

Paradigmes de programmation

Thème : Structure de données

Vocabulaire de la programmation objet : classes, attributs, méthodes, objets.	Écrire la définition d'une classe. Accéder aux attributs et méthodes d'une classe.	On n'aborde pas ici tous les aspects de la programmation objet comme le polymorphisme et l'héritage.
--	---	--

Thème : Langage et programmation

Paradigmes de programmation.	Distinguer sur des exemples les paradigmes impératif, fonctionnel et objet. Choisir le paradigme de programmation selon le champ d'application d'un programme.	Avec un même langage de programmation, on peut utiliser des paradigmes différents. Dans un même programme, on peut utiliser des paradigmes différents.
------------------------------	---	--

Source du cours :

- Cours rédigé par Frédéric Mandon sous licence Creative Commons BY NC SA
- Liste de diffusion NSI
- NSI 24 leçons avec exercices corrigés, édition Ellipse
- David Roche : https://pixees.fr/informatiquelycee/n_site/nsi_term_paraProg_poo.html

Les langages de programmation sont nombreux et variés. On peut les regrouper dans plusieurs classes, correspondantes à des schémas de pensée différents : ce sont les paradigmes de programmation. Il n'est pas inusuel qu'un langage appartienne à plusieurs de ces classes, c'est par exemple le cas de Python (ainsi que de C++, Ruby, Ocaml...). Certains de ces paradigmes sont mieux adaptés que d'autres pour traiter des problèmes spécifiques. On verra ultérieurement qu'il est possible d'utiliser plusieurs paradigmes à l'intérieur d'un même programme. Les paradigmes principaux sont impératif, objet, déclaratif.

I Programmation impérative

Le paradigme sinon le plus ancien, tout au moins le plus traditionnel, est le paradigme impératif. Les premiers programmes ont été conçus sur ce principe :

- Une suite d'instructions qui s'exécutent séquentiellement, les unes après les autres ;
- Ces instructions comportent :
 - Affectations
 - Boucles (pour..., tant que..., répéter...jusqu'à...)
 - Conditions (si...sinon)
 - Branchement/saut sans condition
- La programmation impérative actuelle limite autant que possible les sauts sans condition. Ce sous-paradigme est appelé programmation structurée. Les sauts sont utilisés en assembleur (instructions BR adr, « branch vers adresse »). En Python, on limite ainsi l'utilisation du break à certains cas particuliers. A l'inverse, un programme utilisant de nombreux sauts est qualifié de « programmation spaghetti », pour la « clarté » toute relative avec laquelle on peut le dérouler. Certains langages peuvent donner facilement ce style de code (BASIC, FORTRAN,...)
- L'usage des fonctions comme vu en première est aussi une variante de la programmation impérative, appelée programmation procédurale. Elle permet de mieux suivre l'exécution d'un programme, de le rendre plus facile à concevoir et à maintenir, et aussi d'utiliser des bibliothèques.

II Programmation objet

1) Principes

Comme son nom l'indique, le paradigme objet donne une vision du problème à résoudre comme un ensemble d'objets. Ces objets ont des caractéristiques et des comportements qui leur sont propres, ce sont respectivement les **attributs** et les **méthodes**. Les objets interagissent entre eux avec des **messages**, en respectant leur **interface** (c'est-à-dire l'ensemble des spécifications en version compacte, les **signatures** des méthodes, on verra ce point plus en détail tout au long de l'année).

2) Un exemple « papier »

Créons un jeu vidéo de type « hack and slash ». Dans ce type de jeu, le personnage joué doit tuer un maximum de monstres sur une carte de jeu. A lire le descriptif, 3 objets apparaissent naturellement :

- Le personnage principal ;
- Les monstres ;
- La carte.

On remarque immédiatement que « monstre » aurait plutôt tendance à désigner un type qu'un objet unique : les objets sont tous typés. Le type définit à la fois le nom de l'objet, et ce qu'il fait.

De même, avec cette structure, on peut avoir plusieurs objets « personnages », pour jouer en multi-joueur, et bien sûr plusieurs cartes.

Le « moule » avec lequel on va fabriquer un objet est appelé **classe**.

