TP 4

Simulation de variables aléatoires discrètes

Exercice 1

Soient n et m deux entiers tels que $n < m$. On rappelle que si $U \hookrightarrow \mathscr{U}([0,1[)$, alors $V = n + \lfloor (m-n)U \rfloor \hookrightarrow \mathscr{U}([\![n, m-1]\!])$.

1. Écrire une fonction `uniforme(n,m)` simulant la loi $\mathscr{U}([\![n, m-1]\!])$.
2. Compléter la fonction suivante afin qu'elle renvoie un vecteur contenant N réalisations indépendantes de la loi $\mathscr{U}([\![n, m-1]\!])$.

```
def Uniforme(n,m,N):
    v = np.zeros(N)
    for k in range(N):
        v[k] = ...
    return v
```

1. On peut proposer la fonction suivante :

```
def uniforme(n,m):
    u = rd.random()
    v = n + np.floor((m-n)*u)
    return v
```

2. On procède ainsi :

```
def Uniforme(n,m,N):
    v = np.zeros(N)
    for k in range(N):
        v[k] = uniforme(n,m)
    return v
```

Exercice 2

1. Pour simuler une loi de Bernoulli, on procèdera comme évoqué plus haut : on tire au hasard un nombre dans l'intervalle $[0,1]$ avec la fonction `random()`. On a alors deux cas :
   - si `rd.random()<=p`, ce qui arrive avec une probabilité de p, on renvoie la valeur 1 ;
   - si `rd.random()>p`, ce qui arrive avec une probabilité de $1-p$, on renvoie 0.

Compléter la fonction suivante afin qu'elle simule une loi de Bernoulli de paramètre p.

```
def bernoulli(p):
    u = 0 ;
    if .........
        u = ...
    return u
```

2. Écrire une fonction `Bernoulli(p,N)` renvoyant un vecteur contenant N réalisations indépendantes de la loi $\mathscr{B}(p)$.

1. On peut procéder ainsi :

```
def bernoulli(p):
    u = 0
    if rd.random()<=p:
        u = 1
    return u
```

2. On procède comme précédemment :

```
def Bernoulli(p,N):
    v = np.zeros(N)
    for k in range(N):
        v[k] = bernoulli(p)
    return v
```

Exercice 3

1. Écrire une fonction `binomiale(n,p)` simulant la loi $\mathscr{B}(n,p)$.
2. Écrire une fonction `Binomiale(n,p,N)` donnant un échantillon de taille N de la loi $\mathscr{B}(n,p)$

1. On peut procéder comme suit, en utilisant la fonction `Bernoulli` définie précédemment.

```
def binomiale(n,p):
    u = Bernoulli(p,n)
    return np.sum(u)
```

Une autre possibilité est d'utiliser la commande `np.sum(rd.random(n)<=p)`.

```
def binomiale(n,p):
    return np.sum(rd.random(n)<=p)
```

2. Toujours la même méthode.

```
def Binomiale(n,p,N):
    v = np.zeros(N)
    for k in range(N):
        v[k] = binomiale(n,p)
```

```
    return v
```

Exercice 4

1. Écrire une fonction `geom(p)` simulant la loi $\mathscr{G}(p)$.
2. Écrire une fonction `Geom(p,N)` afin de simuler un échantillon de taille N de la loi $\mathscr{G}(p)$.
3. Simuler un échantillon de taille $N = 10000$ de la loi $\mathscr{G}(0.2)$, puis vérifier que la valeur moyenne de cet échantillon est cohérente avec ce qu'on attend.

1. On peut procéder comme suit :

```
def geom(p):
    u = 1
    while rd.random()>p:
        u=u+1
    return u
```

On peut remplacer la commande `rd.random()>p` par `bernoulli(p)==0`.

2. Toujours pareil.

```
def Geom(p,N):
    v = np.zeros(N)
    for k in range(N):
        v[k] = geom(p)
    return v
```

3. On peut procéder ainsi :

```
>>> u = Geom(0.2,10000)
>>> np.mean(u)
```

On obtient (par exemple) `5.0013`. Cela correspond approximativement à ce qu'on attend. En effet, l'espérance d'une variable suivant une loi géométrique de paramètre $p = 0,2$ est $\frac{1}{p} = 5$.

