Fonctions récursives : Snake

Le célèbre jeu vidéo Snake tiré du jeu d'arcade Blockade a été développé en 1997 par Taneli Armanto, ingénieur chez Nokia. L'original apparaît pour la première fois dans le téléphone monochrome Nokia 6110.
Les graphismes se résument à une suite de carrés noirs et il dispose de quatre directions. Le joueur contrôle une longue et fine ligne semblable à un serpent, qui doit slalomer entre les bords de l'écran et les obstacles qui parsèment le niveau. Pour gagner chacun des niveaux, le joueur doit faire manger à son serpent un certain nombre de pastilles similaires à de la nourriture, allongeant à chaque fois la taille du serpent. Alors que le serpent avance inexorablement, le joueur ne peut que lui indiquer une direction à suivre (en haut, en bas, à gauche, à droite) afin d'éviter que la tête du serpent ne touche les murs ou son propre corps, auquel cas il meurt.

Figure 1: Nokia 6110

Objectif de l'activité : Programmer le module de gestion de la position du snake sur l'écran.

1. Définition du problème

Le snake est composé d'une succession de carrés placés à des coordonnées différentes sur le plan. Par exemple, dans la situation décrite sur la figure 2, le snake est composé de 6 carrés dont les coins en haut à gauche sont de coordonnées :

(5, 3)	(5, 2)	(4, 2)	(3, 2)	(2, 2)	(2, 1)

← Tête ← Queue

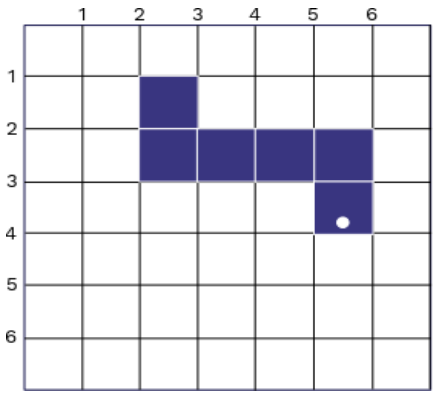

Figure 2: Exemple de serpent

Q1. Déterminer les nouvelles coordonnées du corps de snake s'il avance d'une case dans la direction repérée par la tête sur la figure 2.

La structure de données stockant les différentes coordonnées du corps peut être implémentée par une liste chaînée. Une liste chaînée est une structure de données permettant de regrouper et manipuler des données à partir des primitives suivantes :

- `créer une liste vide`
- `tester si la liste est vide`
- `Lire la valeur de la queue ou de la tête`
- `ajouter un élément en tête ou en queue de la liste`
- `supprimer la tête ou la queue de la liste`
- `Compter le nombre d'éléments`

Q2. Parmi ces primitives, **indiquer** celles à utiliser sur la liste de coordonnées lorsque snake se déplace d'une case et celles lorsqu'il grandit d'une case.

Q3. Décrire en fonction de son orientation (haut, bas, gauche, droite) les calculs à effectuer pour définir les nouvelles coordonnées de la tête de snake.

2.Implémentation d'une liste chaînée

Constitution d'une liste chaînée

Une liste chaînée L peut être vue comme un ensemble de maillons, chaque maillon étant composé de deux parties :

➔ un contenu (valeur) c'est à dire la valeur à stocker

➔ un pointeur qui pointe le maillon suivant. En fait cette deuxième partie contient l'adresse mémoire du maillon suivant.

La figure 3 illustre l'organisation de cette structure de donnée pour le snake défini précédemment :

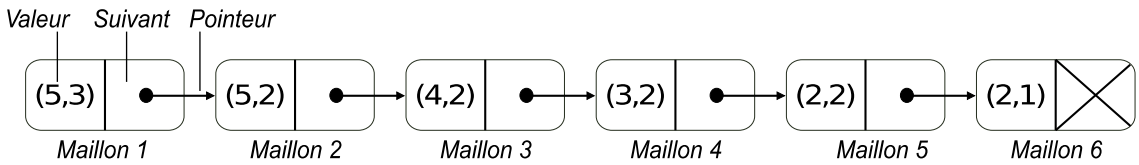

Figure 3: Représentation de Snake par une liste chaînée

Nous avons vu que l'agrandissement du Snake consistait à rajouter en tête un maillon avec les nouvelles coordonnée de la tête.

Q4. La figure 4 illustre la liste chaînée de Snake à un instant ainsi que le nouveau maillon à insérer pour agrandir le serpent. **Relier** le pointeur de ce nouveau maillon à la liste chaînée initiale afin de respecter la structure étudiée jusqu'à maintenant.

