

# 1 Images bitmap

## Bloc-Note 1

Une image numérique de forme rectangulaire peut être découpée en petits carrés élémentaires appelés **pixels** pour *picture element*.

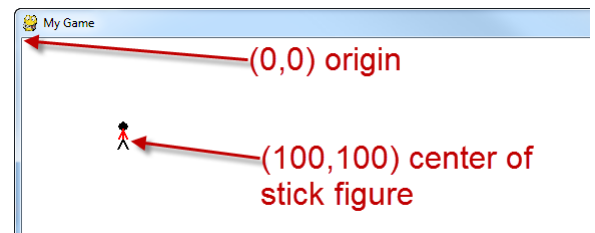
La **définition** de l'image est donnée par le couple (**L**, **H**), représentant le nombre de pixels en largeur et en hauteur.

Un format de fichier **bitmap** permet de stocker les valeurs des pixels sous la forme d'un tableau de **L** colonnes par **H** lignes.

On distingue trois formes pour la valeur d'un pixel selon le type de l'image :

- une valeur parmi deux, sur une échelle de 0 à 1, pour une **image binaire**, en noir et blanc;
- une valeur parmi 256, sur une échelle de 0 à 255, pour une **image en niveaux de gris**;
- trois valeurs de 0 à 255, représentant trois composantes Rouge, Vert et Bleu, pour une **image en couleurs**.

Dans une image **bitmap**, chaque pixel est repéré par sa colonne (abscisse), de 0 à  $L - 1$ , et par sa ligne (ordonnée), de 0 à  $H - 1$ , mesurées de gauche à droite et de haut en bas, par rapport au pixel origine situé dans le coin supérieur gauche de l'image.



Une image **bitmap** est généralement stockée dans un format de **fichier binaire**, comme PNG, JPEG ... Ces fichiers ne sont pas lisibles par l'homme, ce sont des suites d'**octets**. Le nombre d'octets codant un pixel définit la **profondeur** de l'image : par exemple 1 octet soit 8 bits pour une image en niveaux de gris ou 3 octets soit 24 bits pour une image en couleurs. Pour éditer un fichier binaire, il faut un éditeur hexadécimal comme l'éditeur en ligne <https://hexed.it/>.

Une image **bitmap**, peut aussi être stockée, dans un format de **fichier texte**, comme PBM, PGM, PPM. Un fichier texte est une suite de caractères lisibles par l'homme, à travers un éditeur de textes comme BlocNote ou NotePad++.

# 2 Fichiers textes et images bitmap

## Bloc-Note 2 Lecture / écriture de fichiers textes

En Python, l'accès à un fichier texte se fait par l'intermédiaire d'un descripteur de fichier créé à l'aide de `f = open(nom, mode)`. Une fois que les manipulations sont terminées, il faut bien penser à fermer le descripteur de fichier avec `f.close()`.

Lors de l'ouverture d'un fichier on précise l'un des trois modes d'accès : lecture '`r`', écriture '`w`' ou ajout '`a`'. Attention, pour l'ouverture en mode écriture, les modes '`w`' ou '`a`' créent le fichier s'il n'existe pas mais le mode '`w`' écrase le fichier s'il existe déjà.

⚠ Avant de manipuler un fichier texte, il faut connaître sa structure!!!

⚠ Un fichier texte se lit comme une bande magnétique, un curseur se déplace dans le fichier de caractère en caractère ou de ligne en ligne. On ne peut pas accéder à la ligne 100 sans lire les 99 précédentes!!!

