TP4 : Fonction `rd.random()` et simulation de v.a.r. discrètes suivant des lois usuelles

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
```

I. La fonction `rd.random()`

I.1. Que fait-elle ?

► Entrer une dizaine de fois dans la console la commande `rd.random()` (on peut utiliser la flèche du haut pour faire réapparaitre la commande précédemment entrée dans la console). Qu'observe-t-on ?

> Les nombres renvoyés sont des réels entre 0 et 1. La suite de nombres obtenus ne semble pas suivre de logique particulière.

► Entrer dans la console la commande `rd.random(10)`. Que renvoie la commande `rd.random(d)` ?

> La commande `rd.random(d)` renvoie un tableau contenant d nombres choisis aléatoirement entre 0 et 1.

► Recopier le script suivant et le compiler. Comment sont obtenus les tracés ? Quel phénomène voit-on apparaître ?

```
1 for k in range(1,7):
2     T = rd.random(10**k)
3     plt.hist(T)
4     plt.title(f'Répartition de {10**k} nombres générés par la fonction
5               rd.random()')
6     plt.show()
```

> Pour obtenir les tracés, on divise l'intervalle $[0, 1]$ en 10 classes :
>
> $$c_0 = [0, 0.1[,\ c_1 = [0.1, 0.2[,\ \ldots,\ c_8 = [0.8, 0.9[,\ c_9 = [0.9, 1].$$
>
> On compte ensuite le nombre N_i de valeurs dans le tableau `T = rd.random(10**k)` qui sont dans la classe c_i et on représente ce nombre comme une colonne de hauteur N_i au dessus de la classe c_i.
> On voit apparaître le phénomène suivant : plus k est grand, plus les différences de taille entre les colonnes sont petites. Autrement dit, plus k est grand, plus la répartition des nombres générés aléatoirement semble homogène entre les différentes classes c_i.

► On modifie légèrement le script précédent :

```
for k in range(1,7):
    T = rd.random(10**k)
    plt.hist(T, density = True)
    plt.title(f'Répartition normalisée de {10**k} nombres générés par la
              fonction rd.random()')
    plt.show()
```

Quelle est la différence entre les nouveaux tracés et les anciens ?

L'option `density = True` permet de *normaliser* la représentation graphique précédente. La représentation normalisée est telle que l'aire totale des colonnes dessinées est égale à 1.
Pour obtenir cette normalisation, il suffit de diviser chaque N_i par un nombre bien choisi.
Calculons le dans le cas général (`T = rd.random(10**k)`) :
Pour tout $i \in [\![0, 9]\!]$, la colonne au dessus de la classe c_i a une aire égale à $N_i \times 0,1$ (hauteur $\times$ largeur). Ainsi, la somme des aires est égale à

$$0,1(N_0 + N_1 + \cdots + N_9) = 0,1 \times 10^k = 10^{k-1}$$

Il faut donc diviser la hauteur de chaque colonne (c'est-à-dire le nombre N_i) par 10^{k-1}.

► La commande `rd.random()` permet de simuler quelle v.a.r. ?

Elle permet de simuler une v.a.r. à densité qui suit une loi uniforme sur $[0, 1]$ (cf chapitre variables aléatoires à densité).

I.2. A quoi sert-elle ?

► Exécuter une dizaine de fois le script suivant. Qu'observe-t-on ?

```
r = rd.random()
print(r)
print(r < 1/3)
```

Python génère un nombre aléatoire r entre 0 et 1 qu'il affiche puis *teste* si ce nombre est strictement inférieur à $\frac{1}{3}$ ou non. Il renvoie alors un booléen :

- `True` si $r < \frac{1}{3}$
- `False` si $r \geqslant \frac{1}{3}$.

► Quelle est la probabilité que **Python** renvoie `True` lorsqu'on exécute la commande `rd.random() < 1/3` ?

Python renvoie `True` avec probabilité $\frac{1}{3}$.

► En déduire une commande qui renvoie `True` avec probabilité p (le réel $p \in]0, 1[$ étant fixé). Cette commande permet alors de simuler une épreuve de Bernoulli de paramètre p.

La commande `rd.random() < p` renvoie `True` avec probabilité p.

II. Simulation de v.a.r. suivant une loi de Bernoulli

On rappelle qu'une épreuve de Bernoulli est une expérience aléatoire à deux issues : `Succès` et `Echec`. Si X est la v.a.r. qui vaut 1 en cas de `Succès` et 0 en cas d'`Echec`, alors X suit une loi de Bernoulli. Plus précisément, si la probabilité de `Succès` est p, alors X suit la loi de Bernoulli de paramètre p. On le note : $X \hookrightarrow \mathcal{B}(p)$.

- Compléter la fonction suivante pour qu'elle
  - prenne en argument un réel $p \in]0,1[$
  - renvoie une simulation d'une v.a.r. $X \hookrightarrow \mathcal{B}(p)$