La classe personnage comprend par exemple les attributs :

- Points de vie
- Dégâts maximums qu'inflige le personnage
- Position

Elle comprend les méthodes :

- Déplacement
- Attaque
- Et des méthodes qui permettent d'accéder aux attributs, ou bien de les modifier. Ce sont les **accesseurs** et les **mutateurs**. Les attributs sont cachés des objets extérieurs (le principe est l'**encapsulation**), ils sont **privés**. Les méthodes permettant d'y accéder sont à l'inverse **publiques**. Un objet extérieur ne doit pas pouvoir modifier à loisir les attributs d'un autre objet, en effet il doit y avoir un contrôle de l'objet sur ses propres attributs.

Quand on crée un personnage, l'ordinateur crée une **instance** de la classe. C'est-à-dire que tous les objets de la classe auront les mêmes attributs et méthodes, mais que deux objets de la même classe peuvent avoir des valeurs différentes pour les attributs. L'instance est créée grâce à un **constructeur**.

Métaphore :

- Une classe, c'est le plan d'une maison (abstrait) ;
- Un objet, c'est une maison issue du plan (concret). Ce qu'il y a à l'intérieur d'une maison diffère de l'intérieur d'une autre maison (décoration, mobilier, etc...) ;
- L'interface c'est le bouton qui permet de régler le chauffage ;
- L'implémentation (ou la réalisation) de l'interface, c'est la méthode de chauffage/climatisation retenue. L'utilisateur ne connaît pas le détail de l'implémentation, ce qui compte pour lui, c'est le bouton de réglage (donc l'interface)
- *Une interface n'est pas forcément liée à la programmation objet.* Vous avez spécifié –en théorie du moins ☺– les programmes que vous avez faits l'an dernier. Mettre en forme ces spécifications permet de les utiliser sans avoir à les comprendre : cette mise en forme est une interface.

3) Une classe en Python

Le code suivant montre, étape par étape, la classe « personnage » telle qu'on l'a définie au paragraphe précédent.

- Le mot-clé pour définir une classe est **class**. On donne ensuite les spécifications de la classe (on documente).

```

class Personnage:
    """
    Personnage d'un jeu de type hack 'n slash

    Attributs :
        _nom : chaîne de caractères, nom du personnage
        _pv : entier positif ou nul, points de vie du personnage
        _degats : entier strictement positif, dégâts maximum
                  du personnage
        _position : couple d'entiers donnant l'abscisse et l'ordonnée
                  du personnage sur la carte

    Méthodes:
        Init() constructeur de la classe Personnage
        getAttribut() : accesseurs des attributs
        setAttributs(nouvelle_valeur) : mutateurs des attributs.
                  Uniquement pour les attributs _pv et _position
        deplacement(paramètres) : permet de changer la position
                  du personnage
        attaque() : renvoie les dégâts faits à l'adversaire
    """

```

- La première méthode dans la classe est le constructeur, appelé `__init__` en Python. Toutes les méthodes d'une classe ont au moins le paramètre `self`, c'est-à-dire que la méthode s'applique à l'objet lui-même. Dans le constructeur de `Personnage` est aussi passé en argument le paramètre `nom`. Le constructeur initialise les attributs de l'objet (points de vie, dégâts, position). Tous les attributs sont précédés d'un tiret bas « `_` » pour signifier qu'ils sont privés.

```

def __init__(self,nom):
    """
    Constructeur de la classe Personnage
    Données:
        _nom : chaîne de caractères, nom du personnage
        _pv : entier positif ou nul, points de vie du personnage
        _degats : entier strictement positif, dégâts maximum du
                  personnage
        _position : couple d'entiers donnant l'abscisse et
                  l'ordonné du personnage sur la carte

    Résultat :
        ne retourne rien, crée un nouveau Personnage
    """
    self._nom = nom
    self._pv = 80
    self._degats = 8
    self._position = (0,0)

