TP noté : corrigé

PTSI Lycée Eiffel

20 décembre 2019

I. Quelques fonctions classiques sur les listes d'entiers.

Question 1 : Écrire une fonction Python **croissante** qui prend comme argument une liste de nombres et qui renvoie True ou False selon que la liste est ou non triée dans l'ordre croissant.

On se contente ici de parcourir la liste en s'arrêtant à l'avant-dernier élément de la liste, et de comparer chaque élément à celui qui le suit dans la liste. Si on trouve un élément strictement supérieur à l'élément suivant, on peut tout arrêter et retourner False. Sinon, on continue le parcours. Si on atteint la fin de la liste sans avoir retourné False, c'est que la liste est croissante et on renvoie la valeur True.

```
def croissante(l) :
    for i in range(len(l)-1) :
        if l[i]>l[i+1] :
            return False
    return True
```

Question 2 : Écrire une fonction Python **somme** qui prend comme argument une liste de nombres et qui renvoie la somme de ces nombres.

C'est encore plus classique, on stocke la somme dans une variable auxiliaire, qu'on initialise à 0 avant de lui ajouter toutes les valeurs de la liste :

```
def somme(l) :
    s=0
    for i in l :
        s=s+i
    return s
```

Question 3 : Écrire une fonction Python **indiceszeros** qui prend comme argument une liste de nombres et qui renvoie la liste (éventuellement vide) des indices i pour lesquels $l[i]$ est égal à 0.

On crée cette fois une variable de type liste initialement vide, à laquelle on ajoute les indices pour lesquels $l[i]$ vaut 0, en parcourant la liste :

```
def indiceszeros(l) :
    a=[]
    for i in range(len(l)) :
        if l[i]==0 :
            l.append(i)
    return l
```

II. Quelques fonctions faisant intervenir les probabilités.

Question 4 : Écrire une fonction Python **lancerde** qui prend comme argument un entier non nul n et qui renvoie une simulation de lancer de dé à n faces, c'est-à-dire un entier aléatoire entre 1 et n.

Pas besoin de se fatiguer, c'est exactement à ça que sert la fonction randint !

```
def lancerde(n) :
    return randint(1,n)
```

Question 5 : Écrire une fonction Python **simulde** qui prend comme arguments deux entiers n et k et qui effectue k simulations de lancers de dé à n faces, en stockant dans une liste le nombres de fois que chaque face du dé a été obtenue. La fonction renverra bien sûr cette liste.

On peut bien sûr utiliser la fonction précédente k fois de suite. À chaque « lancer de dé », on augmente d'une unité la valeur d'indice $k-1$ (parce que Python va numéroter les indices à partir de 0) d'une liste préalablement crée de longueur n et ne contenant initialement que des zéros :

```
def simulde(n,k) :
    l=[0 for i in range(n)]
    for i in range(k) :
        r=randint(1,n)
        l[r]=l[r]+1
    return l
```

Question 6 : Écrire une fonction Python **premiersix** qui ne prend aucun argument et qui effectue des simulations de lancer de dé à six faces jusqu'à obtenir un 6. La fonction renverra le nombre d'essais nécessaires avant d'obtenir ce premier 6.

On va bien sûr utiliser ici une instruction while : on lance le dé jusqu'à obtenir un 6, en incrémentant à chaque fois une variable qui servira à compter le nombre d'essais :

```
def premiersix() :
    c=1
    while randint(1,6) !=6 :
        c=c+1
    return c
```

Question 7 : Écrire une fonction Python **moyennesix** qui prend comme argument un entier k et qui effectue k fois de suite l'expérience de la question précédente (on compte le nombre d'essais nécessaires avant d'obtenir un 6). La fonction doit cette fois renvoyer le nombre moyen de tentatives nécessaires pour obtenir un 6 lors de ces k expériences.

C'est en fait tout simple : on lance k fois de suite la fonction premiersix en additionnant les résultats obtenus dans une variable, puis on divise le total par k pour retourner la moyenne :

```
def moyennesix(k) :
    s=0
    for i in range(k) :
        s=s+premiersix()
    return s/k
```

III. Modélisation de tas de sable.

Question 8 : Écrire une fonction Python **graphetas** qui prend comme argument une liste d'entiers l et qui renvoie une représentation graphique du tas de sable correspondant (on utilisera le module **matplotlib.pyplot** et on se contentera de relier les points de coordonnées $(i, l[i])$).

```
def graphetas(l) :
    a=range(len(l))
    plt.plot(a,l)
    return l
```

Question 9 : Écrire une fonction Python **connexe** qui prend comme argument une liste d'entiers l et qui renvoie True ou False selon que le tas est connexe ou non.

L'idée est de déterminer si les 0 éventuels de la liste ne se trouvent qu'en début ou en fin de liste. Une façon fort laide de procéder : tant qu'on a des 0 (en début de liste), on avance dans la liste (en s'arrêtant bien sûr si on atteint la fin de la liste). Une fois atteint le premier élément non nul, on continue à avancer dans la liste jusqu'à atteindre un nouveau 0 (ou la fin de la liste). S'il existe un élément non nul dans la liste après ce 0, alors le tas n'est pas connexe. Sinon, il est connexe.

```
def connexe(l) :
    i=0
    n=len(l)
    while i<n and l[i]==0 :
        i=i+1
    while i<n and l[i] !=0 :
        i=i+1
    for j in range(i+1,len(l)) :
        if l[j] !=0 :
            return False
    return True
```

Question 10 : Écrire une fonction Python **nbrepiles** qui prend comme argument une liste d'entiers l et qui renvoie le nombre de piles de sable contenus dans le tas représenté par cette liste.

