Corrections aux TD 5 et 4

Laura S. Mendoza

Mars 2020

Introduction

Voici la correction pour l'exercice 5 du TD 4 et pour le TD 5, destinées aux étudiants du groupe 1b, pour plus des questions sur ce TD, vous pouvez m'envoyer directement un email mendoza@unistra.fr ou par moodle.

TD 4 - Exercice 5

1. Une *grille* est une liste de liste. Son *hauteur* c'est le nombre de listes qu'elle contient, et sa *largeur* le nombre d'éléments que chacune de ces listes contiennent (elles contiennent toutes le même nombre d'entiers, donc on peut prendre le nombre d'éléments de la première liste de la grille).

```
>>> # Creation d'une grille de hauteur 3 et largeur 5:
>>> grille = [ [0, 0, 0, 0, 0],
...            [0, 0, 0, 0, 0],
...            [0, 0, 0, 0, 0] ]
>>> hauteur = len(grille) # nb de liste
>>> largeur = len(grille[0]) # nb d'ele dans une liste

>>> # On verifie que les resultats soient correctes:
>>> print("Hauteur de la grille :", hauteur)
Hauteur de la grille : 3
>>> print("Largeur de la grille :", largeur)
Largeur de la grille : 5
```

2. On veut modifier une grille de taille 4x3 (de hauteur 4 et largeur 3), tel qu'elle devienne:

   ```
   [ [1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4] ]
   ```

   On va devoir créer deux boucles `for` la première va parcourir chaque ligne de la grille, et la deuxième chaque élément de la liste parcourue. De plus, on remarque que les entiers correspondent au numéro de la ligne plus un (vu qu'en `python` la numérotation commence par 0).

```
>>> # Creation d'une grille 4x3 avec des zeros :
>>> grille = [ [0, 0, 0],
...            [0, 0, 0],
...            [0, 0, 0],
...            [0, 0, 0] ]

>>> # On peut recuperer le nombre de lignes (hauteur)
>>> # et de colonnes (largeur) de la grille :
>>> nb_lgn = len(grille)
>>> nb_col = len(grille[0])

>>> # On parcourt chaque ligne de la grille
>>> for i in range(nb_lgn):
...     for j in range(nb_col):
...         grille[i][j] = i + 1
...
>>> print(grille)
[[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
```

3. Cette fois-ci, nous devons créer la même liste mais on ne commence pas d'une liste déjà créer sinon qu'on doit créer la grille en connaissant juste les dimensions de la grille finale. Pour rappel, il existe principalement deux façons de créer une liste de taille `n`.

```
>>> # On donne une valeur à n
>>> n = 15

>>> # Premier methode : par concatenation
>>> liste1 = []
>>> for i in range(n):
...     liste1 += [i]
...
>>> # Deuxieme methode : en utilisant `append`
>>> liste2 = []
>>> for j in range(n):
...     liste2.append(j)
...
>>> # On affiche les deux listes
>>> print("Liste 1:", liste1)
Liste 1: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> print("Liste 2:", liste2)
Liste 2: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Et une grille, comme on a déjà vu, ce n'est qu'une liste de listes. On aura toujours besoins de deux boucles `for`, une pour créer les listes qui

formeront les lignes, et une autre pour créer la grille, la liste de liste. Voici le code en utilisant la méthode de concatenation.

```
>>> # On donne une valeur à haut et larg
>>> haut = 5; larg = 6

>>> # Premier methode : par concatenation
>>> grille = []

>>> for i in range(haut):
...     # on cree la liste qui va contenir chaque ligne
...     ligne = []
...     # et on la remplit
...     for j in range(larg):
...         ligne += [i]
...     # maintenant qu'elle est faite
...     # on l'ajoute a la grille
...     grille += [ligne]
...
>>> # on affiche la grille
>>> for i in range(haut):
...     print(grille[i])
...
[0, 0, 0, 0, 0, 0]
[1, 1, 1, 1, 1, 1]
[2, 2, 2, 2, 2, 2]
[3, 3, 3, 3, 3, 3]
[4, 4, 4, 4, 4, 4]
```

Notez que même si `ligne` est une liste il faut quand même ajouter des crochets lors de la concatenation (`grille += [ligne]`) sinon on n'aurait pas une grille (liste de liste) mais juste une liste `[1, 1, .., 2, 2, .. 3, ...]`. Voici la même solution avec `append` (équivalente)

