TP6 : Simulation de couples de v.a.r. discrètes

Commencer par importer les bibliothèques suivantes dans chaque fichier **Python** utilisé :

```
import numpy as np
import numpy.random as rd
import matplotlib.pyplot as plt
```

On rappelle que l'on a alors accès aux commandes suivantes :

- `rd.random()`
  ↪ génère un nombre aléatoire dans $[0,1]$, *i.e.* simule une v.a.r. $X \hookrightarrow \mathcal{U}([0,1])$
- `rd.binomial(n,p,d)`
  ↪ simule d fois de manière indépendante une v.a.r. $X \hookrightarrow \mathcal{B}(n,p)$ (le paramètre d est optionnel)
  ↪ en choisissant $n=1$, simule d fois de manière indépendante une v.a.r. $X \hookrightarrow \mathcal{B}(p)$
- `rd.randint(a,b,d)`
  ↪ simule d fois de manière indépendante une v.a.r. $X \hookrightarrow \mathcal{U}(\llbracket a, b-1 \rrbracket)$ (le paramètre d est optionnel)
- `rd.geometric(p,d)`
  ↪ simule d fois de manière indépendante une v.a.r. $X \hookrightarrow \mathcal{G}(p)$ (le paramètre d est optionnel)
- `rd.poisson(l,p)`
  ↪ simule d fois de manière indépendante une v.a.r. $X \hookrightarrow \mathcal{P}(l)$ (le paramètre d est optionnel)
- `rd.choice(E,d)`
  ↪ simule un d-tirage équiprobable avec remise sur l'ensemble E où E est un tableau ou une liste (le paramètre d est optionnel)
- `rd.choice(E,d,replace=False)`
  ↪ simule un d-tirage équiprobable sans remise sur l'ensemble E où E est un tableau ou une liste (le paramètre d est optionnel)

I. Expériences en une étape

I.1. Un dé lancé n fois, le plus petit et le plus grand des résultats

Soit $n \geqslant 2$ un entier fixé. On considère un dé à 6 faces équilibré que l'on lance n fois. On note, pour tout $k \in \llbracket 1, n \rrbracket$, X_k le résultat du k^{e} lancer. On note enfin $Y = \min(X_1, \ldots, X_n)$ et $Z = \max(X_1, \ldots, X_n)$.

► On considère dans cette question le cas particulier $n = 2$. Ecrire une fonction **Python** `simulYZ_2` qui renvoie une liste `[Y,Z]` contenant le résultat de la simulation de Y et de Z.

```
1 def simulYZ_2():
2     X1 = rd.randint(1,7)
3     X2 = rd.randint(1,7)
4     return [min(X1,X2), max(X1,X2)]
```

► On revient maintenant au cas général. Compléter la fonction **Python** suivante pour qu'elle renvoie une liste `[Y,Z]` contenant le résultat de la simulation de Y et de Z.

```
def simulYZ(n):
    X = rd.randint(1,7)
    Y = __X__
    Z = __X__
    for k in __range(n-1):__
        X = rd.randint(1,7)
        if X < Y:
            Y = __X__
        elif __X > Z:__
            Z = __X__
    return [Y,Z]
```

I.2. ECRICOME 2022

On dispose de trois urnes U_1, U_2 et U_3, et d'une infinité de jetons numérotés 1, 2, 3, 4, ...
On répartit un par un les jetons dans les urnes : pour chaque jeton, on choisit au hasard et avec équiprobabilité une des trois urnes dans laquelle on place le jeton. Le placement de chaque jeton est indépendant de tous les autres jetons, et la capacité des urnes en nombre de jetons n'est pas limitée.
Pour tout entier naturel n non nul, on note $X_n^{(1)}$ (respectivement $X_n^{(2)}$, $X_n^{(3)}$) le nombre de jetons présents dans l'urne 1 (respectivement l'urne 2, l'urne 3) après avoir réparti les n premiers jetons.
Soit Y le nombre de jetons placés lorsque, pour la première fois, deux urnes exactement sont occupées par au moins un jeton.
Soit Z le nombre de jetons placés lorsque, pour la première fois, les trois urnes contiennent chacune au moins un jeton.

