td1a_correction_session5

December 23, 2020

1 1A.2 - Classes, méthodes, attributs, opérateurs et carré magique (correction)

Correction.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

1.0.1 Exercice 1 : carré magique

[2]:
```
class CarreMagique :
    def __init__(self, coef) :
        self.mat = [ [ coef[i+j*3] for i in range(3) ] for j in range(3) ]
    def __str__(self) :
        return "\n".join ( [ ",".join( [ str(n) for n in row ] ) for row in self.mat ]␣
↪)
    def __add__ (self, carre) :
        coef = []
        for i in range(3) :
            for j in range(3) :
                coef.append ( self.mat[i][j] + carre.mat[i][j])
        return CarreMagique(coef)

c = CarreMagique ( [ 1,3,4, 2,6,9, 8,7,5 ] )
print (c)
print("--")
print (c + c)
```

```
1,3,4
2,6,9
8,7,5
--
2,6,8
4,12,18
16,14,10
```

1.0.2 Exercice 2 : à faire à trois, carré magique (suite)

[3]:
```
class CarreMagique :
    def __init__(self, coef) :
        self.mat = [ [ coef[i+j*3] for i in range(3) ] for j in range(3) ]
    def __str__(self) :
        return "\n".join ( [ ",".join( str(n) for n in row ) for row in self.mat ] )
    def __add__ (self, carre) :
        coef = []
        for i in range(3) :
            for j in range(3) :
                coef.append ( self.mat[i][j] + carre.mat[i][j])
        return CarreMagique(coef)

    def somme_ligne_colonne_diagonale(self):
        tout = [ sum ( ligne ) for ligne in self.mat ] + \
               [ sum ( self.mat[i][j] for i in range(3) ) for j in range(3) ] + \
               [ sum ( self.mat[i][i] for i in range(3) ) ] + \
               [ sum ( self.mat[2-i][i] for i in range(3) ) ]
        return tout

    def coefficient_unique(self):
        d = { }
        for ligne in self.mat :
            for c in ligne :
                d [c] = d.get(c,0) + 1
        return len(d) == 9

    def est_magique(self):
        unique = self.coefficient_unique()
        if not unique : return False
        somme = self.somme_ligne_colonne_diagonale()
        return min(somme) == max(somme)

c = CarreMagique ( [ 1,1,1, 1,1,1, 1,1,1 ] )
print (c.est_magique())
c = CarreMagique ( [ 1,4,8, 5,2,6, 7,9,3 ] )
print (c.est_magique())
c = CarreMagique ( [ 1,6,8, 7,5,3, 2,4,9 ] )
print (c.est_magique())
c = CarreMagique ( [ 2,7,6, 9,5,1, 4,3,8 ] )
print (c.est_magique())
```

```
False
False
False
True
```

1.0.3 Exercice 3 : trouver tous les carrés magiques

La première version est fastidieuse à écrire mais simple à comprendre.

[4]:
```
def tous_les_carre_naif() :
    res = []
    for a1 in range(9) :
        for a2 in range(9) :
```

```
            for a3 in range(9) :
                for b1 in range(9) :
                    for b2 in range(9) :
                        for b3 in range(9) :
                            for c1 in range(9) :
                                for c2 in range(9) :
                                    for c3 in range(9) :
                                        carre = CarreMagique( [a1,a2,a3, b1,b2,b3,␣
↪c1,c2,c3 ])
                                        if carre.est_magique() :
                                            res.append (carre)
                                            print (carre)
    return res

# tous_les_carre_naif() (c'est très long)
```

La seconde version n'est pas plus rapide mais elle contient moins de boucles.