Fonctions	Rôle
<code>f = open('nom_fichier.txt', 'w')</code>	accès en écriture avec création d'un nouveau fichier
<code>f = open('nom_fichier.txt', 'r')</code>	accès à un fichier existant en mode lecture
<code>f = open('nom_fichier.txt', 'a')</code>	accès en écriture à un fichier existant en mode ajout
<code>f.write('texte')</code>	ajout de 'texte' dans le fichier
<code>f.writelines(liste)</code>	ajout d'une liste de lignes dans un fichier
<code>f.read()</code>	lecture de tout le fichier
<code>f.read(8)</code>	lecture des 8 premiers caractères du fichier
<code>f.readline()</code>	lecture de la ligne courante du fichier
<code>f.readlines()</code>	liste de toutes les lignes à partir de la position courante
<code>f.tell()</code>	position courante du curseur
<code>f.seek(16)</code>	place le curseur sur le caractère en position 16
<code>for ligne in f</code>	itération sur les lignes du fichier
<code>f.close()</code>	fermeture du fichier

#### Lecture de tout le fichier

```
f = open('fichier.txt', 'r')
data = f.read()
f.close()
```

#### Lecture ligne par ligne

```
f = open('fichier.txt', 'r')
for ligne in f:
    #traitement sur la ligne
f.close()
```

#### Lecture ligne par ligne

```
f = open('fichier.txt', 'r')
ligne = f.readline()
while ligne != '':
    #traitement sur la ligne
    ligne = f.readline()
f.close()
```

#### Capture dans une liste de toutes les lignes

```
f = open('fichier.txt', 'r')
listlignes = f.readlines()
f.close()
```

#### Écriture

```
f = open('fichier.txt', 'w')
f.write('Une ligne qui ecrase
      tout\n')
f.close()
```

#### Ajout à la fin

```
f = open('fichier.txt', 'a')
f.write('Ligne de plus\n')
f.close()
```

## Exercice 1 Images binaires au format PBM

Le format PBM est un format de fichier texte permettant de stocker des images bitmap en noir et blanc. La première ligne du fichier précise le format avec l'identifiant P1, la seconde définit la largeur L et la hauteur H de l'image et à partir de la troisième ligne le tableau de pixels est stocké ligne par ligne, chaque pixel étant codé par 0 pour un pixel blanc et 1 pour un noir.

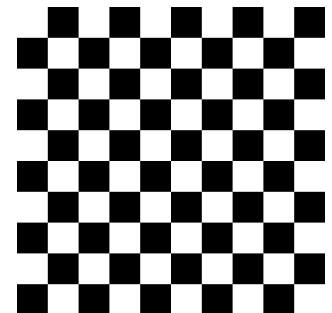
On donne ci-dessous l'exemple d'une image de définition  $4 \times 5$  représentant un F.

```
P1
4 5
1 1 1 1
1 0 0 0
1 1 1 0
1 0 0 0
1 0 0 0
```



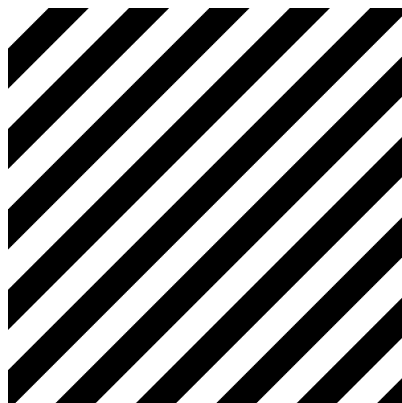
1. Déterminer la colonne et la ligne du pixel blanc situé à droite de la barre centrale du F. On commence la numérotation des lignes et des colonnes à partir de 0.
2. Le code ci-dessous permet de créer l'image d'un damier de définition  $500 \times 500$  dont les cases sont des carrés de 50 pixels. La suite de caractères `'\n'` code un caractère spécial, le **saut de ligne**.

```
L, H = 500, 500
f = open('damier.pbm', 'w')
f.write('P1\n')
f.write(str(L) + ' ' + str(H) + '\n')
for lig in range(H):
    for col in range(L):
        f.write(str((lig // 50 + col // 50) % 2) + ' ')
    f.write('\n')
f.close()
```



Écrire un code qui génère l'image ci-dessous, de définition  $500 \times 500$ , constituée de bandes noires parallèles dont les côtés coïncidant avec un bord de l'image, mesurent 50 pixels.

Image à générer



3. Recopier et compléter la fonction `inverser_couleurs_pbm(source, but)` qui crée un fichier de format PBM `but` obtenu à partir d'un fichier PBM `source` en inversant les valeurs de chaque pixel (0 devient 1 et vice-versa).

```
def inverser_couleurs_pbm(source, but):
    """Lit le fichier pbm source et le recopie dans le fichier pgm
    but en inversant les couleurs"""
    #ouverture de fichiers en lecture pour source et écriture pour but
    f = open(source, 'r')
    g = open(but, 'w')
    #on recopie l'en-tête (les deux premières lignes)
    for k in range(2):
        lig = f.readline()
        g.write(lig)
    #pour les lignes codant les pixels on inverse chaque valeur de pixel
    for lig in f:
        for caractere in lig:
            #à compléter
    g.close()
    f.close()
```

## Exercice 2 Images en niveaux de gris au format PGM

Le format PGM est un format de fichier texte permettant de stocker des images en niveaux de gris. La première ligne du fichier précise le format avec l'identifiant P2, la seconde définit la largeur L et la hauteur H de l'image et la troisième fixe la valeur maximale de niveau de gris.

À partir de la quatrième ligne le tableau de pixels est stocké ligne par ligne, chaque pixel est codé par son niveau de gris entre le minimum 0 et le maximum (255 en général).

On donne ci-dessous l'exemple de l'image `lena.ppm` fournie avec les ressources du DM. Seuls les six premiers pixels des deux premières lignes du tableau de pixels sont représentés.

```
P2
512 512
255
162 162 164 162 161 157 ....
162 162 164 162 161 157 ....
```



1. Dans une image au format PGM, le niveau de gris d'un pixel est codé sur un nombre variable de caractères de 1 à 3, alors que dans un format binaire comme PNG, le niveau de gris est stocké sur un seul octet. Un fichier texte est donc plus lisible par un humain mais un fichier binaire est plus compact et plus facile à manipuler par la machine.

Pour découper en pixels une ligne du tableau de pixels dans un fichier PGM, il faut utiliser la méthode `split` des chaînes de caractères.

```
In [19]: ligne = "162 162 164 162 161 157"

In [20]: liste_pixels = ligne.split()

In [21]: liste_pixels
Out[21]: ['162', '162', '164', '162', '161', '157']
```

À l'aide des indications précédentes, écrire une fonction `rechercher_pixel_ppm(source, lig, col)` qui retourne la valeur entière du pixel situé en ligne `lig` et colonne `col` dans un fichier image `source` au format PGM. Attention, il faut penser à sauter les trois lignes d'en-tête du fichier!

```
In [22]: rechercher_pixel_ppm("lena.ppm", 1, 5)
Out[22]: 157
```

- On a expliqué précédemment comment découper la ligne d'un fichier PGM en liste de pixels. Le `slicing` et la méthode `join` des listes permettent d'inverser la liste et de reconstruire une ligne correspondante en séparant chaque pixel par un espace.

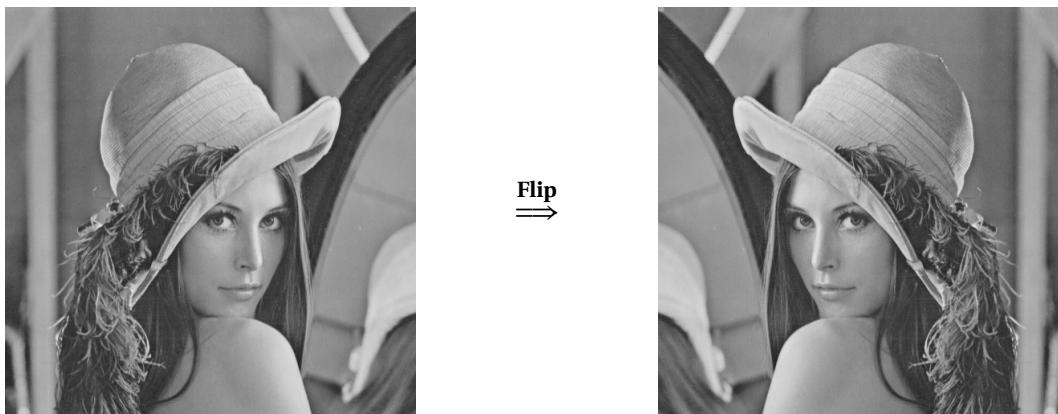
```
In [25]: liste_pixels
Out[25]: ['162', '162', '164', '162', '161', '157']

In [26]: liste_pixels[::-1]
Out[26]: ['157', '161', '162', '164', '162', '162']

In [27]: ' '.join(liste_pixels[::-1])
Out[27]: '157 161 162 164 162 162'
```