Exercice 5

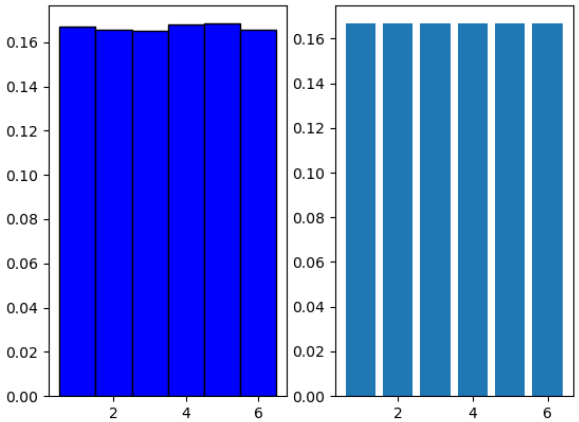
On se donne le code suivant :

```
# Echantillon
x = Uniforme(1,7,100000)

#DeB des fréquences
c = np.arange(0.5,7)
plt.subplot(1,2,1)
plt.hist(x,c,density='True',edgecolor='k',
color='blue',label="DeB des fréq")
#DeB des probas théoriques
u = np.arange(1,7)
v = (1/6)*np.ones(6)
plt.subplot(1,2,2)
plt.bar(u,v,label="DeB des prob
theo")

plt.show()
```

Exécuter ce code et interpréter le graphique ainsi obtenu. En particulier, à quoi correspond le vecteur u ?

Le script commence par créer un échantillon de taille 100000 à l'aide de la fonction `uniforme` définie au début du TP, avec $n = 1$ et $m = 7$. Elle trace ensuite le diagramme en bâton des fréquences empiriques de cet échantillon. Pour cela, on définit les classes dans la variable `c`. Elle trace enfin le diagramme en bâton des probabilités théoriques. En particulier, le vecteur `v` contient `[1/6,1/6,1/6,1/6,1/6,1/6]`, qui correspond bien aux probabilités théoriques d'une loi uniforme sur $[\![1, 6]\!]$.

En exécutant le code proposé, on obtient la représentation graphique suivante :



Les diagrammes en bâtons étant très similaires, on peut en conclure que la fonction `uniforme` renvoie bien ce que l'on souhaite : des entiers de 1 à 6 avec une fréquence de 1/6 environ. La simulation de la loi $\mathscr{U}([\![1, 6]\!])$ à l'aide de la fonction `uniforme(1,7)` est bien pertinente.

Exercice 6 (★★ - Simulation de la loi binomiale)

1. On considère les instructions suivantes :

```
>>> c = np.ones(n+1)
>>> c[1:(n+1)] = np.cumprod(np.arange(n,0,-1)/np.arange(1,n+1))
```

   Que contient le vecteur `c` à la suite de ces instructions ? Expliquer.

   On pourra commencer par exécuter ces instructions pour `n = 3`, `n = 4`.

2. À l'aide d'un diagramme en bâtons, tester la simulation de la loi $\mathscr{B}(10, 0.2)$ obtenue à l'aide de la fonction `Binomiale`.

1. En exécutant ces commandes pour $n = 3$, on obtient `c = np.array([1,3,3,1])`, et pour $n = 4$, `c = np.array([1,4,6,4,1])`. On reconnait la liste des coefficients binomiaux $\binom{n}{k}$. Tentons de l'expliquer.

   `np.arange(n,0,-1)` est le vecteur `[n,n-1,...,2,1]`. Et donc :

   `np.arange(n,0,-1)/np.arange(1,n+1)`

   est le vecteur `[n/1,(n-1)/2,...,2/(n-1),1/n]`. Par conséquent, si on applique à ce

vecteur la fonction `np.cumprod`, le vecteur obtenu a pour k-ème composante :

$$\frac{n}{1} \times \cdots \times \frac{n-k+1}{k} = \frac{n(n-1)\dots(n-k+1)}{k!} = \frac{n!}{k!(n-k)!}\binom{n}{k}.$$

Il nous manque alors dans le vecteur le cas où $k = 0$, qu'on ajoute à l'aide de la commande `np.ones(n+1)`, dont la 0-ième composante vaut 1.