► Compléter la fonction **Python** suivante pour qu'elle simule les n premiers placements des jetons et qu'elle renvoie un tableau `numpy` contenant le résultat des simulations de $X_n^{(1)}$, $X_n^{(2)}$ et $X_n^{(3)}$.

```
def remplissage_urnes(n):
    R = __np.zeros(3)__
    for k in __range(n):__
        indice = __rd.randint(0,3)__
        R[indice] += 1
    return R
```

► On note a (resp. b et c) le nombre de jetons contenus dans l'urne 1 (resp. 2 et 3) à un moment donné de l'expérience. Montrer que :

- Au moins une urne est vide si et seulement si $abc = 0$
- Au moins deux urnes sont vides si et seulement si $(a+b)(b+c)(c+a) = 0$

D'une part : $abc = 0 \iff a = 0$ ou $b = 0$ ou $c = 0 \iff$ l'urne 1 est vide ou l'urne 2 est vide ou l'urne 3 est vide $\iff$ au moins une urne est vide

D'autre part : $(a+b)(b+c)(c+a) = 0 \iff a+b = 0$ ou $b+c = 0$ ou $c+a = 0 \iff a = b = 0$ ou $b = c = 0$ ou $c = a = 0 \iff$ les urnes 1 et 2 sont vides ou les urnes 2 et 3 sont vides ou les urnes 3 et 1 sont vides $\iff$ au moins deux urnes sont vides

▶ Compléter la fonction **Python** suivante pour qu'elle simule l'expérience jusqu'à-ce que les trois urnes contiennent au moins un jeton et qu'elle renvoie une liste $[Y, Z]$ contenant le résultat des simulations de Y et Z.

```
def simulYZ():
    R = __np.zeros(3)__
    rang = 0
    liste = []
    while ____(R[0] + R[1]) * (R[1] + R[2]) * (R[2] + R[0]) == 0:____
        rang = rang + 1
        indice = __rd.randint(0,3)__
        R[indice] += 1
    __liste.append(rang)__
    while __R[0] * R[1] * R[2] == 0:__
        rang = rang + 1
        indice = __rd.randint(0,3)__
        R[indice] += 1
    __liste.append(rang)__
    return liste
```

II. Expériences en deux étapes

II.1. Un dé lancé un nombre aléatoire de fois

On lance une pièce de monnaie équilibrée une infinité de fois. On note N le rang du premier `Pile` obtenu. Si le premier `Pile` a été obtenu au rang n, on lance ensuite n fois un dé à 6 faces équilibré. On note alors X le nombre de 6 obtenus et S la somme des résultats obtenus.

▶ Ecrire une fonction **Python** `simulNX()` qui renvoie une liste `[N,X]` contenant le résultat de la simulation de N et de X.

```
def simulNX():
    N = rd.geometric(1/2)
    X = rd.binomial(N, 1/6)
    return [N,X]
```

▶ Ecrire une fonction **Python** `simulNS()` qui renvoie une liste `[N,S]` contenant le résultat de la simulation de N et de S. On calculera S à l'aide d'une boucle `for`.

```
def simulNS():
    N = rd.geometric(1/2)
    S = 0
    for k in range(N):
        S = S + rd.randint(1,7)
    return [N,S]
```

▶ Ecrire une nouvelle fonction **Python** `simulNS_v2()` qui renvoie une liste `[N,S]` contenant le résultat de la simulation de N et de S. On calculera cette fois-ci S à l'aide de la fonction `np.sum`.

```
def simulNS_v2():
    N = rd.geometric(1/2)
    S = np.sum([rd.randint(1,7) for k in range(N)])
    return [N,S]
```

