IN 101 - Examen

23 octobre 2018

François Pessaux

Énoncé

— **Lisez attentivement les consignes et tout le sujet avant de commencer.**

— Les documents (polys, transparents, TDs, livres ...) sont autorisés.
— Sont **absolument interdits** : le WEB, le courrier électronique, les messageries diverses et variées, le répertoire des camarades, le téléphone (même pour avoir l'heure puisque vous l'avez sur votre ordinateur).

— **Important** : Il vous est demandé de vous conformer aux constructions de PYTHON **3** qui ont été imposées dans ce cours. En particulier, **pas de** `range`, `for`, `in`, `append`, `numPy`, classes (hormis les exceptions). Si vous avez besoin d'une boucle, faites un `while`. Si vous avez besoin d'un test faites un `if`. Un tableau devra être alloué à sa déclaration, avec sa taille calculée **une fois pour toute**. **Pas** d'indices négatifs ou de «slices » de tableau. Une pénalité sera appliquée en cas de non respect. Si vous avez un doute, demandez à votre chargé de TD.
— Votre travail sera (en partie) évalué par un mécanisme automatique. Vous **devez** respecter les règles de **nommage** des fichiers et autres **consignes** qui vous sont données.
— **Sauf indications contraires**, vos programmes doivent **gérer** les cas **d'erreur** pouvant survenir.
— Le nom de la fonction **principale** permettant de lancer votre programme vous est **imposé**. Rien ne vous empêche d'écrire **d'autres** fonctions appelées par cette fonction principale. Bien souvent, ce sera même une bonne chose pour mieux structurer vos programmes.
— Lorsqu'il vous est demandé que votre programme réponde en affichant « **Yes** » ou « **No** », il ne doit **rien** afficher d'autre, et **pas** « Oui » ou « Yes. » ou « no » ou « La réponse est : no ».
Donc pensez à retirer vos affichages de test / debug.
— La **lisibilité** et l'efficacité / simplicité de vos programmes seront **prises en compte** dans l'évaluation.

— **À la fin de l'examen**, vous devrez créer une **archive** contenant **tous** les fichiers **sources** que vous avez écrits (`.py`). Le nom de cette archive devra avoir la structure suivante :
`nom_prenom.zip` ou `.tgz` (selon l'outil d'archivage que vous utilisez).
Par exemple, Harry Potter nommera son archive `potter_harry.zip`.
— Vous devrez **copier** cette archive dans répertoire de rendu se trouvant à `~pessaux/in101rendus/`
Par exemple, le magicien ci-dessus remettra son examen en invoquant la commande : `cp -vi potter_harry.zip ~pessaux/in101rendus/`.
— **N'oubliez pas** d'effectuer cette copie sinon nous devrons considérer que vous n'avez rien rendu !

— Le sujet comporte 7 pages et l'examen dure **2** heures.
— Le barème est **volontairement** approximatif.

1 Suppression des chiffres en double ($\sim 30\%$)

Écrivez une fonction `remd` qui prend en **argument** un entier n **positif ou nul** et **retourne** l'entier correspondant à n dans lequel les occurrences multiples de chiffres ont été supprimées, ne laissant chaque chiffre de n n'apparaître qu'une seule fois.

Attention : Par soucis de simplicité, il ne vous est **pas** demandé de gérer les entiers négatifs.

Attention : On souhaite faire disparaître les occurrences multiples de chiffres en considérant leur apparition dans l'écriture du nombre de **gauche à droite** (cf. dernier exemple ci-dessous).

Attention : On ne veut pas travailler sur des chaînes de caractères : c'est un problème traitant des entiers au travers d'entiers. Évitez-moi la solution naïve et inefficace consistant à faire un coup de `str ()` suivi d'un coup de `int ()`.

Nommage : Le fichier source de ce programme devra s'appeler `remdigit.py`. La fonction principale devra s'appeler `remd`.

