TP2 Python
Quelques modélisations de l'aléatoire

Dans ce TP on utilisera abondamment les générateurs aléatoires de `numpy`, qui s'importent avec :

```
import numpy as np
import numpy.random as rd
```

1 Les basiques

Exercice 1. Programmer de la manière la plus concise possible, à l'aide de la fonction `rd.binomial`, un script qui renvoie la moyenne de 1000 tirages d'une variable $X \hookrightarrow \mathscr{B}(10, 0.4)$.

Exercice 2. Programmer, à l'aide de la fonction `rd.randint`, une fonction sans argument qui renvoie un tirage de $Y = e^X$ où $X \hookrightarrow \mathscr{U}([\![1, 10]\!])$.

Exercice 3. Programmer, à l'aide de la fonction `rd.poisson`, une fonction qui effectue des tirages successifs indépendants d'une variable X suivant la loi de Poisson $\mathscr{P}(4)$, et renvoie la valeur du premier tirage supérieur ou égal à 10.
Modifier cette fonction pour qu'elle renvoie le couple `(a,n)` où `a` est la première valeur supérieure à 10, et `n` le nombre de tirages nécessaires à l'obtention de cette valeur.)

Exercice 4. Programmer, à l'aide de la fonction `rd.geometric`, un script qui effectue 100 tirages indépendants d'une variable X suivant la loi $\mathscr{G}(1/2)$, et affiche la plus grande valeur obtenue.

Exercice 5. Programmer un script qui simule 100 lancers d'une pièce équilibrée et renvoie le nombre de Pile obtenus. On utilisera seulement la fonction `rd.random`.

2 Expériences aléatoires (plus compliquées)

Exercice 6. Programmer un script qui simule des lancers d'une pièce équilibrée et renvoie le nombre de lancers nécessaires pour obtenir pour la première fois 2 Pile successifs.
On pourra, par exemple, maintenir une variable `i` initialisée à 0, qui est incrémentée de 1 si on obtient Pile, et remise à 0 si on obtient Face.

Exercice 7. On considère un jeu vidéo possédant N niveaux.
Un joueur joue à ce jeu ; on suppose que la probabilité de terminer le niveau k est égale à $\dfrac{1}{k}$.
Programmer une fonction Python qui prend comme argument N, et renvoie le nombre de niveaux terminés par le joueur.
On pourra à cet effet introduire une variable k égale au numéro du dernier niveau **réussi**.

Exercice 8. On dispose de 10 urnes : pour tout $k \in [\![1, 10]\!]$, l'urne k contient k boules numérotées $1, 2, \ldots, k$.
Écrire un script Python qui simule le choix aléatoire équiprobable d'une urne, puis le tirage d'une boule dans cette urne, et affiche le numéro de la boule tirée.

Exercice 9.

1. Programmer une fonction `f1(n)` qui effectue `n` tirages indépendants d'une variable suivant $\mathscr{P}(4)$, et renvoie le nombre de valeurs distinctes obtenues.
   Par exemple, si $n = 5$ et que les 5 tirages ont donné 4,10,4,2,2, la fonction devra afficher 3 (car seules les 3 valeurs 2,4,10 sont apparues)

2. Programmer une fonction `f2(n)` qui effectue des tirages indépendants d'une variable suivant $\mathscr{P}(4)$ jusqu'à ce que `n` valeurs distinctes aient été obtenues ; et affiche la liste de ces valeurs.

3. Programmer une fonction `f3(n)` qui effectue des tirages indépendants d'une variable suivant $\mathscr{P}(4)$ jusqu'à ce que toutes les valeurs de $\{0,1,2,...,n\}$ aient été obtenues au moins une fois ; et renvoie le nombre de tirages effectués.

