Correction du DS 2

Julien Reichert

Exercice 1

1) Une chaîne de caractères est immutable en Python, l'instruction `seq2[n-1-i] = seq[i]` provoque alors une erreur [1].

2) La copie effectuée par la ligne `seq2 = seq` rend ces deux listes dépendantes l'une de l'autre. Ainsi, les affectations `seq2[n-1-i] = seq[i]` modifieront aussi les éléments correspondants de `seq`, donc dans la deuxième moitié de la boucle, ce sont les mêmes éléments dans l'autre sens qui seront lus, ce qui n'aura pas d'effet sur la liste. Par exemple, `[6,6,4]` deviendra dès le premier tour dans la boucle la liste composée de trois `6`.

3) Une possibilité est de remplacer l'instruction `seq2 = seq` par une instruction engendrant une copie indépendante, comme `seq[:]` ou `copy` (du module de même nom), voire `deepcopy` (idem) pour la sécurité (après tout, on ne sait pas ce qui va être fait avec le résultat, et la fonction doit être appelable sur des listes de listes, entre autres). Sinon, on peut aussi faire la fonction en place, à condition que la boucle soit réduite de moitié et qu'on procède à des échanges :

```
def miroir(seq):
    n = len(seq)
    for i in range(n//2):
        seq[i],seq[n-1-i] = seq[n-1-i],seq[i]
```

Cependant, puisque la spécification parlait de chaîne, cela ne suffit pas (on retrouve l'erreur de la question 1), donc une version non en place marchant sur des séquences immutables (mais à ne pas utiliser sur des listes, si possible) est :

```
from copy import deepcopy
def miroir2(seq):
    n = len(seq)
    res = seq[0:0] # pour que le type soit le même
    for i in range(n-1,-1,-1):
        res += deepcopy(seq[i:i+1]) # du même type que seq, car seq[i] serait un élément
                                    # et avec copie profonde tant qu'à faire
    return res
```

Attention, ce code ne marche que si l'opération d'addition est acceptée, de même que le slice. Cependant, les séquences de base ne posent pas de problème.

Pour des objets non mutables, une modification en place ne peut être envisagée simplement.

4) Tout simplement : `return seq[::-1]` ou, pour une version en place sur les listes, la méthode `reverse` (mais elle n'existe pas sur les chaînes de caractères [2]). Attention à ne pas écrire `seq[len(seq)-1:-1:-1]` comme pour un objet `range`, car ici le `-1` du milieu est compris comme le dernier élément, et du dernier élément inclus au dernier exclu... c'est vide.

1. `"TypeError: 'str' object does not support item assignment"`
2. Il existe une fonction nommée `reversed` qui retourne un objet contenant le miroir de son argument, mais il faut convertir l'objet en le bon type. Au passage, cet objet est un générateur, et il est dommage que le temps manque pour en parler.

Exercice 2

Pour simuler `arange(deb,fin,pas)` avec un `linspace`, on a besoin de calculer le nombre d'éléments, qui est exactement $\lceil \frac{\texttt{fin-deb}}{\texttt{pas}} \rceil$, et le dernier élément, qui est exactement `deb` plus le nombre d'éléments moins un (soit le nombre d'intervalles, en fait).

Pour simuler `linspace(deb,fin,nombre)` avec un `arange`, on a besoin de calculer le pas, qui est exactement $\frac{\texttt{fin-deb}}{\texttt{nombre-1}}$, et une nouvelle fin exclue, qui doit être strictement supérieure à `fin` mais inférieure ou égale à `fin + pas`. On peut prendre par exemple `fin + pas/42` (mais la division par `42` n'est pas optimisée...).

```
from math import ceil

def arange_to_linspace(deb,fin,pas):
    n = ceil((fin-deb)/pas)
    from numpy import linspace # importation ici afin que la fonction ne dépende pas
        # d'un changement ailleurs dans la façon d'importer
    return linspace(deb,deb+(n-1)*pas,n)

def linspace_to_arange(deb,fin,n):
    pas = (fin-deb)/(n-1)
    from numpy import arange
    return arange(deb,fin+pas/2,pas)
```

Exercice 3

Quoi qu'il en soit, on est amené à affecter `m*n` cases, donc une double boucle est asymptotiquement optimale. L'utilisation de la fonction `eye`, particulièrement adaptée pour résoudre simplement le problème (mais simplement pour l'utilisateur), masque une complexité bien plus mauvaise (même si on espère que le nombre d'opérations nécessaires pour engendrer une matrice nulle en-dehors d'une diagonale est optimisé pour ne pas être quadratique, ce qui n'est à mon avis pas le cas) à double titre, notamment par le nombre linéaire d'additions de matrices, donc un coût cubique.

```
import numpy
def ma_matrice(m,n):
    assert(m >= 0 and n >= 0) # préférable
    mat = numpy.array([[0]*n]*m)
    for i in range(m):
        for j in range(n):
            mat[i][j] = abs(j-i)
    return mat

def ma_matrice2(m,n):
    assert(m >= 0 and n >= 0)
    mat = numpy.eye(m,n,dtype=int)*0 # initialisation qui ne pose pas de souci pour m ou n nuls,
        # et on force le type entier pour la suite, pour faire bonne mesure
    for i in range(-m+1,n):
        mat += numpy.eye(m,n,i) * abs(i)
    return mat
```