[Q1] Listes de diviseurs

Étant donné deux listes d'entiers `l1` et `l2`, écrivez une fonction `diviseurs(l1,l2)` qui retourne un dictionnaire, où chaque clé dans le dictionnaire correspond à un élément de `l1`, et la valeur associée à cette clé est une liste contenant le sous-ensemble d'éléments de `l2` qui sont divisibles par la clé.

Par exemple :

```
diviseurs([3,5,10], [9,15,10,150])
```

retournera le dictionnaire suivant :

```
{3: [9,15,150], 5: [15,10,150], 10: [10,150]}
```

- *9* , *15* et *150* sont multiples de 3
- *15* , *10* et *150* sont multiples de 5
- *10* et *150* sont multiples de 10

Les deux listes `l1` ou `l2` peuvent être vides. Si `l1` est vide, le dictionnaire retourné doit être vide. Si `l2` est vide, les listes associées aux différentes clés dans le dictionnaire seront vides.

La fonction demandée est la suivante:

```
def diviseurs(l1,l2):
    """
    @pre: l1 et l2 sont des listes d'entiers, potentiellement vides
    @post: retourne un dictionnaire avec comme différentes clés
           chaque élément e de l1, et comme valeur associée à cette
           clé e, la liste des éléments de l2 que l'on peut diviser
           par e
```

[Q2] Run-length encoding

Un problème courant en informatique est celui de la *compression* : la création de fichiers plus petits pour stocker ou envoyer des données dans moins d'espace. Un type de compression est le *run-length encoding* .

Étant donnée une liste, un codage « *run-length* » (un procédé de codage des répétitions) se compose de tuples (`i`, `e`), où `i` est un entier qui représente le nombre d'éléments consécutifs égal à `e`. Écrivez une fonction `rle(l)` qui calcule le codage *run-length* d'une liste `l` donnée.

Par exemple :

```
print(rle(['A','A','A','C','C',1,1,1,'A','A','T','G','G']))
```

imprime :

```
[(3,'A'),(2,'C'),(3,1),(2,'A'),(1,'T'),(2,'G')]
```

La liste `l` que la fonction `rle` prend en paramètre peut éventuellement être vide (auquel cas la liste retournée sera vide), peut contenir de caractères, de chaînes de caractères, ou de nombres entiers, mais ne contient pas d'éléments None. La liste retournée par la fonction est une liste de tuples (`i`, `e`). Le premier tuple (`i`, `e`) de la liste retournée veut dire que la liste originale commençait par `i` fois l'élément `e`, et ainsi de suite.

```
def rle(l):
    """
    @pre: l est une liste d'éléments, éventuellement vide,
          contenant soit de caractères, de chaînes de caractères, ou
          de nombres entiers, mais ne contient pas d'éléments None
    @post: retourne une nouvelle liste avec le codage run-length de
           la liste l, comme décrit dans l'énoncé ci-dessus
```

[Q3] Lecture d'un fichier d'étudiants

Le service administratif des étudiants veut offrir un nouveau service permettant aux professeurs d'envoyer des fichiers de texte contenant les données des étudiants participant à un cours. A vous de vérifier si ces fichiers ont le bon format avant qu'ils soient traités par le service administratif. D'abord, présentons le format attendu des fichiers. Idéalement, les fichiers contiennent des lignes comme :

```
00110804,De Bels,Tanguy,debels.tanguy@gmail.com
00120501,Michel,Francois,michel.francois@student.uclouvain.be
```

où chaque ligne du fichier est composée comme suit:

```
NOMA,Nom,Prénom,email
```

Ici, le *NOMA* est une chaîne de 8 chiffres, et doit être suivi par un virgule « , » comme séparateur. Ensuite on a le *Nom* et le *Prénom*, des chaînes de caractères non vides avec au moins un caractère non blanc, chacune séparées par un « , » . Ensuite l'adresse e-mail qui est une chaîne de caractères contenant au moins le caractère @. Aucune autre contrainte n'est imposée sur les types de caractères acceptés dans les noms ou dans l'adresse e-mail. Par exemple, un fichier avec les lignes suivants devrait aussi être accepté :

```
00190304,Martin @,88, m@g
00180501,@ L,L M,@s#a
```

Des lignes vides ou contenant uniquement des espaces vides sont aussi permises dans le fichier. Un fichier vide est donc également accepté.
Par contre, aucune des lignes suivantes ne sera correcte :

```
2019,De Bels,Tanguy,debels.tanguy@gmail.com
20190304,     ,Tanguy,debels.tanguy@gmail.com
20190304,De Bels,  ,debels.tanguy@gmail.com
20190304,De Bels,Tanguy,debels.tanguy
20190304 De Bels Tanguy debels.tanguy@gmail.com
```

Écrivez maintenant une fonction `est_correct(f)` qui, pour un nom de fichier f donné, retourne `True` si le fichier nommé f existe et est dans le format correct, ou retourne `False` si le fichier n'est pas dans le format correct. On vous conseille d'écrire d'abord une fonction auxiliaire pour vérifier si une ligne est dans le format correct.

```
def est_correct(f):
    """
    @pre:  f est le nom d'un fichier
    @post: retourne True si le fichier avec comme nom f existe et
           a le format correct;
           retourne False s'il n'y a pas de fichier nommé f ou si ce
           fichier n'a pas le format correct
    """
    # à compléter
```

[Q4] OXO

Pour pouvoir jouer le jeu OXO, il suffit d'être deux, avoir sous la main une feuille de papier sur laquelle vous avez tracé une grille de *m* x *n* cases. A tour de rôle, chaque joueur doit remplir l'une des cases soit avec un X, soit avec un O. Le but du jeu étant d'aligner la combinaison « OXO » dans des cases consécutives (soit horizontalement, verticalement ou diagonalement), le gagnant étant celui qui a pu aligner le plus de fois cette combinaison.

Implémentez une fonction `oxo(grille)` qui, pour une grille donnée de *m* x *n* cases, calcule le nombre total de combinaisons « OXO » sur cette grille. La grille est représentée par une liste de m listes contenant chacune n valeurs. Une valeur est soit le caractère "O", le caractère "X", ou le caractère " " (c'est-à-dire un espace, représentant une case non remplie).

Voici un exemple d'une telle grille de 4 x 3 cases:

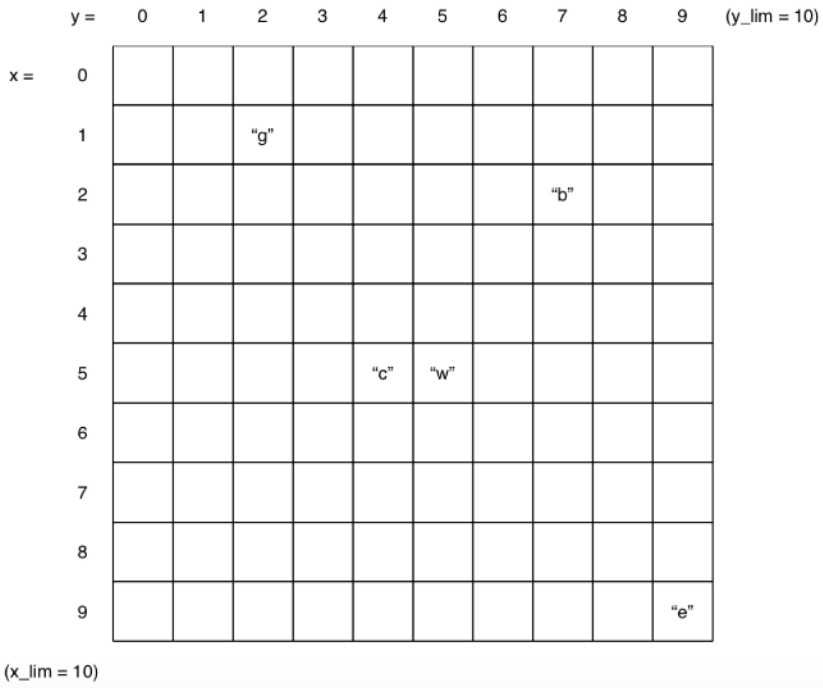
```
grille = [ [ "O", " ", "O" ],
           [ "X", "X", "X" ],
           [ "O", " ", "O" ],
           [ "O", "X", "O" ] ]
```

Pour cette grille la fonction `oxo(grille)` retourne 5 car il existe dans cette grille 2 combinaisons "O","X","O " verticales, 2 diagonales et 1 horizontale.