Exercice 10 (Digicode)**.** Un immeuble possède un digicode qui est un nombre à 4 chiffres. Une personne tente de rentrer dans l'immeuble en pressant des touches au hasard sur le clavier jusqu'à ce que la porte s'ouvre. On note X le nombres de touches qu'il faut presser. On admet que la personne arrive presque sûrement à rentrer.
Par exemple, si le digicode est 1664 et que la personne presse successivement

$$1,6,5,4,6,9,1,6,6,7,8,5,3,7,4,8,2,1,6,6,4$$

X vaudra 21.
On veut modéliser cela par une fonction Python qui prendra pour argument *la liste des 4 chiffres* (donc dans cet exemple, `[1,6,6,4]`).
On rappelle qu'on peut tester l'égalité entre deux objets de type `list` avec la commande `L1==L2`.
Compléter le code suivant :

```
def digicode(L):
    essais = ...
    P = list(rd.randint(....))
    # liste des 4 premières touches enfoncées
    while ... :
        new = ...  # nouvelle touche enfoncée
        essais = ...
        P = ... # P doit contenir les 4 dernières touches enfoncées
    return essais
```

Exercice 11. On s'intéresse à un problème de collection de cartes Pokémon.
On suppose qu'il existe en tout N cartes Pokémon distinctes ; pour alimenter sa collection on peut acheter des paquets de n vignettes. On suppose que les vignettes d'un même paquet sont 2 à 2 distinctes. On souhaite modéliser informatiquement cette situation, et estimer combien de paquets on doit acheter en moyenne pour posséder la collection complète.
Dans tout ce qui suit, les N cartes de la collection seront numérotées de 0 à $N-1$.

1. **Génération aléatoire d'un paquet de n cartes 2 à 2 distinctes.**
   On cherche ici, selon notre modélisation, à générer une liste aléatoire de n entiers de $[\![0,N-1]\!]$, deux à deux distincts, dans laquelle tout entier apparaît de manière équiprobable.
   Programmer une fonction `paquet(n,N)` effectuant ceci de la manière suivante, en supposant que ceci répond bien aux contraintes :

   - On initialise un « paquet » vide sous la forme d'une liste `M` vide ;
   - On génère la liste `L=[0,1,2,...,N-1]`
   - On répète n fois l'opération consistant à choisir aléatoirement un élément de `L`, à l'ajouter à `M` puis à l'enlever de `L`.

2. **Achats de paquets jusqu'à obtenir la collection complète.**
   Programmer une fonction qui simule l'achat de paquets de cartes générés par la fonction précédente, fait évoluer la collection au cours de ces achats, et renvoie le nombre de paquets qu'il aura été nécessaire d'acheter pour avoir la collection complète.
   On pourra pour cela modéliser la collection par une liste `collection` qui sera constituée de `0` et de `1` ; `collection[i]` vaudra 1 si et seulement si le collectionneur est en possession de la carte `i`.

Exercice 12. On considère l'expérience aléatoire suivante (ECRICOME S, 2017) :

On dispose de n urnes initialement vides, numérotées de 1 à n et on dispose d'un grand stock de boules que l'on dépose une à une dans ces urnes. Pour chaque boule, on choisit au hasard, de façon équiprobable, l'urne dans laquelle la boule est déposée. On note X_n le rang du premier tirage pour lequel une des urnes contiendra deux boules.

On modélise cela de la manière suivante :

- On crée une liste L à n composantes, initialement toutes égales à 0 ;
- Chaque tour de boucle correspond à la mise d'une boule dans un urne tirée au sort. On choisit donc aléatoirement cette urne ; et on modifie la liste L pour rendre compte du fait qu'on met la boule choisie dans cette urne.
- Ce processus se répète jusqu'à ce que... ?

Compléter la fonction suivante qui prend en argument le nombre n d'urnes et renvoie un tirage de X_n.

```
def tirage(n):
    L = np.zeros( ... ) # contient le nombre de boules dans chaque urne
    c = 1 # comptera les tirages
    choix_urne = rd.randint( ... , ... ) # choix de l'urne pour la première boule
    while ... :
        L[...] = L[...]+1 # l'urne choisie contient une boule de plus !
        choix_urne = .... #  on prend la boule suivante
        c = .... # et on met à jour le compteur
    return ....
```

Solutions

1

```
np.mean(rd.binomial(10,0.4,1000))
```

2

```
def exp_geometrique():
    return np.exp(rd.randint(1,11))
```

3

```
def poisson_sup_10():
    a=rd.poisson(4)
    while a<10:
        a=rd.poisson(4)
    return a
```

