

Dans ce TP, on va manipuler des images sous forme de matrices de pixels. Pour cela, on commence par définir des fonctions qui nous seront utiles dans la suite. **Évaluez la cellule suivante !**

Entrée [1]:

```
from PIL import Image
import numpy as np
def affiche_image_nb(m, zoom=1):
    x = 255 * np.asarray(m, dtype=np.uint8)
    im = Image.fromarray(x)
    if zoom > 1: im = im.resize((im.size[0]*zoom, im.size[1]*zoom), Image.NEAREST)
    elif max(im.size) < 100: im = im.resize((im.size[0]*10, im.size[1]*10), Image.NEAREST)
    display(im)
def affiche_image(m, zoom=1):
    x = np.asarray(m, dtype=np.uint8)
    im = Image.fromarray(x)
    if zoom > 1: im = im.resize((im.size[0]*zoom, im.size[1]*zoom), Image.NEAREST)
    elif max(im.size) < 100: im = im.resize((im.size[0]*10, im.size[1]*10), Image.NEAREST)
    display(im)
raccoon = np.array(Image.open('raccoon.png').convert('RGB').resize((200,150))).tolist()
raccoon_nb = np.array(Image.open('raccoon.png').convert('L').resize((512,350)), dtype=np.uint8)
raccoon_g = np.array(Image.open('raccoon.png').convert('L').resize((512,350)), dtype=np.uint8)
raccoon_carre = np.array(Image.open('raccoon.png').convert('L').resize((256,256)))
```

1 Noir et blanc

Une image en noir et blanc est une matrice de 0 ou de 1. C'est-à-dire un tableau dont les éléments sont des tableaux de 0 et de 1 représentant chaque ligne de l'image, de haut en bas.

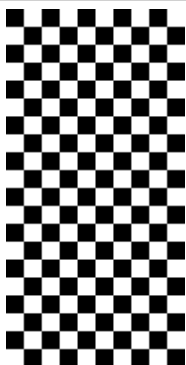
Ainsi pour faire un échiquier, on peut écrire :

Entrée [2]:

```
def echiquier(w,h):
    m = []
    for j in range(h):
        l = [] # la jeme ligne
        for i in range(w):
            l.append( (i+j)%2 )
        m.append(l)
    return m
```

Entrée [4]:

```
affiche_image_nb(echiquier(10,20))
```



Entrée [3]:

affiche_image_nb(racoon_nb)



Les deux fonctions suivantes sont pratiques. `taille` renvoie la taille (largeur, hauteur) d'une image et `image_nb_vide` renvoie une image avec uniquement des 0.

Entrée [3]:

```
def taille(m):  
    return (len(m[0]), len(m))
```

Entrée [4]:

```
def image_nb_vide(w,h):  
    m = []  
    for j in range(h):  
        l = []  
        for i in range(w):  
            l.append(0)  
        m.append(l)  
    return m
```

On peut alors réécrire la fonction précédente de manière plus naturelle.

Attention pour accéder au pixel de coordonnées (i, j) comme il est sur la jème ligne et la ième colonne, il faut écrire `m[j][i]`.

Entrée [5]:

```
m = image_nb_vide(20,10)
x1, x2, y = 3, 8, 5
for x in range(x1, x2+1):
    m[y][x] = 1
affiche_image_nb(m)
```

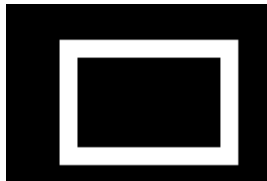


Entrée [9]:

```
list(range(3,8))
```

Out[9]: [3, 4, 5, 6, 7]

Les résultats attendus pour les trois fonctions suivantes sont



Entrée [12]:

