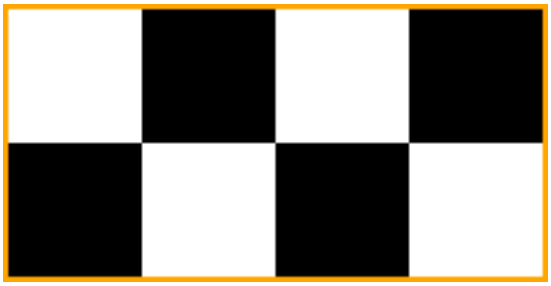
Images et tableaux à plusieurs dimensions

Thème types construits

1 Représentation d'une image bitmap

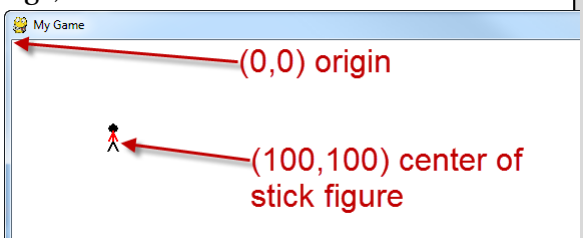
Point de cours 1

On considère l'image constituée de 2 lignes et 4 colonnes de carrés noirs ou blancs ci-dessous. Le cadre orange ne fait pas partie de l'image.



Tous les carrés étant superposables, l'image peut être entièrement décrite par la liste des couleurs et des positions de chaque carré.
Chaque carré est un composant élémentaire de l'image qu'on nomme **picture element** ou **pixel**.
Pour coder la couleur on a besoin de deux entiers et 1 bit d'information suffit : par exemple 0 pour le noir et 1 pour blanc. La quantité d'information nécessaire pour coder la couleur d'un pixel est la **profondeur de l'image**.
Pour repérer la position d'un pixel il est naturel d'utiliser le couple (index de colonne, index de ligne) ou (abscisse, ordonnée) qui apparaît lorsqu'on représente l'image par un tableau à deux dimensions.
Cette représentation d'une image par un tableau à deux dimensions est appelée **représentation bitmap**.
On parle de **matrice de pixels** car toutes les lignes du tableau ont le même nombre de colonnes.
Traditionnellement, les lignes et les colonnes sont indexées à partir de 0, de gauche à droite et de haut en bas, en partant du pixel origine situé dans le coin supérieur gauche de l'image.
Le nombre de lignes ou **hauteur** de l'image et le nombre de colonnes ou **largeur** de l'image, forment un couple noté (**largeur** × **hauteur**) qui donne la **définition** de l'image, ici 4 × 2.

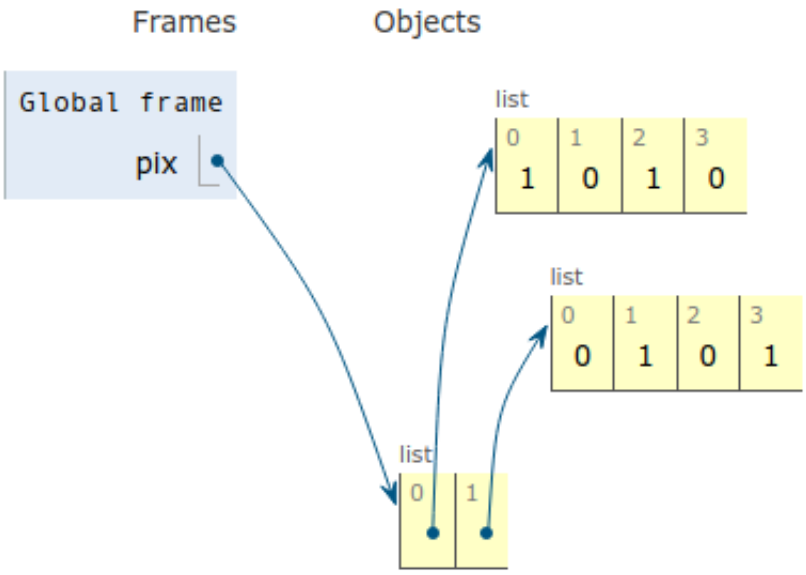
ligne / colonne	0	1	2	3
0	**1**	**0**	**1**	**0**
1	**0**	**1**	**0**	**1**