```
def Bernoulli(p):
    if rd.random() < p:
        return __1__
    else:
        return __0__
```

- Compléter la fonction suivante pour qu'elle
  - prenne en arguments un entier $N \in \mathbb{N}^*$ et un réel $p \in]0,1[$
  - renvoie une liste contenant le résultat de la simulation de N v.a.r. indépendantes suivant toutes la loi $\mathcal{B}(p)$

```
def SampleBernoulli(N,p):
    L = __[]__
    for k in __range(N)__:
        __L.append(Bernoulli(p))__
    return L
```

- Recopier et exécuter le script suivant :

```
N = 10**4
p = 0.3
echantillon = SampleBernoulli(N,p)
effectif = np.arange(0,2)
for k in echantillon:
    effectif[k] += 1
plt.bar([0,1], effectif)
plt.title(f'Répartition des valeurs dans l\'échantillon de {N}
        var suivant la loi B(p)')
plt.show()
```

Interpréter les graphiques obtenus :

Le script précédent permet de tracer le diagramme en barre des effectifs de l'échantillon. Sur les 10000 simulations, on voit environ 7000 fois la valeur 0 et 3000 fois la valeur 1. C'est cohérent avec le fait qu'une var $X \hookrightarrow \mathcal{B}(0.3)$ prend la valeur 1 avec probabilité $0.3 = 30\%$.

III. Simulation de v.a.r. suivant une loi binomiale

On rappelle que lors d'une répétition de n épreuves de Bernoulli identiques et indépendantes, la v.a.r. X qui compte le nombre de succès suit une loi binomiale. Plus précisément, si la probabilité de succès de chaque épreuve de Bernoulli est p, alors X suit la loi binomiale de paramètres n et p. On le note : $X \hookrightarrow \mathcal{B}(n,p)$.
On rappelle également que si $X_1, \dots, X_n$ sont des v.a.r. qui sont indépendantes et suivent toutes la loi $\mathcal{B}(p)$, alors la v.a.r. $X = X_1 + \cdots + X_n$ suit la loi $\mathcal{B}(n,p)$.

- En s'inspirant de la remarque précédente, compléter la fonction suivante pour qu'elle
  - prenne en arguments un entier $n \in \mathbb{N}^*$ et un réel $p \in]0,1[$
  - renvoie une simulation d'une v.a.r. $X \hookrightarrow \mathcal{B}(n,p)$

```
1  def Binomiale(n,p):
2      X = 0
3      for k in __range(n)__:
4          if __rd.random() < p__:
5              X = __X + 1__
6      return __X__
```

- Compléter la fonction suivante pour qu'elle
  - prenne en arguments un entier $N \in \mathbb{N}^*$, un entier $n \in \mathbb{N}^*$ et un réel $p \in]0,1[$
  - renvoie une liste contenant le résultat de la simulation de N v.a.r. indépendantes suivant toutes la loi $\mathcal{B}(n,p)$

```
1  def SampleBinomiale(N,n,p):
2      L = __[]__
3      for k in __range(N)__:
4          __L.append(Binomiale(n,p))__
5      return L
```

- Recopier et compiler le script suivant :

```
1  N = 10000
2  n = 10
3  for k in range(1,10):
4      p = k/10
5      echantillon = SampleBinomiale(N,n,p)
6      effectif = np.arange(n+1)
7      for k in echantillon:
8          effectif[k] += 1
9      plt.bar(np.arange(n+1), effectif)
10     plt.title(f'Répartition des valeurs dans l\'échantillon de N var
11                  suivant la loi B(n,p)')
12     plt.show()
```

Interpréter les graphiques obtenus :

> Sur chaque tracé, il semble que les valeurs des 10000 simulations se concentrent autour d'une valeur. Plus on s'écarte de cette valeur centrale et plus les effectifs diminuent.
> Cette valeur centrale devient de plus en plus grande au fur et à mesure que p croît.

▶ Faire un copié collé du script précédent et le modifier comme ci-dessous :

```
N = 10000
n = 10
for k in range(1,10):
    p = k/10
    echantillon = SampleBinomiale(N,n,p)
    effectif = np.arange(n+1)
    for k in echantillon:
        effectif[k] += 1
    frequence = effectif / N
    plt.bar(np.arange(n+1), frequence)
    plt.title(f'Répartition normalisée des valeurs dans l\'échantillon de {N} var
              suivant la loi B({n},{p})')
    L = [np.math.comb(10,k)*p**k*(1-p)**(10-k) for k in range(11)]
    plt.plot(L,'+r')
    plt.show()
```

▶ Que contient la liste `L` ?

La liste `L` contient la loi d'une v.a.r. $X \hookrightarrow \mathcal{B}(n,p)$.

▶ Interpréter les graphiques obtenus :

Notons N_k le nombre de v.a.r. qui ont pris la valeur k au cours de la simulation.
La proportion de v.a.r. qui ont pris la valeur k est N_k/N.
D'après les graphiques, $\frac{N_k}{N} \simeq \mathbb{P}([X=k])$ où $X \hookrightarrow \mathcal{B}(n,p)$.
Cette égalité est une conséquence d'un résultat important en probabilité qui s'appelle la loi faible des grands nombres (cf chapitre Convergence et approximation).

IV. Simulation de v.a.r. suivant une loi géométrique

On rappelle que lors d'une répétition *infinie* d'épreuves de Bernoulli identiques et indépendantes, la v.a.r. X égale au rang du premier succès suit une loi géométrique. Plus précisément, si la probabilité de succès de chaque épreuve de Bernoulli est p, alors X suit la loi géométrique de paramètre p. On le note : $X \hookrightarrow \mathcal{G}(p)$.

▶ Compléter la fonction suivante pour qu'elle

- prenne en argument un réel $p \in]0,1[$
- renvoie une simulation d'une v.a.r. $X \hookrightarrow \mathcal{G}(p)$

```
def Geometrique(p):
    X = 1
    while  rd.random() >= p  :
        X =  X + 1
    return  X
```

- Est-on sûr que cette fonction s'arrête toujours ?

> La probabilité que cette fonction ne s'arrête jamais est nulle. C'est une conséquence du théorème de la limite monotone.

- Compléter la fonction suivante pour qu'elle
  - prenne en arguments un entier $N \in \mathbb{N}^*$ et un réel $p \in]0,1[$
  - renvoie une liste contenant le résultat de la simulation de N v.a.r. indépendantes suivant toutes la loi $\mathcal{G}(p)$

```
def SampleGeometrique(N,p):
    L = __[]__
    for k in __range(N)__:
        __L.append(Geometrique(p))__
    return L
```

- Recopier et compiler le script suivant :

