TP Python : Générer de l'aléa

Correction

Partie I : La fonction `rand`

- Le but de ce TP est de découvrir un **module** Python très puissant: le module `random` de la bibliothèque `numpy`. (Un module est une "sous-librairie" d'une librairie "mère", plus générale.)
- Ce module contient des fonctions permettant de **générer des nombres aléatoires**.

Pour traiter un problème de probabilité/générer de l'aléatoire avec Python, on importe donc systématiquement le module `random`.

Entrée[3]:
```
1 import numpy as np
2 import numpy.random as rd
3 from math import floor # on importe la fonction floor depuis la b
```

Découvrons quelques fonctions de bases, disponibles dans la bibliothèque `random`, et **essentielles dans le programme d'ECG**.

1. La fonction `randint` permet de générer un **entier** N aléatoire compris entre deux entiers a et b : $a \leq N \leq b$.

   Pour appeler cette fonction on utilise la commande `rd.randint(a,b)`.
2. La fonction `random` du module `random` (oui, il y a répétition du nom) génère un flottant compris entre 0 **inclus** et 1 **exlus**.

   Pour appeler cette fonction, on utilise la commande `rd.random()`.

À vous de faire :

1. Construire une liste N contenant 15 nombres entiers compris entre -2 et 8, puis l'afficher.
2. Construire une liste X contenant 9 nombres réels appartenant à $[0, 1[$, puis l'afficher.
3. Sans l'exécuter, que va renvoyer la commande `rd.randint(3.5,4)` ?

Typesetting math: 100%

Entrée[ ]:
```
1
```

→ Cliquez-ici pour la réponse ←

À vous de faire :

1. Que fait la fonction Python suivante?
2. Que va renvoyer la commande `somysterious(3)` ?

Entrée[4]:
```
1 def somysterious():
2     nb = rd.random()
3     if nb < 1/3:
4         return "P"
5     else:
6         return "F"
7 
8 somysterious()
```

Sortie[4]: `'P'`

→ Cliquez-ici pour la réponse ←

Partie II : Exercices

On va voir dans les trois premiers exercices différentes manières de simuler un dé.

Exercice 1
À l'aide **d'instructions conditionnelles**, écrire une fonction qui simule le lancer d'un dé à quatre faces, tel que la face 1 apparait avec probabilité $\frac{1}{4}$, la face 2 avec probabilité $\frac{1}{2}$, et les faces 3 et 4 avec probabilité $\frac{1}{8}$.

Typesetting math: 100%

Entrée[6]:

```
def Dice4():
    nb = rd.random()
    if nb < 1/4:
        return 1
    elif 1/4 <= nb < 3/4:
        return 2
    elif 3/4 <= nb < 7/8:
        return 3
    else:
        return 4

Dice4()
```

Sortie[6]: 3

Exercice 2
À l'aide de la fonction `floor` de la bibliothèque `math`, écrire une fonction qui simule le lancer d'un dé équilibré à 6 faces.

Indication : soit $x \in [0, 1]$ un réel. Dans quel intervalle appartient le réel $a + kx$ avec a, b deux entiers naturels?

Entrée[7]:

```
def Dice6():
    resultat = floor(1+6*rd.random())
    return resultat
Dice6()
```

Sortie[7]: 5

Exercice 3

1. À l'aide de la fonction `randint`, écrire une fonction `RollTwoDice` qui simule le lancer de deux dés équilibrés à six faces, et qui renvoie 0 si la somme des deux faces obtenues est différente de 7, et 1 sinon.
2. Construire une autre fonction `MutlipleRolls` qui utilisera `RollDice`, prendra en argument un entier N, réalisera N lancers de deux dés, et renverra la fréquence d'apparition du résultat 7.
3. Que devient cette fréquence sur 100 séries, 1000 séries, ... ?
4. Quelle estimation de la probabilité d'apparition du 7 peut-on proposer ?