En `Python`, une matrice de pixels peut être représentée comme un tableau de tableaux, ou liste de listes dans la nomenclature `Python`. On utilise l'opérateur crochet une fois pour accéder à une ligne par son index et deux fois pour accéder à un pixel par ses index de ligne puis de colonne.

```
>>> pix = [[1, 0, 1, 0], [0, 1, 0, 1]]
>>> pix[0] #première ligne
[1, 0, 1, 0]
>>> pix[1] #deuxième ligne
[0, 1, 0, 1]
>>> pix[0][2] #pixel en 1ere ligne et 3eme colonne
1
```

Dans la représentation donnée par http://pythontutor.com, on voit bien que `pix[0]`, de type `list`, est une référence vers le tableau de pixels représentant la première ligne.



Exercice 1

1. Récupérer l'archive `Images-Tableaux2d-materiel.zip`, la copier dans un répertoire pertinent et la déballer.

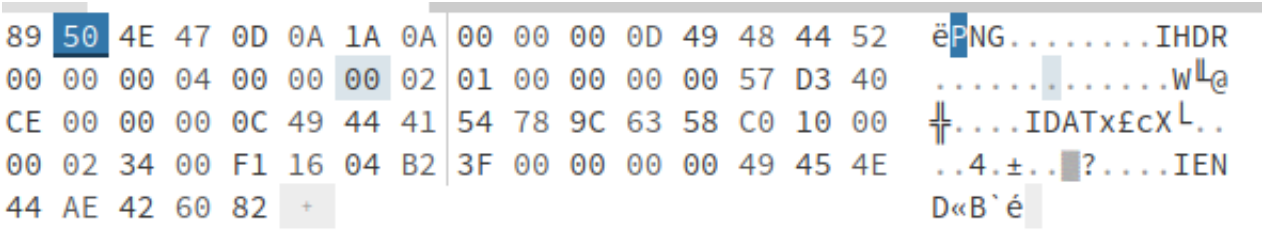
2. Ouvrir le fichier `Images-Tableaux2d-Eleves-Partie1.py` dans un IDE `Python` et le renommer éventuellement.

3. **a.** Créer l'image présentée dans le point de cours 1, en évaluant dans la console l'expression :

```
>>> matrice_to_image([[1,0,1,0],[0,1,0,1]], mode = '1', fichier=
    'exemple_binaire_4x2.png',res=1)
```

   Un fichier au format PNG a été créé sur le disque. Il s'agit d'un **fichier binaire** lisible uniquement par une machine car il s'agit d'une suite d'**octets**.

   b. Éditer le fichier avec un éditeur hexadécimal comme https://hexed.it/. Voici une capture d'écran du résultat avec 16 octets par ligne :



c. Dans cette question on va décoder le contenu de ce fichier binaire où chaque octet est représenté par un nombre de deux chiffres en hexadécimal.

Ouvrir avec un navigateur web la page d'URL https://fr.wikipedia.org/wiki/Portable_Network_Graphics

Pourquoi a-t-on créé le format PNG?

Dans quels cas l'utilise-t-on?

d. La structure d'un fichier PNG est la suivante :

- signature PNG sur 8 octets
- chunk (morceau de fichier) IHDR pour l'en-tête sur 25 octets
- chunk IDAT pour les données de longueur variable
- chunk IEND pour la fin de fichier sur 12 octets

De plus un chunk est composé de 4 parties :

LENGTH	TYPE	DATAS	CRC
Longueur des données	Type de chunk	Données dont la longueur en octets est spécifiée dans LENGTH	Contrôle
4 octets	4 octets	n octets	4 octets

Entourer sur la capture d'écran du fichier, les quatre parties et les octets représentant respectivement : la largeur, la hauteur, la profondeur de l'image (chunk IHDR), les données (chunk IDAT).