[5]:
```
def tous_les_carre_naif2() :
    # on choisit l'ensemble de tous les tableaux de 9 chiffres compris entre 1 et 9
    coef = [ 1 ] * 9
    res = [ ]
    while coef [0] < 10 :
        carre = CarreMagique(coef)
        if carre.est_magique() :
            res.append (carre)
            print (carre)
        coef[-1] += 1
        if coef[-1] >= 10 :
            i = len(coef)-1
            while coef[i] >= 10 and i > 0 :
                coef[i] = 1
                coef[i-1] += 1
                i -= 1

# tous_les_carre_naif2() (c'est très long)
```

La troisième version utilise le fait que les chiffres d'un carré magique sont tous différents. Il suffit de regarder seulement tous les permutations. La variable `stop_after` permet de se limiter seulement aux premiers.

[6]:
```
def tous_les_carres_permutation( permut = None, pos = 0, stop_after = 3):
    if pos == 9 :
        carre = CarreMagique (permut)
        if carre.est_magique() :
            print (carre)
            print ()
            return [ carre ]
        else :
            return []
    else :
        res = [ ]
        if permut == None :
            permut = [ i+1 for i in range(9) ]
        for i in range (pos,9) :
            # on permute les éléments i et pos
```

```
            a = permut[i]
            permut[i] = permut[pos]
            permut[pos] = a

            res += tous_les_carres_permutation(permut, pos+1)

            if stop_after > 0 and len(res) >= stop_after :
                return res

            # on effectue la permutation inverse
            a = permut[i]
            permut[i] = permut[pos]
            permut[pos] = a
        return res

res = tous_les_carres_permutation()
print ("nombre de carrés", len(res))
```

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

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

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

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

nombre de carrés 4
```

Le langage Python propose une fonction qui parcourt toutes les permutations d'un ensemble : itertools.permutation. Cela réduit de beaucoup la longueur du programme.

[7]:
```
import itertools
def tous_les_carres_permutation( stop_after = 3):
    res = [ ]
    firstn = list(range(1,10))
    for permut in itertools.permutations(firstn) :
        carre = CarreMagique (permut)
        if carre.est_magique() :
            res.append( carre )
            if stop_after >= 0 :
                print (carre)
                print ()
                if len(res) >= stop_after :
                    return res
    return res
```

```
res = tous_les_carres_permutation()
print ("nombre de carrés", len(res))
```

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

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

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

nombre de carrés 3
```

1.0.4 Exercice 4 : faire plus rapide

Est-il possible d'aller plus vite que de parcourir l'ensemble des permutations ? La réponse est oui. En parcourant les permutations, la fonction qui teste si les chiffres sont uniques est devenue inutile. Pour vérifier qu'on va plus vite, on peut mesurer le temps que met la fonction pour trouver tous les carrés :

[8]:
```
import time
d = time.perf_counter()
res = tous_les_carres_permutation(-1)
d = time.perf_counter() - d
print ("nombre de carrés", len(res), " en ", d, "seconds")
```

```
nombre de carrés 8  en  32.244037248859705 seconds
```

Pour aller plus vite, il faut utiliser la contrainte des sommes. Comment ? Lorsqu'on permute les nombres, on peut simplement vérifier que les deux premières lignes ont la même somme. L'utilisation de cette contrainte nous permet de d'aller 10 fois plus vite et d'obtenir le résultat en moins d'une seconde. L'inconvénient est que l'optimisation fonctionne parce qu'on ne parcourt pas toutes les permutations. On ne peut plus utiliser la fonction itertools.permutation.

[9]:
```
def tous_les_carres_permutation_ligne12_meme_somme( permut = None, pos = 0):
    if pos == 9 :
        carre = CarreMagique (permut)
        if carre.est_magique() :
            #print (carre)
            #print ()
            return [ carre ]
        else :
            return []
    else :
        if pos >= 6 :                                          # ajout
            if sum ( permut[:3]) != sum(permut[3:6]) :         # ajout
                return [ ]                                     # ajout

        res = [ ]
        if permut == None :
            permut = [ i+1 for i in range(9) ]
```

```
        for i in range (pos,9) :
            # on permute les éléments i et pos
            a = permut[i]
            permut[i] = permut[pos]
            permut[pos] = a

            res += tous_les_carres_permutation_ligne12_meme_somme(permut, pos+1)  #␣
↪changé

            # on effectue la permutation inverse
            a = permut[i]
            permut[i] = permut[pos]
            permut[pos] = a
        return res

import time
d = time.perf_counter()
res = tous_les_carres_permutation_ligne12_meme_somme()
d = time.perf_counter() - d
print ("nombre de carrés", len(res), " en ", d)
```

```
nombre de carrés 112  en  5.413181214436662
```

