Informatique

Dernier TP - Révisions

PSI - Lycée Rabelais

CORRIGÉ

★

Exercices "simples"

★

1 Liste pyramidale

Écrire une fonction `liste_pyramide(n)` qui construit une liste avec des entiers "en pyramide" allant de 1 à n en ordre croissant puis redescend à 0. Par exemple `liste_pyramide(5)` devra renvoyer `[1,2,3,4,5,4,3,2,1]`.

```
def liste_pyramide(n):
    L = []
    for i in range(1,n+1):
        L.append(i)
    for i in range(1,n):
        L.append(n-i)
    return L

print(liste_pyramide(5))
```

2 Le roi Dagobert

Écrire une fonction `Dagobert(mot)` qui écrit le mot `mot` à l'envers. Par exemple, `Dagobert('Roi Dagobert')` devra renvoyer la chaîne `'trebogaD ioR'`.

```
def Dagobert(mot):
    a_l_envers = ''
    n = len(mot)
    for i in range(-1,-(n+1),-1):
        a_l_envers += mot[i]
    return a_l_envers
```

En déduire une fonction `palindrome(mot)` qui renvoie un booléen indiquant si la chaîne de caractère `mot` est un palindrome (`palindrome('kayak')` devra par exemple renvoyer `True`).

```
def palindrome(mot):
    if mot == Dagobert(mot):
        return True
    else:
        return False
```

3 Min, Max et Moy

Écrire une fonction `minMaxMoy(L)` qui reçoit une liste`L` d'entiers et qui renvoie le minimum, le maximum et la moyenne de cette liste.

```
def min_max_moy(L):
    n = len(L)

    mini = L[0]
    maxi = L[0]
    moy = L[0]/n




    for i in range(1,n):
        if L[i]<mini:
            mini = L[i]
        if L[i]>maxi:
            maxi = L[i]
        moy += L[i]/n

    return [mini,maxi,moy]


print(min_max_moy([1,2,3,4,5,6,7,8,9]))
```

4 Calcul des racines

Écrire une fonction `Racine(a,b,c)` qui renvoie, si elles existent dans $\mathbb{R}$, les racines du polynôme $P(x) = a.x^2 + b.x + c$. La fonction renverra également `False`, s'il n'y a pas de racine dans $\mathbb{R}$.

```
import numpy as np

def Racines(a,b,c):
    delta = b**2 - 4*a*c
    if delta<0:
        return False
    else:
        r1 = (-b - np.sqrt(delta))/(2*a)
        r2 = (-b + np.sqrt(delta))/(2*a)
```

```
10          return [r1,r2]
11
12  print(Racines(1,-2,1))
```

★

Exercices "difficiles"

★

5 Fourmi ivre

Une fourmi ivre suit les lignes d'un quadrillage dont les intersections sont représentées par les couples de nombres entiers. Lorsqu'elle arrive à une intersection, alle a autant de chance de continuer tout droit, de tourner à gauche, à droite ou de revenir sur ses pas. On note C_a, le carré formé des couples $[x, y]$ avec $-a \leq x \leq a$ et $-a \leq y \leq a$.

Q1. Écrire une fonction `avancer` sans argument qui renvoie l'un des quatre couples `[1,0]`, `[0,-1]`, `[0,1]` ou `[-1,0]` avec équiprobabilité (on pourra utiliser la fonction `randint` du module `random`).

```
from numpy import sqrt
import random as rd


def avancer():
    choix = [[1,0],[-1,0],[0,1],[0,-1]]
    i = rd.randint(0,3)
    return choix[i]
```

Q2. Simuler une trajectoire de la fourmi en partant de $[0,0]$ puis en faisant afficher les nœuds du quadrillage par où elle passe, jusqu'à ce qu'elle sorte du carré C_2.

```
a = 2

T = [[0,0]]
x = T[-1][0]
y = T[-1][1]
while a >= x and x >= -a and a >= y and y >= -a:
    x = T[-1][0]
    y = T[-1][1]

    dep = avancer()

    xsuivant = x + dep[0]
    ysuivant = y + dep[1]

    T.append([xsuivant,ysuivant])
```

