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 IMPERATIVE

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.

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

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

- 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. Pour information, peu
                            utilisé en Python
        setAttributs(nouvelle_valeur) : mutateurs des attributs. Pour
                            information, peu utilisé en Python
                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). Les attributs peuvent être précédés d'un tiret bas « _ » pour signifier qu'ils sont privés, c'est-à-dire qu'on y accède en lecture ou en écriture en utilisant des méthodes spécifiques (accesseurs et mutateurs).

```
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)
```

- *Complément* : dans de nombreux langages (mais pas trop en Python), on utilise des accesseurs (getters en anglais) et mutateurs (setters en anglais)[2]. 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.

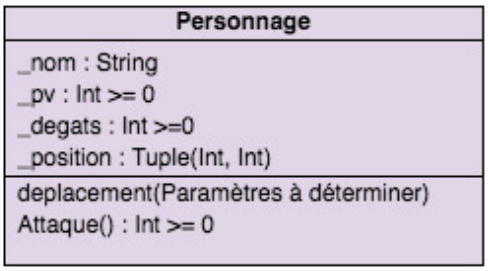
```
#accesseurs des attributs
def getNom(self):
    return self.nom
def getPv(self):
    return self.pv
def getDegats(self):
    return self.degats
def getPosition(self):
    return self.position

#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
```

[2] En Python, les accesseurs et mutateurs sont peu utilisés. A la place, on utilise les propriétés et décorateurs. Ce sont des outils très puissants mais spécifiques à Python. Ils ne sont pas forcément difficiles à comprendre, mais la logique derrière n'est pas évidente, et ces outils ne sont pas au programme de terminale. Les accesseurs et mutateurs sont relativement universels a contrario.

- 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) CREATION D'UNE INSTANCE ET ACCES 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 à un attribut ou le modifier, on code `objet"."attribut` comme suit :

```
>>> perso_1.nom
"Un Seul Bras Les Tua Tous"
>>> perso_1.pv = 100
```

Complément : en général, mais pas en Python :

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.
Par convention, on peut préciser qu'un attribut est privé en mettant un tiret bas « _ » devant son nom. Cela n'empêche pas de le modifier comme ci-dessus, mais le programmeur qui le fait sait qu'il est en faute.
Exemple : ici on demande à ce que le nom ne soit pas modifiable :

```
def __init__(self,nom):
    self._nom = nom
```

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_perso = "joueur"
```

On y accède soit à partir d'une instance, soit à partir de la classe :

```
>>> perso_1.type_perso
'joueur'
>>> Personnage.type_perso
'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
```

Voir aussi en complément avancé : au paragraphe 7, une astuce pour éviter la création de nouveaux attributs, ainsi qu'une modification encadrée de la même manière qu'avec un mutateur, à l'aide de la la méthode `__setattr__`.

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.degats = 12
perso_2 = perso.Personnage("Ventre de Fer")
perso_2.pv = 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.pv - degats
    if pv_perso_2 <= 0:
        print(perso_2.nom," est au tapis !")
        baston = False
    else:
        perso_2.pv = pv_perso_2
        print(perso_2.nom," a subi ",degats, " points de dégats
              et est à ",perso_2.pv," points de vie")

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

```
def __str__(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}'
```

- `__repr__(self)` fonctionne presque comme `__str__(self)`. Elle renvoie une chaîne de caractères, définie dans la méthode, et donnant la description de la classe lorsque l'on tape le nom de l'objet (sans `print`). Cette méthode remplace `__str__`, si cette dernière n'est pas définie.
- `__lt__(self , autre_instance)` permet de faire une méthode de comparaison entre deux objets (lt = less than)

6) HERITAGE 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 surper-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 la méthode de guérison :

```
    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.

```
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.degats))
```

- 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.

7) LA CLASSE PERSONNAGE AVEC UN MUTATEUR PYTHONESQUE

Remarque préliminaire : toujours pas au programme de terminale !
Rajouts dans le code par rapport à ce qui précède :

- création d'une liste avec les noms des attributs (qui sont des clés en fait), pour éviter la création de nouveaux attributs. C'est plus une astuce de codage qu'une méthode de programmation répandue.
- Dans __setattr__ :
  - vérification de l'existence des attributs, sinon refus de le créer/modifier ;
  - en cas de points de vie négatifs, ceux-ci sont ramenés à 0.

```
class Personnage:
    """
    ...
    """
    _cles = ["nom", "pv", "degats", "position"]
    def __init__(self,nom):
        """
        ...
        """
    ...
    def __setattr__(self, cle, valeur):
        if cle in self._cles:
            if cle == "pv" and valeur < 0 :
                valeur = 0
            object.__setattr__(self, cle, valeur)
        else:
            print("Il est interdit d'ajouter un nouvel attribut")
```