1.0.5 Programme complet

[10]:
```
class CarreMagique :
    def __init__(self, coef) :
        self.mat = [ [ coef[i+j*3] for i in range(3) ] for j in range(3) ]
    def __str__(self) :
        return "\n".join ( [ ",".join( [ str(n) for n in row ] ) for row in self.mat ]␣
↪)
    def __add__ (self, carre) :
        coef = []
        for i in range(3) :
            for j in range(3) :
                coef.append ( self.mat[i][j] + carre.mat[i][j])
        return CarreMagique(coef)

    def somme_ligne_colonne_diagonale(self):
        tout = [ sum ( ligne ) for ligne in self.mat ] + \
               [ sum ( self.mat[i][j] for i in range(3) ) for j in range(3) ] + \
               [ sum ( self.mat[i][i] for i in range(3) ) ] + \
               [ sum ( self.mat[2-i][i] for i in range(3) ) ]
        return tout

    def coefficient_unique(self):
        d = { }
        for ligne in self.mat :
            for c in ligne :
                d [c] = d.get(c,0) + 1
        return len(d) == 9

    def est_magique(self):
```

```
        unique = self.coefficient_unique()
        if not unique : return False
        somme = self.somme_ligne_colonne_diagonale()
        return min(somme) == max(somme)

def tous_les_carres_permutation_ligne12_meme_somme( permut = None, pos = 0):
    if pos == 9 :
        carre = CarreMagique (permut)
        if carre.est_magique() :
            #print (carre)
            #print ()
            return [ carre ]
        else :
            return []
    else :
        if pos >= 6 :                                          # ajout
            if sum ( permut[:3]) != sum(permut[3:6]) :         # ajout
                return [ ]                                     # ajout

        res = [ ]
        if permut == None :
            permut = [ i+1 for i in range(9) ]
        for i in range (pos,9) :
            # on permute les éléments i et pos
            a = permut[i]
            permut[i] = permut[pos]
            permut[pos] = a

            res += tous_les_carres_permutation_ligne12_meme_somme(permut, pos+1)  #␣
 ↪changé

            # on effectue la permutation inverse
            a = permut[i]
            permut[i] = permut[pos]
            permut[pos] = a
        return res

import time
d = time.perf_counter()
res = tous_les_carres_permutation_ligne12_meme_somme()
d = time.perf_counter() - d
print ("nombre de carrés", len(res), " en ", d)
```

```
nombre de carrés 8  en  4.024395385971979
```

On peut faire encore plus rapide en utilisant les contraintes pour inférer les autres coefficients (solution venant d'un élève) :

[11]:
```
def tous_les_carres():
    for a1 in range(1,10):
        for a2 in range(1,10):
            for a3 in range(1,10):
                for b1 in range(1,10):
                    somme = a1 + a2 + a3
```

```
                    c1 = somme - a1 - b1
                    b2 = somme - a3 - c1
                    b3 = somme - b1 - b2
                    c2 = somme - a2 - b2
                    c3 = somme - c1 - c2
                    M = CarreMagique([a1,a2,a3,b1,b2,b3,c1,c2,c3])
                    if M.est_magique() and 0 < b2 < 10 and 0 < b3 < 10 and 0 < c1 < 10␣
 ↪and 0 < c2 < 10 and 0 < c3 < 10 :
                        #print(M)
                        #print("---------------")
                        pass
%timeit tous_les_carres()
```

```
1 loops, best of 3: 200 ms per loop
```

[12]: