tp3-bcpst2-Denombrement-Cor

November 13, 2019

1 TP 03 - DENOMBREMENTS

Capacités : Modéliser une situation combinatoire au moyen d'un vocabulaire précis ; mener un calcul de dénombrement;

```
In [1]: %pylab inline
        from math import *
        import numpy as np
        import matplotlib.pyplot as plt
        import random as rdm
```

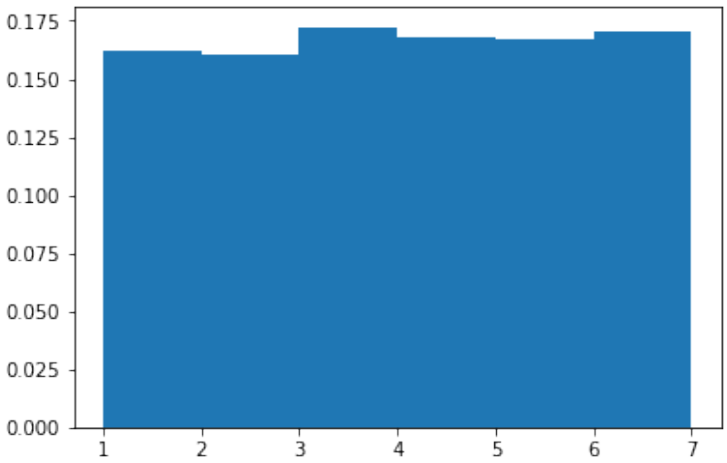
```
Populating the interactive namespace from numpy and matplotlib
```

```
In [2]: print([rdm.randint(1,6) for k in range(10)],' ')
```

```
[2, 1, 3, 3, 1, 4, 4, 3, 1, 5]
```

```
In [3]: L = [rdm.randint(1,6) for k in range(10000)]
        plt.hist(L,range(1,8),density=True)
```

```
Out[3]: (array([0.1622, 0.1602, 0.1723, 0.1678, 0.1668, 0.1707]),
         array([1, 2, 3, 4, 5, 6, 7]), <a list of 6 Patch objects>)
```



```
In [4]: S = ['P','F']
        [rdm.choice(S) for k in range(10)]

Out[4]: ['P', 'P', 'F', 'F', 'P', 'F', 'P', 'P', 'P', 'F']

In [5]: S = range(10)
        rdm.sample(S,10)

Out[5]: [0, 8, 6, 4, 3, 5, 2, 7, 1, 9]
```

1.1 Exercice 2 :

1.1.1 III/ Modélisation :

On précise que `randint(start,stop)` de la bibliothèque `random` fournit un nombre entier aléatoire compris entre `start` et `stop` (ces deux valeurs étant incluses) et que `random()` fournit un nombre réel aléatoire compris dans l'intervalle $[0,1[$.

Modélisation 1 : **Lancer d'une pièce de monnaie.** On lance n fois une pièce de monnaie équilibrée. Écrire une fonction qui simule cette expérience aléatoire et retourne une liste formée de 1 si "Pile" est obtenu, de 0 sinon.

```
In [6]: def simulLancers(n):
            return [rdm.randint(0,1) for k in range(n)]
        print(simulLancers(10))

[1, 1, 1, 1, 0, 1, 0, 1, 0, 1]
```

ou encore :

```
In [7]: def simulLancers2(n):
            S = ['P','F'] # à compléter
            L = [rdm.choice(S) for k in range(n)]
            Lf = [0]*n
            for i in range(n):
                if L[i] == 'P':
                    Lf[i] = 1
            return L,Lf
        print(simulLancers2(10))

(['F', 'F', 'F', 'P', 'P', 'P', 'F', 'F', 'F', 'F'], [0, 0, 0, 1, 1, 1, 0, 0, 0, 0])
```

Modifier votre fonction pour obtenir une fonction `nbePile(n)` qui retourne à la fois la liste des résultats et le nombre de "Pile" obtenus au cours de ces n lancers

```
In [8]: def nbePile(n):
            # retourne la liste des tirages et le nombre de pile
            LT = [rdm.randint(0,1) for k in range(n)]
            return LT,LT.count(1)
        L,nP = nbePile(10)
        print(L,'| nombre de piles = ',nP)

[0, 0, 0, 0, 1, 0, 1, 1, 0, 1] | nombre de piles =  4
```

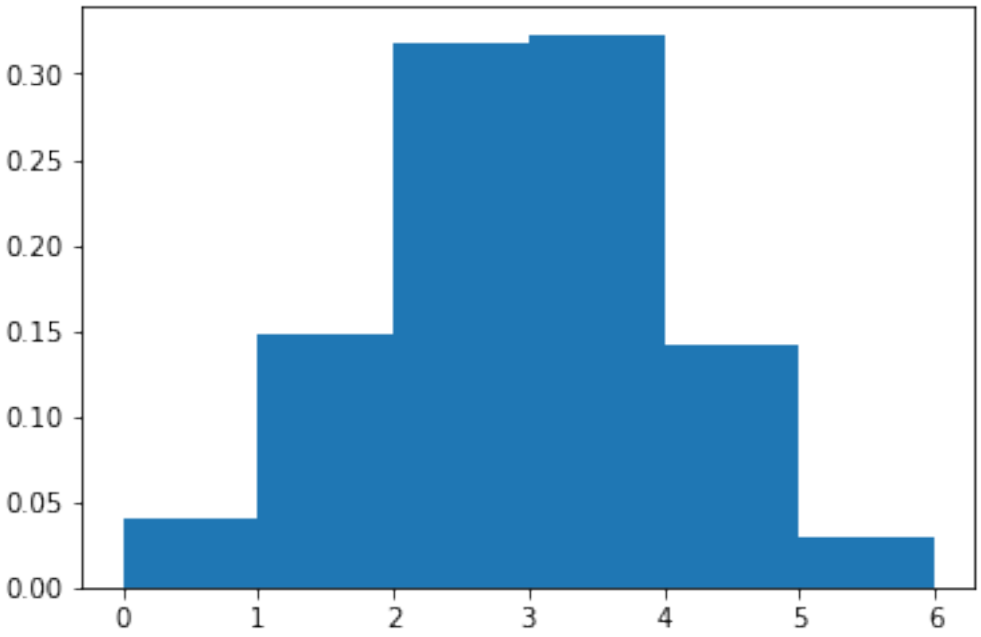
Une autre écriture possible est la suivante :

```
In [9]: def freqPile(n):
            # Autre façon d'écrire la fonction précédente.
            nP = 0
            LT = []
            for k in range(n):
                has = rdm.randint(0,1)
                LT.append(has)
                nP += has # nP = nP + has
            return LT,nP
        L,nP = freqPile(10)
        print(L,'| nombre de piles = ',nP)

[0, 0, 0, 0, 0, 0, 0, 1, 0, 0] | nombre de piles =  1
```

On effectue n lancers d'une pièce équilibrée. Ecrire une fonction permettant d'obtenir sous forme de liste la fréquence relative du nombre k de "Pile" obtenus ($0 \leq k \leq n$) lorsqu'on réalise m fois cette expérience (avec m grand).

```
In [10]: def freqNbePile(m,n):
             # retourne la fréquence relative du nombre k de pile (0 <= k <= n) au cours de m ré
             F = [0]*(n+1)
             for k in range(m):
                 L,nP = nbePile(n)
                 F[nP] = F[nP]+1 # F[nb] += 1
             return [F[i]/m for i in range(n+1)] # ou F/m si on a initialisé F = np.zeros(n+1)

In [11]: freqNbePile(1000,5)

Out[11]: [0.034, 0.143, 0.307, 0.323, 0.162, 0.031]

In [12]: def freqNbePile2(m,n):
             LnP = [nbePile(n)[1] for k in range(m)] # Liste formée du nbe de Piles à chaque épr
             H = plt.hist(LnP,np.arange(n+2),density = 'True')
             return H[0] # H est un tupple avec H[1] qui contient les bornes des classes...

In [13]: freqNbePile2(1000,5)
```

```
Out[13]: array([0.04 , 0.148, 0.318, 0.323, 0.142, 0.029])
```




```
In [14]: from scipy.stats import binom
         n = 5
         print([binom.pmf(k,n,1/2) for k in range(n+1)])
```

```
[0.03125, 0.15624999999999994, 0.3125, 0.3125, 0.15624999999999994, 0.03125]
```

Déterminer la moyenne du nombre de “Pile” obtenus lors de m répétitions de n lancers.

- Réponse 1 : On applique la formule de la moyenne à partir du tableau des fréquences relatives obtenues précédemment.

```
In [15]: n = 10
         F = freqNbePile(10000,n)
         print(sum([i*F[i] for i in range(n+1)]))
```

```
5.0226
```

- Réponse 2 : On écrit une fonction qui détermine la liste du nombre de “Pile” obtenus au cours de chacune des m répétitions de n lancers et on en fait la moyenne.

```
In [16]: def moyPile(m,n):
             # on répète m fois une série de n lancers
             # on retourne le nombre moyen de "Pile".
```

```
            LNbPile = []
            for i in range(m):
                LT,nP = nbePile(n)
                LNbPile.append(nP)
            return np.mean(LNbPile)
```

```
In [17]: n = 10
         print(moyPile(10000,n))
```

```
5.0051
```

Modélisation 2 : **Tirages avec remise.**

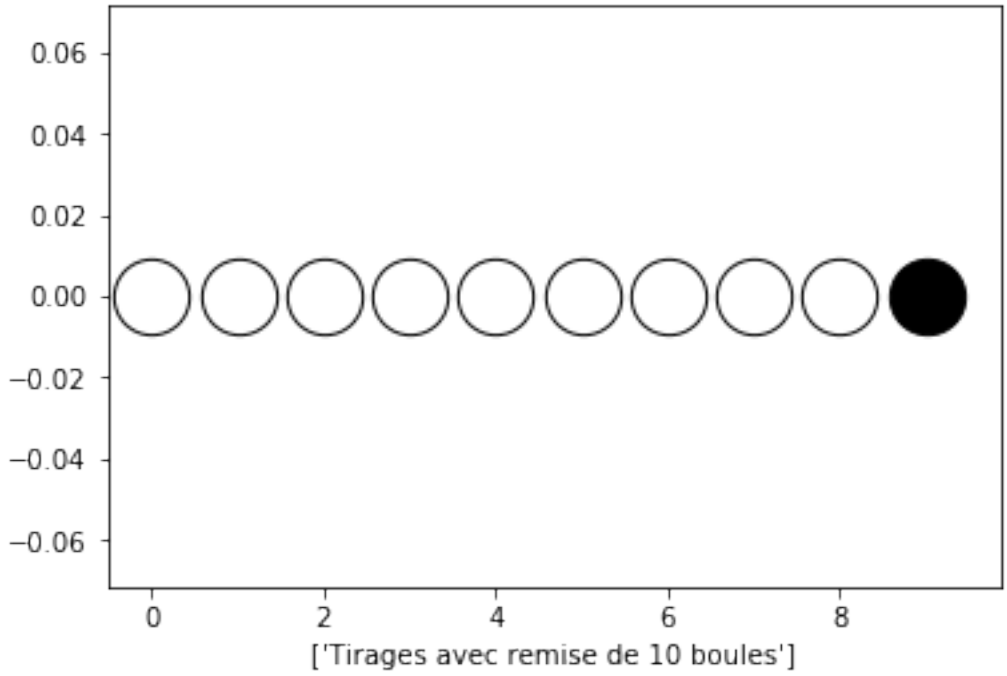
- Version 1 : On effectue n tirages avec remise dans une urne composée de 7 boules blanches et 3 boules noires. Écrire une fonction `tirageARv1.py` qui utilise la fonction `randint()` et simule cette expérience aléatoire en retourne une liste formée de 1 à chaque fois qu'une boules blanches est tirée, 0 sinon.

```
In [18]: Nb = 7 # Nombre de boules blanches
         N = 10 # Nombre de boules dans l'urne (p1 = Nb/N et p2 = (N-Nb)/N)def tirageARv1(n):

         def tirageARv1(n):
             # retourne une liste avec 1 si blanche, 0 sinon
             tirages=[0]*n # initialisation d'une liste de zéros
             for k in range(n):
                 has=rdm.randint(1,N) # tirage dans l'urne
                 if has <= Nb: # On a une blanche
                     tirages[k]=1
             return tirages

         tirageARv1(10)
```

```
Out[18]: [1, 1, 0, 1, 1, 1, 1, 1, 1, 1]
```

```
In [19]: def represente_tirage(T):
             plt.clf()
             for x in range(n):
                 if T[x]==1:
                     plt.scatter(x, 0, s=800 , c = 'w', edgecolors = 'k')
                 else:
                     plt.scatter(x, 0, s=800 , c = 'k')
             plt.xlabel(['Tirages avec remise de '+str(n)+' boules'])
```

```
In [20]: n = 10
         T1 = tirageARv1(n)
         represente_tirage(T1)
```