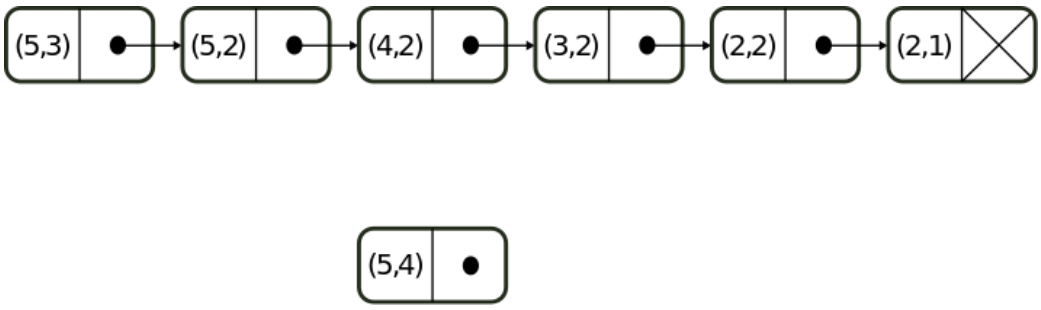

Figure 4: Agrandissement de Snake

Q5. Dans le cas de l'utilisation d'un tableau, **indiquer** les opérations à réaliser pour placer une valeur en tête. **Justifier** alors l'intérêt d'implémenter dans notre problème, une liste chaînée plutôt qu'un tableau (liste Python).

3.Implémentation récursive d'une liste chaînée

Implémentation d'un maillon

L'implémentation d'un maillon sera effectuée en programmation orientée objet avec une classe `Maillon`. Cette classe est composée de deux attributs publics `valeur` et `suivant` et d'un constructeur (voir figure 5). L'attribut `valeur` stockera la valeur de l'élément de la liste et `suivant` stockera l'adresse du maillon suivant (pointeur).

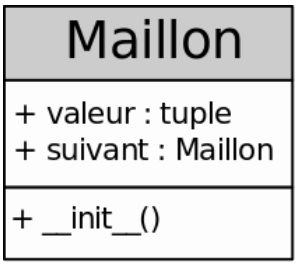

Figure 5: Diagramme de classe Maillon

Codage de la classe Maillon et association de maillons

Un pointeur est en fait ni plus ni moins qu'une référence à un autre objet Maillon. Concrètement pour implémenter ce pointeur, il suffit d'affecter à l'attribut `suivant` un nouvel objet `Maillon`. En bout de chaîne, le dernier maillon ne pointant vers aucun autre, la valeur `None` sera affectée à l'attribut suivant spécifiant ainsi la fin de la liste chaînée.

Codage de la classe Maillon

```
class Maillon :
    def __init(self, val, maillon_suivant) :
        self.valeur = val
        self.suivant = maillon_suivant
```

Association de trois maillons

```
>>>m1 = Maillon(10, None)
>>>m2 = Maillon(20, m1)
>>>m3 = Maillon(22, m2)
```

Q6. L'association des trois maillons `m1`, `m2`, `m3` constitue une liste chaînée, **indiquer** le nom du maillon placé en tête de cette liste et celui placé en queue de liste.

Manipulation de la liste chaînée par récursion

Une liste chaînée peut être vue comme un maillon contenant un autre maillon contenant un autre maillon etc. Ceci fait donc apparaître une dimension récursive de notre liste (voir figure 6). Chaque maillon contient dans l'attribut `suivant` une sous liste d'éléments.

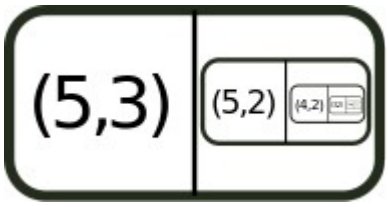

Figure 6: Dimension récursive d'une liste chaînée

Compter le nombre d'éléments

Compter le nombre d'élément d'une liste chaînée consiste donc à rajouter 1 à la longueur de la sous liste contenue dans le maillon. Le calcul de la longueur de la sous liste pouvant être appelée récursivement jusqu'à arriver en bout de chaîne.

