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

I 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 leurs 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 crée 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)

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 : chaine de caratères, nom du personnage
        _pv : entier positif ou nul, points de vie du personnage
        _degats : entier strictement positif, dégats 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 : chaine de caratères, nom du personnage
            _pv : entier positif ou nul, points de vie du personnage
            _degats : entier strictement positif, dégats 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.

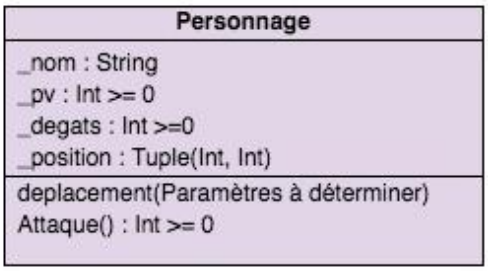
```
    #accesseurs des attributs
    def getNom(self):
        return self._nom
    def getPv(self):
        return self._pv
    def getDegats(self):
        return self._degats
    def getPosition(self):
        return self._position

    #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 :



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.

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)
```

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 des 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)