III PROGRAMMATION FONCTIONNELLE

1) PRINCIPES

La programmation fonctionnelle est un paradigme de programmation de type déclaratif, c'est-à-dire décrivant le « quoi », c'est-à-dire le résultat final à obtenir, plutôt que le « comment », la manière d'y parvenir (cette dernière méthode relevant d'un paradigme impératif). HTML, SQL sont des langages déclaratifs. Pour donner une image, imaginons que l'on se retrouve face à un meuble d'une célèbre marque nordique à monter. L'utilisation d'un paradigme impératif est alors la notice tout aussi

fameuse où l'on finit par monter un meuble qui ne ressemble en rien au résultat escompté, tandis que le paradigme déclaratif donnerait juste la photographie du meuble monté.
La programmation fonctionnelle considère un programme comme l'exécution de fonctions mathématiques. L'analogue informatique des fonctions mathématiques est la fonction pure : le résultat ne dépend que des paramètres d'entrée.
Conséquences :

- Il n'y a pas de « variation d'état » ni de mutation des objets, comme il peut y en avoir lorsque l'on fait une affectation du type `n = n + 1` ou `tableau.append(truc)`. Il n'y a pas d'effet de bord.
- Le code est une succession d'appels de fonctions.
- Il y a transparence référentielle : une fonction mathématique renvoie le même résultat si l'on rentre les mêmes données, ce qui n'est pas le cas par exemple si l'on utilise des variables globales dans la fonction.
- Les boucles n'existent pas, elles sont remplacées par des appels récursifs (qui mathématiquement correspondent à la composition d'une fonction par elle-même).
- Les fonctions sont des données comme les autres : une fonction peut être passée en argument dans une fonction, ou bien peut être le résultat d'une fonction.

Avantages et inconvénients.

- Sans effet de bord, le code est facile à maintenir et à tester
- Le calcul parallèle est simplifié
- Le code est concis, avec une lecture assez simple… lorsqu'on y est habitué. Attention concision ≠ lisibilité.
- Se combine bien avec l'objet ou l'impératif dans une programmation « impure ».
- o La récursivité est limitée par la taille de la pile d'appels récursifs.
- o La récursivité est souvent plus difficile à comprendre/maîtriser que l'itérativité (beau barbarisme).
- o Non employable pour les bases de données ou serveurs.
- o Données non modifiables.
- o Les fonctions mathématiques permettent de faire la même chose que la programmation itérative, mais parfois c'est bien plus compliqué que d'utiliser l'impératif.

2) QUELQUES INSTRUCTIONS TYPIQUES

A/ LES FONCTIONS ANONYMES

Les fonctions anonymes, comme leur nom l'indique, n'ont pas de nom. En Python, elles sont basées sur le λ-calcul, inventé par Alonzo Church dans les années 1930.
Une fonction λ, ou expression λ, s'écrit sous la forme :

```
f = lambda x : 2*x + 1
```
f est la fonction affine $x \mapsto 2x+1$

`f(3)` renvoie `7`

Remarque : ici on a donné un nom à la fonction… elle n'est donc pas anonyme !
On peut utiliser une fonction λαμβδα sans lui donner de nom :

`(lambda x : 2*x +1)(3)` renvoie `7`

Ou plus utile, et très le style fonctionnel :

```
def calculer_fct_sur_tableau(fonction, tableau) :
    nouveau_tableau = []  # on ne modifie pas le tableau passé en paramètre
    for élément in tableau :
        nouveau_tableau.append(fonction(élément))
    return nouveau_tableau

tableau = [i for i in range(42)]
new_tab = calculer_fct_sur_tableau(lambda x:x**2, tableau)
# renvoie un tableau avec tous les éléments mis au carré
```

On peut mettre une condition si…sinon dans une fonction λ :

```
# Vérification qu'un nombre entier est un carré parfait
from math import sqrt
carre_parfait = lambda a : True if int(sqrt(a))**2 == a else False
```

B/ MAP

La fonction `map(fonction, itérable)` applique une fonction sur tous les éléments d'un tableau, d'un dictionnaire, ou de tout objet itérable (cf. ci après « pour aller plus loin » pour ceux qui veulent créer leurs propres itérables). La fonction peut être définie avec « def » ou par une expression lambda.