Q3. Écrire une fonction `traj` d'argument `a` qui renvoie une trajectoire tirée au hasard, partant du centre $[0,0]$ et s'arrêtant lorsque la fourmi tente de sortir du carré C_a.

```
def traj(a):
    T = [[0,0]]
```

```
    x = T[-1][0]
    y = T[-1][1]
    while a >= x and x >= -a and a >= y and y >= -a:
        x = T[-1][0]
        y = T[-1][1]

        dep = avancer()

        xsuivant = x + dep[0]
        ysuivant = y + dep[1]

        T.append([xsuivant,ysuivant])

        x = T[-1][0]
        y = T[-1][1]

    tx = []
    ty = []
    for i in range(0,len(T)):
        tx.append(T[i][0])
        ty.append(T[i][1])
    return [tx,ty]
```

Q4. Tracer sur un même graphique 6 trajectoires différentes pour $a = 10$.

```
import matplotlib.pyplot as plt

for i in range(0,7):
    a = 10
    tx,ty = traj(a)

    plt.plot(tx,ty)
plt.grid(True)
plt.xlim([-a-2,a+2])
plt.ylim([-a-2,a+2])
```

Q5. Soit L_a la longueur du trajet effectué par une fourmi pour sortir du carré C_a en partant de son centre. Définir une fonction `LM` de deux arguments `a` et `N` qui renvoie la moyenne des longueurs de trajet de N fourmis (N réalisations de la variable aléatoire L_a).

```
def LM(a,N):
    Lmoy = 0
    for i in range(0,N):
        La = len(traj(a)[0])
        Lmoy += La/N
    return Lmoy
```

Q6. Tracer pour a entier compris entre 1 et 20, une estimation de $E(L_a)$, l'espérance de L_a.

```
Liste_a = []
```

```
Liste_LM = []
N = 1000
for ai in range(1,21):
    Liste_a.append(ai)
    Liste_LM.append(LM(ai,N))

plt.figure()
plt.plot(Liste_a,Liste_LM)
```

6 Suites récurrentes

On considère un jeu de 32 cartes. Il est formé de couples de 8 valeurs ordonnées, `valeurs=["7","8", "9", "10", "V", "D", "R", "A"]`, et de 4 "couleurs", `couleurs=["T", "K", "C", "P"]` (Trèfle, Carreau, Cœur, Pique). On distribue au hasard une "main", c'est-à-dire 5 cartes distinctes, et on s'intéresse à des mains particulières :

- les "une paire" (2 cartes de même valeur et 3 cartes de valeurs différentes) ;
- les "deux paires" (2 cartes d'une même valeur, 2 cartes d'une même autre valeur, et une carte d'une troisième valeur) ;
- les "fulls" (3 cartes d'une même valeur et 2 cartes d'une même autre valeur) ;
- les "carrés" (les 4 cartes d'une même valeur, et une autre).

Q1. Construire la liste `cartes` des 32 cartes à partir des deux listes `valeurs` et `couleurs`, chaque carte étant représentée par la paire `[valeur,couleur]`. Vérifier que le nombre de cartes obtenu est correct.

```
from numpy import sqrt
import random as rd

valeurs=["7","8", "9", "10","V", "D", "R", "A"]
couleurs=["Trèfle", "Carreau", "Coeur", "Pique"]


cartes = []
for v in valeurs:
    for c in couleurs:
        cartes.append([v,c])

## NOTA : len(cartes) renvoie bien 32.
```

Q2. Écrire une fonction `tirerMain` sans argument qui renvoie une liste de 5 cartes distinctes tirées au hasard. On pourra utiliser la fonction `sample` du module `random` dont la documentation est fournie ci-dessous.

```
def tirerMain():
    main = rd.sample(cartes,5)
    return main
```

Q3. Écrire une fonction `resultat` en vous aidant de la fonction tri appelée `sorted` et déjà implémentée sur `Python`. La fonction `resultat` doit renvoyer :

- 0 si la meilleure combinaison est un "une paire" ; ;
- 1 si la meilleure combinaison est un "deux paires" ;

