Tutorat de Programmation
L1 Informatique

Tuto 6

SNAKE

Table of Contents

I – Introduction

Le but de ce tuto est de vous faire réaliser un programme beaucoup plus gros que ce que vous avez pu faire précédemment : il s'agit d'un jeu « Snake » Lien_Wiki tel qu'il était implémenté sur les premiers téléphones portables Nokia.

Pour pouvoir faire ce tuto dans de bonnes conditions, il vous est recommandé de travailler sous Linux. Pour ceux qui n'ont pas encore installé Linux, des instructions sont disponibles sur le forum du cours pour l'installation dans une machine virtuelle. Lien_Forum

Certaines fonctionnalités utilisées par le code qui vous est fournis sont spécifiques aux terminaux Linux. Néanmoins, vous pouvez faire une grande partie du projet sous Windows et les fonctions spécifiques à Linux seront indiquées dans la suite du document.

II – Consignes de Rendu

1) – Rapport

Vous devez rendre un rapport expliquant votre code, les problèmes que vous avez rencontrés et les choix d'implémentation effectués. Pour cela, vous pouvez compléter le fichier README.md (au format Markdown).

Ce rapport nous permet de vérifier que vous avez bien compris le code que vous nous rendez, il s'agit donc non pas d'expliquer chaque ligne de votre programme, mais plutôt l'idée des différentes parties du programme ainsi que la méthodologie utilisée.

C'est aussi l'occasion d'indiquer les problèmes et les erreurs que vous n'avez peut-être pas réussi à résoudre, et d'expliquer ce que vous avez tenté comme solution.

2) – Format

Vous devez rendre une archive **ZIP**. (Pas d'autre format) Le nom de l'archive doit être : Prenom_Nom_L1_tuto6.zip (Sans accents ni espaces), et elle doit contenir un dossier nommé Prenom_Nom_L1_tuto6 (Sans accents ni espaces) contenant tous les fichiers sources que vous aurez complété.

Pour guider votre développement, un squelette de code vous est fournis avec ce document. L'essentiel du code doit être fait dans le fichier **snake.py**. Vous devez directement modifier ce fichier, et implémenter toutes les fonctions dont les prototypes vous sont fournis à l'intérieur.

Pour chaque fonction, le prototype contient une chaîne de documentation qui comprends des séries de tests. Vous devez utiliser ces tests pour vérifier que vous avez correctement implémenté chaque fonction. Vous ne devez pas modifier ces chaînes de documentation sauf si cela vous est explicitement demandé.

```
def gen_grid(nRows, nCols):
    ''' (int, int) -> list of list of chars {{grid}}

    Generates and returns a grid of nRows by nCols.
```

```
>>> gen_grid(2, 3)
[[' ', ' ', ' '], [' ', ' ', ' ']]
>>> gen_grid(1, 4)
[[' ', ' ', ' ', ' ']]
>>> gen_grid(3, 4)
[[' ', ' ', ' ', ' '], [' ', ' ', ' ', ' '], [' ', ' ', ' ', ' ']]
'''
```

Pour vous faciliter la vie et produire une vue plus restreinte des tests qui ne sont pas validés, il vous est recommandé de lancer ces tests en exécutant le fichier fourni **runtests.py**. Le code à l'intérieur de ce fichier permet d'arrêter le déroulement des tests dès qu'un test n'est pas validé.

2) – Style

Respectez au minimum les règles de Style décrites dans le tuto3 – Lien_Tuto3

3 points sur **20** seront attribués au respect des règles de style.

Il est indispensable que vous configuriez votre éditeur de texte pour utiliser **4 espaces** par niveau d'indentation (**Pas de tabulations**) les fichiers fournis utilisent tous cette norme et si vous n'avez pas correctement configuré votre éditeur cela risque de produire des erreurs que vous aurez du mal à comprendre. Voir comment faire ici : Lien_Forum

2 points de **Bonus** seront attribués si vous écrivez tout votre code en **Anglais**. (noms de variables, commentaires)

L'Anglais est la langue utilisée en développement au niveau international.

III – Concepts Nécessaires

Pour réaliser ce programme, vous allez avoir besoin d'un nombre de concepts que vous avez vu en TP/TD/CM :

- Expressions
- Control Statements
- Loops
- Functions
- Code Reuse
- Strings Manipulation
- Lists, List Methods
- Modules

Comme pour le dernier TP, une grande partie de ce programme tourne autours de listes à deux

dimensions (listes de listes)

IV – Description des objets

On va utiliser différents objets pour représenter les concepts nécessaires pour le jeu :

1) – Grid

Cet objet va nous permettre de modéliser la surface de l'aire de jeu.
Il s'agit d'une zone rectangulaire dont chaque case est positionnée selon des coordonnées (x, y)

```
>>> grid = [['8', ' ', ' '], [' ', ' ', '*']] ; display_grid(grid)
+ + + + +
+ 8     +
+     * +
+ + + + +
```

Ici, vous voyez en exemple une grille qui possède deux lignes et trois colonnes. Le caractère '`8`' a pour coordonnées (0, 0) (Les caractères '`+`' sont utilisés pour délimiter la grille lors de l'affichage, mais ils n'en font pas partie.

La coordonnée x d'un point augmente vers la droite, et la coordonnée y augmente vers le bas.

Le caractère '`*`' a pour coordonnées (x, y) == (2, 1) (Colonne 3, ligne 2).

2) – Snake

Un Snake est représenté par une liste dont les éléments représentent des propriétés du serpent.

```
>>> snake = [[1, 2], [0, 1], 'red', True, 1]
```

Ceci est la représentation explicite 'littéraire' d'un Snake.

- `snake[0]` est une liste de deux entiers qui représente les coordonnées de la tête du Snake sur une grille. (Ici : Colonne 2, Ligne 3)

- `snake[1]` est une liste de deux entiers qui représente le vecteur vitesse de la tête du Snake, autrement dit la direction dans laquelle la tête du Snake se déplace.
(Ici : (vx, vy) == (0, 1) donc la coordonnée x de la tête ne change pas, mais la coordonnée y augmente de 1 unité. → Le Snake se déplace verticalement et vers le bas.)

- `snake[2]` est là pour permettre de différentier un Snake d'un autre mais vous n'avez pas besoin de vous en soucier dans la première partie de ce projet.

- `snake[3]` est un Booléen qui représente l'état (Vivant / Mort) du Snake

- `snake[4]` représente la taille (et donc le score) du Snake.

V – Fonctions Fondamentales

Vous trouverez ci-après une description des fonctions que l'on vous demande d'implémenter et qui ensemble, définissent la structure fondamentale du programme de jeu.

1) – genGrid

```
def genGrid(nRows, nCols):
    ''' (int, int) -> list of list of chars {{grid}}

    Generates and returns a grid of nRows by nCols.

    >>> genGrid(2, 3)
    [[' ', ' ', ' '], [' ', ' ', ' ']]
    '''
```

Cette fonction sert à générer la grille du jeu. Les deux paramètres (nRows, nCols) permettent de spécifier la taille de la grille. Ils sont respectivement le nombre de lignes et le nombre de colonnes.
La grille est une liste à deux dimensions dont les éléments des listes interne sont des chaînes de caractères de longueur 1. (Vous devez initialiser la grille avec des espaces blanc)
La fonction doit renvoyer la grille qui a été générée (`return`)

Attention : les listes internes ne doivent pas être des alias les unes des autres sinon des problèmes vont apparaître comme dans l'exemple suivant :

```
>>> l1 = [0, 0, 0]
>>> alias = l1
>>> alias[1] = 4
>>> l1
[0, 4, 0]
```

2) – displayGrid

```
def displayGrid(grid):
    ''' (grid) -> NoneType

    Display the grid to standard output. The grid must be surrounded by '+'
    symbols.

    >>> s = [['8', ' ', ' '], [' ', ' ', '*']] ; displayGrid(s)
    + + + + +
    + 8     +
    +     * +
    + + + + +
    '''
```

Cette fonction est la fonction d'affichage (c'est la seule fonction du programme qui doit contenir une expression `print`.)
Pour aider à visualiser l'aire de jeu, la grille doit être entourée par des symboles '+' des 4 côtés.

3) – pointInsideGrid

```
def pointInsideGrid(grid, point):
    ''' (grid, (int, int)) -> Boolean

    Check if the point is somewhere inside the grid.

    >>> pointInsideGrid(genGrid(4, 5), (4, 5))
    False
    >>> pointInsideGrid(genGrid(6, 5), (4, 5))
    True
    '''
```

Cette fonction est là pour vérifier si un jeu de coordonnées représentant un point tombent à l'intérieur d'une grille. Le point est représenté par un tuple (x, y) de deux valeurs entières.
Gardez à l'esprit que l'indexing d'une liste en Python est toujours basé sur 0.

4) - placeObject

```
def placeObject(grid, x, y, o):
    ''' (grid, int, int, char) -> NoneType

    Place a character (the object o) at the given coordinates (x, y) on the
    grid. You can assume that (x, y) are valid coordinates inside the grid.

    >>> g = [[' ', ' '], [' ', ' ']] ; placeObject(g, 1, 0, 'o') ; g
    [[' ', 'o'], [' ', ' ']]
    '''
```

Cette fonction est utilisée pour changer la chaîne de caractères contenu dans une case de la grille. Comme précédemment les coordonnées (x, y) où insérer la nouvelle chaîne (o) sont passées en paramètres mais cette fois il s'agit de deux paramètres (les coordonnées ne sont pas regroupées dans un tuple)

5) – dropBlock et dropFood

```
def dropBlock(grid, x, y):
    ''' (grid, int, int) -> NoneType

    Place a roadblock type object on the grid at the given (x, y) coordinates.
    A roadblock is identified by the character "#".

    >>> l = [[' ', ' '], [' ', ' ']] ; dropBlock(l, 1, 0) ; l
    [[' ', '#'], [' ', ' ']]
    '''
```

Ces deux fonctions sont essentiellement des raccourcis pour placer certains objets, ils doivent réutiliser la fonction **placeObject**.

6) - genSnake

```
def gen_snake(position, speed, color):
    ''' (tuple, tuple, str) -> [[int, int], [int, int], str, Boolean, int]
                                                    {{snake}}

    Generates a new snake.
    A snake is represented by a list of 5 Values:

    - A list [x, y] representing the current position of the snake's head.
    - A list [vx, vy] representing the speed vector of the snake's head.
    - A string identifying the snake by a color.
    - A Boolean that is True if the snake is alive and False otherwise.
    - An integer representing the size of the snake (starts at 1)

    >>> gen_snake((1, 1), (1, 0), 'red')
    [[1, 1], [1, 0], 'red', True, 1]
    '''
```

Génère et retourne un nouveau snake correspondant aux paramètres. Vous pouvez transformer un tuple en liste avec un appel de la fonction list() :

```
>>> l = list((3, 4))
>>> l
[3, 4]
```

7) - isAlive et killSnake

```
>>> isAlive([[0, 0], [1, 0], 'red', True, 1])
True
>>> isAlive([[0, 0], [1, 6], 'green', False, 1])
False
>>> s = gen_snake((0, 0), (1, 0), 'red') ; killSnake(s) ; s
[[0, 0], [1, 0], 'red', False, 1]
```

isAlive vérifie si un snake est vivant et renvoie True si c'est le cas, False sinon. Pour rappel, c'est le 4eme élément dans la liste représentant un snake qui dit si le snake est vivant ou pas.

killSnake met à jours le status d'un snake passé en argument, en remplaçant la valeur du 4eme élément dans la liste par '`False`'.

8) - getNextPosition

```
def getNextPosition(snake):
    ''' (snake) -> (int, int)

    Returns the next position the (head of the) snake will move to.

    >>> getNextPosition([[0, 0], [1, 0], 'red', True, 1])
    (1, 0)
    '''
```

Cette fonction sert à calculer la prochaine position d'un snake passé en argument en fonction de sa position actuelle et de son vecteur vitesse.

Pour calculer la nouvelle position il suffit d'ajouter le vecteur vitesse à la position. Dans l'exemple la postion est [0, 0] et le vecteur [1, 0] donc le résultat est la position (1, 0) ce qui correspondra à un déplacement d'une case vers la droite.

Attention cette fonction est uniquement indicative : elle ne doit pas faire bouger (modifier) le snake, juste renvoyer sa prochaine position.

Notez également que cette fonction est indépendante d'une grille.

9) – turnRight et turnLeft

```
def turnRight(snake):
    ''' (snake) -> NoneType

    Change the speed vector of the snake to make it turn right.

    >>> s = gen_snake((0, 0), (1, 0), 'red') ; s
    [[0, 0], [1, 0], 'red', True, 1]
    >>> turnRight(s) ; s
    [[0, 0], [0, 1], 'red', True, 1]
    '''
```

Ces fonctions servent à faire tourner le serpent. 'Faire tourner' veut dire seulement changer le vecteur vitesse.
Dans l'exemple le snake commence avec un vecteur (1, 0) (il se déplace vers la droite) et après avoir tourné à droite se retrouve avec un vecteur (0, 1) (il se déplace vers le bas)

Ces fonctions doivent changer le vecteur vitesse dans le snake et ne doivent rien renvoyer.

10) – updateSnake

```
def updateSnake(grid, snake):
    ''' (grid, snake) -> NoneType

    Update the given snake, relative to the grid.
        - If the snake steps over one side of the grid, its position switches
          to the other side of the grid.
        - If the snake's next position on the grid is occupied by a roadblock
          or a snake, the snake should be killed.
        - Else, Update the snake's position.
        - If the new position has snakefood, increase size element of the snake

    >>> s = genSnake((1, 0), (-1, 0), 'red') ; g = genGrid(3, 5) ; s
    [[1, 0], [-1, 0], 'red', True, 1]
    >>> updateSnake(g, s) ; s
    [[0, 0], [-1, 0], 'red', True, 1]
    >>> updateSnake(g, s) ; s #3
    [[4, 0], [-1, 0], 'red', True, 1]
    >>> updateSnake(g, s) ; s
    [[3, 0], [-1, 0], 'red', True, 1]
    >>> dropFood(g, 2, 0) ; updateSnake(g, s) ; s
    [[2, 0], [-1, 0], 'red', True, 2]
    >>> dropBlock(g, 1, 0) ; updateSnake(g, s) ; s
    [[2, 0], [-1, 0], 'red', False, 2]
    '''
```

Avec cette fonction, on attaque la partie difficile.

Cette fonction doit modifier le snake passé en argument en fonction de la grille passée en argument et de l'état interne du snake.

- Attention, cette fonction ne doit pas modifier la grille.
- Cette fonction doit mettre à jours la position du snake en fonction de son vecteur vitesse
- Si la prochaine position du snake sort d'un côté de la grille, il doit ré-apparaitre de l'autre côté. Dans l'exemple #3 au dessus, le snake a disparu du côté gauche de la grille pour réapparaître du coté droit.
- Si la prochaine position du snake sur la grille contient un caractère 'roadblock' (#) le snake doit être tué avec la fonction killSnake et ne pas bouger
- Si la prochaine position du snake sur la grille contient un caractère 'snakefood' (o) la taille du snake doit augmenter.

11) – updateGrid

```
def update_grid(grid, snake):
    ''' (grid, snake) -> NoneType

    Update the given grid, relative to the snake.
        - Wipe any snake symbol off the grid.
        - If the snake is not dead, repaint it on the grid

    >>> s = gen_snake((2, 2), (-1, 0), 'red') ; g = gen_grid(3, 3)
    >>> placeObject(g, 2, 2, '*') ; display_grid(g)
    + + + + +
    +       +
    +       +
    +     * +
    + + + + +
    >>> update_snake(g, s) ; update_grid(g, s) ; display_grid(g)
    + + + + +
    +       +
    +       +
    +   *   +
    + + + + +
    '''
```

Cette fonction va servir de complément à la fonction updateSnake. Son rôle est de faire ce que la fonction snake ne faisait pas : mettre à jours la grille.

