Informatique - TP 4 **Simulation de variables aléatoires - 2** 29 septembre - 3 octobre 2025

Code de partage avec Capytale : ef48-7365099 **Corrigé**

On utilisera (entre autres) la bibliothèque `numpy.random` sous l'alias `rd`

Exercice 1

Le score d'un joueur joueur lors d'un lancer de fléchette est modélisé par une variable aléatoire X telle que $X(\Omega) = \{0, 2, 5, 10\}$ et

$$P(X = 0) = \frac{1}{5}, \quad P(X = 2) = \frac{1}{2}, \quad P(X = 5) = \frac{1}{5} \quad \text{et} \quad P(X = 10) = \frac{1}{10}$$

1. Calculer l'espérance et la variance de X

   De manière immédiate en appliquant respectivement la définition et la formule de Kœnig-Huygens, on obtient $E(X) = 3$ et $V(X) = 8$

2. Découper l'intervalle $[0, 1]$ en quatre intervalles où pourrait tomber un nombre aléatoire généré avec `rd.random()` dans le but de simuler X. Écrire alors une fonction `simulX()` qui renvoie une simulation de la variable X

   Puisque la fonction `rd.random()` renvoie de manière aléatoire un nombre dans l'intervalle $[0, 1[$, il y aura respectivement une chance sur 5, sur 2, sur 5, sur 10 que son résultat se trouve dans les intervalles $\left[0, \frac{1}{5}\right[$, $\left[\frac{1}{5}, \frac{7}{10}\right[$, $\left[\frac{7}{10}, \frac{9}{10}\right[$ ou $\left[\frac{9}{10}, 1\right[$
   On le décrit avec les conditions suivantes dans le programme (on utilise `elif` qui signifie « sinon si » et qui permet de ne pas répéter les commandes précédentes.

```
def simulX():
    a=rd.random()
    if a<1/5:
        return 0
    elif a<7/10:
        return 2
    elif a<9/10:
        return 5
    else :
        return 10
```

3. Que fait le programme suivant ?

```
def sampleX(n):
    S = []
    for k in range(n):
        S.append(simulX())
    return S
print(np.mean(sampleX(1000)))
```

Ce programme réalise 1 000 simulations de la variable aléatoire X grâce à la fonction de la question **2.** (et stocke les valeurs dans une liste) et en fait la moyenne. On s'attend à trouver un résultat proche de la valeur théorique de l'espérance de X, qui est 3, comme nous l'avons vu à la question **1.**

Exercice 2

On désigne par n un entier naturel supérieur ou égal à 2. On lance n fois une pièce équilibrée (c'est-à-dire donnant « Pile » avec la probabilité $\frac{1}{2}$), les lancers étant supposés indépendants.
On note Z la variable aléatoire qui vaut 0 si l'on n'obtient aucun Pile pendant ces n lancers et qui, dans le cas contraire, prend pour valeur le rang du premier pile.

1. Compléter la fonction suivante permettant de simuler la variable aléatoire Z

```
def simulZ(n) :
    for i in range(1,n+1):
        if  rd.binomial
    (1,0.5) >0 :
            return i
    return 0
```

On répète, avec la boucle, n lancers qui peuvent être modélisées par `rd.binomial(1,0.5)` (loi de Bernoulli), `rd.randint(1,3)` ou `rd.random()` (avec l'intervalle $\left[0, \frac{1}{2}\right[$). Il semblerait que la fonction « renvoie » (du fait du `return`) un résultat à chaque « Pile ». En fait dès lors qu'un `return` est effectué, lors du premier « Pile » donc, la fonction ne renvoie plus de résultat. Et si aucun résultat n'a été renvoyé lors de la boucle, ce sera le 0, par défaut, à l'issue.

2. Réécrire cette fonction à l'aide d'une boucle `while`.

   Ici on n'effectue pas les n lancers, puisqu'on s'arrête dès que « Pile » a été obtenu, mais cela correspond pour la variable aléatoire. Il faut préciser des conditions, à la fois dans la boucle, si « Pile » n'a pas été obtenu au bout de n lancers, et à l'issue car si le compteur contient $n+1$ c'est que « Pile n'a jamais été obtenu » et il faut alors que la fonction renvoie 0

```
def simulZ2(n):
    i=1
    while rd.randint(1,3)==1
    and i<=n:
        i=i+1
    if i==n+1 :
        return 0
    else :
        return i
```

Exercice 3

On dispose de $n \geqslant 1$ 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 dans cette urne et on note X_n la variable aléatoire qui prend la valeur du numéro de la boule piochée.