Avec la variante

```
def poisson_sup_10():
    a=rd.poisson(4)
    n=1
    while a<10:
        a=rd.poisson(4)
        n=n+1
    return (a,n)
```

4

```
print(max(rd.geometric(0.5,100)))
```

5

```
n=0
for k in range(100):
    if rd.random()<1/2:
        n=n+1
print(n)
```

6

```
def deux_pile():
    n=0
    i=0
    while i<2:
        if rd.random()<1/2:
            i=i+1
        else:
            i=0
        n=n+1
    return n
```

7 Pas si facile à mettre en place ! La solution que je propose n'est sans doute pas unique ; la variable que je décide de maintenir ici est le nombre de niveaux réussis (et non pas le niveau qu'on joue actuellement)

```
def jeu(N):
    k = 0 # k est le dernier niveau réussi. Au début on n'a rien réussi
    while rd.random()<1/(k+1) and k<N: # si le dernier réussi est k,
                                      # on teste si on passe le k+1...
                                      # sauf si k=N ! (car alors on a gagné)
        k = k+1 # si on passe le niveau (k+1), on incrémente
    return k
```

8

```
urne=rd.randint(1,11)
boule=rd.randint(1,urne+1)
```

9

```
def f1(n):
    tirages = rd.poisson(4,n)
    val=[]
    for k in tirages:
        if k not in val:
            val.append(k)
    return tirages,val



def f2(n):
    val=[]
    while len(val)<n:
        tirage=rd.poisson(4)
        if tirage not in val:
            val.append(tirage)
    return val



def f3(n):
    L=np.zeros(n+1)
    nb_tir = 0
    while np.sum(L)<n+1:
        L[rd.poisson(4)] = 1
        nb_tir+=1
    return nb_tir
```

10 Se souvenir de la syntaxe qui permet de tirer en bloc plusieurs variables suivant la même loi ! Ici par exemple on effectue 4 tirages $\hookrightarrow \mathscr{U}([\![0,9]\!])$ par la commande `rd.randint(0,10,4)`.
Remarques :

- On pose `P = list(rd.randint(0,10,4))` car il est bon que `P` soit une liste et non un `np.array`, pour le fonctionnement de la comparaison `L==P`.
- `P` doit toujours contenir les 4 dernières touches enfoncées : à chaque tout on enlève la première composante (donc on garde le bloc `P[1:]`) et on ajoute la nouvelle touche à la fin.

```
def digicode(L):
    essais = 4 # on tire en bloc les 4 premières touches
    P = list(rd.randint(0,10,4))
    while L!=P: # tant qu'on n'a pas le bon code
        new = rd.randint(0,10) # nouvelle touche
        essais = essais+1 # compteur
        P=P[1:]+[new] # et on met à jour la liste P
    return essais
```

11

```
def paquet(n,N):
    """ paquet de n cartes dans une collection
    de N cartes numérotées de 0 à N-1"""
    L=list(range(N))
    tirage=[]
    for k in range(n):
        p=rd.randint(len(L))
        tirage.append(L[p])
        del L[p]
    return(tirage)
```

```
def temps_obtention(n,N):
    collection = np.zeros(N)
    obtenues = 0 # nombre de cartes distinctes obtenues
    nb_paquets = 0 # nombre de paquets achetés
    while obtenues < N :
        p = paquet(n,N)
        nb_paquets = nb_paquets + 1
        for carte in p:
            if collection[p] == 0: # si la carte n'a pas été obtenue précédemment
                collection[p] = 1 # on l'ajoute à la collection
            obtenues = obtenues + 1 # et on ajuste le compteur de cartes obtenues
    return nb_paquets
```

12

```
def tirage(n):
    L = np.zeros(n)
    c = 1
    choix_urne = rd.randint(1,n+1)
    while L[choix_urne-1] == 0 : # l'urne dans laquelle on souhaite mettre la
                                 # boule est vide
        L[choix_urne-1] = L[choix_urne-1]+1
        choix_urne = rd.randint(1,n+1)
        c = c+1
    return c
```

(Rappel sur l'origine des indices : l'urne numéro 1 correspond à la première composant de L, donc à `L[0]`. il faut donc incrémenter `L[choix_urne-1]`).