Cette fonction doit faire cela de la manière suivante :

1. Parcourir la grille entièrement et dès qu'un symbole représentant un snake (*) est trouvé, l'enlever et le remplacer par un espace.
2. Puis, remettre le symbole représentant le snake à la position actuelle du snake.

L'idée c'est que le snake a été modifié avant par un appel à updateSnake, mais la grille n'a pas encore été mise à jours pour prendre en compte ce changement de position. Le but de cette fonction est de prendre en compte ce changement.

VI – Premier test du programme

Si vous avez correctement implémenté toutes les fonctions vues jusqu'ici, vous pouvez maintenant procéder à un test qui vous donnera un aperçut du programme final.

→ Exécutez le script fourni **main_v1.py**

Si tout se passe correctement, à la fin du programme le snake devrait avoir mangé 6/7 des caractères 'o', en laissant celui qui est le plus haut au début et avoir un score de 7. Le programme devrait se terminer parce que le snake finit par se manger un obstacle.

VII – Contrôle Par l'Utilisateur

Maintenant qu'on a une structure de jeu sur laquelle on peut développer, il est temps de se soucier du contrôle du snake par l'utilisateur du programme.

1) – cmdFromString

```
def cmdFromString(string, cmd_chars):
    ''' (str, str) -> int

    Gets a command from a string.
    A command consists of a single char.
    The cmd_chars parameters is a string of two chars, each char corresponds to
    a single command: "Turn Left" or "Turn Right". The first char is left, the
    second is right.

    Return the last command contained in the string, from left to right, as
    an int:
        -1 -> Turn Left
         0 -> No command found in string
         1 -> Turn Right

    >>> cmdFromString("ahpauyozdhgapzeaz", "op")
    1
    >>> cmdFromString("ahpauyozdhgapzoeaz", "op")
    -1
    >>> cmdFromString("ahauyzdhgazeaz", "op")
    0
    '''
```

Cette fonction va nous permettre d'analyser une chaîne de caractères tapés au clavier pour en extraire des commandes à passer au snake.
Dans la logique du jeu, on va lier deux touches du clavier aux commandes 'turnRight' et 'turnLeft'.
Dans l'exemple, on suppose que les deux touches en questions sont « o » et « p ».
La fonction prends en paramètre une chaîne à analyser et les deux caractères correspondants aux deux commandes.
La fonction analyse la chaîne string et renvoie des valeurs différentes en fonctions des caractères qu'elle y trouve :

- 0 si aucun des caractères de commande n'est présent dans la chaîne.
- 1 si le dernier caractère parmi les deux command chars est celui de droite (indice 1 dans cmd_char)
- -1 si le dernier caractère parmi les deux command chars est celui de gauche (indice 0 dans cmd_char)

« Le dernier caractère » veut dire le dernier vu en parcourant la chaîne de gauche à droite.

2) – doCmd

```
def doCmd(s, cmd):
    ''' (snake, int) -> None

    Apply cmd on the snake s.
    If cmd is -1, turn the snake to the left. If it's 1, to the right.
    Do nothing if the command is 0.

    >>> s = genSnake((1, 2), (-1, 0), 'red') ; doCmd(s, -1) ; s
    [[1, 2], [0, 1], 'red', True, 1]
    >>> doCmd(s, -1) ; s
    [[1, 2], [1, 0], 'red', True, 1]
    >>> doCmd(s, 0) ; s
    [[1, 2], [1, 0], 'red', True, 1]
    >>> doCmd(s, 1) ; s
    [[1, 2], [0, 1], 'red', True, 1]
    '''
```

Cette fonction prends en paramètre un snake, et va également récupérer le résultat d'une précédente invocation de la fonction cmdFromString.

En fonction de ce résultat, elle va appliquer sur le snake une commande turnLeft si cmd est -1, turnRight si c'est 1 et ne rien faire si cmd vaut 0.

3) – Fonction fournie inputWithTimeout

```
def inputWithTimeout(timeout):
    ''' (float) -> NoneType

    Gets input from standard input, for 'timeout' seconds.
    '''
```

Je vous fournis une fonction toute faite qui est une variation sur la fonction input() que vous avez l'habitude d'utiliser.