Exemple :

`map(lambda x:x**2, [i for i in range(42)])` renvoie un itérateur (cf. ci-dessous, considérez pour l'instant que c'est une sorte de tableau) avec tous les éléments mis au carré.

Pour obtenir la liste des résultats, on transforme avec list :

```
tab_au_carré = liste(map(lambda x:x**2, [i for i in range(42)]))
```

En effet l'objet `map` est un objet de type itérateur (cf. ci-après « pour aller plus loin »)

C/ REDUCE

La fonction `reduce(fonction, itérable)` est un peu plus compliquée à comprendre. Elle calcule un résultat unique à partir de tous les éléments d'un itérable. Son fonctionnement est le suivant :

- Le résultat de la fonction est calculé sur les deux premiers éléments ;
- Puis la fonction est appliqué sur ce résultat et le 3ème élément ;
- Ainsi de suite jusqu'au dernier élément de l'itérable.
- Le résultat final est renvoyé

Exemple :

```
# calcul de la somme des éléments d'une liste
from functools import reduce
somme_elements_liste = reduce(lambda a, b : a + b, liste)
```

D/ FILTER

`filter(fonction, itérable)` filtre les résultats d'un itérable suivant la fonction booléenne (qui renvoie `true` ou `False` forcément). `filter` renvoie un itérable, donc comme pour `map` on rajoute `list` pour obtenir un tableau dynamique Python.

Exemple :

```
# Obtenir les nombres pairs dans une liste
filter(lambda x : x%2 == 0, [i for i in range(10)])
```

ou, en plus compliqué, un exemple avec un `if` :

```
list(filter(lambda x : True if x%2 == 0 else False,
             [i for i in range(10)]))
```

E/ POUR ALLER PLUS LOIN : ITERATEURS ET GENERATEURS

Les itérateurs sont des objets permettant de représenter des flux de données. Un objet `it` de type `iterator` (`list_iterator`, `dict_keyiterator` …) dispose d'une fonction `next(it)` et d'une méthode `it__next__()`. Les itérateurs ne font qu'aller de l'avant, une fois un élément « consommé » par l'appel de `next()`, cet élément a définitivement disparu du flux.

Exemple : testez le code suivant

```
it = iter([i for i in range(10)])
type(it)
while True:
      next(it)
```

ainsi que :

```
itbis =iter({i: i + 1 for i in range(10)})
type(itbis)
```

```
while True:
      itbis.__next__()
```

Puis relancez un des `while True`

On peut créer un itérateur maison avec une classe objet. L'itérateur dispose de deux méthodes qui lui sont propres, à savoir `__iter()__` et `__next()__`. La méthode `__iter()__` initialise l'itérateur.

Exemple :

```
class MyIteratorDeCarre():
    def __init__(self):
        self.i = 0
        self.c = 0
    def __iter__(self):
        print("On a appelé __iter__")
        return self
    def __next__(self):
        print("On a appelé __next__")
        self.c += 2 * self.i + 1  # une jolie méthode pour calculer
                                  # les carrés
        #self.c = self.i**2       #à la place de celle à cette ligne
        self.i += 1
        if self.c > 100:
            raise StopIteration()
        return self.c


for e in MyIteratorDeCarre():
    print(e)
```

Les générateurs sont des fonctions qui retournent des flux de valeurs, non pas avec le mot clé `return`, mais avec le mot clé `yeld`, qui fait continuer la fonction, en renvoyant les item générés au fur et à mesure. De manière semblable à la fonction `map`, un générateur est un objet de type `generator`, que l'on peut transformer en liste.

Exemple : le générateur suivant renvoie les nombres dont le carré commence par 3

```
from math import log
def carresCommencantPar3(maxi):
    for i in range(1,maxi):
        c=i**2
        if c//10**int(log(c,10)) == 3 :
            yield i

print(type(carrésCommencantPar3(100)))
nombres = list(carrésCommencantPar3(100))
print(nombres)
print(list(map(lambda x : x**2, nombres)))
```

Remarque : itérateurs et générateurs sont des outils de programmation fonctionnelle, dans le sens où les données ne sont pas stockées en mémoire, mais traitées au fur et à mesure que leur flux est crée. Il est notamment possible de créer des itérateurs/générateurs infinis, ou du moins très grands, sans encombrer la mémoire.