```

- Accesseurs (getters en anglais) et mutateurs (setters en anglais). On ne documente pas les accesseurs, on peut le faire pour les mutateurs. Les accesseurs n'ont pas de paramètre (à part `self`), les mutateurs ont la nouvelle valeur. Il n'y a pas forcément de mutateurs (ni d'accesseurs) pour tous les attributs : le nom du personnage n'est pas modifiable ici.

```

#accesseurs des attributs
def getNom(self):
    return self._nom
def getPv(self):
    return self._pv
def getDegats(self):
    return self._degats
def getPosition(self):
    return self._position

#mutateurs des attributs
def setPv(self,nouveaux_pv):

```

```

"""
Les points de vie d'un personnage sont positifs ou nul.
"""
if nouveaux_pv < 0:
    self._pv = 0
else:
    self._pv = nouveaux_pv
def setDegats(self,nouveaux_degats):
    self._degats = nouveaux_degats
def setPosition(self,nouvelle_pos):
    # un contrôle sur la position doit se faire en communiquant avec
    # l'objet carte: x et y doivent être compatibles.
    # Il y aura des instructions du type carte.getDimensions(),
    # cartes.getObstacles() etc. dans cette méthode
    self._position = nouvelle_position

```

- Méthodes de la classe. Comme précédemment, les méthodes ont `self` en premier paramètre. Dans cet exemple, la méthode de déplacement reste à programmer suivant le jeu.

```

def deplacement(self,paramètres):
    """
    Données: paramètres dépendant des règles
    Résultat: renvoie le booléen tout_s_est_bien_passe Vrai si le
               déplacement est possible
    à programmer suivant les règles,
               le return est assez inélégant ici !
    """
    tout_s_est_bien_passe = True
    #...code...
    if tout_s_est_bien_passe :
        return True
    else:
        return False
def Attaque(self):
    """
    Données: pas de paramètre dans cette méthode
    Résultat: renvoie un entier aléatoire compris entre 1 et _degats
    """
    return(randint(1,self._degats))

```

On représente la classe comme ceci :

Personnage
<code>_nom : String</code> <code>_pv : Int >= 0</code> <code>_degats : Int >= 0</code> <code>_position : Tuple(Int, Int)</code>
<code>deplacement(Paramètres à déterminer)</code> <code>Attaque() : Int >= 0</code>

4) Création d'une instance et accès aux attributs.

Création d'un objet :

```
>>> perso_1 = Personnage("Un Seul Bras Les Tua Tous")
```

l'appel à `mon_perso` renvoie l'adresse de l'objet :

```
>>> perso_1
<__main__.Personnage object at 0x110c892b0>
```

Pour accéder aux attributs, on utilise l'accesseur, sans préciser le paramètre `self` :

```
>>> perso_1.getNom()
'Un Seul Bras Les Tua Tous'
```

Pour modifier un attribut, on utilise le mutateur, sans préciser le paramètre `self` :

```
>>> perso_1.setDegats(12)
>>> perso_1.getDegats()
12
```

Attributs publics, attributs privés et Python.

En programmation objet, indépendamment du langage, on considère que les attributs doivent être privés, encapsulés à l'intérieur de la classe et accessibles uniquement par mutateurs. En Python avancé la situation est différente : les propriétés et décorateurs, que l'on ne verra pas cette année, évitent qu'un objet extérieur modifie un attribut sans en respecter les spécifications.

Les attributs dans le constructeur ne sont plus précédés du double tiret, le code devient :

```
def __init__(self, nom):
    """
    Constructeur de la classe Personnage
    """
    self.nom = nom
    self.pv = 80
    self.degats = 8
    self.position = (0,0)
```

Après création du personnage, on peut alors accéder et modifier les attributs sans getters ni setters :

```
>>> perso_1 = Personnage("Un Seul Bras Les Tua Tous")
>>> perso_1.degats = 12
>>> perso_1.degats
12
```

Vous trouverez sur le web de nombreux exemples de code rédigés de cette manière, sans forcément savoir si les propriétés avancées de Python ont été utilisées. Nous utiliserons également ce type de code plus tard dans l'année, pour simplifier l'écriture des programmes.