Cette fonction capture les touches pressées au clavier pour une certaine durée spécifiée en secondes sous forme d'un nombre à virgule flottante. (0.5 secondes par exemple)

Cette fonction retourne après le temps spécifié, même si vous n'avez pas pressé la touche entrée.

Attention : **Cette fonction est spécifique à Linux.**

I have neither the time nor the will to develop a similar function for Windows. If you are a Windows Hacker and have an idea on how to do that feel free to drop your code on the forum so that everyone else can enjoy it. Top off my mind I'd say maybe using an ncurse layer, something like that.

Note : vous n'avez pas besoin de comprendre comment cette fonction marche pour pouvoir l'utiliser. Cela dit, je vous encourage à lire le code et essayer de comprendre si ça vous intéresse.

Vous pouvez constater que cette fonction vous est fournie dans un fichier **timedinput.py.** Pour l'utiliser vous pouvez donc utiliser la directive python « `from timedinput import inputWithTimeout` »

VIII – Seconde version du programme

Vous allez maintenant écrire une seconde version du programme fourni **main_v1.py**.

Écrivez un programme **main_v2.py**, basé sur **main_v1.py** mais qui utilise les fonctions **inputWithTimeout**, **cmdFromString** et **doCmd** pour contrôler le snake.

Utilisez les touches « op » pour controler gauche / droite.

Initialisez la grille de la même manière que dans **main_v1.py** en spécifiant manuellement les cases « block » et « food ».

Vous n'avez plus besoin dans ce programme d'utiliser time.sleep(), puisque la fonction **inputWithTimeout** va se charger de fixer l'intervalle de rafraîchissement de l'affichage.

IX – Faire en sorte que le serpent ressemble à un serpent

Pour ceux qui connaissent le jeu, vous vous êtes peut être déjà fait la réflexion que le snake ne grossit pas quand il mange.

Normalement, à chaque fois que le snake assimile un item de type snakefood, ('o') il devrait grossir d'une unité de longueur.

Nous allons maintenant corriger ça.

Il y a au moins deux manières d'approcher le problème :

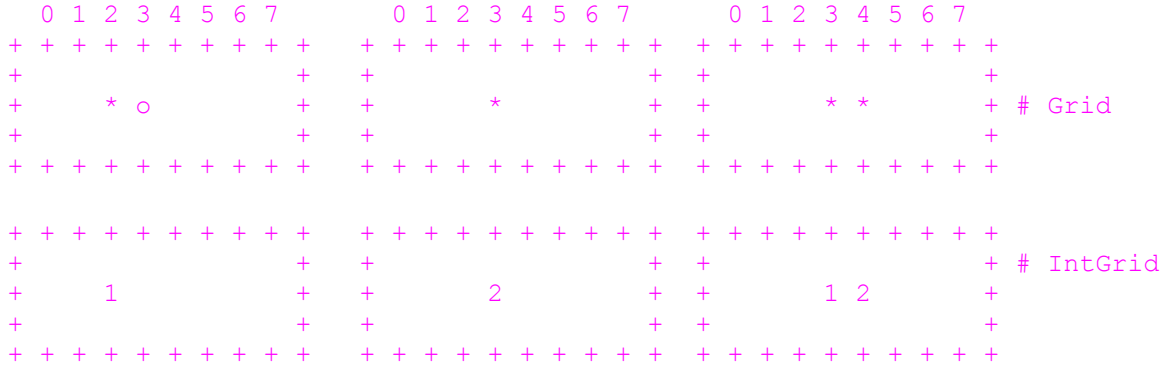
1. On pourrait maintenir une liste des positions occupées par le snake, et parcourir cette liste à chaque affichage et à chaque mise à jours du snake pour corriger les coordonnées de chaque partie du snake
2. On peut utiliser une méthode détournée de compteurs en conjonction avec la taille du snake

Personnellement je préfère la seconde méthode donc c'est celle que je vais vous expliquer par la suite. Si il vous prends l'envie d'implémenter une version utilisant la première approche, vous pouvez bien entendu le faire (en plus).