Quel est le rôle des CRC?

2 Tableaux à 2 ou n dimensions

Méthode *Manipulations de tableaux à 2 ou n dimensions*

Un tableau `Python`, de type `list`, est un conteneur de type séquence qui peut contenir toutes sortes de valeurs, y compris des tableaux. On obtient ainsi une structure de conteneurs imbriqués , on parle de **tableau à plusieurs dimensions**. Les opérations sur ces tableaux sont les mêmes que sur les tableaux à une dimension présentés dans un chapitre précédent, il faut juste les répéter à chaque niveau d'imbrication.
Par exemple `len` permet d'obtenir la taille d'un tableau qu'il soit conteneur ou élément.
La dimension d'un tableau imbriqué est le nombre de niveaux d'imbrication. Il est possible que tous les éléments n'aient pas la même dimension mais nous ne manipulerons pas ce type de tableaux.

- **Construction**
  - On peut construire un tableau à plusieurs dimensions par **extension**. Lorsque le tableau a deux dimensions et que toutes les lignes sont de même taille, on peut le qualifier de **matrice**.

```
>>> t1 = [[1,2], [3,4]]  #tableau/matrice à 2 dimensions
>>> t2 = [[1,2], [3,4],[5,6,7]] #tableau à 2 dimensions
>>> t3 = [[[1], [2,3]],[4,5,6],[7]] #tableau mixte
>>> len(t2)  #t2 contient 2 tableaux éléments
3
>>> len(t2[0]) #t2[0] est un tableau contenant 2 entiers
2
>>> len(t2[2]) #t2[2] est un tableau contenant 3 entiers
3
```

– On peut construire un tableau à plusieurs dimensions par **compréhension** :

```
>>> t4 = [[0] * 3 for _ in range(2)]
>>> t4
[[0, 0, 0], [0, 0, 0]]
>>> t5 = [[ [0] * 4 for i in range(3)] for j in range(2)]
>>> t5
[[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0,
    0, 0, 0], [0, 0, 0, 0]]]
```

- **Lecture / écriture**

  On peut lire ou écrire dans un tableau à plusieurs dimensions en traversant les différents niveaux d'imbrication de l'extérieur vers l'intérieur avec l'opérateur crochet :

```
>>> t1[0][1]
2
>>> t1[0][1] = 734
>>> t1
[[1, 734], [3, 4]]
>>> t5[1][2][3] = t1[0][1]
>>> t5
[[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0,
    0, 0], [0, 0, 0, 734]]]
```

- **Parcours**

  Pour parcourir un tableau à plusieurs dimensions, il faut parcourir chaque niveau d'imbrication : par index ou élément par élément. Il faut donc connaître la structure du tableau avant de le parcourir.

```
>>> def parcours_tableau2d_index(tab):
...     for i in range(len(tab)):
...         for j in range(len(tab[i])):
...             print('Element en ligne {} colonne {} : '.format(i,j)
    ,tab[i][j])
...
>>> parcours_tableau2d_index(t1)
Element en ligne 0 colonne 0 : 1
Element en ligne 0 colonne 1 : 734
```

```
Element en ligne 1 colonne 0 : 3
Element en ligne 1 colonne 1 : 4
>>> def parcours_tableau2d_element(tab):
...     for ligne in tab:
...             for element in ligne:
...                     print(element)
...
>>> parcours_tableau2d_element(t1)
1
734
3
4
```

Exercice 2

1. Le site http://pythontutor.com/visualize.html fournit un outil de visualisation de l'état courant d'un programme très pratique.

   Exécutez le code ci-dessous dans http://pythontutor.com/visualize.html#mode=edit puis commentez.

```
M = [ [0, 0, 0] for i in range(3) ]
N = M
P = [e for e in M ]
Q = [ e[:] for e in M ]
M[2][1] = 3
```

2. Compléter la fonction `maxi_tab2d(tab)` pour qu'elle retourne le maximum d'un tableau de nombres à deux dimensions.

```
def max_tab2d(tab):
    """Retourne le maximum d'un tableau à 2 dimensions"""
    maxi = float('-inf')
    for y in range(len(tab)): #boucle sur les lignes
        for x in range(len(tab[y])): # boucle sur les colonnes
            "à  compléter"
```

   Les assertions suivantes doivent être vérifiées :

```
assert max_tab2d([[-1,-2],[-2,-3,-0.5]]) == -0.5
assert max_tab2d([[1,2],[float('inf'),10]]) == float('inf')
assert max_tab2d([[1,2],[8,0]]) == 8
assert max_tab2d([[8, float('-inf')],[]]) == 8
```

3. Écrire une fonction `moyenne_tab2d(tab)` qui retourne la valeur moyenne des valeurs d'un tableau de nombres à deux dimensions.

   Les assertions suivantes doivent être vérifiées :

```
assert moyenne_tab2d([[-1,-2],[-2,-3,-0.5]]) == -1.7
assert moyenne_tab2d([[1,2],[float('inf'),10]]) == float('inf')
assert moyenne_tab2d([[1,2],[8,0]]) == 2.75
```

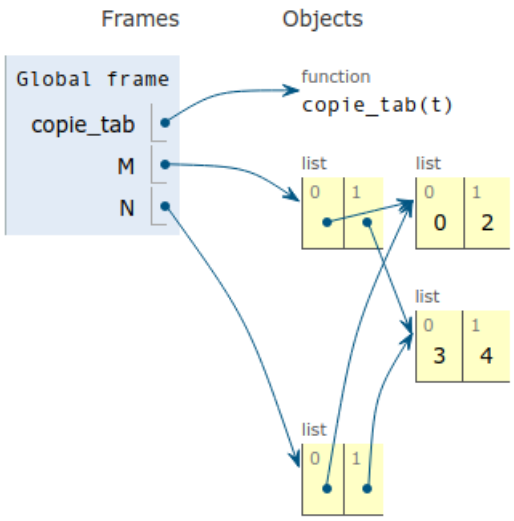
```
assert moyenne_tab2d([[8, float('-inf')],[]]) == float('-inf')
```

Méthode *Copie d'un tableau à plusieurs dimensions*

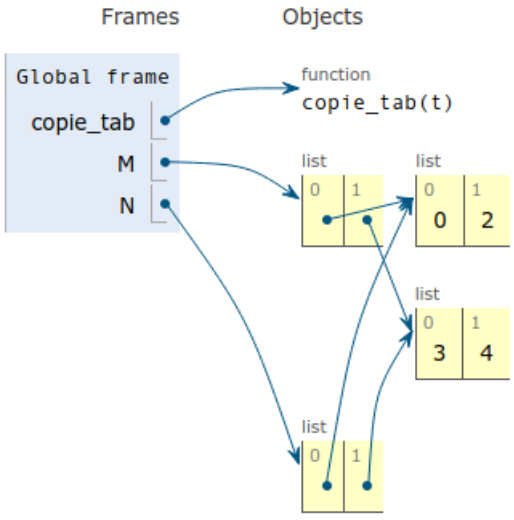
La valeur d'un tableau étant une référence vers une séquence de données en mémoire, un tableau à deux dimensions est une référence vers une séquence de références, ce qui nécessite un soin particulier pour déréférencer les tableaux imbriqués lors d'une opération de copie.
La fonction `copie_tab` ci-dessous ne retourne qu'une **copie superficielle** du tableau passé en paramètre, elle ne déréférence pas les éléments du tableau `M`.

```
1 def copie_tab(t):
2     res = []
3     for x in t:
4         res.append(x)
5     return res
6
7 M = [[1,2],[3,4]]
8 N = copie_tab(M)
9 M[0][0] = 0
```

Après l'exécution de la ligne 8, les éléments du tableau `N`, copie superficielle de `M`, sont des alias des éléments de `M` :



Après l'exécution de la ligne 9, la modification de `M` se traduit par un effet de bord sur `N` :



Pour copier en profondeur un tableau à plusieurs dimensions on peut utiliser la fonction `deepcopy()` du module `copy`.

```
>>> M = [[1,2],[3,4]]
>>> from copy import deepcopy
>>> N = deepcopy(M)
>>> M[0][0] = 0
>>> N
[[1, 2], [3, 4]]
```

Une explication très claire du caractère modifiable des listes est donnée dans cette video :

https://d381hmu4snvm3e.cloudfront.net/videos/MhgBaG50LRrH/SD.mp4

Exercice 3 *QCM type E3C2*

1. On considère le tableau `t` suivant.

```
t = [[1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6]]
```

Quelle est la valeur de `t[1][2]` ?

Réponses :

a. 1 **b.** 3 **c.** 4 **d.** 2

2. Quelle est la valeur de la variable image après exécution du script Python suivant?

```
image = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
for i in range(4):
    for j in range(4):
        if (i+j) == 3:
            image[i][j] = 1
```

Réponses :

a. `[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [1, 1, 1, 1]]`

b. `[[0, 0, 0, 1], [0, 0, 0, 1], [0, 0, 0, 1], [0, 0, 0, 1]]`

c. `[[0, 0, 0, 1], [0, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0]]`

d. `[[0, 0, 0, 1], [0, 0, 1, 1], [0, 1, 1, 1], [1, 1, 1, 1]]`

3. On définit une grille G remplie de 0, sous la forme d'une liste de listes, où toutes les sous-listes ont le même nombre d'éléments.

```
G = [ [0, 0, 0, ..., 0],
[0, 0, 0, ..., 0],
[0, 0, 0, ..., 0],

......
[0, 0, 0, ..., 0]]
```

On appelle *hauteur* de la grille le nombre de sous-listes contenues dans G et *largeur* de la grille le nombre d'éléments dans chacune de ces sous-listes. Comment peut-on les obtenir?

Réponses :

a.

```
hauteur = len(G[0])
largeur = len(G)
```

b.

```
hauteur = len(G)
largeur = len(G[0])
```

c.
```
hauteur = len(G[0])
largeur = len(G[1])
```

d.
```
hauteur = len(G[1])
largeur = len(G[0])
```

4. Quelle est la valeur de l'expression `[[0] * 3 for i in range(2)]` ?

   Réponses :

   a. `[[0,0], [0,0], [0,0]]`
   b. `[[0,0,0], [0,0,0]]`
   c. `[[0.000], [0.000]]`
   d. `[[0.00], [0.00], [0.00]]`

5. On exécute le script suivant :

```
asso = []
L=[['marc','marie'],['marie','jean'],
    ['paul','marie'], ['marie','marie'],
    ['marc','anne']]
for c in L :
    if c[1]=='marie':
        asso.append(c[0])
```

   Que vaut `asso` à la fin de l'exécution ?

   Réponses :

   a. `['marc', 'jean', 'paul']`
   b. `[['marc','marie'], ['paul','marie'], ['marie','marie']]`
   c. `['marc', 'paul', 'marie']`
   d. `['marie', 'anne']`

3 Traitement d'image

Exercice 4

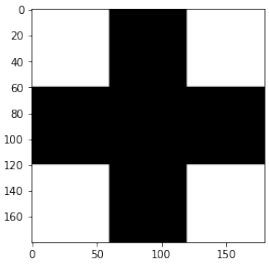
Ouvrir le fichier `Images-Tableaux2d-Eleves-Partie1.py` dans un IDE `Python`.

1. Le prototype d'une fonction doit indiquer son nom, le type de la valeur de retour et le type des paramètres. En `Python`, la liste des paramètres suit la déclaration de la fonction avec l'instruction `def`, les paramètres ne sont pas obligatoirement typés et le type de retour n'est pas précisé mais on doit pouvoir les déduire de la documentation. Déterminer le prototype de la fonction `matrice_to_image` qui est fournie.