- Version 2 : Une urne est composée de boules blanches en proportion p_1 ($0 < p_1 < 1$). On effectue n tirages avec remise dans cette urne. Écrire une fonction `tirageARv2.py`dont les paramètres d'entrée sont p_1 et n, qui utilise cette fois la fonction `random()` et qui modélise sous forme de liste le résultats de ce tirage.

```
In [21]: def tirageARv2(n):
             # retourne une liste avec 1 si blanche, 0 sinon
             tirages=[0]*n # initialisation d'une liste de zéros
             p1 = Nb/N
             p2 = 1-p1
             for k in range(n):
                 has=rdm.random() # valeur dans ]0,1[ ;tirage dans l'urne
                 if has <= p1: # On a une blanche
                     tirages[k]=1
             return tirages
         tirageARv2(10)

Out[21]: [1, 0, 1, 1, 1, 1, 1, 1, 1, 0]
```

- Proposer une fonction Python `freqBoulesBlanches(m,n,p)` permettant de simuler m expériences (avec m grand) telles que celle décrite ci-dessus et retournant la fréquence d'apparition de k boules blanches pour $0 \leq k \leq n$.

```
In [22]: def freqBoulesBlanches(m,n):
             # retourne la fréquence du nombre k de boules blanches (0 <= k <= n) en proportion
```

```
            F = [0]*(n+1)
            for k in range(m):
                tirages = tirageARv2(n)
                nb = tirages.count(1) # ou np.sum(tirages) : Nbre de boules blanches
                F[nb] = F[nb]+1 # F[nb]+=1
            return [F[i]/m for i in range(n+1)]
```

```
In [23]: m,n = 100000,10
         F1 = freqBoulesBlanches(m,n)
         print(F1)
```

```
[1e-05, 0.00015, 0.00135, 0.00935, 0.0366, 0.1027, 0.20104, 0.26582, 0.23304, 0.1211, 0.02884]
```

On reconnait un shéma binomial et les résultats précédents pourront être comparés à :

```
In [24]: from scipy.stats import binom
         n = 10
         print([binom.pmf(k,n,7/10) for k in range(n+1)])
```

```
[5.904900000000006e-06, 0.0001377810000000004, 0.0014467005000000008, 0.00900169200000001, 0.036
```

