Paradigmes de programmation

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 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[1]. 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.

[1] Indispensable pour certains profs en sortant de certains cours, afin de sublimer.

Le « moule » avec lequel on va fabriquer un objet est appelé **classe**.
La classe personnage comprend par exemple les attributs :

- Points de vie
- Dégâts maximum 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**), il 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)
- *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 : 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.

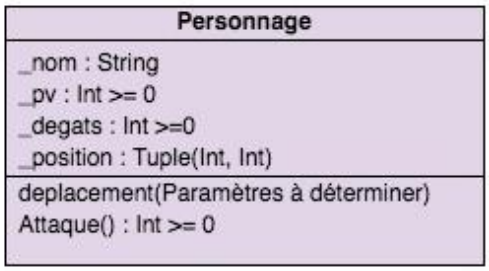
```
#accesseurs des attributs
def getNom(self):
    return self._nom
def getPv(self):
    return self._pv
def getDegats(self):
    return self._degats
def getPosition(self):
    return self._position

#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) <u>Création d'une instance et accès aux attributs.</u>

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 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
- `__str__(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. De même `__repr__(self)` fait la même chose quand on écrit le nom de l'objet directement dans la console. `__str__(self)` appelle `__repr__(self)` s'il n'est pas décrit explicitement.

```
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 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 méthode 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 Guerrisseur
      """
      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.

exercices programmation objet

Exercice 1 : des grandes boîtes et des petites boîtes et des moyennes boîtes…

1. Créer une classe Boite.
   Cette classe a pour attributs :
   - Longueur
   - Largeur
   - Hauteur
   - Ces trois attributs sont dans un ordre décroissant longueur ≥ largeur ≥ hauteur

   Elle a pour méthodes :
   - Volume, qui comme son nom l'indique donne le volume d'une boite
   - RentreDans (autre_boite), qui renvoie vrai si l'objet Boite rentre dans autre_boite.
2. Créer aléatoirement une liste d'une vingtaine de boîtes (on peut choisir des dimensions entre 1 et 50).
3. A l'aide d'un algorithme glouton, donner une suite de boîtes aussi grande que possible qui rentrent les unes dans les autres.
   Rappel : pour trier les boîtes, on fera appel à la fonction `sorted`.
   Syntaxe :

```
def cle_titre(ligne):
    """
    Renvoie la valeur du champ 'title' d'un enregistrement de la table
    """
    return ligne['title']

films_tries = sorted(mesDonnées, key = cle_particulière)
```

   On peut rajouter `reverse = True` pour avoir l'ordre décroissant.
4. Pour les révisions (algorithmes à connaître pour le bac), il peut être intéressant de créer une méthode __lt__(self) afin de comparer deux boîtes, puis d'utiliser cette méthode pour faire le tri préalable à l'algorithme glouton ci-dessus. On utilisera bien sûr un des algorithmes à connaître, à savoir le tri par insertion et/ou le tri par sélection.

Exercice 2 : une horloge.

Créer une classe horloge, puis la tester.
Attributs :

- heures
- minutes
- secondes

Méthodes :

- ticTac : cette méthode augmente l'horloge d'une seconde, et éventuellement sonne le réveil
- reveil(h , mn , s)
- __repr__

Exercice 3 : des chiens.

Créer une classe chien, puis la tester.
Attributs :

- nom
- points_de_santé
- aboiement : chaîne de caractères

Méthodes :

- mordre(autre_chien) : fait baisser les points de santé d'un autre chien
- manger : augmente les points de santé

- grogner : renvoie « Grrr… » + son aboiement
- machouiller(chaîne) : renvoie la chaîne mélangée
- __repr__

Exercice 4 : tableaux bidimensionnels (© livre Numérique et Sciences Informatiques, Balabonski – Conchon – Filliâtre – Nguyen , éditions Ellipses)

On souhaite construire des tableaux pouvant avoir des indices négatifs, c'est-à-dire que les indices du tableau vont de i_{min} à i_{max}, où $i_{min} \leq 0$ et $i_{max} \geq 0$. Le tableau vide correspond à $i_{min} = 0$ et $i_{max} = -1$. Les tableaux sont extensibles par la droite et par la gauche.