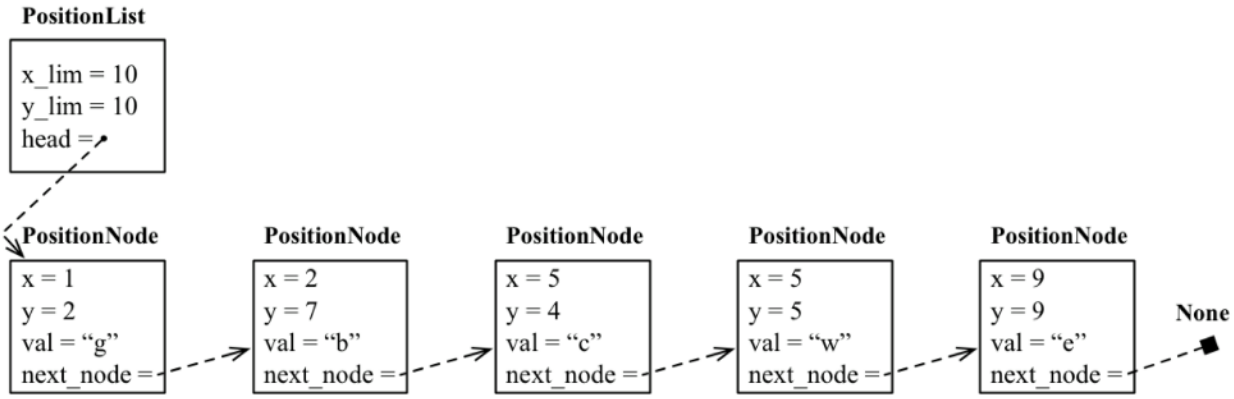
```
def oxo(grille):
    """
    @pre:  grille est une liste de m listes contenant chacune
           n valeurs ; chaque valeur est soit le caractère "O",
           le caractère "X", ou le caractère " "
    @post: retourne le nombre total de combinaisons "O","X","O"
           dans des cases consécutives (soit horizontalement,
           verticalement ou diagonalement) sur cette grille
```

Ceci est le contexte et le code utilisés dans la question 5 et la question 6, il est conseillé de commencer par le lire.
Pour rappel: il est possible d'ouvrir une fenêtre supplémentaire d'INGInious.

Une matrice creuse est une matrice contenant beaucoup d'éléments vides. L'image ci-dessous montre une telle matrice creuse de taille *x_lim* * *y_lim* où *x_lim* = *y_lim* = *10*, contenant une valeur seulement aux positions *(1,2)*, *(2,7)*, *(5,4)*, *(5,5)* et *(9,9)*.



Une manière peu efficace de stocker une telle matrice serait d'utiliser une liste de listes pour représenter chaque case de la grille, même les cases vides, et associer une valeur None à chaque case vide. Une manière plus optimale pour représenter une matrice creuse, qu'on vous demande d'implémenter dans cette question, est de ne stocker que les cases utilisées. A cette fin, on utilise une liste chaînée de nœuds (implémentée par une classe **PositionList**), où chaque nœud (implémenté par une classe **PositionNode**) représente une position *(x,y)*, la valeur val stockée à cette position, et une référence vers le nœud suivant. Dans cette liste chaînée on ne stockera que les positions ayant une valeur ; les cases vides ne seront pas stockées. Par exemple, la matrice ci-dessus est représentée par la structure suivante :



La liste est ordonnée par ordre croissant de coordonnées *x* et *y* : un nœud à la position *(x',y')* suivra un nœud *(x,y)* si $x' > x$ ou si $x' = x$ et $y' > y$. C'est le cas dans l'exemple ci-dessus.