Points: 1

ID: cell-6c289a54e968141f

Manually graded answer ▼

```
def ligne_horiz(m, x1, x2, y):
    '''Trace une ligne horizontale de pixels à 1
    allant de (x1,y) à (x2,y)'''
    # BEGIN SOLUTION
    for x in range(x1, x2+1):
        m[y][x] = 1
    # END SOLUTION
m = image_nb_vide(15,10)
ligne_horiz(m, 3, 12, 3)
ligne_horiz(m, 5, 10, 6)
affiche_image_nb(m)
m
```



```
Out[12]: [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

Entrée [13]:

Points: 1

ID: cell-a6120a4019b2edf7

Manually graded answer ▼

```
def ligne_vert(m, x, y1, y2):
    '''Trace une ligne verticale de pixels à 1
    allant de (x,y1) à (x,y2)'''
    # BEGIN SOLUTION
    for y in range(y1, y2+1):
        m[y][x] = 1
    # END SOLUTION
m = image_nb_vide(15,10)
ligne_vert(m, 3, 2, 8)
affiche_image_nb(m)
```



Entrée [14]:

- ▼

```
def rectangle(m, x1, y1, x2, y2):
    ligne_horiz(x1, x2, y1)
    ligne_horiz(x1, x2, y2)
```

Entrée [15]:

Points:

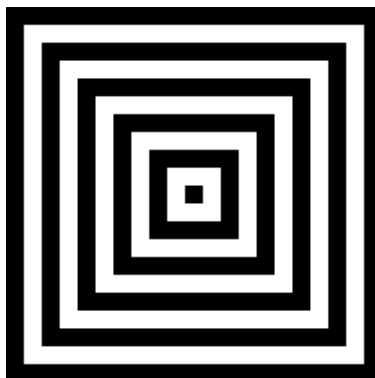
ID:

Manually graded answer ▼

```
def rectangle(m, x1, y1, x2, y2):  
    '''Trace un rectangle de (x1,y1) à (x2,y2)'''  
    # BEGIN SOLUTION  
    ligne_horiz(m, x1, x2, y1)  
    ligne_horiz(m, x1, x2, y2)  
    ligne_vert(m, x1, y1, y2)  
    ligne_vert(m, x2, y1, y2)  
    # END SOLUTION  
m = image_nb_vide(15,10)  
rectangle(m, 3, 2, 12, 8)  
affiche_image_nb(m)
```



Pour la prochaine fonction vous devez créer une image de dimension $(2n + 1, 2n + 1)$ présentant des carrés concentriques comme ici



On va commencer par faire une fonction qui étant donné une image de dimension $(2n+1) \times (2n+1)$ et un entier $i \leq n$ va afficher le carré centré dont le coin supérieur gauche est (i, i) .

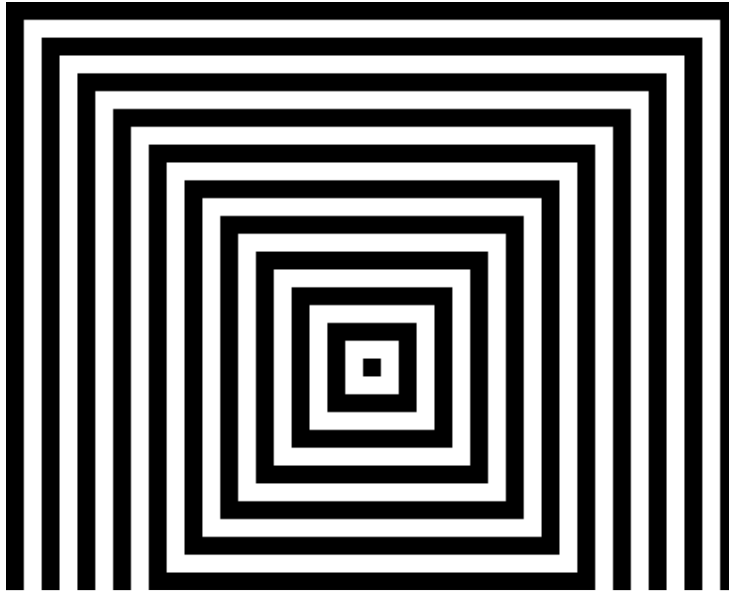
Entrée [19]:

Points: 1

ID: cell-32b3d80406c2c0e2

Manually graded answer ▼