On va construire une classe TaBiDir pour cela

Attributs :

- tabG : le tableau contenant les éléments d'indice strictement négatif. L'élément d'indice -1 sera tabG[0], celui d'indice -2 sera tabG[-1] etc.
- tabD : le tableau de droite contenant les éléments d'indice positif ou nul

Méthodes :

- __init__(self, tabG , tabD) : le constructeur prend en paramètres les deux tableaux de gauche et de droite.
- iMin et iMax : renvoient l'indice minimal/maximal du tableau
- append(élément) : ajoute un élément à droite du tableau. Si le tableau est vide, le place à l'indice 0.
- prepend(élément) : ajoute un élément à gauche du tableau. . Si le tableau est vide, le place à l'indice -1.
- __getitem__(self, i) : encore une méthode particulière. Comme son nom l'indique, renvoie l'élément d'indice i du tableau. On peut traiter les exceptions si on le souhaite, cf. TP.
- __setitem__(self, i , valeur) : idem précédente sauf que l'on met l'élément d'indice i du tableau à valeur.
- __repr__

Exercice 5 : un peu d'écologie (scientifique).

On va modéliser le fonctionnement d'un ecosystème, avec de l'herbe, des moutons dans un premier temps, puis des loups dans un deuxième temps. Cette simulation est discrète : tous les « tours de jeu » (tous les ticks d'horloge), les moutons ainsi que les loups se déplacent, mangent éventuellement, meurent parfois, et l'herbe pousse.

1. Les moutons et l'herbe

   On va d'abord créer deux classes pour le monde et les moutons.

   Classe Monde. Cette classe est la carte sur laquelle évoluent les moutons. C'est une matrice carrée de `dimension` 50 (ou plus), qui représente une prairie. Chaque élément de la matrice représente un carré sur lequel pousse de l'herbe. Cette matrice est une matrice d'entiers. Si la valeur de l'entier est inférieure à l'entier `duree_repousse` il n'y a pas d'herbe, sinon il y en a. L'herbe repousse à une vitesse donnée (entier `duree_repousse` entre 1 et 100, on peut mettre 30). Le coefficient de la matrice est réinitialisé à 0 lorsqu'un mouton « occupe » la case et mange l'herbe. A chaque tick d'horloge on augmente l'entier de 1 (on verra la gestion du temps dans la classe Simulation).

   La classe Monde a comme attributs :

   - `dimension` ;
   - `duree_repousse` ;
   - `carte` ;

   et comme méthodes :

   - `herbePousse` ;
   - `herbeMangee(i,j)`, où `i` et `j` sont les indices du coefficient de la matrice carte ;
   - `nbHerbe` qui renvoie le nombre de carrés de la carte herbus ;
   - `getCoefCarte(i,j)`, où `i` et `j` sont les indices du coefficient de la matrice carte. C'est un getter de type particulier qui renvoie la valeur du coefficient $a_{i,j}$ de la carte ;

- plus les getters usuels.

On pourra initialiser les couples de carte avec 50% de carrés herbus, et pour ceux qui n'ont pas d'herbe on initialisera le coefficient de la matrice entre 0 et `durée_repousse` aléatoirement.

La classe Mouton a comme attributs

- `gain_nourriture` : le gain d'énergie apporté par la consommation d'un carré d'herbe (on peut mettre 4)
- `position`, couple d'entiers indice de la carte ;
- `energie`, entier positif (ou nul). Quand l'énergie d'un mouton est à 0, il meurt et l'objet est supprimé. Sera initialisé entre 1 et 2 x `gain_nourriture`.
- `taux_reproduction`, entier compris entre 1 et 100 (on peut mettre 4). Ce taux représente le pourcentage de chance qu'un mouton se reproduise (par parthénogénèse ici, ce qui risque de contrarier ceux d'entre vous qui ont des connaissances minimales en biologie).