Écrire une fonction `flip_ppm(source, but)` qui ouvre un fichier image `source` au format PGM et le recopie dans un fichier image `but` au format PGM en appliquant à l'image un flip c'est-à-dire une symétrie axiale par rapport au bord droit de l'image.

Tester la fonction sur l'image `lena.ppm`.



- Dans un fichier texte, on ne peut accéder à la dernière ligne qu'en parcourant d'abord toutes les précédentes. La méthode `readlines` permet de stocker dans une liste toutes les lignes d'un fichier texte à partir de la position courante et l'itérateur `reversed` permet de parcourir une liste dans l'ordre inverse. Le code ci-dessous permet de recopier le fichier `endroit.txt` dans le fichier `envers.txt` en gardant la première ligne mais en inversant l'ordre des lignes suivantes.

## Code

```
f = open('endroit.txt', 'r')
g = open('envers.txt', 'w')
#saut de la première ligne
lig = f.readline()
g.write(lig)
liste_lignes = f.readlines()
for lig in reversed(liste_lignes):
    g.write(lig)
f.close()
g.close()
```

endroit.txt

```
#En-tete
0 0 0
1 1 1
2 2 2
```

envers.txt

```
#En-tete
2 2 2
1 1 1
0 0 0
```

À l'aide des indications précédentes, écrire une fonction `flop_ppm(source, but)` qui ouvre un fichier image `source` au format PGM et le recopie dans un fichier image `but` au format PGM en appliquant à l'image un flop c'est-à-dire une symétrie axiale par rapport au bord supérieur de l'image.

Tester la fonction sur l'image `lena.ppm`.



Flop  
⇒



### 3 Fichiers binaires et images bitmap

Le fichier `lenagray.png` fourni avec les ressources du DM, stocke la même image que `lena.ppm` mais comme un fichier binaire au format PNG, c'est-à-dire comme une suite d'octets qui n'est pas lisible par l'homme. Après un en-tête spécifique au format et précisant les dimensions de l'image, le niveau de gris de chaque pixel est codé sur un octet (trois si l'image était en couleur, voir plus si on code aussi la transparence).

Si on édite le fichier avec un éditeur hexadécimal comme Okteta ou l'éditeur en ligne <https://hexed.it/>, on obtient à gauche les octets codés sur deux chiffres en hexadécimal (base 16) et à droite leurs traductions en caractères dans un jeu de caractères ASCII (ici le jeu étendu ISO-8859-1). Par exemple, le niveau de gris  $75 = 4 \times 16 + 11$  correspondra à l'octet `4b` en hexadécimal et au caractère `K`.

_lenagray.png			
0000:0000	89 50 4E 47	0D 0A 1A 0A	00 00 00 0D 49 48 44 52
0000:0010	00 00 02 00	00 00 02 00	08 00 00 00 00 D1 13 8B
0000:0020	26 00 00 00	09 70 48 59	73 00 00 0B 13 00 00 0B
0000:0030	13 01 00 9A	9C 18 00 00	00 07 74 49 4D 45 07 DE
0000:0040	0C 13 0E 1B	30 E6 CD 14	B3 00 00 20 00 49 44 41
0000:0050	54 78 DA EC	BB C1 D2 2D	49 72 9C E7 EE 11 59 E7
0000:0060	FC 3D D3 1C	42 32 A3 E9	99 20 00 03 F2 ED B8 D5
			.PNG.....IHDR
			.....N..
			&....pHYs.....
			.....tIME.P
			...0æI.³...IDA
			TxÜi»Ä0-Ir.çi.Yç
			ü=0.B2fé. .ôi.0