Format de sortie : Pas d'affichage, valeur retournée.

Ex. tests :

- `remd (123)` $\longrightarrow$ 123
- `remd (52223)` $\longrightarrow$ 523
- `remd (1242353)` $\longrightarrow$ 12435 et **non** 14253 où les chiffres «en double » ont été supprimés en partant de la droite.

2 Affichage de damier ($\sim 15\%$)

On souhaite afficher un damier **carré** contenant des cases **carrées** à l'aide de caractères `'*'` et d'espaces. Le damier sera caractérisé par le nombre de cases du côté du damier et le nombre de caractères (`'*'` ou espaces) par côté d'une case. Le damier commencera **toujours** avec la case en haut à gauche étant **noire**. Un élément de case noire sera affiché avec un caractère `'*'`, un blanc avec un espace.

Écrivez une fonction `print_checker` prenant en argument (et dans **cet ordre**) la taille du côté du damier (en nombre de cases) et la taille du côté d'une case (en nombre de caractères) et `affiche` le damier.

Nommage : Le fichier source de ce programme devra s'appeler `checker.py`. La fonction principale devra s'appeler `print_checker`.

Format de sortie : Le damier avec un retour à la ligne final.

Rappel : Vous pouvez empêcher l'affichage d'un retour à la ligne lors d'un appel à `print` en utilisant la variante `end=''` :

```
>>> print ("foobar")
foobar
>>> print ("foobar", end='')
foobar>>>
```

Ex. tests :

— `print_checker (4, 2)` ⟶

```
**  **
**  **
  **  **
  **  **
**  **
**  **
  **  **
  **  **
```

— `print_checker (5, 3)` ⟶

```
***   ***   ***
***   ***   ***
***   ***   ***
   ***   ***
   ***   ***
   ***   ***
***   ***   ***
***   ***   ***
***   ***   ***
   ***   ***
   ***   ***
   ***   ***
***   ***   ***
***   ***   ***
***   ***   ***
```

— `print_checker (2, 3)` ⟶

```
***
***
***
   ***
   ***
   ***
```

3 Labyrinthe (~ 40%)

On souhaite calculer et afficher le nombre de mouvements possibles dans chaque case d'un labyrinthe. Un mouvement est un déplacement d'**une** case, **verticalement** ou **horizontalement**. Un labyrinthe est décrit par un fichier texte comportant sur sa première ligne sa largeur (en nombre de cases), sur sa seconde ligne sa hauteur (en nombre de cases), puis une suite de lignes de texte dans lesquelles un caractère `'X'` représente un mur et un espace représente une case vide.

```
9
7
XXXXXXXXX
X       X
X XXXXX X
X     X X
XXX X XXX
X   X   X
XXXXXXX X
```

Les coordonnées **commencent à 0** et l'origine du repère est en **haut à gauche**. Dans le labyrinthe ci-dessus, depuis la case (1, 1) 2 mouvements sont possibles (vers le bas et vers la droite). Depuis la case (3, 7) 1 seul mouvement est possible (vers le haut). Depuis la case (0, 0) aucun mouvement n'est possible (c'est un mur).

On ne souhaite afficher les nombres de mouvements **que pour** les cases où **au moins un** mouvement est **possible**.

Écrivez une fonction `how_many_moves` prenant en argument un **nom de fichier** et **affichant** les nombres de mouvements possibles dans les cases du labyrinthe contenu dans ce fichier, tel que spécifié ci-dessus. Vous pourrez (et devrez pour une meilleure structuration de votre programme) bien entendu définir d'autres fonctions pour effectuer les différents traitements du programme.

Votre programme devra gérer les cas d'erreur en levant une exception `BadFormat` que vous définirez.

Nommage : Le fichier source de ce programme devra s'appeler `movesmaze.py`. La fonction principale devra s'appeler `how_many_moves`.

