```
"""
MP* / IE1 / 21-11-2016
Corrigé : propositions de codes
"""

## Exercice N°1
from random import randint

def Simul(p,n,N):
    # LongMoy = longueur moyenne des listes obtenues
    # ValMoy  = valeur moyenne des éléments des listes obtenues
    LongMoy, ValMoy = 0, 0
    # Boucle principale : nombre de simulations
    for i in range(N):
        L = []
        # Boucle secondaire : nombre de simulations
        for j in range(n):
            k = randint(1,2*p)
            if k%2 :
                L = []
            else:
                L.append(k)
        LongMoy += len(L)
        ValMoy  += sum(L)
    LongMoy /= N
    ValMoy  /= N
    return(LongMoy,ValMoy)

"""
N = int(input("Veuillez saisir le nombre total de simulations : "))
n = int(input("Veuillez saisir le nombre total de tirages par simulation : "))
P = 1
while P%2 :
    P = int(input("Veuillez saisir la borne supérieure (nombre pair) de l'intervalle
de tirage des entiers : "))
p = P//2
(a,b) = Simul(p,n,N)
print("Longueur moyenne des listes :",a)
print("Valeur moyenne des éléments des listes :",b)
"""

## Exercice N°2
from math import log

# Un code pour l'exponentiation rapide
def QuickExpo(a,n):
    if n == 0:
        return 1
    else:
        q,r = divmod(n,2)
        prod = QuickExpo(a,q)
        prod *= prod
        if r == 1:
            prod *= a
        return prod

# Fonction NewExpo_nr (non récursive)
def NewExpo_nr(a,n):
    res = 1
    while n > 0:
        new_exp = 2**(int(log(n,2)))
        res *= QuickExpo(a,new_exp)
        n -= new_exp
    return(res)

# Fonction NewExpo_nr (version récursive non terminale)
def NewExpo_r(a,n):
```

```
    if n == 0:
        return 1
    else:
        new_exp = 2**(int(log(n,2)))
        return(QuickExpo(a,new_exp) * NewExpo_r(a,n - new_exp))

# Fonction NewExpo_nr (version récursive terminale)
def NewExpo_rt(a,n,res=1):
    if n == 0:
        return res
    else:
        p = int(log(n,2))
        p2 = 2**p
        res *= QuickExpo(a,p2)
        return(NewExpo_rt(a,n-p2,res))

## Exercice N°3

class TriMat(list):
    # La classe TriMat est créée en tant que sous-classe de la classe "list".

    """
    Méthode __init__ : le seul attribut est appelé "content" et est une liste.
    Pour une matrice triangulaire supérieure, la longueur de la liste L[i] est
    égale à len(L)-i tandis que pour une matrice triangulaire inférieure, cette
    longueur est égale à i+1. Pour effectuer plusieurs tests "simultanément", on
    utilise la fonction "all".
    """
    def __init__(self,L):
        n =len(L)
        if all([len(L[i]) == n-i for i in range(n)]) or all([len(L[i]) == i+1 for i
in range(n)]):
            self.content = L
        else:
            raise ValueError("Argument incorrect ! Création impossible.")

    """
    Méthode ___repr___ : on renvoie simplement la liste self.content en la
    transformant en une chaîne de caractères.
    """
    def __repr__(self):
        return(str(self.content))

    """
    Méthode TriMat_test.
    Détermine si une matrice triangulaire est inférieure ou supérieure.
    Une matrice triangulaire est inférieure si sa première ligne ne contient
    qu'un seul coefficient (le premier). Dans ce cas, la première liste
    de la liste sera de longueur 1.
    """
    def TriMat_test(self):
        if len(self.content[0]) == 1 :
            return("Tinf")
        else:
            return("Tsup")

    """
    Méthode TriMat_det.
    Calcule le déterminant d'une matrice triangulaire
    (produit des éléments diagonaux).
