Programmation Orientée Objet (POO)

BARBIER J.M. - LGT Dumezil - NSI

29 septembre 2025

Table des matières

1 Programme

1.1 Partie “structures de donneés”

Contenus : Vocabulaire de la programmation objet :

— classes
— attributs

— méthodes
— objets

Capacités attendues :

— écrire la définition d'une classe
— accéder aux attributs et méthodes d'une classe

Commentaires :

On n'aborde pas ici tous les aspects de la programmation objet comme le polymorphisme et l'héritage.

2 Vocabulaire de la POO

2.1 Notion de classe

2.1.1 Définition

Une **classe** est une construction de langage qui permet de **réunir les données et les fonctions** agissant sur ces données.

2.1.2 Exemple

```
class Robot:
    # Constructeur (appelé quand la classe est instanciée)
    def __init__(self):
        # x : attribut de la classe (ou propriété)
        self.x = 0
        # y : attribut de la classe (ou propriété)
        self.y = 0

    # méthode "bouge" de la classe Robot
    def bouge(self, dx, dy):
        self.x += dx
        self.y += dy

    # méthode "position" de la classe Robot
    def position(self):
        return (self.x, self.y)

# walle est un objet, instance de la classe Robot
walle = Robot()
# evee est un objet, autre instance de la classe Robot
evee = Robot()

walle.bouge(3,3)
evee.bouge(1,2)

print("WallE", walle.position())
```

```
print("Evee",  evee.position())
print("x WallE", walle.x) # accès direct
```

2.1.3 Représentation en mémoire

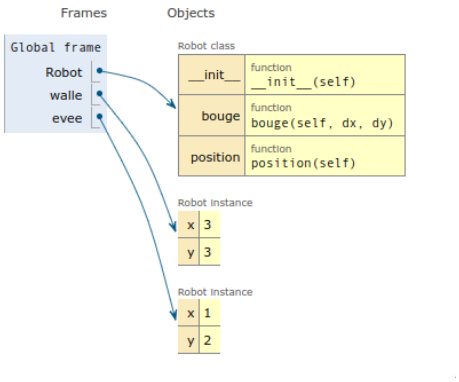

FIGURE 1 – Représentation en mémoire

2.1.4 Vocabulaire

- **classe** : description de données appelées *attributs* et d'opérations appelées *méthodes* ; c'est un "modèle" permettant de fabriquer des *objets* (instances)
- **constructeur** : fonction d'initialisation permettant de créer une *instance*
- **instance** : objet créé à partir d'un "modèle" (*classe*)
- **attribut** : données d'une instance
- **méthode** : fonctions agissant sur une instance

en python :

```
class NomClasse():  # définition de classe, CamelCase
    def __init__(self): # constructeur, self est une référence à l'instance
        self.attribut = .. # attribut initialisé
    def methode(self): # méthode
        ...
obj = NomClasse() # instanciation (obj = instance de NomClasse)
obj.attribut # accès direct à l'attribut
obj.methode() # appel d'une méthode, self est passé automatiquement
# équivalent à NomClasse.methode(obj)
```

2.1.5 Attributs

Les **attributs** sont initialisés dans le **constructeur** (méthode `__init__`).

Ils peuvent être déclarés avant le constructeur (pour indiquer leur type).

```
class Robot:
    x: int
    y: int
    def __init__(self):
```

```
        self.x = 0
        self.y = 0
```

Une valeur par défaut peut être donnée aux attributs dans la déclaration (cf attributs de classe).

2.1.6 Méthodes

Les **méthodes** sont des fonctions définies dans la classe, qui agissent sur une instance de la classe. Le premier argument d'une méthode est toujours `self`, qui est une référence à l'instance sur laquelle la méthode est appelée.

Les attributs de l'instance sont accessibles via `self.attribut`.

```
class Robot:
    ...
    def bouge(self, dx, dy):
        self.x += dx
        self.y += dy
```

2.1.7 Encapsulation

Il est préférable de ne pas agir directement sur les données d'un objet, mais de passer par des méthodes, qui jouent le rôle d'**interface** entre l'objet et le programme.

On peut **protéger** les attributs d'une classe en faisant débuter leur nom par un double underscore (`self.__x` par exemple)

```
class Robot:
    def __init__(self):
        self.__x = 0
        self.__y = 0
r2d2 = Robot()
r2d2.__x # ===> boum erreur pas permis
```

Remarque : certains langages (Java, C++, …) permettent de définir des attributs **privés** (non accessibles en dehors de la classe) et des attributs **publics** (accessibles en dehors de la classe). En python, tous les attributs sont publics, mais on peut **simuler** des attributs privés en utilisant le double underscore.

3 Infos utiles

3.1 Méthodes "magiques" / spéciales

3.1.1 Référence

Elles permettent (entre autres) d'implémenter les opérations "classiques" en python :

https ://docs.python.org/fr/3/reference/datamodel.html

3.1.2 Exemple

```
class Point():
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, p):
        return Point(self.x + p.x, self.y + p.y)

p1 = Point(2, 4)
p2 = Point(3, 5)
p = p1 + p2
print(p.x, p.y) # 5, 9
```

Exemples : `__repr__`, `__str__`, `__eq__`, `__lt__`, `__add__`, `__sub__`, ...