Remarques :

- On peut créer des attributs de classe, qui seront identiques pour toutes les instances. Ces attributs ne sont pas dans le constructeur :

```
class Personnage:
    type = "joueur"
```

Si on ne met pas d'accesseur, on y accède soit à partir d'une instance, soit à partir de la classe :

```
>>> perso_1.type
'joueur'
>>> Personnage.type
'joueur'
```

A éviter absolument :

Bien évidemment, si vous essayez d'accéder à un attribut qui n'existe pas, cela ne fonctionne pas :

```
>>> perso_1.taille
Traceback (most recent call last):
  File "<ipython-input-5-5bf2379b8aff>", line 1, in <module>
    perso_1.taille
AttributeError: 'Personnage' object has no attribute 'taille'
```

Mais on peut créer un attribut à la volée, ce qui est *a priori* de la mauvaise programmation (je ne vois pas d'exemple où c'est indispensable, d'ailleurs à ma connaissance Python est le seul langage permettant ceci).

```
>>> perso_1.taille = 175
>>> perso_1.taille
175
```

5) Interaction entre deux objets.

Remarque préliminaire ; le fichier `classe_personnage.py` comprend notre classe, ainsi que l'import de `randint`.

Créons un deuxième personnage et faisons les se combattre avec le code suivant :

- On importe la classe dans notre programme principal ; en effet il est conseillé de faire un fichier par classe. On donne un alias plus court (perso).

```
import classe_personnage as perso
```

- On crée les personnages et on modifie leurs attributs

```
perso_1 = perso.Personnage("Un Seul Bras Les Tua Tous")
perso_1.setDegats(12)
perso_2 = perso.Personnage("Ventre de Fer")
perso_2.setPv(120)
```

- Bagarre en mode automatique.

Remarques :

- on utilise ici le paradigme impératif, on utilise deux paradigmes différents dans le même programme.
- Bien comprendre l'utilisation des méthodes, notamment Attaque()
- Le code n'est pas optimisé : on répète deux fois la même séquence d'instruction, d'où :
- Exercice : optimiser ce code

```
baston = True
while baston:
    degats = perso_1.Attaque()
    pv_perso_2 = perso_2.getPv() - degats
    if pv_perso_2 <= 0:
        print(perso_2.getNom(), " est au tapis !")
        baston = False
    else:
        perso_2.setPv(pv_perso_2)
        print(perso_2.getNom(), " a subi ", degats, " points de dégats
              et est à ", perso_2.getPv(), " points de vie")

    degats = perso_2.Attaque()
    pv_perso_1 = perso_1.getPv() - degats
    if pv_perso_1 <= 0:
        print(perso_1.getNom(), " est au tapis !")
        baston = False
    else:
        perso_1.setPv(pv_perso_1)
        print(perso_1.getNom(), " a subi ", degats, " points de dégats
              et est à ", perso_1.getPv(), " points de vie")
```

Remarques :

- Ne pas donner le même nom à une méthode et à un attribut dans une classe !
- Plusieurs classes peuvent avoir les mêmes noms de méthodes sans que cela soit problématique. En effet l'appel d'une méthode passe par objet.méthode() , ce qui permet de savoir dans quelle classe chercher la méthode. La classe définit son espace de noms.
- On peut définir des méthodes privées, avec la même convention que pour les variables privées (avec _ devant le nom). On ne devrait pas s'en servir cette année.
- On peut définir également des méthodes de classe. On ne met pas self dans les paramètres. Cette possibilité ne devrait pas nous plus nous être utile cette année.

Méthodes particulières

- `Personnage.__doc__` permet d'obtenir les spécifications de la classe
- `__repr__(self)` renvoie une chaîne de caractères, définie dans la méthode, et donnant la description de la classe lorsque qu'on demande un `print` de l'objet

```
def __repr__(self) :  
    return f'{self._nom} a {self._pv} points de vie,\ninflige  
{self._degats} points de dégats au plus, \net est en position  
{self._position}'
```
- `__lt__(self, autre_instance)` permet de faire une méthode de comparaison entre deux objets (`lt` = less than)

6) Héritage et polymorphisme.

Remarque préliminaire : ces notions ne sont pas au programme de terminale. Vous ne pouvez pas être interrogé au bac sur ce thème. Par contre, pour les projets, ces notions sont très utiles.