```
N = 100000
p = 0.3
echantillon = SampleGeometrique(N,p)
effectif = np.arange(1,20)
for k in echantillon:
    if k < 20:
        effectif[k-1] += 1
frequence = effectif / N
plt.bar(np.arange(1,20), frequence)
plt.title(f'Répartition normalisée des valeurs dans l\'échantillon de {N} var
          suivant la loi G({p})')
A = [k for k in range(1,20)]
L = [p*(1-p)**(k-1) for k in range(1,20)]
plt.plot(A,L,'+r')
plt.show()
```

Interpréter les graphiques obtenus :

> Notons N_k le nombre de v.a.r. qui ont pris la valeur k au cours de la simulation.
> La proportion de v.a.r. qui ont pris la valeur k est N_k/N.
> D'après les graphiques, $\frac{N_k}{N} \simeq \mathbb{P}([X=k])$ où $X \hookrightarrow \mathcal{G}(p)$.
> Cette égalité est une conséquence d'un résultat important en probabilité qui s'appelle la loi faible des grands nombres (cf chapitre Convergence et approximation).
> En particulier, environ 30% des valeurs sont égales à 1 ($\mathbb{P}([X=1]) = p(1-p)^0 = p = 0,3$).
> On remarque également que dès que k prend une valeur supérieure ou égale à 15, la probabilité $\mathbb{P}([X=k])$ devient extrêmement petite. Ainsi, la plupart du temps les v.a.r. suivant la loi $\mathcal{G}(p)$ prennent de petites valeurs lorsque $p = 0,3$.

V. Simulation de v.a.r. suivant une loi uniforme discrète

▶ On considère deux entiers naturels a et b tels que $a \leqslant b$. Décrire ce que renvoient les commandes suivantes :

- `rd.random()`
- `(b-a+1)*rd.random()`
- `np.floor((b-a+1)*rd.random())`
- `a + np.floor((b-a+1)*rd.random())`

- `rd.random()` renvoie un réel choisi aléatoirement et uniformément dans $[0, 1]$.
- `(b-a+1)*rd.random()` renvoie un réel choisi aléatoirement et uniformément dans $[0, b-a+1]$.
- `np.floor((b-a+1)*rd.random())` renvoie un entier choisi aléatoirement et uniformément dans $[\![0, b-a]\!]$.

  Expliquons pourquoi la valeur $b-a+1$ ne peut pas être atteinte.

  Soit $x \in [0, b-a+1]$. Alors $\lfloor x \rfloor = b-a+1 \iff x = b-a+1$. Or, la probabilité de choisir exactement le nombre $b-a+1$ lorsqu'on choisit un nombre dans $[0, b-a+1]$ en suivant la loi uniforme continue sur $[0, b-a+1]$ est égale à 0.
- `a + np.floor((b-a+1)*rd.random())` renvoie un entier choisi aléatoirement et uniformément dans $[\![a, b]\!]$.

▶ Compléter la fonction suivante pour qu'elle

- prenne en arguments deux entiers naturels a et b tels que $a \leqslant b$
- renvoie une simulation d'une v.a.r. $X \hookrightarrow \mathcal{U}([\![a, b]\!])$

```
1 def Uniforme(a,b):
2     return    a + np.floor((b-a+1)*rd.random())
```

▶ Compléter la fonction suivante pour qu'elle

- prenne en arguments un entier $N \in \mathbb{N}^*$ et deux entiers naturels a et b tels que $a \leqslant b$
- renvoie une liste contenant le résultat de la simulation de N v.a.r. indépendantes suivant toutes la loi $\mathcal{U}([\![a, b]\!])$

```
1 def SampleUniforme(N,a,b):
2     L = __[]__
3     for k in __range(N)__:
4         __L.append(Uniforme(a,b))__
5     return L
```

- Recopier et compiler le script suivant :

```
for k in range(1,7):
    N = 10**k
    a = 0
    b = 4
    echantillon = SampleUniforme(N,a,b)
    effectif = np.arange(a, b+1)
    for k in echantillon:
        effectif[k-1] += 1
    plt.bar(np.arange(a, b+1), effectif)
    plt.title(f'Répartition des valeurs dans l\'échantillon de {N} var
              suivant la loi U([|{a},{b}|])')
    plt.show()
```

Interpréter les graphiques obtenus :

On voit apparaître le phénomène suivant : plus k est grand, plus les différences de taille entre les colonnes sont petites. Autrement dit, plus k est grand, plus les effectifs pour chaque valeur (de 0 à 4) sont similaires. C'est une illustration de l'équiprobabilité.

Commentaire

Avoir en tête les diagrammes à bâtons théoriques des lois usuelles permet d'émettre des conjectures sur la loi suivie par une variable aléatoire inconnue, que l'on sait simuler, à partir du diagramme à bâtons (ou de l'histogramme) des fréquences obtenues sur un grand échantillon de la variable.