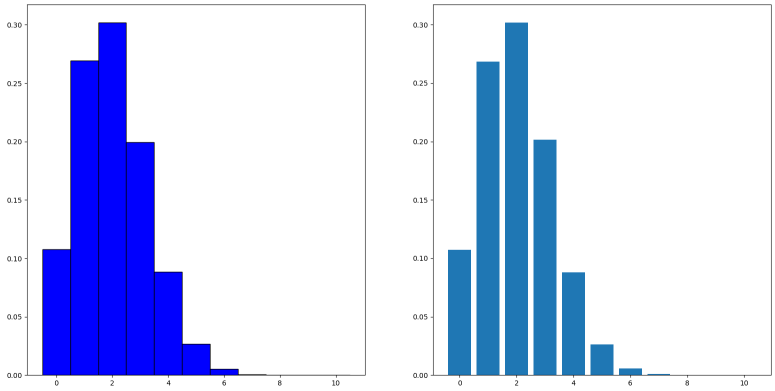
2. On commence par générer un échantillon de taille $N = 100000$ grâce à la fonction `Binomiale`, et tracer l'histogramme (diagramme en bâtons) des fréquences empiriques de l'échantillon.

```
# Echantillon
p = 0.2
n = 10
N = 100000
x = Binomiale(n,p,N)

#DeB des fré quences
c = np.arange(-0.5,11)
plt.subplot(1,2,1)
plt.hist(x,c,density='True',edgecolor='k',color='blue',
label="DeB des fréq")
```

On trace ensuite le diagramme en bâtons des probabilités théoriques. Pour cela, on construit un vecteur `v` dont la k-ème composante est $\binom{10}{k}p^k(1-p)^{10-k}$.

```
#DeB des probas thé oriques
s = np.arange(0,11,1)
c = np.ones(n+1)
c[1:(n+1)] = np.cumprod(np.arange(n,0,-1)/np.arange(1,n
+1))
u = p**np.arange(0,11)
v = (1-p)**np.arange(10,-1,-1)
t = c*u*v

plt.subplot(1,2,2)
plt.bar(s,t,label="DeB des prob theo")
plt.show()
```

On obtient la représentation graphique suivante.



On observe que les diagrammes en bâtons sont pratiquement identique. Il est donc pertinent de simuler la loi $\mathscr{B}(10, 0.2)$ à l'aide de la fonction `binomiale`.

Exercice 7 (★★ - Simulation d'une loi géométrique à partir d'une loi exponentielle)
Soit $\lambda \in]0, +\infty[$. On a montré en TD que si $X \hookrightarrow \mathscr{E}(\lambda)$, alors $Y = \lfloor X \rfloor + 1 \hookrightarrow \mathscr{G}(1 - e^{-\lambda})$.

1. Écrire une fonction `Geom2(p,N)` simulant un échantillon de taille N de la loi $\mathscr{G}(p)$. On utilisera pour cela la fonction `rd.exponential(1/λ,N)` pour simuler un échantillon de taille N de la loi exponentielle $\mathscr{E}(\lambda)$.

2. On souhaite comparer avec la loi théorique les simulations d'une variable Y suivant une loi $\mathscr{G}(0.3)$ obtenues à l'aide des fonctions `Geom` et `Geom2`. Pour ce faire, simuler $N = 10000$ réalisations d'une telle variable à l'aide de ces deux fonctions, puis tracer les histogrammes correspondants ainsi que le diagramme en bâtons des probabilités théoriques.

   On fait remarquer que Y ne prend qu'exceptionnellement des valeurs au-delà de 20, si bien qu'on pourra considérer des histogrammes pour des classes se limitant à la valeur 20. On pourra afficher les trois diagrammes de front (pour mieux les comparer) à l'aide de la commande `subplot`.

1. On souhaite simuler une loi géométrique de paramètre p. Pour cela, on va utiliser une simulation d'une loi exponentielle $\mathscr{E}(\lambda)$ de telle sorte que :

$$p = 1 - e^{-\lambda} \quad \Leftrightarrow \quad \lambda = -\ln(1-p).$$

   On propose le code suivant.

```python
def Geom2(p,N):
    lbd = -np.log(1-p)
    u = rd.exponential(1/lbd, N)
    return np.floor(u)+1
```

2. On commence par tracer les diagrammes en bâtons des fréquences empiriques pour les fonctions `Geom` et `Geom2`.

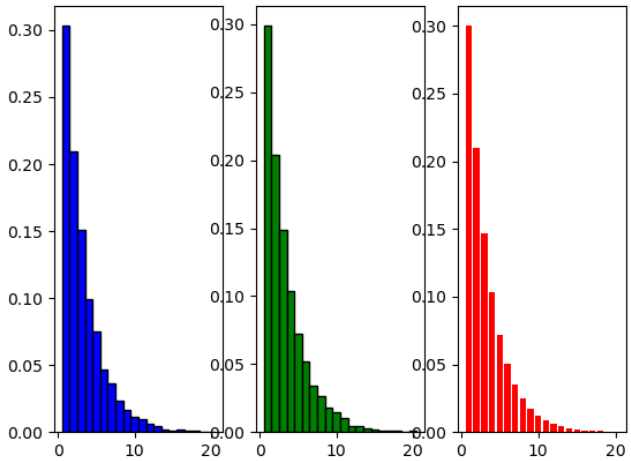
```python
# Echantillons
p = 0.3
N = 10000
x1 = Geom(p,N)
x2 = Geom2(p,N)
```

```python

#DeB des fréquences
c = np.arange(0.5,21)
plt.subplot(1,3,1)
plt.hist(x1,c,density='True',edgecolor='k',color='blue',label="DeB des fréq")
plt.subplot(1,3,2)
plt.hist(x2,c,density='True',edgecolor='k',color='green',label="DeB des fréq")
```

On trace ensuite le diagramme en bâtons des probabilités théoriques. Pour cela, on construit un vecteur `v` dont la k-ème composante est $(1-p)^{k-1}p$.

```python
#DeB des probas théo
u = (1-p)**np.arange(0,20)
v = p*u
s = np.arange(1,21)
plt.subplot(1,3,3)
plt.bar(s,v,color='red',label="DeB des prob theo")
plt.show()
```

On obtient la représentation graphique suivante :



Les diagrammes en bâtons étant tous similaires, les simulations obtenues à l'aide des fonctions `Geom` et `Geom2` sont bien pertinentes.

Exercice 8 (★)

On souhaite simuler une loi uniforme $\mathscr{U}([\![1,6]\!])$.

1. Déterminer le découpage correspondant de l'intervalle $[0,1]$ pour cette loi.
2. Écrire une fonction `unif()` simulant la loi $\mathscr{U}([\![1,6]\!])$ à l'aide de la méthode d'inversion.

1. On découpe l'intervalle $[0,1]$ en 6 sous-intervalles $I_k =]\frac{k-1}{6}, \frac{k}{6}]$ de même longueur $1/6$ où $k = 1,\dots,6$ (on prendra $I_1 = [0,1/6]$).

2. On détermine une réalisation t d'une loi uniforme sur $[0,1]$ à l'aide de la fonction `rd.random()`. On détermine l'intervalle I_k contenant t, et on renvoie l'entier k. On obtient la fonction suivante.

```
def unif():
    t = rd.random()
    k = 1
    while t>k/6:
        k = k+1
    return k
```

Lorsque la boucle `while` s'arrête, la variable k satisfait $t \leq \frac{k}{6}$ et $t > \frac{k-1}{6}$, donc t appartient à l'intervalle I_k. Cet évènement se produit avec une probabilité de $\dfrac{1}{6}$. On renvoie alors la valeur k.

Exercice 9 (★★ - Simulation de la loi de Poisson par la méthode d'inversion)

On souhaite simuler une loi de Poisson de paramètre $\lambda > 0$ par la méthode d'inversion.

1. On tire au hasard un nombre $t \in [0,1]$. Dans quel intervalle I_k le nombre t doit se trouver pour qu'on retourne k ?
2. On se donne le code suivant :

```
def poisson(lbd):
    t = rd.random()
    k = 0
    u = np.exp(-lbd)
    s = u ;
    while s<t :
        k = k+1
        u = u*(lbd/k)
        s = s+u
    return k
```

Expliquer le fonctionnement de la fonction `poisson`.

3. Écrire une fonction `Poisson(lbd,N)` renvoyant un vecteur contenant N réalisations indépendantes de la loi $\mathscr{P}(\lambda)$.
4. À l'aide d'un diagramme en bâtons, juger de la pertinence de cette simulation pour $\lambda = 5$. On admet pour cela que les cas où la variable prend une valeur supérieure à 10 sont négligeables.

1. Selon la méthode d'inversion expliquée ci-dessus, on renverra la valeur $k = 0$ si t appartient à l'intervalle $I_0 = [0, \frac{\lambda^0}{0!}e^{-\lambda}]$, et la valeur $k \in \mathbb{N}^*$ si t appartient à l'intervalle $I_k = \left]\displaystyle\sum_{i=0}^{k-1} \frac{\lambda^i}{i!}e^{-\lambda}, \sum_{i=0}^{k} \frac{\lambda^i}{i!}e^{-\lambda}\right]$.
2. On cherche à l'aide d'une boucle `while` l'intervalle I_k contenant t. On renvoie alors l'entier k. Notons que les variables u et s contiennent respectivement $\dfrac{\lambda^k}{k!}$ et $\displaystyle\sum_{i=0}^{k} \frac{\lambda^i}{i!}e^{-\lambda}$ à chaque nouveau passage dans la boucle.

3. On procède comme d'habitude.

```
def Poisson(lbd,N):
    X = np.zeros(N)
    for k in range(N):
        X[k] = poisson(lbd)
    return X
```

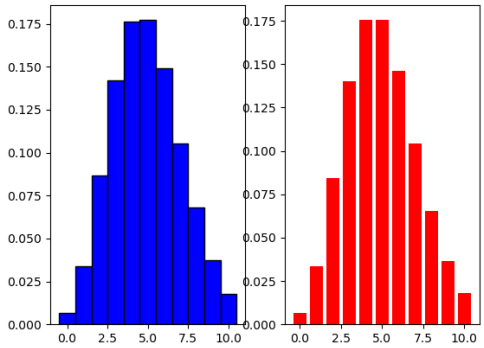
4. On utilise le code suivant pour tracer le diagramme en bâtons des fréquences empiriques

```
# Echantillons
lbd = 5
N = 100000
x = Poisson(lbd,N)

#DeB des fré quences
c = np.arange(-0.5,11)
plt.subplot(1,2,1)
plt.hist(x,c,density='True',edgecolor='k',color='blue',
label="DeB des fréq")
```

On trace ensuite le diagramme en bâtons des probabilités théoriques. Pour cela, il faut construire un vecteur `u` contenant en k-ème composante $\frac{\lambda^k}{k!}e^{-\lambda}$. Il faut traiter le $k = 0$ à part, ce qu'on fait avec la première commande. Le vecteur `v` contient ensuite les λ^k, et le vecteur `w` les $k!$. Reste alors à faire les opérations coefficients par coefficients.

```
#DeB des probas théo
u = np.exp(-lbd)*np.ones(11)
v = lbd**np.arange(1,11)
w = np.cumprod(np.arange(1,11))
u[1:11] = np.exp(-lbd)*v/w

s = np.arange(11)

plt.subplot(1,2,2)
plt.bar(s,u,color='red',label="DeB des prob theo")
plt.show()
```

On obtient la représentation graphique suivante :



> Les diagrammes en bâtons étant presque identiques, on en déduit que les simulations renvoyées par la fonction `poisson` sont pertinentes.

Exercice 10 (★★★ - Simulation de la loi géométrique par méthode d'inversion)
Écrire une fonction `geom3` qui prend en paramètre un nombre $p \in]0,1[$, puis à l'aide de la méthode d'inversion, simule une loi géométrique de paramètre p.

> On propose la fonction suivante.

```
def geom3(p):
    t = rd.random()
    k = 1
    u = p
    s = u
    while s<t :
        k = k+1
        u = u*(1-p)
        s = s+u
    return k
```

> La variable `u` contient à chaque étape $(1-p)^{k-1}p$. La variable s contient à chaque étape $\sum_{i=0}^{k}(1-p)^{i-1}p$. La boucle `while` permet de déterminer l'entier k satisfaisant :
>
> $$\sum_{i=0}^{k-1}(1-p)^{i-1}p < t \leq \sum_{i=0}^{k}(1-p)^{i-1}p.$$
>
> Et on a bien une probabilité de $\sum_{i=0}^{k}(1-p)^{i-1}p - \sum_{i=0}^{k-1}(1-p)^{i-1}p = (1-p)^{k-1}p$ que cela se réalise.

Exercice 11 (★)
Soit n un entier naturel non nul et N un entier supérieur ou égal à 2. On dispose de N urnes notées $U_1, \ldots, U_N$. Pour tout $k \in [\![1, N]\!]$, l'urne U_k contient $k-1$ boules blanches et $N-k$ boules noires. On lance un dé équilibré à N faces numérotées de 1 à N. On note k le numéro obtenu et on effectue n tirages successifs d'une boule avec remise dans l'urne U_k. On note X la variable aléatoire égale au nombre de boules blanches obtenues.
Écrire une fonction d'en-tête `def simul(n,N)` qui simule X pour des valeurs de n et N entrées par l'utilisateur.

> On commence par lancer un dé équilibré à N faces, ce qui revient à exécuter la commande `rd.randint(1,N+1)` (loi uniforme sur $[\![1, N]\!]$) et on stocke le résultat dans la variable `k`. On effectue alors n tirages avec remise dans l'urne U_k, et on compte le nombre de boules blanches obtenues. Ceci correspond à une loi binomiale $\mathscr{B}(n, \frac{k-1}{N-1})$ ($\frac{k-1}{N-1}$ étant la proportion de boules blanches dans l'urne k), et correspond à exécuter la commande `n = rd.binomial(n,(k-1)/(N-1)`.
>
> On propose donc la fonction suivante.

```
def simul(n,N):
    k = rd.randint(1,N+1)
    p = (k-1)/(N-1)
```

```
    n = rd.binomial(n,p)
    return n
```

Exercice 12 (★★)

On dispose d'une urne contenant sept boules dont trois sont blanches et quatre sont noires. On effectue dans cette urne deux tirages successifs d'une boule sans remise. On note X (respectivement Y) la variable aléatoire prenant la valeur 1 si la première (respectivement la deuxième) boule tirée est blanche et 0 sinon.

Compléter le programme suivant afin qu'il simule l'expérience et affiche la valeur du couple (X, Y).

```
if rd.random()<3/7 :
    X = ...
else :
    X = ...
if rd.random()< ... :
    Y = ...
else :
    Y = ...
print (X,Y)
```

La condition `rd.random()<3/7` est réalisée avec une probabilité de 3/7. Elle correspond à la probabilité de l'évènement « tirer une boule blanche au premier tirage ». Si cet évènement est réalisé, alors on attribue à X la valeur 1, sinon on lui attribue la valeur 0.

À la suite du premier tirage, il reste dans l'urne $3 - X$ boules blanches et 6 boules en tout dans l'urne, soit une proportion de $(3 - X)/6$ boules blanches dans l'urne. On a donc une probabilité de $(3-X)/6$ de tirer une boule blanche au deuxième tirage. Si `rd.random()<(3-X)/6` est réalisé, ce qui arrive avec une probabilité de $(3 - X)/6$, on attribue alors à Y la valeur 1. Sinon, on lui attribue la valeur 0.

On peut maintenant compléter le programme comme suit.

```
if rd.random()<3/7 :
    X = 1
else :
    X = 0
if rd.random()< (3-X)/6 :
    Y = 1
else :
    Y = 0
print (X,Y)
```

Exercice 13 (★★)

On lance indéfiniment un dé équilibré à 6 faces numérotées de 1 à 6.

Écrire le script d'un programme qui permet de simuler ce lancer de dé et qui renvoie le rang du lancer où l'on obtient pour la première fois un numéro déjà obtenu.

On effectue des lancers de dés successifs à l'aide de la commande `rd.random(1,7)` qui simule une réalisation d'une loi $\mathscr{U}(\llbracket 1,6 \rrbracket)$.

Il faut qu'à chaque lancer, on garde une trace du numéro obtenu pour savoir si on l'obtient de nouveau ensuite. On va pour cela créer un vecteur `u = np.zeros(6)` avec des zéros pour commencer (puisqu'on a obtenu zéro fois chaque entiers 1, ..., 6), la composante `u[k-1]` correspondant au nombre de fois où l'entier k est apparu. À chaque fois qu'on lance le dé avec la commande `k = rd.random(1,7)`, on augmente la $k-1$-ème composante du vecteur `u` de 1 à l'aide de la commande `u[k-1] = u[k-1]+1`.

On continue à lancer le dé tant qu'on ne tombe pas sur un numéro déjà obtenu, ce qu'on testera avec la condition `np.max(u)<=1`. On va donc faire une boucle `while` avec cette condition. On stockera enfin le nombre de lancers de dés dans la variable `n` qu'on renverra à la fin.

On propose donc le script suivant (on nous demande un programme et pas une fonction !) :

```
u = np.zeros(7)
n = 0
while np.max(u) <=1 :
     k = rd.randint(1,7)
     u[k-1] = u[k-1]+1
     n = n+1
print(n)
```

Exercice 14 (★★)

À un guichet, des clients peuvent venir expédier ou retirer un colis. Au cours d'une journée, le nombre de clients N qui s'y présentent suit la loi de Poisson de paramètre λ (où $\lambda \in \mathbb{R}_+^*$). Chaque client a une probabilité p (où $p \in]0,1[$) de venir pour expédier un colis et $1-p$ pour en retirer un.
On note C le nombre de colis expédiés dans la journée.

1. Pour tout $k \in \mathbb{N}$, déterminer l'espérance de C conditionnellement à l'événement $[N=k]$.
2. En déduire l'espérance de C.
3. On suppose dans cette question que $p = 0.3$ et $\lambda = 2.5$.
   (a) Écrire un programme renvoyant un vecteur `C` contenant $n = 10000$ réalisations de la variable aléatoire C.
   (b) Comparer alors l'espérance empirique obtenue avec l'espérance théorique.
   (c) On souhaite représenter le diagramme en bâtons des fréquences de l'échantillon.
      i. Exécuter la commande `np.mean(C<=4)`. Que dire des valeurs prises par `C` ?
      ii. Représenter le diagramme en bâtons des fréquences de l'échantillon.

1. La loi de C conditionnellement à l'évènement $[N=k]$ est la loi binomiale $\mathscr{B}(k,p)$. On a donc $E(C \mid [N=k]) = kp$.
2. On applique le théorème de l'espérance totale avec le SCE $([N=k])_{k\in\mathbb{N}}$:
   - Pour tout $k \in \mathbb{N}$, $E(C \mid [N=k])$ existe bien et vaut kp.
   - On étudie la convergence (absolue) de la série de terme général (si $k \geq 1$) :

$$E(|C| \mid [N=k])P(N=k) = kp\frac{\lambda^k}{k!}e^{-\lambda} = \frac{\lambda^{k-1}}{(k-1)!}\lambda p e^{-\lambda}.$$

On reconnait ici le terme général d'une série exponentielle qui converge donc (absolument).