On peut déchiffrer à partir du deuxième octet du fichier l'identifiant PNG du format. Ainsi le caractère P, qui a pour ordinal 80 dans le jeu de caractères, correspond à l'octet 50 en hexadécimal puisque  $80 = 5 \times 16 + 0$ .

```
In [17]: int('0x50', 16) #conversion en base dix de 50 en base seize
Out[17]: 80

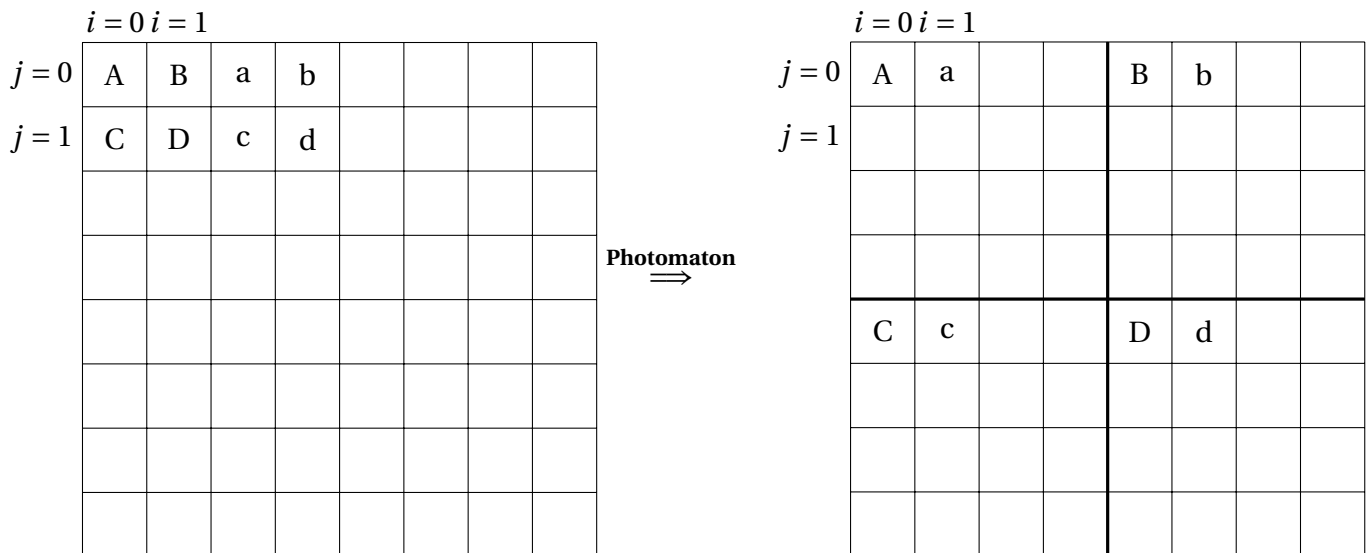
In [18]: chr(80)        #caractère unicode d'ordinal 80
Out[18]: 'P'
```

### Exercice 3 Transformation du photomaton

On considère une image bitmap de dimensions  $n \times n$  avec  $n$  pair, comme lenagray.png de dimensions  $512 \times 512$ .

Les lignes sont indexées de  $j = 0$  à  $j = n - 1$  et les colonnes de  $i = 0$  à  $i = n - 1$ .

À partir du tableau de pixels de cette image source, on construit le tableau de pixels d'une image but selon la transformation suivante appelée **transformation du photomaton**.