- 2 si la meilleure combinaison est un "full" ;
- 3 si la meilleure combinaison est un "carré" ;
- 4 sinon.

```
def tirerMain():
    main = rd.sample(cartes,5)
    return main

main = tirerMain()
val = [maini[0]for maini in main]
print(sorted(val))

def resultat():
    main = tirerMain()
    val = [maini[0] for maini in main]
    if val[0]==val[1] and val[0]==val[2] and val[0]==val[3] or val[1]==val[2]
   and val[1]==val[3] and val[1]==val[4] :
        return 3
    elif val[0]==val[1] and val[0]==val[2] and val[3]==val[4] or val[0]==val
   [1] and val[2]==val[3] and val[2]==val[4] :
        return 2
    elif val[0]==val[1] and val[2]==val[3] or val[1]==val[2] and val[3]==val
   [4] or val[0]==val[1] and val[3]==val[4] :
        return 1
    elif val[0]==val[1] or val[1]==val[2] or val[2]==val[3] or val[3]==val[4]
   :
        return 0
    else :
        return 4
```

Q4. À partir de 50000 tirages aléatoires de mains, estimer les probabilités d'obtenir une "une paire", "deux paires", un full et un carré.

```
proba = [0.,0.,0.,0.,0.]
ntirage = 100000
for i in range(0,ntirage):
    proba[resultat()] += 1/ntirage
print(proba)
```

Q5. Comparer ces estimations avec les probabilités obtenues par dénombrement (ou par recherche sur internet).

7 Transformation

Q1. On considère un nombre n. Que donne la commande `list(str(n))` ?

...

Q2. Comment récupérer le chiffre des unités d'un nombre entier donné ? Celui des dizaines ? Tester cette méthode sur le nombre 2015.

...

Q3. Écrire une fonction `transforme` d'argument un entier naturel n et qui renvoie le nombre obtenu par la méthode suivante :

- les chiffres de rang impair (en partant des unités) sont inchangés.
- les chiffres de rang pair (en partant des unités) sont doublés, si ce double est supérieur ou égal à 10, on le remplace par la somme de ses chiffres (par exemple, 3 est remplacé par 6 et 7 est remplacé par 5).

```
def transforme(n):
    ln = list(str(n))



    i = -1
    while i >= -len(ln) :
        ln[i] = int(ln[i])
        if i%2 == 0 : ## rang impair
            double = 2*ln[i]
            if double < 10:
                ln[i] = double
            else:
                ln[i] = double//10 + double%10
        i = i - 1



    ## transformation en entier
    entier = 0
    i = -1
    while i >= -len(ln) :
        puissance = - i - 1

        entier = entier + ln[i]*10**puissance

        i = i - 1

    return entier
```

Exemple : 43281 est transformé en 46271.

Q4. Déterminer les nombres inférieurs à 10000 invariants par la fonction `transforme`. Expliquer le résultat obtenu.

```
verif = []
for test in range(0,10001):
    if test == transforme(test):
        verif.append(test)

print(verif)
```

Il suffit de remarquer que les chiffres 0 et 9 sont invariants. Donc tous les nombres qui ont un chiffre de rang pair égal à 0 ou 9 sont invariants.

8 Type J

Dans la liste des entiers naturels non nuls, on barre un nombre sur 2 en commençant par barrer le deuxième :

$$1, \not{2}, 3, \not{4}, 5, \not{6}, 7, \not{8}, 9, \ldots$$

Puis dans la liste restante, on barre un nombre sur 3 en commençant par barrer le troisième :

$$1, 3, \not{5}, 7, 9, \not{11} \ldots$$

Et ainsi de suite... A l'infini, on obtient "*la liste des nombres de type J*".

Q0. Créer une liste comprenant tous les entiers allant de 1 à 100 (inclus).

```
liste = [i for i in range(1,101)]
```

Pour la suite, on s'interdit l'utilisation de l'instruction pop.

Q1. Écrire une fonction `enlever` de deux arguments, une liste `L` et un entier naturel `i`, qui renvoie une liste `S` construite en ne gardant dans la liste `L` que les éléments dont le rang n'est pas multiple de `i`. Par exemple :

`enlever([1,3,5,7,9,11,13],3)` doit donner `[1,3,7,9,13]`.

```
def enlever(L,i):

    nouvelle_liste = []

    for j in range(0,len(L)):
        if (j+1)%i != 0 :
            nouvelle_liste.append(L[j])

    return nouvelle_liste
```