Typesetting math: 100%

Entrée[29]:

```
def RollTwoDice():
    N = rd.randint(1,6)
    M = rd.randint(1,6)
    S = N+M

    if S ==7:
        return 1
    else:
        return 0

def MultipleRolls(N):
    compt = 0

    for k in range(N):
        result = RollTwoDice()
        if result == 1 :
            compt = compt +1

    freq = compt/N
    return freq

print("La fréquence d'apparition du nombre 7 pour 10 lancers est
print("La fréquence d'apparition du nombre 7 pour 100 lancers est
print("La fréquence d'apparition du nombre 7 pour 1000 lancers es
print("La fréquence d'apparition du nombre 7 pour 10000 lancers e
```

```
La fréquence d'apparition du nombre 7 pour 10 lancers est  0.2

La fréquence d'apparition du nombre 7 pour 100 lancers est  0.16

La fréquence d'apparition du nombre 7 pour 1000 lancers est  0.163

La fréquence d'apparition du nombre 7 pour 10000 lancers est  0.159
4
```

Exercice 4

1. Construire une fonction `piece` qui ne prends rien en argument, et qui code une pièce équilibrée. On renverra 0 pour ``pile'' et $1$ pour ``face''.
2. Constuire une fonction `success` qui prend en argument une liste L de nombres appartenant à $\{0, 1\}$, compte le nombre de 0 dans la liste, et **affiche** la phrase "Gagné!" (resp. ``Perdu!'') lorsque ce nombre est supérieur à 50.
   *Notez que cette fonction affiche des phrases... Elle ne retourne rien. Ne pas oubliez le `return` en fin de fonction qui est **obligatoire**!!!*
3. Construire une liste `lancers` contenant 100 résultats de lancers de pièce, et tester votre fonction `success` .

Typesetting math: 100%

Entrée[57]:

```
def piece():
    N = rd.random()
    if N<1/2:
        return 0
    else :
        return 1

def success(L):
    compteur = 0
    for k in range(len(L)):
        if L[k] ==0:
            compteur = compteur +1
    if compteur>=50 :
        print("Gagné!")
    else :
        print("Raté!")

    return

lancers = []
for k in range(101):
    lancers.append(piece())


success(lancers)
```

```
Gagné!
```

Exercice 5

1. Écrire une fonction `ReachEight` tirant des nombres aléatoires entre 0 et 1, jusqu'à ce que la somme des nombres tirés dépasse 8, et qui retourne le nombre X de tirages qui ont été faits.
2. Écrire un programme pour que l'expérience soit faite 10 fois, et affiche à chaque fois le nombre de tirages effectués X pour que la somme dépasse 8.
3. Conjecturer combien de tirages sont nécessaires en moyenne pour que la somme dépasse 8.
4. Écrire une fonction prenant en argument N le nombre de fois où l'expérience est effectuée, et qui renvoie la moyenne du nombre de tirages X. Tester cette fonction pour $N = 10$, $N = 100$, et $N = 1000$.

Typesetting math: 100%

Entrée[40]:

```
def ReachEight():
    # on initialise un compteur qui est notre nombre de tirages t
    X = 0
    # on initialise la somme qui commence à zéro
    s = 0
    while s <8:
        s = s + rd.random()
        X = X +1
    return X

# on effectue 10 fois la même expérience
for k in range(10):
    nb = ReachEight()
    print("Pour le lancer numéro",k+1,"il aura fallu",nb,"tirages
```

```
Pour le lancer numéro 1 il aura fallu 16 tirages pour dépasser 8.

Pour le lancer numéro 2 il aura fallu 16 tirages pour dépasser 8.

Pour le lancer numéro 3 il aura fallu 18 tirages pour dépasser 8.

Pour le lancer numéro 4 il aura fallu 13 tirages pour dépasser 8.

Pour le lancer numéro 5 il aura fallu 16 tirages pour dépasser 8.

Pour le lancer numéro 6 il aura fallu 19 tirages pour dépasser 8.

Pour le lancer numéro 7 il aura fallu 14 tirages pour dépasser 8.

Pour le lancer numéro 8 il aura fallu 17 tirages pour dépasser 8.

Pour le lancer numéro 9 il aura fallu 21 tirages pour dépasser 8.

Pour le lancer numéro 10 il aura fallu 14 tirages pour dépasser 8.
```

Typesetting math: 100%

Entrée[38]:

```
def MeanEight(N):
    # on initialise la somme des X
    s = 0
    for k in range(N):
        # on effectue la k-ième expérience : X est le nb de tirag
        X = ReachEight()
        # on somme
        s = s +X
    # on moyenne
    m = s/N
    return m

print("La moyenne du nombre de tirages nécessaires pour ",10,"exp
print("La moyenne du nombre de tirages nécessaires pour ",100,"ex
print("La moyenne du nombre de tirages nécessaires pour ",1000,"e

```

```
La moyenne du nombre de tirages nécessaires pour  10 expériences es
t de 16.3 .

La moyenne du nombre de tirages nécessaires pour  100 expériences e
st de 16.31 .

La moyenne du nombre de tirages nécessaires pour  1000 expériences
est de 16.58 .
```

Partie III : Diagrammes en bâtons

Réflexe :
Importer le module nécessaire à la représentation graphique en Python.