La notion d'héritage est au cœur de la programmation objet. Elle permet notamment d'utiliser sur des objets de type différent les mêmes méthodes, en les appelant par un même nom. L'effet dépendant du type précis de l'objet.

On souhaite créer un deuxième type de personnage, un guérisseur qui peut (se) soigner, mais rate parfois son attaque. Plutôt que de créer une nouvelle classe à partir de zéro, on va utiliser le principe d'héritage. La classe `Personnage` est la super classe, ou classe mère, la classe `Guérisseur` est la sous classe, ou classe fille.

On pourrait aussi créer une classe monstre, plus généraliste, avec les mêmes attributs et méthodes, et quelques variantes. Dans ce cas, il est inutile de créer deux classes différentes. On créera une sur-classe « bonhomme » et deux sous classes « personnage » et « monstre ». Par exemple, on pourrait ajouter l'attribut `nom` pour le personnage, et modifier la méthode de déplacement suivant la sous-classe. Pour le personnage, le déplacement serait guidé par les actions du joueur (clavier ou souris), tandis que pour les monstres, le déplacement serait automatique (aléatoire, dirigé vers le personnage, ou bien un mélange des deux). Il y a **héritage** de classe et **polymorphisme** (plusieurs formes)

- Notre classe `Guérisseur` hérite de `Personnage` avec la syntaxe qui suit. Cela signifie que tous les attributs et toutes les méthodes de `Personnage` se retrouvent dans `Guérisseur`. Dans cet exemple, le fichier `classe_personnage.py` est importé. On lui donne un alias (`perso`) que l'on utilise par la suite, comme dans `perso.Personnage`. Si les deux classes `Personnage` et `Guérisseur` sont dans le même fichier, on ne met pas d'alias, et on fait référence directement à `Personnage`.

```
import classe_personnage as perso  
class Guerisseur(perso.Personnage):  
    """  
    Personnage guérisseur d'un jeu de type hack 'n slash  
    Hérite de la classe personnage  
    Attributs :  
        _soins : montant maximum des points de vie soignés  
    Méthodes:  
        Init() : constructeur de la classe Guerisseur  
        Soigner() : renvoie le montant des points de vie guéris  
    """
```

- Le constructeur de la classe `Guérisseur` utilise celui de la classe `Personnage`, en :
 - Rajoutant éventuellement des attributs (ici `_soins`);
 - Modifiant éventuellement des attributs par rapport à la classe mère (ici `_pv = 60`).

```
def __init__(self, nom):  
    """  
    Constructeur de la classe Guerisseur  
    Données:  
        Attributs de la classe Personnage
```

```

        _soins : entier strictement positif,
                  montant maximum des points de vie soigné
Résultat :
    ne retourne rien, crée un nouveau Guerisseur
"""
perso.Personnage.__init__(self,nom)
self._pv = 60
self._soins = 8

```

- On peut rajouter des méthodes, comme ici l'accesseur à `_soins` et la méthode de guérison :

```

def getSoins(self):
    return self._nom

def Guerir(self):
    """
    Données: pas de paramètre dans cette méthode
    Résultat: renvoie un entier aléatoire compris
              entre 1 et _soins
    """
    return(randint(1,self._soins))

```

- On peut aussi redéfinir des méthodes (polymorphisme), ici l'attaque. Suivant les langages, on accède directement aux attributs de la classe mère ou bien par un accesseur : observer la différence dans la ligne `return` entre les méthodes `Guerir()` et `Attaque()`.

```

def Attaque(self):
    """
    Données: pas de paramètre dans cette méthode
    Résultat: renvoie un entier aléatoire compris
              entre 0 et _degats
    """
    if randint(1,4) == 0 :
        return 0
    else :
        return(randint(1,self.getDegats()))    #attention à l'accesseur

```

- Pour utiliser une méthode de la classe mère, on rajoute le mot clé `super()`.
- En Python, une classe peut hériter de plusieurs classes-mère. C'est à manier avec précaution, notamment lorsque les classes-mère ont des méthodes du même nom.