Q2. Écrire une suite d'instructions donnant *la liste des nombres de type J* inférieurs ou égaux à 100.

```
i = 2

while i<=len(liste):
    liste = enlever(liste,i)
    i += 1
```

Q3. Écrire une fonction `LJ` d'argument `n` renvoyant *la liste des nombres de type J* inférieurs ou égaux à `n`.

```
def LJ(n):

    i = 2
    liste1 = [j for j in range(1,n+1)]

    while i<=len(liste1):
        liste1 = enlever(liste1,i)
        i += 1
```

```
    return liste1
```

Q4. Écrire une fonction `U` d'argument `n` renvoyant u_n, le nombre de *nombres de type J* inférieurs ou égaux à `n`.

```
def U(n):
    return len(LJ(n))
```

Q5. Vers quelle limite l_i semble tendre $4.n/u_n^2$ quand n tend vers l'infini ?

```
import matplotlib.pyplot as plt
listeni = []
listeuni = []
for ni in range(10,50000,500):
    listeni.append(ni)
    listeuni.append(4*ni/(U(ni))**2)
plt.plot(listeni,listeuni)
```

La limite l_i semble être π !

Q6. Déterminer le premier n pour lequel la différence en valeur absolue entre $4.n/u_n^2$ et l_i est inférieure à 10^{-3}.

```
ni = 1
import numpy as np
li = np.pi
while abs(4*ni/(U(ni))**2 - li) > 10**(-3):
    ni += 1
print(ni)
```

9 Échiquier

Un échiquier est un plateau de 8 lignes et 8 colonnes. Ces lignes et ces colonnes seront,dans cet exercice, numérotées de 0 à 7. Une position de l'échiquier est un couple $[i, j]$ d'entiers compris entre 0 et 7 inclus, avec i le numéro de ligne et j le numéro de colonne. Un cavalier placé sur l'échiquier se déplace en bougeant de 2 cases dans une direction (verticale ou horizontale) et de une case perpendiculaire. Si le cavalier est loin des bords de l'échiquier, il y a 8 possibilités de déplacements, mais il en a moins s'il est près des bords.

Q1. Illustrer sur un brouillon les deux cas énoncées précédemment.

...

Q2. Écrire une fonction `valide` prenant en argument deux entiers relatifs i et j et vérifiant que le couple $[i, j]$ est bien une position de l'échiquier. `valide` doit renvoyer un booléen.

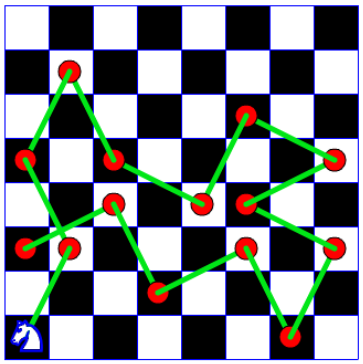
```
def valide(c):
    i = c[0]
    j = c[1]
    if (0 <= i and i <= 7 and 0 <= j and j <= 7):
        return True
    else:
        return False
```

Q3. Écrire une fonction `coupsSuivants` prenant en argument une position $[i, j]$ et renvoyant la liste des positions que peut atteindre un cavalier placé en $[i, j]$ en un seul coup.

```
def coupsSuivants(c):
```

```
    i = c[0]
    j = c[1]

    lsuivant = []

    for dep in [[-2,1],[-1,2],[1,2],[2,1],[2,-1],[1,-2],[-1,-2],[-2,-1]]:

        isuiv = dep[0] + i
        jsuiv = dep[1] + j

        if valide([isuiv,jsuiv]):
            lsuivant.append([isuiv,jsuiv])
    return lsuivant
```

On veut écrire un script permettant de résoudre le problème du cavalier. Ce problème algorithmique consiste à déplacer un cavalier sur le plateau pour parcourir toutes les cases sans passer deux fois sur la même.



On notera `T`, une liste de liste de dimension 8×8 définie telle que :

```
T = [[False,False,False,False,False,False,False,False], \
     [False,False,False,False,False,False,False,False], \
     [False,False,False,False,False,False,False,False], \
     [False,False,False,False,False,False,False,False], \
     [False,False,False,False,False,False,False,False], \
     [False,False,False,False,False,False,False,False], \
     [False,False,False,False,False,False,False,False], \
     [False,False,False,False,False,False,False,False]]
```

On notera `False` lorsque la case n'a pas été parcourue et on remplira le tableau au fur et à mesure en remplaçant `False` par le numéro du n-ième déplacement.