II.2. L'embarras du choix parmi les urnes

Soit $n \geqslant 2$ un entier fixé. On dispose de n urnes, numérotées de 1 à n. Pour chaque $k \in [\![1, n]\!]$, l'urne k est composée de k boules numérotées de 1 à k. On choisit une urne au hasard puis on tire une boule au hasard dans cette urne. On note

- X_n la v.a.r. égale au numéro de l'urne choisie
- Y_n la v.a.r. égale au numéro de la boule tirée

▶ Ecrire une fonction `simulXY` qui
- prend en argument un entier $n \geqslant 2$
- renvoie une liste `[X,Y]` contenant le résultat de la simulation de X_n et de Y_n

```
def simulXY(n):
    X = rd.randint(1,n+1)
    Y = rd.randint(1,X+1)
    return [X,Y]
```

II.3. Une expérience de savant fou

Un savant fou s'ennuie dans sa tour isolée et dispose d'une pièce usée qui tombe sur `Pile` avec probabilité $p \in]0, 1[$. Il décide de se lancer dans une expérience aléatoire dont le protocole est décrit ci-dessous :

1. Il lance la pièce jusqu'à-ce qu'elle tombe sur `Pile`. On note N la variable aléatoire égale au rang du premier `Pile`.

2. Si la pièce est tombée sur `Pile` pour la première fois au n^{e} lancer, alors il remplit une urne de la manière suivante. Pour tout entier $k \in [\![1, n]\!]$,
- si k est pair, alors il lance un dé à 6 faces. Il place ensuite dans l'urne autant de jetons numérotés k que le résultat indiqué sur le dé.
- si k est impair, alors il lance un dé à 8 faces. Il place ensuite dans l'urne autant de jetons numérotés k que le résultat indiqué sur le dé.

3. Une fois l'urne remplie, il procède à n tirages successifs et sans remise dans cette urne, note les résultats obtenus et fait leur somme. On note S la variable aléatoire égale au résultat obtenu.

On souhaite simuler le couple (N, S). Pour cela, on rappelle que :

× Si k est un entier, la commande `k % 2 == 0` renvoie `True` si k est pair et renvoie `False` si k est impair.

× Si `L` est une liste **Python** possédant plus de n éléments, alors la commande `rd.choice(L, n, replace = False)` simule une succession de n tirages sans remise dans cette liste et renvoie une liste contenant les résultats successifs.

▶ Compléter la fonction **Python** suivante pour qu'elle renvoie une liste `[N,S]` contenant le résultat de la simulation de N et de S.

```
def savantfou(p):
    N = rd.geometric(p)
    urne = []
    for k in range(1, N+1):
        if k % 2 == 0:
            urne = urne + [k]*rd.randint(1,7)
        else:
            urne = urne + [k]*rd.randint(1,9)
    resultats = rd.choice(urne, N, replace=False)
    S = np.sum(resultats)
    return [N,S]
```

II.4. Tests de dépistage

Chaque jour, le nombre de personnes subissant un test de dépistage dans une certaine pharmacie est modélisé par une v.a.r. X suivant une loi de Poisson de paramètre $m = 5$. Chaque test a une certaine probabilité $p = \frac{1}{10}$ d'être positif et les tests sont indépendants les uns des autres. On note N le nombre de tests positifs un jour donné.

▶ Ecrire une fonction **Python** nommée `simulN()` qui simule la v.a.r. N.

```
def simulN():
    X = rd.poisson(5)
    return rd.binomial(X,1/10)
```

▶ Créer un échantillon de taille 1000 de cette v.a.r. et le comparer, avec un histogramme, à un échantillon de même taille d'une loi de Poisson de paramètre $mp = \frac{1}{2}$. Que peut-on conjecturer ?

```
L = []
for k in range(1000):
    L.append(simul_N())
P = rd.poisson(0.5,1000)
plt.hist(L, [k for k in range(10)])
plt.hist(P, [k for k in range(10)])
plt.show()
```

On peut conjecturer que $N \hookrightarrow \mathcal{P}\left(\frac{1}{2}\right)$.