    """
    def TriMat_det(self):
        n = len(self.content)
        d = 1
        # Cas d'une triangulaire supérieure
        # On va multiplier entre eux les éléments diagonaux, c'est à dire
        # les premiers éléments des listes dans self.
```

```
    # On positionne donc la variable indice à 0
    if self.TriMat_test() == "Tsup":
        indice = 0
    # Cas d'une triangulaire inférieure
    # On va multiplier entre eux les éléments diagonaux, c'est à dire
    # les derniers éléments des listes dans self.
    # On positionne donc la variable indice à -1
    else:
        indice = -1
    # On effectue le produit des éléments diagonaux
    for i in range(n):
        d *= self.content[i][indice]
    return(d)

"""
Méthode TriMat_transpose.
Calcule la transposée d'une matrice triangulaire.
"""
def TriMat_transpose(self):
    n = len(self.content)
    # Cas particulier
    if n == 1:
        return(TriMat(self.content))
    else:
        # Initialisation de TR
        TR = []
        # Cas d'une triangulaire supérieure
        if self.TriMat_test() == "Tsup":
            for i in range(n):
                TR.append([self.content[j][i-j] for j in range(i+1)])
        # Cas d'une triangulaire inférieure
        else:
            for i in range(n):
                TR.append([self.content[j][i] for j in range(i,n)])
        # Fin
        return(TriMat(TR))

"""
Méthode TriMat_inverse.
Calcule l'inverse d'une matrice triangulaire après avoir testé son
inversibilité.
"""
def TriMat_inverse(self):
    if self.TriMat_det() == 0:
        return(False)
    else:
        n = len(self.content)
        # Cas d'une triangulaire supérieure
        if self.TriMat_test() == "Tsup":
            # Initialisation de l'inverse TI
            TI = []
            # Eléments diagonaux
            for i in range(n):
                TI.append([1/self.content[i][0]])
            # Autres éléments (ligne par ligne)
            for i in range(n):
                # Pour une ligne (i) donnée, on calcule les éléments
                # d'indices (informatiques) de colonne de i+1 à n-1
                for k in range(i+1,n):
                    S = 0
                    for j in range(k-i):
                        S += TI[i][j]*self[i+j][k-j-i]
                    TI[i].append(-S/self[k][0])
            # Fin
        # Cas d'une triangulaire inférieure
        else:
            # On transpose self pour se ramener au cas précédent...
```

```
                T = self.TriMat_transpose()
                X = T.TriMat_inverse()
                TI = X[1].TriMat_transpose()
            return(True,TriMat(TI))

    """
    Surcharge de l'opérateur __mul__.
    On peut multiplier entre elles deux triangulaires supérieures ou deux
    triangulaires inférieures.
    Pour la multiplication par un entier ou un flottant, se reporter à la
    surcharge de l'opérateur __rmul__.
    """
    def __mul__(self,T):
        if type(T) != TriMat:
            raise TypeError("Le type du deuxième facteur n'est pas valable !")
        elif self.TriMat_test() == 'Tinf' and T.TriMat_test() == 'Tinf':
            # Produit lorsque les deux matrices sont triangulaires inférieures
            n = len(self.content)
            L = []
            for i in range(n):
                L2 = []
                for j in range(i+1):
                    S = 0
                    for k in range(j,i+1):
                        S += self.content[i][k]*T[k][j]
                    L2.append(S)
                L.append(L2)
            return(TriMat(L))
        else:
            # Produit lorsque les deux matrices sont triangulaires supérieures
            # On utilise la transposition...
            return((T.TriMat_transpose()*self.TriMat_transpose()).TriMat_transpose())

    """
    Surcharge de l'opérateur __rmul__.
    On peut multiplier une matrice triangulaire par un entier ou un flottant.
    """
    def __rmul__(self,x):
        if type(x) != int and type(x) != float:
            raise TypeError("Le premier facteur doit être un int ou un float !")
        else:
            return(TriMat([[self.content[i][j] * x for j in
range(len(self.content[i]))] for i in range(len(self.content))]))
```