3.2 Attributs de classe

3.2.1 Utilisation

Un **attribut de classe** est défini au niveau de la classe, mais sert aussi de valeur par défaut aux attributs de la classe. On peut donc s'en servir pour stocker une information au niveau de la classe elle-même (bien) et/ou pour donner une valeur par défaut aux attributs (pas bien, mais couramment utilisé dans les faits)

```
class Point():
    cpt = 0
    x = 0
    def __init__(self, y):
        self.y = y
        Point.cpt += 1
    def pos(self):
        print(self.x, self.y)
```

3.2.2 Exemple d'utilisation

```
p1 = Point(2)
p1.pos() # 0 2
p1.x = 2
p1.pos() # 2 2
Point.x = 3
p1.pos() # 2 2
p2 = Point(3)
p2.pos() # 3 3
print(Point.cpt) # 2
```

De manière générale, attention aux valeurs par défaut de **type mutable** (listes, objets, ...)

4 Exercices

4.1 Contact et carnet adresses

4.1.1 Contact

Définir et tester une classe `Contact` permettant de

- stocker les coordonnées d'une personne (`nom`, `prenom`, `email`)
- afficher les coordonnées d'une personne sous la forme `NOM Prénom <email>` via la méthode `__str__`
- vérifier si le nom, le prénom ou l'email contiennent une sous-chaine `cible` de caractères donnée via la méthode `search(self, cible)`

4.1.2 Carnet

Définir et tester une classe `Carnet` permettant de

- stocker les coordonnées de plusieurs personnes (via `Contact`)
- ajouter une personne au carnet d'adresse via la méthode `ajoute(self, nom, prenom, email)`
- renvoyer toutes les personnes dont les coordonnées contiennent une sous-chaîne donnée via la méthode `search(self, cible)`

4.2 Date et heure

4.2.1 Heure

Définir et tester une classe `Heure` qui permet de

- stocker une heure sous la forme de 3 attributs : `heure`, `minute`, `seconde`
- renvoyer via la méthode `__str__` la date sous forme de chaîne de caractères au format `8h48m13s`
- tester si une heure est supérieure à une autre via la méthode `__gt__(self, autre_heure)` (ou inférieure via `__lt__`)
- ajouter `x` secondes à une heure via la méthode `ajoute(self, x)`
  - ajouter `x` à l'attribut `seconde`
  - si `seconde >= 60`, ajouter `seconde // 60` à `minute` et mettre `seconde = seconde % 60`
  - si `minute >= 60`, ajouter `minute // 60` à `heure` et mettre `minute = minute % 60`
  - si `heure >= 24`, mettre `heure = heure % 24`

4.2.2 Date

Définir et tester une classe `Date` qui permet de

- stocker une date sous la forme de 3 attributs numériques : `année`, `mois`, `jour`

— renvoyer via la méthode `__str__` la date sous forme de chaîne de caractères au format `28 mai 1978`
— ajouter un jour à une date (sans prendre en compte les années bissextiles) via une méthode `inc`; utiliser un tableau `nb_jours` donnant le nombre de jours dans chaque mois pour passer au mois suivant et à l'année suivante si besoin
— tester si une date est égale à une autre via la méthode `__eq__(self, autre_date)`

4.3 Ligne et points

4.3.1 Point

Définir et tester une classe `Point` permettant de :

— stocker les coordonnées `x`, `y` et `z` d'un point
— renvoyer via `coords()` les coordonnées du point sous la forme d'un triplet `(x, y, z)`
— translater le point de `dx`, `dy`, `dz` via la méthode `bouge(self, dx, dy, dz)`

4.3.2 Ligne

Définir et tester une classe `Ligne` permettant de :

— stocker les coordonnées a et b des 2 points (instances de `Point`) qui définissent la ligne
— translater la ligne de `dx`, `dy` et `dz` via la méthode `translate(self, dx, dy, dz)` (la translation agit sur les 2 points, utiliser la méthode `bouge` de `Point`)
— renvoyer les coordonnées d'un point situé à p entre a et b (si $p = 0$, on est sur a, si $p = 1$ on est sur b, et de manière générale $c(p) = c(a) + p * (c(b) - c(a))$) via la méthode `curseur(self, p)`

4.4 Projet pyxel

4.4.1 Objectif

Réaliser un mini-jeu avec pyxel, en utilisant la programmation orientée objet.

Le jeu est un jeu de type "évite les obstacles" : une pluie de météorites tombe, et le joueur doit les éviter en se déplaçant sur l'écran.

Vous sauvegarderez chaque étape de votre travail en utilisant `git`.