- On découpe l'image en petits carrés de dimensions  $2 \times 2$  et l'image en quatre secteurs carrés de dimensions  $\frac{n}{2} \times \frac{n}{2}$ .
- Pour chaque petit carré :
  - Le pixel en haut à gauche de coordonnées  $(i, j)$  avec  $i$  pair et  $j$  pair est envoyé dans le secteur en haut à gauche de l'image, en position  $(i//2, j//2)$ .
  - Le pixel en haut à droite de coordonnées  $(i, j)$  avec  $i$  impair et  $j$  pair est envoyé dans le secteur en haut à droite de l'image, en position  $((i + n)//2, j//2)$ .
  - Le pixel en bas à gauche de coordonnées  $(i, j)$  avec  $i$  pair et  $j$  impair est envoyé dans le secteur en bas à gauche de l'image, en position  $(i//2, (n + j)//2)$ .
  - Le pixel en bas à droite de coordonnées  $(i, j)$  avec  $i$  impair et  $j$  impair est envoyé dans le secteur en bas à droite de l'image, en position  $((i + n)//2, (j + n)//2)$ .
- Ainsi, on construit une correspondance unique entre un pixel de l'image source et un pixel de l'image but. La transformation du photomaton est une **transformation bijective** du tableau de pixels.



- Voici un exemple d'application de la transformation du photomaton à un tableau de pixels de dimensions  $4 \times 4$  :

1	2	3	4		1	3	2	4
5	6	7	8	Photomaton	9	11	10	12
9	10	11	12	⇒	5	7	6	8
13	14	15	16		13	15	14	16

Si on applique par itérations successives la transformation du photomaton à partir de `lennagray.png`, image source de dimensions  $512 \times 512$ , voici les images itérées qu'on obtient :

Source



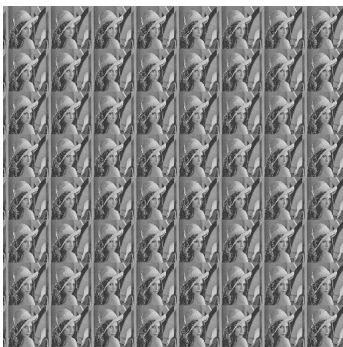
Itération 1



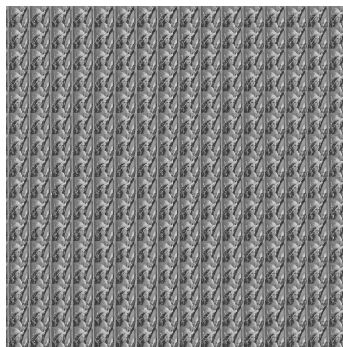
Itération 2



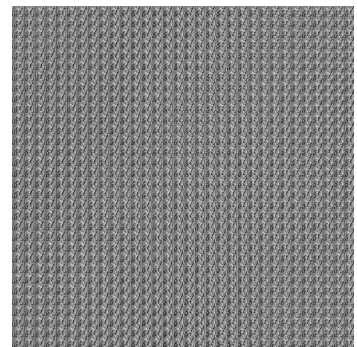
Itération 3



Itération 4



Itération 5



### Bloc-Note 3 Modules PIL et numpy

- Nous allons naturellement représenter le tableau de pixels d'une image bitmap par une liste de listes ou tableau à deux dimensions (matrice en mathématiques). Pour ouvrir les fichiers images, on va utiliser `Image` le sous-module du module `PIL`, qui va nous renvoyer des objets spécifiques qui ne se convertissent pas en listes de listes mais en `ndarray` avec le module `numpy`. Nous manipulerons ces derniers comme des listes de listes.
- Les modules `numpy` et `PIL` (en fait son fork `pillow`) sont inclus dans la distribution `Anaconda` mais si on a juste installé la distribution standard de `Python`, on peut les installer depuis un interpréteur de commandes du système (`cmd.exe` sous `Windows`) avec les commandes :

```
C:\python3 -m pip install numpy
C:\python3 -m pip install pillow
```

- Pour extraire le tableau de pixels d'une image bitmap stockée dans un fichier binaire comme `lennagray.png`, on utilise `Image.open` du module `PIL` puis `np.asarray` du module `numpy`.