Pour le déplacement du cavalier, on choisira aléatoirement celui-ci parmi les différents coups possibles (on pourra éventuellement modifier la fonction `valide`). On utilisera également la fonction `randint` du module `random` pour générer un entier aléatoire.

Q4. Écrire une liste d'instructions afin de remplir le tableau `T`. On remplira le tableau tant qu'un déplacement est possible.

```
def valide(c,T):
    i = c[0]
```

```
    j = c[1]
    if (0 <= i and i <= 7 and 0 <= j and j <= 7):
        if T[i][j] == 0:
            return True
    else:
        return False

def coupsSuivants(c,T):
    i = c[0]
    j = c[1]

    lsuivant = []

    for dep in [[-2,1],[-1,2],[1,2],[2,1],[2,-1],[1,-2],[-1,-2],[-2,-1]]:

        isuiv = dep[0] + i
        jsuiv = dep[1] + j

        if valide([isuiv,jsuiv],T):
            lsuivant.append([isuiv,jsuiv])
    return lsuivant


depart = [4,4]

T = [[False,False,False,False,False,False,False,False], \
     [False,False,False,False,False,False,False,False], \
     [False,False,False,False,False,False,False,False], \
     [False,False,False,False,False,False,False,False], \
     [False,False,False,False,False,False,False,False], \
     [False,False,False,False,False,False,False,False], \
     [False,False,False,False,False,False,False,False], \
     [False,False,False,False,False,False,False,False]]

numero_coup = 1
T[depart[0]][depart[1]] = numero_coup

print('depart',T)

import random as rd

toutes_les_pos_suiv = coupsSuivants(depart,T)
choix_suiv = rd.randint(0,len(toutes_les_pos_suiv)-1)
pos_suiv = coupsSuivants(depart,T)[choix_suiv]
```

```
while toutes_les_pos_suiv != [] :


    numero_coup += 1
    choix_suiv = rd.randint(0,len(toutes_les_pos_suiv)-1)

    print('num_coup',numero_coup)
    # print('choix',choix_suiv)

    ##nouvelle position choisie
    pos_suiv = toutes_les_pos_suiv[choix_suiv]
    T[pos_suiv[0]][pos_suiv[1]] = numero_coup

    toutes_les_pos_suiv = coupsSuivants(pos_suiv,T)

print(T)
```

Q5. Combien de déplacements sont nécessaires pour remplir entièrement l'échiquier ? Votre code permet-il de trouver une solution à ce problème ?

Normalement, il faut 63 coups pour remplir l'ensemble de l'échiquier.

```
ncoupmax = 0
for essai in range(0,100000):


    depart = [4,4]

    T = [[False,False,False,False,False,False,False,False], \
         [False,False,False,False,False,False,False,False], \
         [False,False,False,False,False,False,False,False], \
         [False,False,False,False,False,False,False,False], \
         [False,False,False,False,False,False,False,False], \
         [False,False,False,False,False,False,False,False], \
         [False,False,False,False,False,False,False,False], \
         [False,False,False,False,False,False,False,False]]

    numero_coup = 1
    T[depart[0]][depart[1]] = numero_coup



    import random as rd

    toutes_les_pos_suiv = coupsSuivants(depart,T)
    choix_suiv = rd.randint(0,len(toutes_les_pos_suiv)-1)
    pos_suiv = coupsSuivants(depart,T)[choix_suiv]

```

```
27
28      while toutes_les_pos_suiv != [] :
29
30
31          numero_coup += 1
32          choix_suiv = rd.randint(0,len(toutes_les_pos_suiv)-1)
33
34
35          # print('choix',choix_suiv)
36
37          ##nouvelle position choisie
38          pos_suiv = toutes_les_pos_suiv[choix_suiv]
39          T[pos_suiv[0]][pos_suiv[1]] = numero_coup
40
41          toutes_les_pos_suiv = coupsSuivants(pos_suiv,T)
42
43
44
45      if numero_coup>ncoupmax:
46          ncoupmax=numero_coup
47          Tsol = T
48
49
50
51
52  print(ncoupmax)
```