```
def carres_concentriques(n):
    m = image_nb_vide(2*n+1,2*n+1)
    # BEGIN SOLUTION
    for i in range(n+1):
        if i % 2 == 1:
            rectangle(m, i, i, 2*n-i, 2*n-i)
    # END SOLUTION
    return m
affiche_image_nb(carres_concentriques(20))
```



On vous demande maintenant de faire une fonction qui inverse une image en changeant tous les 0 en 1 et tous les 1 en 0.

ATTENTION Il faudra renvoyer une *nouvelle* image et non modifier l'image passée en entrée. C'est important : il en va de la survie des *ratons-laveurs* !

Entrée [24]:

```
m = carres_concentriques(5)
m
```

```
Out[24]: [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0],
 [0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0],
 [0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0],
 [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
 [0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0],
 [0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0],
 [0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0],
 [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

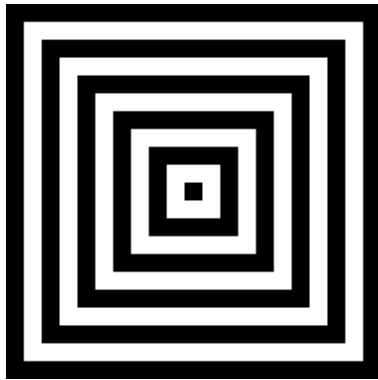
Entrée [13]:

Points: 1

ID: cell-0d468a758c1b24fd

Manually graded answer ▼

```
def inverse_nb(m):  
    w, h = taille(m)  
    m2 = image_nb_vide(w, h)  
    # BEGIN SOLUTION  
    for i in range(w):  
        for j in range(h):  
            m2[j][i] = 1 - m[j][i]  
    # END SOLUTION  
    return m2  
affiche_image_nb(inverse_nb(carres_concentriques(10)))  
affiche_image_nb(inverse_nb(racoon_nb))
```



On va maintenant écrire des fonctions opérant des symétries et des rotations d'image. Les résultats attendus sont





Entrée [8]:

Points:

1

ID:

cell-727bc39ae36b87bf

Manually graded answer ▼

```
def sym_vert(m):  
    '''Effectue une symétrie verticale de l'image, c'est-à-dire que le haut dev.  
    et réciproquement'''  
    w, h = taille(m)  
    m2 = image_nb_vide(w, h)  
    # BEGIN SOLUTION  
    for i in range(w):  
        for j in range(h):  
            m2[h-j-1][i] = m[j][i]  
    # END SOLUTION  
    return m2  
affiche_image_nb(sym_vert(racoon_nb))
```



Entrée [15]:

Points:

1

ID:

cell-0b6c7a5853fcabf5

Manually graded answer ▼

```
def sym_horiz(m):  
    '''Effectue une symétrie horizontale de l'image, c'est-à-dire que la gauche  
    et réciproquement'''  
    w, h = taille(m)  
    m2 = image_nb_vide(w, h)  
    # BEGIN SOLUTION  
    for i in range(w):  
        for j in range(h):  
            m2[j][w-i-1] = m[j][i]  
    # END SOLUTION  
    return m2  
affiche_image_nb(sym_horiz(racoon_nb))
```



Entrée [9]:

Points:

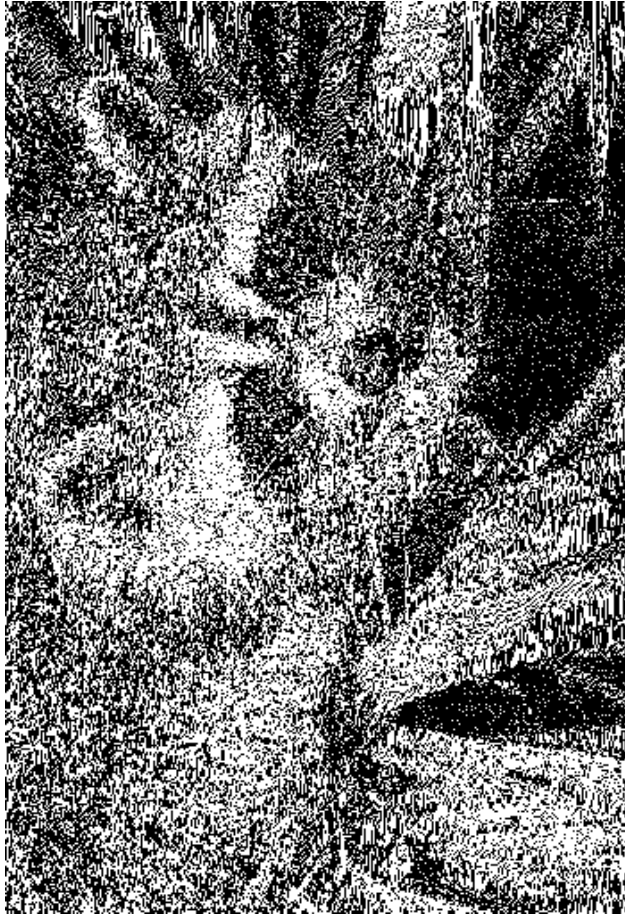
1

ID:

cell-ec27328b0672b27f

Manually graded answer ▼

```
def rot90(m):  
    '''Effectue une rotation de l'image d'un quart de tour dans le sens trigonom  
    w, h = taille(m)  
    m2 = image_nb_vide(h, w)  
    # BEGIN SOLUTION  
    for i in range(w):  
        for j in range(h):  
            m2[i][j] = m[j][i]  
    # END SOLUTION  
    return sym_vert(m2)  
affiche_image_nb(rot90(racoon_nb))
```



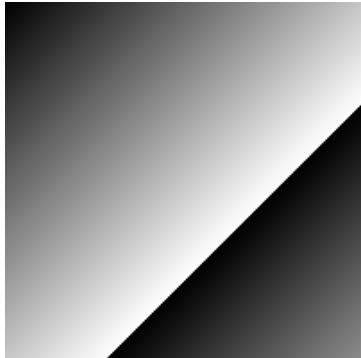
- ▼

2 Niveaux de gris

Les images en niveaux de gris sont des matrices d'entiers où la couleur va de 0 (noir) à 255 (blanc). Ainsi 128 sera un gris médian

Entrée [17]:

```
def dégradé(w,h):  
    m = []  
    for j in range(h):  
        l = []  
        for i in range(w):  
            l.append( (i+j)%256 )  
        m.append(l)  
    return m  
affiche_image(degradé(200,200))
```



Entrée [18]:

```
affiche_image(racoon_g)
```



Normalement, si vous avez fait ce qu'il faut vos fonctions de symétries et de rotations continuent de fonctionner. On pourra même directement utiliser `image_vide_nb` car 0 est noir ici aussi.

Entrée [19]:

`affiche_image(rot90(racoon_g))`



Résultat attendu pour la fonction suivante :



Entrée [20]:

Points:

1

ID:

cell-2a077e2799718d06

Manually graded answer ▼

```
def inverse_g(m):  
    '''Inverse le niveau de gris.'''  
    w, h = taille(m)  
    m2 = image_nb_vide(w, h)  
    # BEGIN SOLUTION  
    for i in range(w):  
        for j in range(h):  
            m2[j][i] = 256 - m[j][i]  
    # END SOLUTION  
    return m2  
affiche_image(inverse_g(racoon_g))
```



Pour faire un effet de flou, on peut remplacer chaque valeur par la moyenne des 9 valeurs constitué par le pixel et ses 8 voisins immédiats dans la grille, en ignorant les bords. On vous demande de faire cette fonction. Le résultat attendu est



On pourra laisser ainsi les bords en noir.