⚠ Attention, pour que Python trouve le fichier image, il est recommandé de placer ce fichier dans le même dossier que le script et de régler le répertoire de travail avec par exemple l'option « Exécuter le fichier en tant que script » sous Pyzo.

```
In [5]: from PIL import Image

In [6]: import numpy

In [7]: im = Image.open('lenagray.png')

In [8]: tab = numpy.asarray(im)

In [9]: tab
Out[9]:
array([[162, 162, 164, ..., 166, 153, 129],
       [162, 162, 164, ..., 166, 153, 129],
       [162, 162, 164, ..., 166, 153, 129],
       ...,
       [ 54,  54,  59, ..., 110, 105, 106],
       [ 53,  53,  63, ..., 109, 111, 113],
       [ 53,  53,  63, ..., 109, 111, 113]], dtype=uint8)

In [10]: tab[1][2] #pixel en deuxième ligne et troisième colonne
Out[10]: 164
```

👉 ⚠ Attention, comme on le voit ci-dessus le pixel de coordonnées (colonne, ligne) = (2, 1) est accessible par `tab[1][2]`. On indexe d'abord la ligne (l'ordonnée) puis la colonne (l'abscisse) et on rappelle que les deux sont numérotées à partir de 0.

👉 Réciproquement, une fois le traitement sur le tableau de pixels effectué, on peut construire l'image associée. Dans un premier temps, on convertit le tableau en `ndarray` pour l'interface avec `Image` puis on utilise `Image.fromarray` pour construire une image qu'on enregistre l'image sur le disque avec la méthode `save`.

⚠ Pour que le tableau de pixels corresponde bien à une image en niveau de gris, il faut préciser le type `'uint8'` lors de la conversion en `ndarray`.

```
In [12]: tab = [ [0 for col in range(512)] for lig in range(512)]

In [13]: type(tab)
Out[13]: list

In [14]: tab = numpy.array(tab, dtype = 'uint8') #liste -> ndarray

In [15]: type(tab)
Out[15]: numpy.ndarray

In [16]: im = Image.fromarray(tab)

In [17]: im.save('tableau-noir.png') #sauver l'image sur le disque

In [18]: im.show() #afficher l'image avec la visionneuse
```

1. Écrire une fonction `transformation(i, j, n)` qui retourne les coordonnées de l'image du pixel de coordonnées  $(i, j)$  par la transformation du photomaton.
2. On donne ci-dessous le code d'une fonction `copie(tableau)` qui retourne la copie profonde d'un tableau de pixels (liste de listes) sous forme de `ndarray`.

```
def copie(tableau):  
    """Retourne la copie profonde d'un tableau à deux dimensions"""  
    return numpy.array([[tableau[j][i] for i in range(len(tableau[j]))]  
                        for j in range(len(tableau))], dtype = 'uint8')
```

À l'aide de cette fonction, écrire une fonction `photomaton(tableau)` qui retourne le tableau calculé après application de la fonction `photomaton` au tableau de pixels passé en paramètre.

3. Recopier et compléter le code de la fonction `photomaton_iterer(source, k)` qui itère  $k$  fois la fonction `photomaton` à partir du fichier image `source` et enregistre à chaque itération au format `png`, l'image correspondant au tableau de pixels obtenu.

```
def photomaton_iterer(source, k):  
    im = Image.open(source)  
    nom, extension = source.split('.')  
    tab = numpy.asarray(im)  
    for iteration in range(1, k + 1):  
        #to be completed
```

4. Que remarque-t-on si on itère 9 fois la fonction `photomaton` à partir de l'image `lenagray.png`, de dimensions  $512 \times 512$ ?
5. Tester la fonction `photomaton_iterer(source, k)` sur des images sources de dimensions  $128 \times 128$  ou  $256 \times 256$  en essayant de retrouver le même phénomène que pour une image de dimensions  $512 \times 512$ .  
Quelle conjecture peut-on faire?