On vous fournit déjà une implémentation complète de la classe **PositionNode** ainsi qu'une implémentation partielle de la classe **PositionList** à compléter :

```
class PositionNode:

    def __init__(self, x, y, val, next_node=None):
        self.x = x
        self.y = y
        self.val = val
        self.next_node  = next_node

    def __str__(self):
        """
        @pre:  -
        @post: retourne une chaine de caractères représentant
               textuellement ce noeud dans le format "(x,y,val)"
        """
        return "(" + str(self.x) + "," + str(self.y) + "," \
               + str(self.val) + ")"
```

```
class PositionList:

    def __init__(self, x_lim, y_lim):
        self.head = None
        self.x_lim = x_lim
        self.y_lim = y_lim

    def __str__(self):
        """
        @pre:  -
        @post: retourne une chaine de caractères représentant
               textuellement la liste chainée dans le format
               [(x1,y1,val1),…,(xn,yn,valn)] dans le même ordre
               que les noeuds apparaissant dans la liste
        """
        str = "["
        node = self.head
        while node is not None :
            str += node.__str__()
            node = node.next_node
        str += "]"
        return str
```

[Q5] Matrice creuse – méthode existe

Implémentez dans la classe **PositionList** la méthode `existe(self,x,y)` qui vérifie si une valeur à la position *(x,y)* existe déjà. Si oui, cette valeur est retournée par la méthode. Sinon la méthode retourne `False`. Si *x* et *y* ne sont pas dans les bornes de la matrice, la méthode retourne `"out-of-bounds"`

Pour rappel: le contexte est disponible dans une autre tâche.

```
def existe(self,x,y):
    """
    @pre:  x et y sont des entiers
    @post: retourne val si un noeud avec une valeur val existe déjà
           à la position (x,y);
           retourne la chaine de caractères "out-of-bounds" si x n'est pas dans
           les bornes 0 <= x < x_lim ou si y n'est pas dans les bornes 0 <= y < y_lim;
           retourne False si x et y sont dans les bornes mais qu'il n'existe
           encore aucun noeud pour cette position dans la liste chaînée
    """
    # à compléter
```

Évidemment, si nécessaire vous pouvez aussi ajouter d'autres méthodes auxiliaires qui seront également insérés au sein de cette même classe.

[Q6] Matrice creuse – méthode insert

Implémentez dans la classe **PositionList** une méthode `insert(self,x,y,val)` qui insère un nouveau nœud avec une valeur `val` pour la position *(x,y)* tenant compte de l'ordre suivante : *(x1,y1)* précède *(x2,y2)* si *x1 < x2* ou si *x1 = x2* et *y1 < y2*. Pour la matrice creuse illustrée précédemment, l'ordre d'insertion sera donc *(1,2) < (2,7) < (5,4) < (5,5) < (9,9)*, les positions les plus petites se trouvant en tête de la liste. Si un nœud existe déjà à la position *(x,y)*, aucun nouveau nœud sera créé mais la valeur du nœud existant sera simplement remplacé par `val`. Si *x* et *y* ne sont pas dans les bornes de la matrice, la méthode retourne `"out-of-bounds"`

Pour rappel: le contexte est disponible dans une autre tâche.

```
def insert(self,x,y,val):
    """
    @pre:  x et y sont des entiers
    @post: retourne la chaine de caractères "out-of-bounds" si x n'est pas dans
           les bornes 0 <= x < x_lim ou si y n'est pas dans les bornes 0 <= y < y_lim;
           retourne True s'il n'y a pas encore un noeud pour la position (x,y)
           dans la liste chainée et insère un nouveau nœud (un objet de la classe PositionNode)
           avec une valeur val à la position (x,y);
           retourne True s'il y a déjà un noeud pour la position (x,y) dans la
           liste chainée et remplace sa valeur par val;
           Les noeuds sont ordonnés de plus petits (en tête de la liste) au plus grands
           (en queue de la liste) selon l'ordre suivant : (x1,y1) précède (x2,y2)
           si x1 < x2 ou si x1 = x2 et y1 < y2
    """
    # à compléter
```

Évidemment, si nécessaire vous pouvez aussi ajouter d'autres méthodes auxiliaires qui seront également insérés au sein de cette même classe.