj_2) : (not equal) Redéfinition de Obj_1 != Obj_2
- __lt__(Obj_1, Obj_2) : (lesser than) Redéfinition de Obj_1 < Obj_2
- __le__(Obj_1, Obj_2) : (lesser or equal) Redéfinition de Obj_1 <= Obj_2
- __gt__(Obj_1, Obj_2) : (greater than) Redéfinition de Obj_1 > Obj_2
- __ge__(Obj_1, Obj_2) : (grater or equal) Redéfinition de Obj_1 >= Obj_2
- __add__(Obj_1, Obj_2) : (add) Redéfinition de Obj_1 + Obj_2
- __sub__(Obj_1, Obj_2) : (subtract) Redéfinition de Obj_1 – Obj_2
- __neg__(self) : (negative) Redéfinition de – Obj
- __mul__(Obj_1, Obj_2) : (multiply) Redéfinition de Obj_1 * Obj_2
- __div__(Obj_1, Obj_2) : (divide) Redéfinition de Obj_1 / Obj_2
- __str__(self) : (string) Redéfinition de str(Obj)
- __len__(self) : (length) Redéfinition de len(Obj)

2. Gestion des erreurs

a. Exceptions

Les « bugs » dans le déroulement d'un programme envoient un message d'erreur dans la console, on les appelle des exceptions, les plus fréquentes sont :

- **SyntaxError** : Erreur de syntaxe (l'erreur sera détectée lors de la lecture du programme !)
- **ZeroDivisionError** : Division par zéro
- **NameError** : Accès à une variable non définie
- **IndexError** : Accès à un indice non valide dans une liste, un tuple ou une chaîne de car.
- **KeyError** : Accès à une clé non définie dans un dictionnaire
- **TypeError** : Opération entre des données incompatibles.
- **AssertionError** : Erreur sur une assertion.

b. Anticiper une erreur

On a vue en 1ère NSI, dans le chapitre 10 – Tests & boucles, que l'on pouvait tester un bloc d'instruction et réagir quand une erreur est trouvée, avant que le programme ne soit stoppé, avec les instructions :

try *bloc_1* **except** *bloc_2*

c. Lever une erreur

On peut déclencher soi-même un message d'erreur avec l'instruction : **raise** *nom_exception*

d. Assertions

Pour tester une fonction, on peut utiliser des assertions avec l'instruction : **assert** *condition*

Exemple : assert 1 + 1 == 2 ne provoque aucune erreur
assert 1 + 1 == 3 provoque le message d'erreur AssertionError