TP Python 5 - Boucles imbriquées

Avant de commencer ce TP, on contemplera les exemples suivants :

```
>>> for i in range(3):
...     for j in range(3):
...         print(i, j)
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
>>> for i in range(3):
...     for j in range(i):
...         print(i, j)
1 0
2 0
2 1
```

Rectangles et triangles

1) Écrire une fonction `rectangle` qui, recevant deux entiers naturels n et p, affiche un rectangle d'astérisques à n lignes et p colonnes.

```
>>> rectangle(3, 6)
******
******
******
```

On rappelle que pour afficher `*` (par exemple) sans passer à la ligne, on peut écrire `print('*', end="")` (ou `print('*', end=" ")` pour afficher `*` suivi d'un espace). Pour passer à la ligne sans rien afficher, il suffit d'écrire `print()`.

2) Écrire une fonction `triangle` qui, recevant un entier naturel n, affiche un triangle à n lignes comme ci-dessous.

```
>>> triangle(4)
*
**
***
****
```

3) Écrire une fonction `remplissage_lignes` qui, recevant deux entiers naturels non nuls n et p, affiche un tableau rectangulaire à n lignes et p colonnes contenant les premiers entiers naturels non nuls rangés en lignes comme dans l'exemple ci-dessous.

```
>>> remplissage_lignes(5, 3)
1 2 3
4 5 6
7 8 9
10 11 12
13 14 15
```

4) Écrire une fonction `remplissage_colonnes` qui, recevant deux entiers naturels non nuls n et p, affiche un tableau rectangulaire à n lignes et p colonnes contenant les premiers entiers naturels non nuls rangés en colonnes comme dans l'exemple ci-dessous.

```
>>> remplissage_colonnes(5, 3)
1 6 11
2 7 12
3 8 13
4 9 14
5 10 15
```

5) Écrire une fonction `floyd` qui, recevant un entier naturel non nul n, affiche un triangle à n lignes contenant les premiers entiers naturels non nuls comme dans l'exemple ci-dessous.

```
>>> floyd(5)
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
```

Tableaux bidimensionnels

On appellera ici tableau bidimensionnel une liste de listes de même longueur, par exemple `[[1, 2, 3], [4, 5, 6]]` .

On notera que si `t` est un tel tableau, alors `t[i]` est la $\texttt{i}^e$ ligne de `t` et donc `t[i][j]` est le $\texttt{j}^e$ terme de la $\texttt{i}^e$ ligne. Ainsi, si `t` est le tableau ci-dessus, alors `t[0][0]` vaut `1`, `t[1][0]` vaut `4`, `t[0][1]` vaut `2`, etc.

1) Écrire une fonction `table` qui, recevant un entier naturel non nul `n`, renvoie un tableau bidimensionnel représentant la table de multiplication de `1` à `n`.

```
>>> table(4)
[[1, 2, 3, 4], [2, 4, 6, 8], [3, 6, 9, 12], [4, 8, 12, 16]]
```

2) a) Écrire une fonction `est_dans` qui, recevant un objet `x` et un tableau bidimensionnel `t`, renvoie `True` si `x` appartient à `t` et `False` sinon. On n'utilisera pas `in`.

```
>>> est_dans(5, [[1, 2, 3], [4, 5, 6]])
True
>>> est_dans(0, [[1, 2, 3], [4, 5, 6]])
False
```

b) Combien de comparaisons sont effectuées par la fonction `est_dans` quand on l'applique à un tableau à n lignes et p colonnes ? On distinguera le meilleur et le pire des cas.

3) Écrire une fonction `max_2D` qui, recevant un tableau bidimensionnel, renvoie son plus grand élément. On n'utilisera pas la fonction `max`.

```
>>> max_2D([[2, 1, 5, 4, 1], [-2, 7, 3, 0, 12], [4, -11, 3, 2, -1]])
12
```

4) Écrire une fonction `coords_max_2D` qui, recevant un tableau bidimensionnel, renvoie les coordonnées de son plus grand élément.

```
>>> coords_max_2D([[2, 1, 5, 4, 1], [-2, 7, 3, 0, 12], [4, -11, 3, 2, -1]])
(1, 4)
```

Valeurs les plus proches dans une liste

1) Écrire une fonction `distance` qui, recevant deux nombres (entiers ou flottants), renvoie leur distance.

```
>>> distance(3, 8)
5
>>> distance(8, 3)
5
```

2) Écrire une fonction `valeurs_les_plus_proches` qui, recevant une liste d'entiers ou de flottants, renvoie les deux valeurs les plus proches de cette liste. On se contente ici d'une méthode brutale en testant tous les écarts possibles.

```
>>> valeurs_les_plus_proches([10, 50, 32, 141, 25, 106, 41])
(25, 32)
```

3) Déterminer le nombre de soustractions effectuées par la fonction précédente quand on l'applique à une liste de longueur n.

4) Écrire une fonction `valeurs_les_plus_proches_liste_triee` qui, recevant une liste triée d'entiers ou de flottants, renvoie les deux valeurs les plus proches de cette liste. On s'efforcera de minimiser le nombre d'opérations effectuées.

```
>>> valeurs_les_plus_proches_liste_triee([1, 10, 16, 23, 32, 51])
(10, 16)
```

5) Écrire une fonction `valeurs_les_plus_proches_2` qui, recevant une liste d'entiers ou de flottants, renvoie les deux valeurs les plus proches de cette liste en commençant par trier la liste (avec `sorted`) et en utilisant la fonction précédente.

6) On se propose de comparer la rapidité des deux fonctions.

Pour la question suivante on pourra utiliser la fonction `randint` du module `random`.

```
>>> from random import randint
>>> randint(1, 6)  # renvoie un entier choisi au hasard entre 1 et 6
3
```

a) Écrire une fonction `liste_aleatoire` qui, recevant deux entiers `n` et `N`, renvoie une liste comprenant `n` entiers aléatoires compris entre 1 et `N`.

```
>>> liste_aleatoire(10, 100)
[20, 85, 59, 7, 11, 63, 82, 4, 30, 64]
```

b) Créer une liste de 1000 entiers compris entre 1 et 10^9 et lui appliquer les fonctions `valeurs_les_plus_proches` et `valeurs_les_plus_proches_2`. Recommencer avec une liste de longueur 10000. Conclusion ?