Entrée [21]:

Points:

1

ID:

cell-041bf358a574daef

Manually graded answer ▼

```
def flou(m):
    w, h = taille(m)
    m2 = image_nb_vide(w, h)
    # BEGIN SOLUTION
    for i in range(1, w-1):
        for j in range(1, h-1):
            s = 0
            for di in range(i-1, i+2):
                for dj in range(j-1, j+2):
                    s += m[dj][di]
            m2[j][i] = s // 9
    # END SOLUTION
    return m2
affiche_image(flou(racoon_g))
```



- ▼

Pour la fonction suivante, on suppose que l'image a des dimensions toutes paires ($2p, 2q$), on peut donc regrouper les pixels en carrés 2x2 ainsi :

```

a11 b11 a12 b12 ... a1p b1p
c11 d11 c12 d12 ... c1p d1p
...
aql bql ... aqp bqp
cql dql ... cqp dqp

```

et on vous demande de les réorganiser ainsi :

```

a11 a12 ... a1p b11 b12 ... b1p
...
aql aq2 ... aqp bq1 bq2 ... bqp
c11 c12 ... c1p d11 d12 ... d1p
...
cql cq2 ... cqp dq1 dq2 ... dqp

```

En quelque sorte, on veut l'image des pixel en haut à gauche, l'image des pixels en haut à droite, ...

Le résultat attendu ici est

Entrée [22]:

Points: 1

ID: cell-2a32c0aa2c1c7ca3

Manually graded answer ▼

```

def split4(m):
    w, h = taille(m)
    m2 = image_nb_vide(w, h)
    # BEGIN SOLUTION
    p = w//2
    q = h//2
    for i in range(w):
        for j in range(h):
            mi = i % 2
            mj = j % 2
            m2[j//2 + mj * q][i//2 + mi * p] = m[j][i]
    # END SOLUTION
    return m2
affiche_image(split4(racoon_carre))

```



- ▼

Dans la cellule suivante on effectue une boucle d'affichage en itérant la fonction précédente sur racoon_carre : racoon_carre , split4(racoon_carre) , split4(split4(racoon_carre)) , ...

Vous pouvez expliquer ?

Entrée [23]:

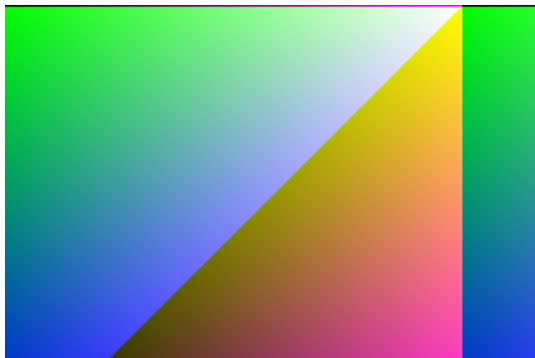
```
m = racoon_carre
for i in range(10):
    # Enlevez le commentaire suivant, il empeche de sauvegarder le notebook...
    # affiche_image(m)
    m = split4(m)
```

3 Images en couleur

On peut également faire des images en couleurs en rajoutant une dimension : chaque pixel n'est plus un entier, mais une liste ou un tuple à trois éléments [r,g,b] donnant les quantités, de 0 à 255, de chaque composante rouge, verte ou bleue.

Entrée [24]:

```
def dégradé_couleur(w,h):
    m = []
    for j in range(h):
        l = []
        for i in range(w):
            l.append( [ i%256, (256-j)%256, (i+j)%256 ] )
        m.append(l)
    return m
affiche_image(dégradé_couleur(300,200))
```



ID: cell-091a493c7fb3ffd6

Read-only

On donne aussi une image pour faire les tests appelées racoon .

Entrée [25]:



ID: cell-d2f2f610e0bf2c17

Read-only

```
affiche_image(racoon)
```



ID: cell-4a8f98b1cfcf16f1

Read-only

Reprendre les fonctions précédentes dans le cas de la couleur !



ID: cell-b5133b395e540834

Read-only

4 Bonus

Allez chercher la définition de la fractale de Mandelbrot et faites un calcul de cette fractale dans notre format d'image.