Prise en main : Ouvrez un fichier `tp1.py` avec votre éditeur de texte favori. Pour intérpréter le code que vous y écrirez, il faudra exécuter la commande `$ python tp1.py` dans un terminal. Alternativement, si vous travaillez sous emacs (en python-mode), vous pouvez ouvrir un intepréteur via C-c C-p, et interpréter le contenu du fichier via C-c C-c directement depuis emacs.

On prendra la bonne habitude de tester au fil de l'eau chacune de ses fonctions.

Question 1 (chaînes de caractères, boucle *for*) Écrire une fonction `miroir` qui prend une chaîne de caractères, et en renvoie l'image miroir, c'est-à-dire la chaîne contenant les mêmes caractères, mais dans l'ordre inverse.

Pour éviter trop d'arithmétique sur les indices, on réécrira ensuite cette fonction avec une `range` au *pas* (le troisième argument de `range`) négatif.

Question 2 (listes) Écrire une fonction `entrelacement` qui attend deux listes `l1` et `l2` en argument, et qui renvoie la liste, qui comprend, de manière alternée tant que faire se peut, les éléments de `l1` et de `l2`.
Par exemple, `entrelacement([1,2,3],[4,5,6,7,8])==[1,4,2,5,3,6,7,8]`.
Astuce : on pourra utiliser la notation par tranche (`l[i:j]`) pour se simplifier le travail dans le cas où les listes sont de longueurs différentes.

Question 3 (classes, listes, exceptions) En utilisant uniquement `del` et `append`, implémenter une classe `File` qui représente une file. En particulier,

- une file sera initialement vide ;
- elle possèdera une méthode `ajouter()` qui ajoute un élément à la file,
- ainsi qu'une méthode `retirer()`, qui retire un élément à la file et le renvoie, ou lève une exception `FileVide` (qu'il faudra définir) si la file est vide.

On attend bien entendu que le comportement des ajouts/retraits soit de type *premier arrivé, premier sorti* (FIFO).

On se place maintenant en dehors de la classe `File`. En tant que clients de cette classe, on fera mine de ne pas en connaître les rouages internes : on ne pourra donc accéder à une `File` que via ses méthodes `ajouter()` et `retirer()`.
Avec cette contrainte en tête, écrire une fonction `vider()` qui dépile en intégralité une file passée en argument, en en affichant chaque élément. Il ne faudra bien sûr pas que cette fonction laisse échapper une exception `FileVide`.

Question 4 (itérateurs ☠) Reprendre la Question 1, mais sous forme itérable : on cherche donc à construire une classe `Miroir` dont chaque instance, qui accepte une chaîne de caractères à la création, soit un itérable qui épelle cette chaîne de caractères à l'envers.
Le comportement attendu est donc le suivant :

```
m = Miroir("bar")
for c in m:
    print(c)
# affiche "r" puis "a" puis "b"
```

Question 5 (itérateurs ☠) Nous allons maintenant généraliser la Question 2 à deux itérables quelconques, et plus seulement deux listes. De plus, on souhaite que le résultat soit lui-même un itérable.
Plus précisément, écrivez une classe itérable `Entrelacement`, dont le constructeur attend deux itérables `i1` et `i2`, et dont un itérateur émettra tantôt un élément de `i1`, tantôt un élément de `i2`, jusqu'à ce que les deux soient épuisés.
On garantira donc le comportement suivant :

```
e = Entrelacement("Python",[1,2,3])
for x in e:
    print(x)
# affiche, sur autant de lignes : P 1 y 2 t 3 h o n
```