IN 101 - Examen

22 octobre 2019

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`, `list`, `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. En cas de non respect, l'exercice concerné ne sera pas pris en compte. 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 / complexité 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, John Wick nommera son archive `wick_john.zip`.
— Vous devrez **copier** cette archive dans répertoire de rendu se trouvant à `~pessaux/in101rendus/`
Par exemple, le spécialiste de la gâchette ci-dessus remettra son examen en invoquant la commande : `cp -vi wick_john.zip ~pessaux/in101rendus/`.
— **N'oubliez pas** d'effectuer cette copie sinon nous devrons considérer que vous n'avez rien rendu !

— Le sujet comporte **6** pages et l'examen dure **2** heures.
— Le barème est **volontairement** approximatif.

Remarque : Vous pourrez débuter vos fichiers par une explication de la structure de votre algorithme, de votre raisonnement en quelques lignes ou insérer des commentaires dans le source. Ceci permettra d'expliquer la réflexion préalable à l'écriture de votre code. Si vous souhaitez utiliser des lettres accentuées dans vos commentaires, vous devez débuter votre fichier avec la ligne suivante :

```
# -*- coding: utf-8 -*-
```

1 Les chiffres d'un entier sont ordonnés ? ($\sim 20\%$)

Écrivez une fonction `incrdigits` qui prend en **argument** un entier n et détermine si les chiffres composant n, lus de **gauche à droite**, sont en ordre **croissant non strict**.

Attention : On ne veut pas travailler sur des chaînes de caractères : c'est un problème traitant des entiers au travers d'entiers. Il est donc inutile de soumettre une solution avec des `str ()`, `int ()`.

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

Format de sortie : **retourne** la valeur de vérité disant si les chiffres sont ordonnés de manière croissante.

Ex. tests :

- `incrdigits (12)` ⟶ `True`
- `incrdigits (143)` ⟶ `False`
- `incrdigits (777)` ⟶ `True`

2 Affichage de croix ($\sim 30\%$)

On souhaite afficher une croix à l'aide de caractères `'*'` et d'espaces. La croix est caractérisée par le nombre `n` de « points » composant chacune de ses branches autour de son centre (cf. exemples).

Écrivez une fonction `cross` prenant en argument le nombre de « points » `n` et `affichant` la croix correspondante.

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

Format de sortie : **affichage** de la croix 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 :

— `cross (1)` ⟶

```
* *
 *
* *
```

— `cross (2)` ⟶

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

— `cross (4)` ⟶

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

3 Histogrammes ($\sim 40\%$)

On représente un histogramme à barres **verticales** dans un fichier **texte** par une suite de caractères `'*'`, `'\n'` (retour à la ligne) et d'espaces. Une `'*'` représente une partie pleine d'une barre, un espace représente une partie vide. Chaque ligne est terminée par un retour à la ligne. Regardez les exemples qui suivent.

Une ligne peut éventuellement comporter des espaces finaux «inutiles» qui représentent des zones vides. Il est également possible de ne pas les faire figurer si plus aucune zone pleine ne les suit (cf. exemple 1).

La fin de l'histogramme est signalée par une ligne contenant le texte `END` en **début de ligne**, suivi d'un **retour à la ligne** final.

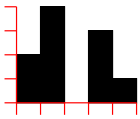
Chaque fichier **débute** par une ligne contenant un entier représentant le nombre de zones pleines que l'on s'attend à trouver dans l'histogramme. Si ce nombre ne correspond pas au nombre effectivement trouvé de zones pleines dans la suite du texte, l'histogramme est aussi ***incorrect***.

De plus, comme dans tout histogramme, une barre verticale ne doit pas comporter de «trou» : elle doit être «coloriée» complètement entre son extrémité haute et la base de l'histogramme. Dans le cas contraire, l'histogramme est ***incorrect***.

Exemples

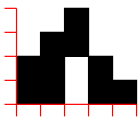
L'histogramme suivant (fichier texte à gauche et représentation graphique à droite) est ***correct*** puisqu'il comporte bien 10 zones pleines et ses barres verticales sont bien continues. Le signe ↵ permet de visualiser les retours à la ligne et ␣ les espaces. Notez que la première ligne (après le `10`) contient des espaces «inutiles» mais pas les suivantes.

```
10↵
␣*␣␣␣↵
␣*␣*↵
**␣*↵
**␣**↵
END↵
```

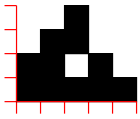


A contrario, les trois histogrammes suivants sont ***incorrects***. Le premier a sa barre du milieu qui n'atteint pas la base de l'histogramme. Le second a sa barre du milieu non continue. Le troisième comporte 14 zones pleines alors que le fichier texte indique qu'il devrait en comporter 5.

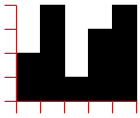
```
10↵
␣␣*↵
␣**↵
**␣*↵
**␣**↵
END ↵
```



```
11↵
␣␣*↵
␣**↵
**␣*↵
*****↵
END ↵
```



```
5↵
␣*␣␣*↵
␣*␣**↵
**␣**↵
*****↵
END ↵
```



Écrivez une fonction qui prend en argument un nom de fichier et vérifie si son contenu représente un histogramme ***correct***.

Attention : On ne vous demande pas de gérer les cas où le fichier comporte des caractères incorrects, ne débute pas par un entier ou ne termine pas par la chaîne `END`.

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

Format de sortie : **retourne** la valeur de vérité disant si le fichier contient un histogramme ***correct***.

Ex. tests : Quatre fichiers texte vous sont donnés que vous pouvez utiliser pour commencer à tester votre programme.

- `h0.txt` : histogramme ***incorrect*** ⟶ `False`
- `h1.txt` : histogramme ***correct*** ⟶ `True`
- `h2.txt` : histogramme ***incorrect*** ⟶ `False`
- `h3.txt` : histogramme ***correct*** ⟶ `True`

4 Point fixe d'un tableau ($\sim 20\%$)

On appelle « point fixe d'un tableau » un indice `i` tel que `t[i] == i`. Écrivez une fonction `fixpt` qui prend en argument un tableau `t` et détermine s'il **existe** un point fixe dans `t`. **Par hypothèse**, `t` contient des entiers **naturels** (donc, nombres positifs), **tous** différents et triés en ordre **croissant** strict.

Attention : la complexité de votre algorithme sera déterminante pour l'évaluation de cet exercice.

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

Format de sortie : **retourne** la valeur de vérité disant s'il existe un point fixe dans le tableau.

—— Fin du sujet ——