EXERCICES PROGRAMMATION OBJET

EXERCICE I : 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 II : 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)

EXERCICE III : 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

EXERCICE IV : 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 $i_{max} \geq -1$. 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
- iMin et iMax : renvoient l'indice minimal/maximal du tableau

Méthodes :

- __init__(self, tabG , tabD) : le constructeur prend en paramètres les deux tableaux de gauche et de droite.
- 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 V : UN PEU D'ECOLOGIE SCIENTIFIQUE

On va modéliser le fonctionnement d'un écosystème, 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.

Il est fourni un fichier Python permettant de visualiser les courbes de population. Le code est recopié en fin d'exercice, il est très court.

1. Les moutons et l'herbe

   On va d'abord créer deux classes pour le monde et les moutons. Cette classe sera dans un fichier appelé par exemple classe_monde_mouton.py .

   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 :

   - Le constructeur peut avoir un paramètre facultatif `dimension = 50` ;
   - `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 ;

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`, liste de deux entiers x et y (ne pas mettre un couple, car un tuple n'est pas modifiable) ;
- `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 $2 \times$ `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 :

- Le constructeur a comme paramètre `dimension`, la dimension du monde, afin de placer correctement les moutons ;
- `variationEnergie` : diminue de 1 l'énergie du mouton s'il n'est pas sur un carré d'herbe, sinon augmente 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.

Enfin on crée une classe Simulation qui, comme son nom l'indique, gère la simulation. Cette classe est dans un autre fichier, par exemple simulation.py.
On y importe les classes Monde et Mouton : `import classe_monde_mouton_loup as mml` (ici en plus la classe Loup). On utilisera les classes Monde et Mouton de la même manière que l'on utilise une bibliothèque, par exemple pour créer un mouton on fera : beeh = mml.Mouton(dim)
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 (il n'est pas interdit de la décomposer en plusieurs fonctions, loin de là) :

- `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`. Un mouton qui se reproduit a son énergie divisée par 2. on peut mettre la naissance dans une des 9

cases du voisinage du mouton (modulo la dimension de la carte), y compris la case où est le mouton reproducteur.
        - 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 `resultats_herbe` et `resultats_moutons` , `resultats_loups` si les loups ont été rajoutés.
    - 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.

Il est fortement conseillé d'utiliser le programme `courbes_ecologie` pour visualiser les résultats (sous Spyder ou EduPython). Vous verrez dans ce programme que le fichier avec la classe simulation s'appelle `simulation_ecologie.py` (en ligne 13), importé « `as simul` » et qu'on y appelle la méthode `simMouton()` (en ligne 18). Si vous avez fait la version sans les loups, il faut :

- décommenter la ligne 17 ;
- mettre en commentaire les lignes 18, 33 et 36 (tout ce qui concerne Y3, soit les loups).

Il est aussi fourni un programme donnant une interface avec animation –en théorie–. Ce programme est sur le site, mais il n'est pas du tout fini. Pour l'instant seul fonctionne le tour par tour, et seule une « photographie » du monde est donnée à chaque étape, sans animation intermédiaire. Pour utiliser ce programme, il faut créer dans la classe `Simulation` une méthode supplémentaire : `simMoutonGUITour()`. Cette méthode n'exécute qu'un tick d'horloge, et renvoie les attributs `monde`, `moutons`, `loups` et `horloge`.

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 4 ou 5, et un gain sur la nourriture de 19 ou 20. 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 : un certain seuil doit être atteint et il y a automatiquement reproduction.
   - Faire une classe abstraite Animal, mère de Mouton et de Loup.

4. Code pour le tracé des courbes.

```
# -*- coding: utf-8 -*-
#REMPLACEZ si ncéssaire le nom du fichier à importer(simulation_ecologie)
#REMPLACEZ si nécessaire le nom de la méthode "sim.simMouton" pour la simulation

import numpy as np
import matplotlib.pyplot as plt
import simulation_ecologie as simul

#Pour un test sans les loups, supprimez Y3
sim = simul.Simulation()
#Y1,Y2 = sim.simMouton()
Y1,Y2 , Y3 = sim.simMouton()
```

```
#print("Herbe : ", Y1)          # si on veut voir les listes
#print("Moutons : ", Y2)
#print("Loups : ", Y3)
for i in range(len(Y1)):
    Y1[i] = Y1[i]/4

X = np.linspace(0, len(Y1), len(Y1), endpoint=True)

plt.plot(X,Y1, 'g')
plt.plot(X,Y2 , 'b')
plt.plot(X,Y3 , 'k')
plt.plot(X, Y1, color="green", linewidth=2.5, linestyle="-", label="Herbe/4")
plt.plot(X, Y2, color="blue",  linewidth=2.5, linestyle="-", label="Moutons")
plt.plot(X, Y3, color="black",  linewidth=2.5, linestyle="-", label="Loups")
plt.legend(loc='upper left', frameon=True)
plt.show()
```

EXERCICES PROGRAMMATION FONCTIONNELLE

Bien sûr, tous les exercices présentés ci-après doivent être faits suivant les principes de la programmation fonctionnelle. On utilisera autant que possible les fonctions `map()`, `reduce()`, `filter()`.