- On suppose cette fois l'urne composée de N boules de trois couleurs distinctes. Proposer une fonction `tirageAR-3C.py` qui modélise sous forme de liste n tirages avec remise dans cette urne.

```
In [25]: def tirageAR_3C(n,p1,p2):
             # n : entier égale au nombre de tirages
             # p1,p2 : réels égales aux proportions respectives de boules blanches et rouges (au
             tirages=[0]*n
             for k in range(n):
                 has=rdm.random()
                 if has <= p1: # c'est une blanche
                     tirages[k]=1
                 elif has <= p1+p2: # c'est une rouge
                     tirages[k]=2
             return tirages
```

```
In [26]: def tirageAR_3C_2(n,N,Nb,Nr):
             # n : entier égale au nombre de tirages
             # Nb, Nr : entiers égales aux nombre respectif de boules blanches et rouges (avec N
             tirages=[0]*n
             for k in range(n):
                 has = rdm.randint(1,N) # à compléter
                 if has <= Nb:
                     tirages[k] = 1
                 elif has <= Nb+Nr:
                     tirages[k] = 2
             return tirages
```

```
In [27]: N = 10
         Nb,Nr,Nv = 3,2,5
         p1,p2 = Nb/N,Nr/N # et p3 = Nv/N = 1-(p1+p2)
         T = tirageAR_3C(10,p1,p2)
         print(T)
         T2 = tirageAR_3C_2(10,N,Nb,Nr)
         print(T2)
```

```
[2, 2, 0, 0, 0, 0, 2, 1, 0, 1]
[2, 1, 2, 2, 0, 1, 0, 0, 0, 2]
```

En terme de probabilités, on reconnaît un shéma multinomial dont les paramètres sont n et p1, p2, proportions respectives des deux premières couleurs.
Si on voulait valider la modélisation précédente, on pourrait écrire les deux fonctions suivantes :

```
In [28]: def loiMultinomiale(n,p1,p2,i,j):
             # retourne la probabilité d'avoir i C1 et j C2
             return factorial(n)/(factorial(i)*factorial(j)*factorial(n-i-j))*p1**i*p2**j*(1-p1-
```

```
In [29]: p11 = loiMultinomiale(10,p1,p2,4,2)
         print(p11)
```

```
0.06378749999999998
```

```
In [30]: def freqBoulesCouleurs1Et2(m,n,p1,p2):
             # retourne la fréquence du nombre i de boules C1 et j de boules C2 (0 <= i,j <= n)
             F = np.zeros((n+1,n+1))
             for k in range(m):
                 tirages = tirageAR_3C(n,p1,p2)
                 i = tirages.count(1) # ou np.sum(tirages) : Nbre de boules blanches
                 j = tirages.count(2)
                 F[i,j] = F[i,j]+1 # F[nb]+=1
             return F/m
```

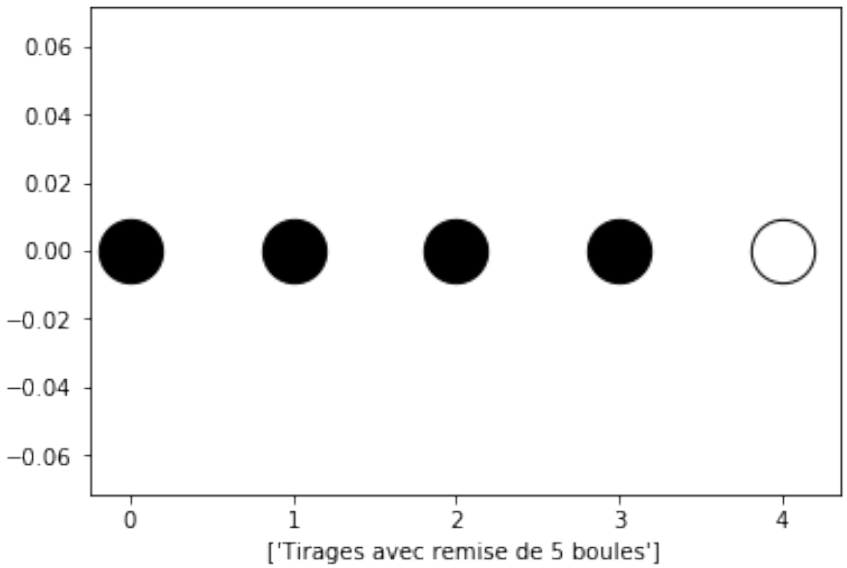
```
In [31]: m = 100000
         n,p1,p2 = 10,3/10,2/10
         T = freqBoulesCouleurs1Et2(m,n,p1,p2)
         print(T[4,2])
```

```
0.06271
```