L'idée c'est que l'on va maintenir une seconde grille, de la même dimension que la grille du jeu mais contenant des entiers au lieu de chaînes de caractères. Toutes les valeurs dans cette grille vont être décrémentées par 1 si elles sont supérieures à 0 à chaque tour dans le jeu. Lors de la mise à jours de la grille de jeu, trois choses vont changer :

1. La procédure de décrémentation sera invoquée sur la grille d'entiers.
2. Un caractère représentant un snake ne sera supprimé que si la position correspondante à celle du caractère dans la grille d'entiers a atteint zéro.
3. Si le serpent est vivant, on repeint sa « tête » sur la grille, et on donne à la case correspondante à la tête dans la grille d'entiers la valeur spécifiée par la taille du serpent. (`snake[4]`)

Le schéma suivant représente ce qu'il se passe quand un snake de taille 1 « mange » :



X - Fonctions pour réaliser la modification d'affichage

1) – genIntGrid

```
def genIntGrid(grid):
    ''' (grid) -> list of list of int {{intgrid}}


    Given a grid, returns an integer grid of the same dimentions, where every
    element is initialized at 0.

    >>> genIntGrid(genGrid(2, 3))
    [[0, 0, 0], [0, 0, 0]]
    >>> genIntGrid(genGrid(3, 4))
    [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
    '''
```

Cette fonction doit fonctionner exactement comme genGrid, à la différence que la grille générée doit être initialisée avec toutes les cases à 0 et et calquer son nombre de lignes / colones sur la grille passée en paramètre.

2) – decrIntGrid

```
def decrIntGrid(intgrid):
    ''' (intgrid) -> NoneType

    Decrement every value in intgrid by one if it is bigger than 0.

    >>> ig = genIntGrid(genGrid(2, 3)) ; ig[0][1] = 2 ; ig[1][2] = 9
    >>> decrIntGrid(ig) ; ig
    [[0, 1, 0], [0, 0, 8]]
    >>> ig[0][0] = 4 ; decrIntGrid(ig) ; ig
    [[3, 0, 0], [0, 0, 7]]
    >>> decrIntGrid(ig) ; ig
    [[2, 0, 0], [0, 0, 6]]
    '''
```

Cette fonction doit passer sur l'intégralité de la grille et décrémenter d'une unité toutes les valeurs supérieures à 0.

3) - updateGrids

```
def updateGrids(grid, intgrid, snake):
    ''' (grid, , intgrid, snake) -> NoneType

    Update the given grids, relative to the snake.
        - Wipe any snake symbol off the grid if the corresponding counter in
          the intgrid has reached zero.
        - If the snake is not dead, repaint it on the grid and reset the
          counter for the snake's position.

    >>> s = genSnake((2, 2), (-1, 0), 'red') ; g = genGrid(3, 3)
    >>> placeObject(g, 2, 2, '*') ; dropFood(g, 1, 2) ; displayGrid(g)
    + + + + +
    +       +
    +       +
    +   o * +
    + + + + +
    >>> ig = genIntGrid(g) ; updateSnake(g, s) ; updateGrids(g, ig, s)
    >>> displayGrid(g)
    + + + + +
    +       +
    +       +
    +   *   +
    + + + + +
    >>> updateSnake(g, s) ; updateGrids(g, ig, s) ; displayGrid(g)
    + + + + +
    +       +
    +       +
    + * *   +
    + + + + +
    '''
```

Version sous amphétamines de updateGrid. Les modifications sont telles que décrites dans la partie IX. Notez le paramètre supplémentaire « intgrid ».

XI – Troisième version du programme

Dernier truc à faire pour ce tuto n°6 : mettre à jours votre programme pour utiliser les nouvelles fonctions et corriger ainsi l'affichage.

→ Écrivez un programme **main_v3.py** basé sur **main_v2.py** qui utilise **updateGrids**.

Si vous êtes encore motivés, vous êtes libres de proposer votre propre amélioration (par exemple un mode multi-joueur pour rendre ce programme capable de gérer plusieurs snakes en même temps).

Happy Hacking !