On utilise le même principe que le programme précédent : tant qu'on a des 0 on avance. Tant qu'on a des éléments différents de 0 on avance. Une fois ces deux procédures effectuées, on a isolé une pile, et on recommence (jusqu'à atteindre la fin de la liste). Encore un programme extrêmement moche, d'autant plus que de la façon dont je l'ai écrit, le programme va compter une pile de trop si la liste se termine par des 0, d'où le if ajouté après la boucle principale pour enlever une pile dans ce cas :

```
def nbrepiles(l) :
    i=0
    p=0
    n=len(l)
    while i<n :
        while i<n and l[i]==0 :
            i=i+1
        while i<n and l[i] !=0 :
            i=i+1
        p=p+1
```

```
    if l[-1]==0 :
        p=p-1
    return p
```

Question 11 : Écrire une fonction Python **nbresommets** qui prend comme argument une liste d'entiers l et qui renvoie le nombre de sommets du tas de sable correspondant.

On retourne à des choses plus classiques : on parcourt les valeurs dans la liste (en évitant les deux extrémités), et on incrémente une variable de comptage à chaque fois qu'on en trouve une plus grande que les deux qui l'entourent :

```
def nbresommets(l) :
    s=0
    for i in range(1,len(l)-1) :
        if l[i]>l[i-1] and l[i]>l[i+1] :
            s=s+1
    return s
```

Question 12 : Écrire une fonction Python **stable** qui prend comme argument une liste d'entiers l et qui renvoie True ou False selon que le tas est stable ou non.

Exactement le même principe qu'à la question précédente, seul le test n'est pas le même :

```
def stable(l) :
    for i in range(1,len(l)) :
        if l[i]-l[i-1]>2 or l[i]-l[i-1]>2 :
            return False
    return True
```

Question 13 : Écrire une fonction Python **stabilise** qui prend comme argument une liste d'entiers l et qui renvoie la liste représentant le tas de sable stabilisé correspondant

On utilise une boucle while : tant que le tas n'est pas stable on le parcourt en déplacant les grains :

```
def stabilise(l) :
    while not stable(l) :
        for i in range(1,len(l)) :
            if l[i]-l[i-1]>2 :
                l[i-1]=l[i-1]+1
                l[i]=l[i]-1
            elif l[i]-l[i-1]>2 :
                l[i-1]=l[i-1]+1
                l[i]=l[i]-1
    return l
```

Question 14 : Écrire une fonction Python **ajoutegrain** qui prend comme arguments une liste d'entiers l et un indice i et qui ajoute un grain dans la colonne i du tas de sable en respectant les indications précédentes.

Une solution bien immonde (en utilisant pour tricher un peu une fonction récursive pour gérer les descentes successive, mais même comme ça le nombre de cas à gérer avec les cas particuliers des

extrêmités et les tirages aléatoires est affreux ; il y a sûrement moyen de faire beaucoup plus léger en réflechissant) :

```
def ajoutegrain(l,i) :
    if i==0 :
        if l[0]<=l[1] :
            l[0]=l[0]+1
                return l
        else :
            if randint(0,1)==0 :
                l[0]=l[0]+1
                return l
            else :
                return ajoutegrain(l,1)
    elif i==len(l)-1 :
        if l[i]<=l[i-1] :
            l[i]=l[i]+1
            return l
        else :
            if randint(0,1)==0 :
                l[i]=l[i]+1
                return l
            else :
                return ajoutegrain(l,i-1)
    else :
        if l[i]<=l[i+1] and l[i]<=l[i-1] :
            l[i]=l[i]+1
            return l
        elif l[i]<=l[i+1] :
            if randint(0,1)==0 :
                l[i]=l[i]+1
                return l
            else :
                return ajoutegrain(l,i-1)
        elif l[i]<=l[i-1] :
            if randint(0,1)==0 :
                l[i]=l[i]+1
                return l
            else :
                return ajoutegrain(l,i+1)
        else :
            r=randint(0,2)
            if r==0 :
                l[i]=l[i]+1
                return l
            elif r==1 :
                return ajoutegrain(l,i+1)
            else :
                return ajoutegrain(l,i-1)
```

Question 15 : Écrire une fonction Python **simulationtas** qui prend comme arguments deux entiers n et k et qui crée un tas de sable initialement vide de longueur n, puis qui y ajoute k grains de sable dans des colonnes choisies aléatoirement à chaque fois. La fonction tracera l'allure du tas de sable obtenu avant de renvoyer la liste correspondante.

Bon, ça c'est beaucoup plus simple, surtout qu'on peut utiliser des fonctions précédemment écrites :

```
def simulationtas(n,k) :
    l=[0 for i in range(n)]
    for i in range(k) :
        r=randint(0,n-1)
        ajoutegrain(l,r)
    graphetas(l)
    return l
```

Les résultats obtenus ne sont en fait pas très intéressants : le fait d'ajouter les grains à des endroits aléatoires combiné aux glissements de grains rend les tas obtenus très uniformes. On pourra comparer à ce qui se passe si on ajoute tous les grains à gauche du tas (cette fois-ci, à la suite d'effondrements, on doit obtenir des tas de type « exponentielle décroissante » si on met beaucoup de grains).