***Modélisation 3 :* Tirages sans remise.** ** Question 1 :** Compléter la fonction `tirageSR.py` afin de modéliser un tirage successif sans remise de n boules dans une urne composée de N boules dont un proportion p_1 est blanche et p_2 est rouge (on notera que $N \cdot p_1$ est le nombre de boules blanches tandis que $N \cdot P_2$ est le nombre de boules rouges).
Cette fonction retournera une liste `tirages` formée de 0 et de 1 selon qu'une boule blanche est extraite ou pas.

```
In [32]: def tirage_sans_remise(N,n,p1):
             # On admettra que n <= N mais on pourrait le vérifier...
             tirages=[0]*(n)
             NT=N # nombre total initial de boules dans l'urne
             Nb=NT*p1 # nombre de boules blanches initialement.
             for k in range(n):
                 has=rdm.random()
                 if has <= Nb/NT:
                     tirages[k]=1
                     Nb = Nb-1
                 NT=NT-1
             return tirages

In [33]: N=10 # A titre d'exemple dix boules dont 3 sont blanches.
         n=5
         p1 = 3/10 # proportion de boules blanches ou Nb = N*p1 = 3 blanches
         LT = tirage_sans_remise(N,n,p1)
         print(LT)
         represente_tirage(LT)
```

```
[0, 0, 0, 0, 1]
```



Question 2 : Compléter la fonction `freqB-tirageSR.py` qui simule la réalisation de $m = 1000$ tirages avec remise de n boules dans une urne composée d'une proportion p_1 de boules blanches et qui retourne un tableau de 2 lignes, $n + 1$ colonnes, formé sur la première ligne du nombre de boules blanches possibles et sur la seconde des fréquences respectives du nombre de boules blanches obtenues au cours des 1000 tirages.

```
In [34]: def freqB_tiragesSR(m,N,n,p1):
             tab=np.zeros((2,n+1))#tableau de 2 lignes, n+1 colonnes
             tab[0,:]=range(n+1)# première ligne de 0 à n = nbre de boules blanches possibles
             for j in range(m):
                 LT = tirage_sans_remise(N,n,p1)
                 nb = LT.count(1) # valeur entre 0 et n
                 tab[1,nb]=tab[1,nb]+1/m
             return tab

In [35]: N,p1 = 10,3/10
         n = 5
         Tab = freqB_tiragesSR(1000,N,n,p1)
         print(Tab)

[[0.    1.    2.    3.    4.    5.   ]
 [0.083 0.412 0.423 0.082 0.    0.   ]]
```

Remarque : On doit reconnaître un **shéma Hypergéométrique**.
Ce qu'on peut vérifier ci-dessous :

```
In [36]: from scipy.stats import hypergeom
         print([hypergeom.pmf(k,N,p1*N,n) for k in range(n+1)],end = " ")
         # Remarque : Attention à l'ordre des paramètres.
                                #On écrit en France : X suit la loi H(N,n,p)

[0.08333333333333337, 0.4166666666666665, 0.4166666666666665, 0.08333333333333337, 0.0, 0.0]
```