Entrée[1]:

```
import matplotlib.pyplot as plt
```

La commande `bar(x, y)` du module `matplotlib.pyplot` permet de représenter un diagramme en bâtons.
Plus précisément, en notant les arguments $x = [x_1, \dots, x_n]$ et $y = [y_1, \dots, y_n]$ (listes de mêmes longueurs) elle permet de positionner un bâton de hauteur y_i en face de la valeur x_i.

Exemple :
Exécuter le programme suivant.

Typesetting math: 100%

Entrée[4]:

```
p = 1 /2
K = [ k for k in range(1,10)]
p_th = [ p * ( 1 -p) ** (k -1) for k in K]
plt.bar(K , p_th )
plt.show()
```

Exercice 6
Représenter le diagramme en bâtons dont les hauteurs des bâtons sont les valeurs **théoriques** de la loi uniforme U(⟦1, N⟧) pour $N = 5$.

Entrée[126]:

```
# paramètre de la loi uniforme
N = 5

# valeurs possibles de la loi uniforme
values = [k for k in range(1,N+1)]
# probabilité théorique de chaque valeur
p_theo = [ 1/N for k in range(1,N+1)]

# Création du diagramme en bâtons (les options "color ="  et "ec
plt.clf()
plt.bar(values, p_theo, color='pink', edgecolor='red')

# Ajout de labels et titre
plt.xlabel('Valeurs')
plt.ylabel('Probabilité')
plt.title('Loi uniforme discrète U(⟦1, N⟧) avec N=5')
plt.xticks(values)  # Affiche uniquement les valeurs discrètes s

# Affichage
plt.show()
```

- Les diagrammes en bâtons sont adaptés au cadre des variables aléatoires dont la loi est finie.
- Il est souvent très efficace d'utiliser de tels diagrammes pour conjecturer sur la loi suivie (par exemple si tous les bâtons ont à peu près la même hauteur, il est raisonnable de penser que la loi est uniforme!

L'idée est alors de créer un n-échantillon (avec n grand) et de créer un diagramme à bâtons dont la liste x en abscisses est celles des valeurs obtenues par simulation et la liste y en ordonnées celle des fréquences de chaque valeur dans l'échantillon.

Exercice 7
Une urne contient des boules numérotées de 1 à n, indiscernables au toucher. On effectue alors n 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-ième tirage.

1. Que fait la fonction `shuffle` du module `random` ? (*Cette fonction est hors programme*).
2. Compléter la fonction python `simul_X` ci-dessous, d'arguments n et k qui

Typesetting math: 100%

renvoie la simulation de X_k.

Entrée[120]:

```
def simul_X(k, n):
    if k > n:
        print("Erreur : k doit être inférieur ou égal à n.")
        return
    else :
        urne = list(range(1, n + 1))
        rd.shuffle(urne)

    return urne[k-1]


simul_X(3,7)
```

Sortie[120]: 5

2. On ajoute les commandes suivantes. Expliquer ce qu'elles font. On insistera que la commande de la ligne 6.

Entrée[125]:

```
n = 6
k = 4

# Générer les échantillons
echantillon = [simul_X(k,n) for repeat in range(1000)]

# Initialiser les fréquences
freq = [0] * n  # Une case pour chaque boule (0 à n-1)

# Comptage des occurrences
for result in echantillon:
    freq[result - 1] += 1  # Décalage nécessaire : `result` va d

# Normaliser les fréquences
freq = [f/1000 for f in freq]

# Diagramme en bâtons
plt.clf()
plt.bar([j for j in range(1, n + 1)], freq, color='pink', edgecc
plt.title(f"Fréquence des résultats pour {len(echantillon)} tira
plt.xlabel("Valeur de la boule tirée")
plt.ylabel("Fréquence")
plt.show()

```

Typesetting math: 100%