Ecrire une fonction `simulX(n)` qui simule X_n

On simule ici successivement deux variables aléatoires suivant des lois uniformes, respectivemnet sur $[\![1, n]\!]$ et sur $[\![1, k]\!]$ où k est le résultat du premier tirage.

```
def simulX(n) :
    a=rd.randint(1,n+1)
    return rd.randint(1,a+1)
```

Exercice 4

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

1. Ecrire une fonction permettant simuler la variable X

   Nous sommes à nouveau confrontés à la succession de deux variables aléatoires suivant des lois usuelles, en l'occurence, une loi de Poisson, puis une loi binomiale (où le test positif représente le succès).

```
def simulX() :
    a=rd.poisson(5)
    return rd.binomial(a,0.1)
```

2. Créer un échantillon de taille 1 000 de cette variable aléatoire et le comparer, avec un diagramme en bâtons, à un échantillon de même taille d'une loi de Poisson de paramètre $\lambda \times p = \frac{1}{2}$

   Emettre une conjecture sur la loi de X

   Rappel : on pourra utiliser la fonction `plt.bar` pour le diagramme en bâtons.

On crée dans un premier temps une liste contenant 1 000 réalisations de la variable aléatoire X grâce à la fonction précédente, que l'on transforme en une liste y de fréquences pour les valeurs allant de 0 (le minimum possible pour la loi de Poisson) au maximum obtenu dans la simulation. Enfin, on une deuxième liste pour la représentation contenant les valeurs théoriques de la loi de Poisson indiquée (il faut une fonction factorielle pour cela).
Enfin on représente les deux diagrammes en bâtons, en ajustant les couleurs et la largeur pour faciliter la comparaison (on rappelle que le diagramme en bâtons requiert une liste d'abscisses, qui dépend du maximum obtenu à la simulation.
Les deux diagrammes sont très proches et il semblerait donc que la « succession » d'une loi de Poisson de paramètre 5 puis d'une loi binomiale dont le paramètre de succès est $\frac{1}{10}$ soit une loi de Poisson de paramètre $5 \times \frac{1}{10} = \frac{1}{2}$

```
x=[simulX() for i in range
    (1000)]
m= max(x)
y = np.zeros(m+1)
for i in range(0,m+1):
    y[i]=x.count(i)
y=y/1000

def fact(n):
    if n==0 :
        return 1
    else :
        return n*fact(n-1)

plt.close()
a=[i for i in range(0,m+1)]
z=[(0.5**i/fact(i))*np.exp
    (-0.5) for i in range(0,m
    +1)]
plt.bar(a,z)
plt.bar(a,y,width=0.4)
plt.show()
```

3. En commençant par expliciter $P_{[N=n]}(X = k)$, démontrer le résultat précédemment conjecturé.
C'est un exercice en soi, à tester !

Exercice 5

Une urne contient des boules numérotées de 1 à n, indiscernables au toucher. On effectue alors des tirages sans remise dans l'urne et on note, pour tout $k \in [\![1, n]\!]$, X_k la variable aléatoire qui prend la valeur de la boule obtenue au $k^{\text{ième}}$ tirage.

1. Ecrire une fonction Python `simulX` qui prend en argument deux entiers naturels non nuls k et n, et qui renvoie une simulation de X_k
Il s'agit à nouveau d'une succession de loi uniformes, mais la difficulté ici est de retirer progressivement des valeurs dans la liste puisque le tirage est sans remise, donc la composition de l'urne change. On va le faire avec la commande `del L[i]` qui retire la valeur d'indice i dans une liste (donc la $i+1^{\text{ième}}$) et en raisonnant sur l'indice pour le tirage aléatoire. Le `b` ne sert qu'à la dernière étape (pour conserver la dernière valeur choisie et retirée inutilement).

```
def simulX(k,n):
    L=[i for i in range(1,n
    +1)]
    for i in range (0,k-1) :
        a=rd.randint(0,n-i)
        b=L[a]
        del L[a]
    return b
```

2. On exécute le programme suivant, expliquer ce qu'il fait (la ligne 7 en particulier).

```
n=6
k=4
echantillon = [simulX(k,n) for j in range(1000) ]
freq = np.zeros(6)

for j in range(1000):
    freq[echantillon[j]-1] = freq[echantillon[j]-1]+1

freq = freq/1000
```

```
plt.bar([j for j in range(1,n+1)],freq)
plt.show()
```

Puis émettre une conjecture sur la loi de X_n

Ce programme réalise 1 000 simulations de la variable aléatoire et calcule (lignes 6 et 7) la fréquence d'apparition de chaque valeur en parcourant la liste et en incrémentant le compteur multiple (qui contient 6 zéros initialement). Une subtilité, quand la valeur i est rencontrée, il faut incrémenter la valeur d'indice $i-1$ du compteur puisque les 1 sont comptés dans `freq[0]`, de même pour $2, \dots 6$
On obtient un diagramme en bâtons qui semble représenter une loi uniforme. Cela est cohérent avec l'idée que les nombres choisis aléatoirement (retirés puis conservé pour le dernier) le sont de manière équiprobable et tous les entiers entre 1 et n ont donc la même probabilité de sortir à la fin.