```
>>> # On donne une valeur à haut et larg
>>> haut = 5; larg = 6

>>> # Premier methode : par concatenation
>>> grille = []

>>> for i in range(haut):
...     # on cree la liste qui va contenir chaque ligne
...     ligne = []
...     # et on la remplit
...     for j in range(larg):
...         ligne.append(i)
```

```
...     # maintenant qu'elle est faite
...     # on l'ajoute a la grille
...     grille.append(ligne)
...
>>> # on affiche la grille
>>> for i in range(haut):
...     print(grille[i])
...
[0, 0, 0, 0, 0, 0]
[1, 1, 1, 1, 1, 1]
[2, 2, 2, 2, 2, 2]
[3, 3, 3, 3, 3, 3]
[4, 4, 4, 4, 4, 4]
```

TD5 : Fonctions

1 - Le codage des caractères

1. Comme toutes les lettre minuscules sont codée en python par des entiers entre 97 et 122, pour savoir si une chaine de caractères est une lettre minuscule, il suffit de savoir si son code est un entier dans cet intervale.

```
>>> def estMinuscule(lettre):
...     if 97 <= ord(lettre) <= 122:
...         return True
...     else:
...         return False
...
>>> # quelques tests:
>>> estMinuscule("a")
True
>>> estMinuscule("z")
True
>>> estMinuscule("m")
True
>>> estMinuscule("G")
False
```

Remarque : on peut écrire la fonction ci dessus en une seule ligne. En effet l'instruction `97 <= ord(lettre) <= 122` donne un booléen. On peut alors écrire:

```
>>> def estMinuscule(lettre):
...     return 97 <= ord(lettre) <= 122
...
>>> # quelques tests:
>>> estMinuscule("a")
True
>>> estMinuscule("z")
True
>>> estMinuscule("m")
True
>>> estMinuscule("G")
False
```

1.2.2 - Le chiffrement de César

1. Au premier abord, on pourrait penser que pour obtenir la lettre chiffrée, c'est à dire la lettre décalée d'un certain décalage `decal`, il suffit d'obtenir son numéro de codage et de lui additionner `decal`:

```
>>> # avec un decalage positif
>>> decal = 5
>>> chiffrement = ord("a") + decal
>>> chr(chiffrement)
'f'

>>> # avec un decalage negatif
>>> decal2 = -3
>>> chiffrement2 = ord("z") + decal2
>>> chr(chiffrement2)
'w'

>>> # mais...
>>> decal3 = 1
>>> chiffrement3 = ord("z") + decal3
>>> # si on decale de 1 "z" on voudrait obtenir
>>> # la lettre "a"
>>> chr(chiffrement3)
'{'
>>> # on n'obtient pas cela, voyons le chiffrement:
>>> print(chiffrement3)
123
```

On se rend compte qu'il faut être sure que le chiffrement obtenu soit aussi compris entre 97 et 122. Dans l'exemple ci-dessus, on a obtenu 123, on aurait voulu obtenir 97. Pour un decalage positif, ob risque alors de déborder par la borne supérieur, et par la bonne inférieur si le décalge est négatif. Pour s'assurer que le code se trouve dans l'intervalle $[97, 122]$, il suffit d'utiliser un modulo

$$(\text{code} + \text{decal} - 97) \,\%\, 26 + 97$$

Avec code+decal on applique le décalge, on soustrait 97 pour que l'intervalle souhaité soit entre $[0, 26]$ (pour pouvoir utiliser le modulo), on applique le modulo pour obtenir un nombre entre 0 et 26, et on se re-décale à l'intervalle $[97, 122]$ en ajoutant 97. On obtient alors la fonction:

```
>>> def decalageCar(lettre, decal):
...     # on commence par obtenir le code de la lettre
...     code_lettre = ord(lettre)
...     # on le decale
...     code_decale = code_lettre + decal
...     # on s'assure qu'on est bien dans l'intervalle
...     # [97, 122] en appliquant la formule:
```

```
...     bon_code = (code_decale - 97) % 26 + 97
...     # on retourne la lettre correspondante
...     return chr(bon_code)
...
>>> # On teste avec les exemples de l'enonce:
>>> decalageCar("a",4) # devra retourner "e"
'e'
>>> decalageCar("p",4) # devra retourner "t"
't'
>>> decalageCar("z",4) # devra retourner "d"
'd'
>>> decalageCar("d",-4) # devra retourner "z"
'z'
```

2. On va utiliser les deux fonctions précédentes, `estMinuscule` et `decalageCar`, pour décaler une chaine de caractères `message`. Pour rappel, on peut accéder à l'i-ème caractère d'une chaine de caractères de la forme suivante

   `lettre = message[i]`

```
>>> def decalageStr(message, decal):
...     # On commence par creer une chaine de caractere
...     # qui sera le message code
...     msg_code = ""
...     # On parcourt la chaine de caracteres:
...     for i in range(len(message)):
...         if estMinuscule(message[i]):
...             msg_code += decalageCar(message[i], decal)
...         else:
...             # on ne modifie pas les caracteres qui ne
...             # sont pas des lettre minuscules
...             msg_code += message[i]
...     return msg_code
...
>>> # On teste avec les exemples de l'enonce:
>>> decalageStr("Bonjour, comment tu vas ?", 4)
'Bsrnsyv, gsqqirx xy zew ?'
>>> decalageStr("Moi, je vais tres bien !", 4)
'Msm, ni zemw xviw fmir !'
```

   Une version légèrement plus courte reviendrait à faire la boucle directement sur la lettre de la chaine de caractères. C'est à dire au lieu de faire une boucle `for` sur `i`, récuperer la lettre directement du message dans la boucle: "`for lettre in message:`".

   Ce qui donne:

```
>>> def decalageStr(message, decal):
...     # On commence par creer une chaine de caractere
...     # qui sera le message code
...     msg_code = ""
...     # On parcourt la chaine de caracteres:
...     for lettre in message:
...         if estMinuscule(lettre):
...             msg_code += decalageCar(lettre, decal)
...         else:
...             # on ne modifie pas les caracteres qui ne
...             # sont pas des lettre minuscules
...             msg_code += lettre
...     return msg_code
...
>>> # On teste avec les exemples de l'enonce:
>>> decalageStr("Bonjour, comment tu vas ?", 4)
'Bsrnsyv, gsqqirx xy zew ?'
>>> decalageStr("Moi, je vais tres bien !", 4)
'Msm, ni zemw xviw fmir !'
```

3. Pour la fonction `codage`, il suffit d'utiliser la fonction `decalageStr` avec le paramètre `decal` à 5.

```
>>> def codage(message):
...     return decalageStr(message, 5)
...
>>> # Voici des exemples:
>>> codage("Bonjour, comment tu vas ?")
'Btsotzw, htrrjsy yz afx ?'
```

4. Pour décoder un message il suffit d'applique le décalage en négatif, donc pour décoder un message codé avec un décalage de 5, on va appliquer `decalageStr` avec le paramètre `decal` à -5.

```
>>> def decodage(message):
...     return decalageStr(message, -5)
...
>>> # On peut verifier en codant et decodant un message
>>> # on devrait obtenir le message d'origine
>>> msg_code = codage("Bonjour, comment tu vas ?")
>>> msg_code
'Btsotzw, htrrjsy yz afx ?'
>>> decodage(msg_code)
'Bonjour, comment tu vas ?'
```

2 - À propos du projet labyrinthe

1. Il y a 4 voisins à chaque cellule (haut, bas, gauche, droite)

2. Pour une cellule aux coordonnées (l, c) ses quatre voisins sont: la cellule du haut $(l, c+1)$, du bas $(l, c-1)$, de droite $(l+1, c)$, et celle de gauche $(l-1, c)$.