EXERCICE I : LES NOTES EN TERMINALE NSI / PATISSERIE

1. On se donne une liste de résultats scolaires sous la forme de tuples `(Nom, Prénom, matière, note)`. Donner les élèves ayant la moyenne en NSI.
2. Le prof de spécialité NSI trouve qu'il a noté trop large, il décide d'enlever 3 points à chaque élève. Par contre, le prof de spécialité Pâtisserie augmente les moyennes de 10 %. Transformer les notes précédentes, tout en restant dans le système de notation usuel.
3. Donner la moyenne des élèves en NSI.

EXERCICE II : AVEC DES CHAINES

Pensez aux méthodes sur les chaînes `split()`, `upper()` *etc.*

1. Initiales : Écrire un programme qui affiche la première lettre de chacun des mots d'une phrase en majuscule. Les mots peuvent être séparés par une ou plusieurs espaces. Par exemple : `"Rentre avec tes pieds"` donnera `RATP`. Celui-ci est sympathique également `"Bientôt au chômage"`.
2. Doublons : Un utilisateur entre une phrase en minuscules et sans accent, par exemple : `"cette personne babille et grelotte"` et en sortie on affiche, pour chaque mot, les lettres où il y a eu un doublement. Pour l'exemple, on obtiendra donc les lettres `tnltt`.
3. Chasse au trésor : Au pied d'un arbre, dans un petit coffre enterré, vous trouvez un parchemin avec le texte suivant : `"Pirate, pour arriver au trésor, tu devras faire 10 pas dans chacune de ces directions : ONNESEE"` Écrire un programme qui donne, par rapport à l'arbre, l'endroit où il faut creuser.
4. Url : on donne une liste d'url pas forcément bien formées, systématiquement en protocole http et avec www. Renvoyer les url correctes.
   Exemple :
   ```
   urls = ['http://www.maths-info-lycee', 'http://www.wikipedia.fr',
         'http:www.orange.fr',   'inexistant.com',   'httpz://www.monsite.fr',
         'http://www.pixees.fr']
   ```
   renvoie
   `'http://www.wikipedia.fr' et 'http://www.pixees.fr'`

EXERCICE III : CONSTRUCTION DE $\mathbb{Z}$ A PARTIR DE $\mathbb{N}$

On se donne une liste représentant les entiers naturels jusqu'à un certain rang [0, 1, 2, 3, 4, 5, 6, …]
Construire la liste qui aux entiers pairs associe leur moitié, et aux entiers impairs l'opposé de la moitié de ces nombres + 1. Essayez de trouver plusieurs solutions pour cet exercice : en utilisant map(), avec un itérateur, avec un générateur. Question : est-ce possible avec une liste par compréhension ?
Exemple : la liste [0, 1, 2, 3, 4, 5, 6] donne [0, -1, 1, -2, 2, -3, 3]
Remarque : cet exercice montre qu'il y a autant de nombres dans $\mathbb{N}$ que dans $\mathbb{Z}$. 😲 🤔
Or on sait que $\mathbb{N} \subset \mathbb{Z}$ et $\mathbb{Z} \not\subset \mathbb{N}$, ce qui pourrait laisser croire que $\mathbb{Z}$ contient plus d'éléments que $\mathbb{N}$: l'intuition est facilement fausse quand on s'attaque aux nombres infinis !

EXERCICE IV : ITERATEURS ET GENERATEURS

Créer un itérateur et un générateur, a priori infinis, pour :

1. la suite de Fibonacci : $u_0 = 1, \quad u_1 = 1, \quad u_{n+2} = u_{n+1} + u_n$
2. la suite de Collatz :
$$\begin{cases} u_0 = n \text{ donné} \\ u_{n+1} = u_n / 2 \text{ si } u_n \text{ est pair} \\ u_{n+1} = 3u_n + 1 \text{ si } u_n \text{ est impair} \end{cases}$$

   On s'arrêtera lorsque l'on obtiendra $u_n = 1$. On pourra également calculer le « temps de vol », c'est-à-dire la valeur de n correspondante.

EXERCICES DU LIVRE

N° 36, 37, 42, 43, 44 p 100