Par le théorème de l'espérance totale, $E(C)$ existe et on a :

$$\begin{aligned} E(C) &= \sum_{k=0}^{+\infty} E(C \mid [N=k])P(N=k) = \sum_{k=0}^{+\infty} kp\frac{\lambda^k}{k!}e^{-\lambda} = \sum_{k=1}^{+\infty} kp\frac{\lambda^k}{k!}e^{-\lambda} \\ &= \left(\sum_{k=1}^{+\infty} \frac{\lambda^{k-1}}{(k-1)!}\right)\lambda p e^{-\lambda} = e^{\lambda}\lambda p e^{-\lambda} = \lambda p. \end{aligned}$$

3. (a) On simule une loi de Poisson pour obtenir le nombre N de clients, puis une loi binomiale de paramètres N et p pour le nombre de colis. On répète cela $n = 10000$ fois à l'aide d'une boucle `for`. On peut donc procéder comme suit.

```
p = 0.3
lbd = 2.5
n = 10000

C = np.zeros(n)
for k in range(n):
    N = rd.poisson(lbd)
    C[k] = rd.binomial(N,p)
print(C)
```

(b) L'espérance empirique (ou moyenne) s'obtient avec la commande suivante.

```
>>> np.mean(C)
```

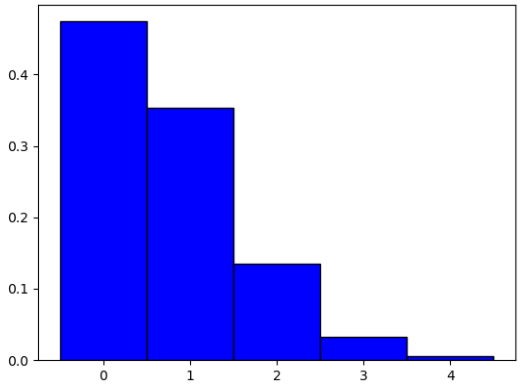
On obtient par exemple `0.735`. L'espérance théorique étant $\lambda p = 0,75$, on obtient bien une moyenne proche de cette valeur.

(c) i. La commande `np.mean(C<=4)` renvoie (dans le cas de notre simulation) `0.999`. Il y a donc $99,9\%$ des valeurs prises par C qui sont entre 0 et 4. On peut donc négliger le cas des valeurs supérieures, et considérer `C` comme étant à valeurs dans $[\![0,4]\!]$

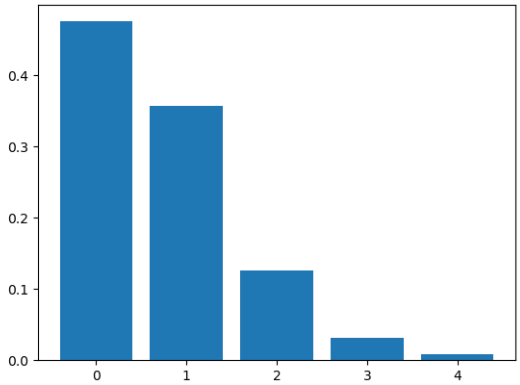
ii. **À l'aide de la commande `plt.hist`.** On prend pour classes $]-0.5, 0.5]$, $]0.5, 1.5]$, $]1.5, 2.5]$, $]2.5, 3.5]$, $]3.5, 4.5]$. Cela se fait à l'aide de la commande `c = np.arange(-0.5, 5)` par exemple. Reste à tracer l'histogramme.

```
c = np.arange(-0.5,5)
plt.hist(C,c,density='True',edgecolor='k',color
='blue',label="DeB des fréq")
plt.show()
```

Ce qui donne :



À l'aide de la commande `plt.bar`. Pour $0 \leq k \leq 4$, on détermine la fréquence d'apparition de l'entier k dans le vecteur `C`, ce qu'on peut faire à l'aide de la commande `np.mean(C==k)`. On stocke cette valeur dans la k-ème composante d'un vecteur v. On trace alors le diagramme en bâtons associé.

```
v = np.zeros(5)
for k in range(5):
    v[k] = np.mean(C==k)
u = np.arange(5)
plt.bar(u,v)
plt.show()
```

Ce qui donne :