```
>>> def afficheVoisins_laby_inf(lgn, col):
...     print("voisin du haut   : ", lgn, col + 1)
...     print("voisin du bas    : ", lgn, col - 1)
...     print("voisin de gauche : ", lgn - 1, col)
...     print("voisin du droite : ", lgn + 1, col)
...     return
...
>>> afficheVoisins_laby_inf(5, 7)
voisin du haut   :  5 8
voisin du bas    :  5 6
voisin de gauche :  4 7
voisin du droite :  6 7
```

3. On fait la même chose mais au lieu de les afficher on crée des tuples et on les renvoit dans une liste

```
>>> def voisins_laby_inf(lgn, col):
...     h = (lgn, col + 1)
...     b = (lgn, col - 1)
...     g = (lgn - 1, col)
...     d = (lgn + 1, col)
...     voisins = [h, b, d, g]
...     return voisins
...
>>> print("Liste des voisins : ", voisins_laby_inf(5,7))
Liste des voisins :  [(5, 8), (5, 6), (6, 7), (4, 7)]
```

On peut faire la même fonction avec une boucle `for`

```
>>> def voisins_laby_inf(lgn, col):
...     # on initialise la liste des voisins
...     voisins = []
...     # on definit les decalages necessaires pour
...     # obtenir les voisins
...     decal = [(0,1), (0,-1), (1,0), (-1,0)]
...     for direc in decal:
...         voisins += [(lgn + direc[0], col + direc[1])]
...     return voisins
```

```
...
>>> print("Liste des voisins : ", voisins_laby_inf(5,7))
Liste des voisins :  [(5, 8), (5, 6), (6, 7), (4, 7)]
```

4. La cellule de coordonnées (lgn, col) est dans le labyrinthe si

$$0 \leq \text{lgn} < \text{nb\_lignes}$$
$$0 \leq \text{col} < \text{nb\_colonnes}$$

```
>>> def estDedans(lgn, col, nb_lignes, nb_colonnes):
...     return (0 <= lgn < nb_lignes
...             and 0 <= col < nb_colonnes)
...
>>> # Test
>>> estDedans(5, 7, 10, 10)
True
>>> estDedans(3, 17, 10, 10)
False
>>> estDedans(-1, 0, 10, 10)
False
```

5. Il y a 3 différents cas: si une cellule est à l'intérieur du labyrinthe, elle contient 4 voisins, si elle est dans un coin elle en a que deux, et si elle sur un bord (mais n'est pas un coin) elle en a trois.

6. Pour créer la fonction `voisins_laby_fin`, on peut utiliser la fonction `voisins_laby_inf` et tester si chacun des voisins calculés est dans le labyrinthe grace à `estDedans`

```
>>> def voisins_laby_fin(lgn, col, nb_lignes, nb_colonnes):
...     # on obtient tous les voisins
...     voisins = voisins_laby_inf(lgn, col)
...     # on initialise la liste des voisins dedans
...     voisins_dedans = []
...     # on verifie un a un s'il est dedans
...     for v in voisins:
...         if estDedans(v[0], v[1], nb_lignes, nb_colonnes):
...             # on l'ajoute a la liste que s'il est dedans
...             voisins_dedans += [v]
...     return voisins_dedans
...
>>> # voisins d'un point milieu
>>> voisins_laby_fin(5, 5, 10, 10)
[(5, 6), (5, 4), (6, 5), (4, 5)]
```

```
>>> # voisins d'un coin
>>> voisins_laby_fin(0, 9, 10, 10)
[(0, 8), (1, 9)]
>>> # voisins d'une cellule bord
>>> voisins_laby_fin(9, 5, 10, 10)
[(9, 6), (9, 4), (8, 5)]
```