Q7. La fonction `longueur_liste(m)` suivante doit renvoyer le nombre de maillons d'une liste chaînée, **compléter** en langage Python cette fonction.

```
def longueur_liste(m) :
    if _____________ : # test si m est le dernier maillons
        return 1
    else :
        return 1+longueur_liste(_______ )
```

Q8. Tester la fonction `longueur_liste(m)` à l'aide du programme de test test_maillons.py. Ce programme initialise une liste chaînée de test et contient les diverses fonctions de manipulation de liste chaînée à compléter et à tester.

Vérifier si la chaîne contient une valeur

Notre programme de gestion de position de Snake devra détecter la collision de la tête du serpent avec son corps. Cette collision peut être programmée en vérifiant si la liste chaînée contient les coordonnées de la tête. Vérifier la présence d'une valeur dans la liste peut être implémenté récursivement en testant si le maillon contient la valeur demandée et renvoyer vrai dans ce cas sinon vérifier par récursion si la sous chaîne contient la valeur.

Q9. Définir dans le fichier test_maillons.py la fonction `test_valeur(v : int, m : Maillon)`. **Tester** son fonctionnement.

Retirer l'élément de queue

Retirer l'élément de queue de la liste chaînée revient à appeler récursivement le retrait de la queue de la sous liste. La condition d'arrêt étant que le maillon suivant est le dernier maillon.

Q10. Ecrire une instruction qui supprime dans un maillon, le maillon suivant. **Ecrire** le test à effectuer pour tester si le maillon contient un maillon de queue.

Q11. Définir dans le fichier test_maillons.py la fonction `supprime_queue(m : Maillon)`. **Tester** son fonctionnement.

4.Implémentation de la classe Snake

Le jeu sera implémenté à partir d'une classe Snake représenté par le diagramme de classe figure 7.

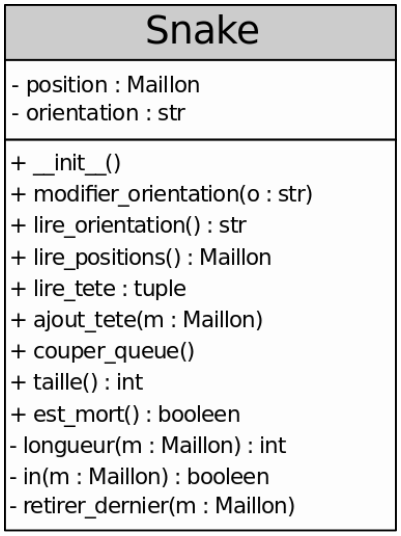

Figure 7: Diagramme de classe Snake

Description des attributs et méthodes

Nom	Description
Attributs	
__position	Attribut privé stockant la liste chaînée de position des éléments du corps de Snake
__orientation	Attribut privé stockant l'orientation de la tête (‘*droite*’, ‘*gauche*’, ‘*haut*’, ‘*bas*’).
Méthodes	
__init__()	Constructeur de la classe. Initialise l'attibut `__position` avec un maillon de queue de valeur *(50,30)* et `__orientation` avec la chaîne de caractère ‘*bas*’
modifier_orientation(o : str)	Mutateur de l'attribut `__orientation`. Affecte à `__orientation` la chaîne de caractères o
lire_orientation() : str	Accesseur à l'attribut `__orientation`.
lire_positions() : Maillon	Accesseur à l'attribut `__position`.
lire_tete() : tuple	Renvoie la valeur du maillon de tête de l'attribut `__position`
ajout_tete()	Ajoute en tête de l'attribut `__position` un nouveau maillon dont la valeur est calculée en fonction de l'orientation définie dans `__orientation`
__retirer_dernier(m : Maillon)	Retire la queue de la liste chaînée `m` en suivant la démarche détaillée Q11
couper_queue()	Retire la queue de l'attribut `__position` en appliquant la méthode `__retirer_dernier()`
__longueur(m : maillon)	Renvoie le nombre de maillons contenus dans `m` en suivant la démarche détaillée en Q8
taille()	Renvoie le nombre de maillons contenus dans `__position` en appliquant la méthode `__longueur(m)`
__in(val : tuple, m : Maillon) : booleen	Teste si la valeur val est présente dans la liste chaînée `m`. Renvoie *True* si vrai (démarche détaillée Q9)
est_mort()	Renvoie un booléen indiquant si le serpent est mort (*True* si mort). La sortie de l'écran du Snake ne sera pas pris en compte, seule la collision de la tête du serpent avec le corps ne sera considérée dans l'évaluation de la mort du serpent.

Q12. A partir de cette description, implémenter la classe Snake. **Tester** son fonctionnement