Format de sortie : Une ligne par case ayant au moins un mouvement possible, avec entre crochets l'abscisse de la case, suivie entre crochets de son ordonnée, suivie de `:` suivi du nombre de mouvements suivi d'un retour à la ligne. L'ordre d'affichage devra se faire ordonnées croissantes et pour chaque ligne d'ordonnée donnée, par abscisses croissantes (autrement dit dans la même sens que l'ont lit un labyrinthe sur l'écran).

Rappel : Vous pouvez empêcher l'affichage d'espaces entre les différents éléments affichés par un appel à `print` en utilisant la variante `sep=''` :

```
>>> print (1, 2, 3)
1 2 3
>>> print (1, 2, 3, sep='')
123
```

Attention : Pour créer un tableau à 2 dimensions, **il ne faut pas** naïvement utiliser la construction `[ [0] * 2 ] * 3`. Ceci crée bien un tel tableau mais chaque « ligne » du tableau **partage le même tableau** `[0, 0]`. Autrement dit, modifier une case dans une « ligne » modifiera la même case dans toutes les lignes.

```
>>> t = [ [0] * 2 ] * 3
>>> t
[[0, 0], [0, 0], [0, 0]]
>>> t[0][0] = 100
>>> t
[[100, 0], [100, 0], [100, 0]]
```

À la place, construisez un premier tableau contenant des tableaux vides et effectuez ensuite une **boucle** pour initialiser chaque case de ce tableau avec un **nouveau** tableau pour former la seconde dimension.

```
>>> t = [ [] ] * 3
>>> i = 0
>>> while ........
...     t[i] = [0] * 2
...     ........
>>> t
[[0, 0], [0, 0], [0, 0]]
>>> t[0][0] = 10
>>> t
[[10, 0], [0, 0], [0, 0]]
```

Ex. tests : Trois fichiers d'exemple vous sont fournis (`maze1.dat`, `maze2.dat` et `maze3.dat`) mais rien ne vous empêche de créer les vôtres.

— `---- maze1.dat ----`

```
5
6
XXXXX
X   X
X   X
X   X
X   X
XXXXX
how_many_moves ("maze1.dat) ⟶
[1][1]:2
[1][2]:3
[1][3]:2
[2][1]:3
[2][2]:4
[2][3]:3
[3][1]:3
[3][2]:4
[3][3]:3
[4][1]:2
[4][2]:3
[4][3]:2
```

— `---- maze2.dat ----`

```
5
6
XXXXX
X X X
X   X
XXX X
X   X
XXXXX
how_many_moves ("maze2.dat) ⟶
[1][1]:1
[1][3]:1
[2][1]:2
[2][2]:2
```

```
[2][3]:3
[3][3]:2
[4][1]:1
[4][2]:2
[4][3]:2
```

4 Tableau de pairs et d'impairs ($\sim 25\%$)

Soit un tableau d'entiers contenant **par hypothèse** autant de nombres pairs que de nombres impairs. On souhaite réarranger en place (c'est-à-dire en modifiant les éléments **directement dans** le tableau) de telle manière que les valeurs paires (resp. impaires) soient à des indices pairs (resp. impairs).

Écrivez une fonction `arrange` qui prend en argument un tel tableau, le réarrange comme décrit précédemment et le **retourne** en résultat.

Attention : Des complexités temporaire et spatiale **linéaires** ($O(n)$) seront un **réel plus** dans l'évaluation de cet exercice.

Nommage : Le fichier source de ce programme devra s'appeler `aarray.py`. La fonction principale devra s'appeler `arrange`.

Format de sortie : Pas d'affichage, valeur retournée.

Ex. tests : Le tableau initial

```
[1 9 4 0 6 2 5 8 7 3]
```

pourra être réarrangé en

```
[0 9 4 1 6 5 2 7 8 3].
```

Notez que ce n'est pas la seule permutation possible, cela dépendra de votre algorithme.

—— Fin du sujet ——