4.4.2 Étapes

1. Définir la classe `Jeu` avec 2 attributs : `width` et `height` (par défaut 160 et 120) qui sont la largeur et la hauteur de l'écran, et 3 méthodes :
   — `play` (lance le jeu : `pyxel.init(self.width, self.height)` et `pyxel.run(self.update, self.draw)`)
   — `update` (met à jour l'état du jeu, ne fait rien pour l'instant)

— `draw` (dessine l'état du jeu ; pour l'instant elle ne fait que effacer l'écran). La classe doit comporter 2 attributs, `width` et `height`, qui sont la largeur et la hauteur de l'écran.

L'appel du jeu se fait via :

```
if __name__ == "__main__":
    jeu = Jeu(160, 120)
    jeu.play()
```

2. Définir la classe `Joueur` avec 4 attributs : `x`, `y`, `width` et `height` (par défaut 8 et 8) qui sont la position et la taille du joueur, et 3 méthodes :

— `update()` (met à jour la position du joueur en fonction des touches fléchées sur le clavier)
— `draw()` (dessine le joueur sous la forme d'un carré plein)
— `get_rect()` (renvoie un tuple `(x, y, width, height)` représentant le rectangle du joueur)

Le constructeur de la classe doit initialiser `x` et `y` pour que le joueur soit positionné au centre de l'écran.

Ajouter un attribut `joueur` à la classe `Jeu`, et modifier les méthodes `update` et `draw` pour utiliser les méthodes de la classe `Joueur`, ainsi que le constructeur de `Jeu` pour initialiser l'attribut `joueur`.

3. Définir la classe `Meteore` avec 6 attributs : `x`, `y`, `width`, `height`, `speed` et `color` `speed` qui sont la position, la taille, la vitesse et la couleur de la météorite, et 3 méthodes :

— `update()` (met à jour la position de la météorite en fonction de sa vitesse). Lorsque la météorite sort de l'écran (i.e. `y > Jeu.height`), elle doit être repositionnée en haut de l'écran (i.e. `y` est remis à une valeur aléatoire entre `-Jeu.height` et 0, et `x` est remis à une valeur aléatoire entre 0 et `Jeu.width`)
— `draw()` (dessine la météorite sous la forme d'un carré plein)
— `get_rect()` (renvoie un tuple `(x, y, width, height)` représentant le rectangle de la météorite)

Le constructeur de la classe doit initialiser `x` à une valeur aléatoire entre 0 et `Jeu.width`, `y` à une valeur aléatoire entre 0 et `-Jeu.height` (au dessus de l'écran pour qu'elle apparaisse en tombant à un moment aléatoire), `width` et `height` à une valeur aléatoire comprise entre deux bornes à choisir et `speed` à une valeur aléatoire entre 1 et 3. Idem pour la couleur.

Ajouter un attribut `level` à la classe `Jeu`, qui est un entier représentant le niveau actuel du jeu (initialisé à 1), et un attribut `meteores` à la classe `Jeu`, qui est une liste de météorites, et modifier les méthodes `update` et `draw` pour utiliser les méthodes de la classe `Meteore`, ainsi que le constructeur de `Jeu` pour initialiser l'attribut `meteores` avec une liste de `level*10` météorites (où `level` est un attribut de `Jeu` initialisé à 1 dans le constructeur).

4. Ajouter un attribut `phase` à la classe `Jeu`, qui peut prendre les valeurs :

— `"pre"` : phase de préparation
— `"play"` : phase de jeu
— `"gameover"` : phase de fin de jeu

Modifier les méthodes `update` et `draw`de la classe `Jeu` pour gérer les différentes phases :

- en phase `"pre"`, afficher “Appuyez sur ESPACE pour commencer” au centre de l’écran dans la méthode `draw`, et passer en phase `"play"` si la touche ESPACE est pressée dans la méthode `update`
- en phase `"play"`, mettre à jour et dessiner le joueur et les météorites
- en phase `"gameover"`, afficher “Game Over! Appuyez sur ESPACE pour rejouer” au centre de l’écran dans la méthode `draw`, et repasser en phase `"pre"` si la touche ESPACE est pressée.

5. Ajouter une méthode `collision(self, rect1, rect2)` à la classe `Jeu` qui teste si deux rectangles (donnés sous la forme de tuples `(x, y, width, height)`) se chevauchent. Le test se fait en vérifiant que les rectangles ne sont pas disjoints :

   ```
   x1, y1, w1, h1 = rect1
   x2, y2, w2, h2 = rect2
   return not (x1 + w1 < x2 or x2 + w2 < x1 or y1 + h1 < y2 or y2 + h2 < y1)
   ```

   Modifier la méthode `update` de la classe `Jeu` pour tester si le rectangle du joueur (obtenu via `joueur.get_rect()`) chevauche le rectangle d’une météorite (obtenu via `meteore.get_rect()`). Si c’est le cas, passer en phase `"gameover"`.
6. (optionnel) Ajouter toutes les améliorations que vous souhaitez (son, score, niveaux, sprites, effets visuels, multijoueur…)