Ses méthodes sont :

- `variationEnergie` : diminue de 1 l'énergie du mouton s'il n'est pas sur un carré d'herbe, sinon augment de `gain_nourriture`. Renvoie `energie`.
  Remarques :
  - si plusieurs moutons sont sur la même case herbue, seul le premier bénéficie du gain d'énergie
  - Réfléchissez bien aux paramètres de cette méthode
- `deplacement` : le mouton se déplace aléatoirement dans une des huit cases adjacentes. Il est plus facile de considérer le monde comme torique (comme dans les jeux vidéo, sortir par le haut de la carte renvoie en bas etc.), on utilise alors l'opérateur modulo (%). On peut accepter le fait que le mouton ne se déplace pas à tous les coups, cela simplifie la programmation.
- Il faut un setter pour la position (utile lors des naissances)

Enfin on crée une classe Simulation qui, comme son nom l'indique, gère la simulation.
Attributs :

- `nombre_moutons` : entier
- `horloge` : entier initialisé à 0
- `fin_du_monde` : entier donnant le temps maximum de la simulation
- `moutons` : liste d'objets Mouton (il y en a nombre_moutons…)
- `monde` : une instance de l'objet Monde
- `resultats_herbe` : liste construite petit à petit, donnant le nombre de carrés d'herbe à chaque tick d'horloge
- `resultats_moutons` : idem que la précédente, mais pour les moutons

Méthode unique, que l'on peut décomposer en plusieurs fonctions :

- `simMouton`: gère la simulation en créant une boucle qui
  - augmente `Horloge` de 1 à chaque appel ;
  - fait pousser l'herbe de monde ;
  - appelle `variationEnergie` pour chaque mouton. Si l'énergie d'un mouton est nulle, l'instance est supprimée (un « `remove` » dans la liste des moutons)
  - Fait se reproduire les moutons. Un nouveau mouton apparaît sur la même case que son parent, avec un pourcentage de naissance donné par `taux_reproduction`
  - Fait se déplacer les moutons.
  - Sauvegarde dans `resultats_herbe` et `resultats_moutons` les nombres de carrés herbus et de moutons du tour.
  - On peut arrêter la simulation s'il n'y a plus de moutons, ou s'il y en a plus qu'un nombre fixé (qu'on rajoutera en attribut). Dans ce dernier cas, les moutons ont conquis le monde… On l'arrête dans tous les cas lorsque l'horloge sonne la fin du monde.

    - Cette fonction renvoie les deux listes de résultats.
- Pour les tests, mettre `fin_du_monde` à 10. Réfléchissez aussi si à ce qui peut se passer en ce qui concerne le nombre de moutons d'une génération à la suivante. Les tests peuvent être longs dès que l'on dépasse 100 ticks d'horloge suivant votre ordinateur.

Vous pouvez utiliser le programme courbes_ecologie pour visualiser les résultats.

2. On rajoute les loups
   Les loups fonctionnent comme les moutons, sauf qu'ils ne mangent pas d'herbe mais plutôt des moutons… Un loup ne mange qu'un seul mouton par tour. On peut leur mettre un taux de reproduction de 5, et un gain sur la nourriture de 19. Si le gain est trop petit les loups meurent tous, s'il est trop grand les loups se multiplient trop, et avec les données de naissances la croissance devient exponentielle. Suivant votre programmation, il peut être nécessaire d'augmenter. Il est logique de mettre au moins deux fois moins de loups que de moutons. Avec ces données et les précédentes, on obtient un système qui se stabilise. Il est intéressant de trouver des valeurs initiales qui donnent un système pseudo-périodique (des sortes de sinusoïdes décalées), ou bien des systèmes chaotiques.

3. Dans les améliorations potentielles, on peut :
   - rajouter le fait que le loup se déplace vers un mouton s'il y en a un dans les 8 ou 15 cases adjacentes.
   - Faire se reproduire les animaux non en fonction d'un pourcentage, mais en fonction de l'énergie. On peut fixer un seuil, par exemple égal à `gain_nourriture` ´ 1,5 (au hasard ici), et diviser l'énergie (par 2) entre le parent et l'enfant.
   - Faire une classe abstraite Animal, mère de Mouton et de Loup.