2. Le paramètre `res` permet de définir le côté en pixels sur l'écran d'un pixel défini dans la matrice de pixels passée en paramètre. Cela permet de visualiser les images de petites dimensions. L'expression suivante permet d'obtenir l'image du point de cours n°1 avec des blocs de 100 pixels sur l'écran :

```
>> matrice_to_image([[1,0,1,0],[0,1,0,1]], fichier='exemple_binaire.
    png',res=50)
```

Quelle expression permet de générer l'image ci-dessous avec des blocs de 60 pixels écran ?



Point de cours 2

La **représentation bitmap** d'une image par une matrice de pixels ne se limite pas à des **images binaires** en noir (0) et blanc (1).
En modifiant la **profondeur** de l'image, on peut coder plus de couleurs dans un pixel :

- Avec une profondeur de 8 bits (1 octet), on peut stocker $2^8 = 256$ nuances par pixel. On utilise en général ce mode pour les **images en nuances de gris** du plus foncé 0 (noir) au plus clair 255 (blanc) mais on le retrouve pour des images qui n'ont pas besoin d'une palette de couleurs étendue comme dans le format GIF.

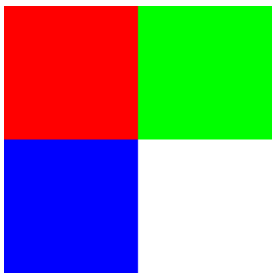
  Expression pour générer l'image en niveaux de gris ci-contre :

```
>>> pix = [[0,50,100],[150,200,255]]
>>> matrice_to_image(pix,mode = 'L',
fichier='exemple.png',res=100)
```

- Avec une profondeur de 24 bits (3 octets), on peut stocker $2^{24} \approx 16 \times 10^6$ nuances par pixel. En général on utilise ce mode pour la représentation des couleurs par synthèse additive de trois couleurs primaires (Rouge, Vert, Bleu). Un pixel est alors représenté par une liste de trois entiers entre 0 et 255 pour mesurer les intensités des trois composantes.

  Expression pour générer l'image en couleurs ci-contre :

```
>>> pix = [
          [[255,0,0],[0,255,0]],
          [[0,0,255],[255,255,255]]
        ]
>>> matrice_to_image(pix,mode = 'RGB',
fichier='exempleRGB.png',res=100)
```



Exercice 5

Compléter le code de la procédure `generer_croix` pour qu'elle génère une image de croix avec la couleur passée en paramètre :

```
def generer_croix(couleur):
    blanc = [255,255,255]
    croix = ..............
    matrice_to_image(croix, mode = 'RGB', res = 10, fichier='croix.png')
```

Par exemple, l'évaluation de `croix([255,0,0])` donne :

Exercice 6

1. Compléter le code de la fonction `matrice_vide` pour qu'elle retourne une matrice de pixels représentant une image noire dans le mode choisi (`'1'`, `'L'` ou `'RGB'`) avec `nlig` lignes et `ncol` colonnes :

```
def matrice_vide(ncol, nlig, mode):
    """Retourne une matrice de pixels de n lignes et m colonnes
    représentant une image noire dans le mode d'image choisi"""
    assert mode in ['1', 'L', 'RGB'], "mode doit appartenir à ['1',
        'L', 'RGB']"
    #compléter par des tableaux définis en compréhension
    if mode in ['1', 'L']:
        return ........................
    else:
        return ........................
```

2. Compléter le code de la fonction `drapeau_3bandes_verticales` ci-dessous pour que cette expression génère le drapeau national :

```
matrice_to_image(drapeau_3bandes_verticales
    (3,6,[0,0,255],[255,255,255], [255,0,0]), mode='RGB', fichier='
    drapeau-france.png', res = 100)
```

```
def drapeau_3bandes_verticales(nlig, ncol, couleur1, couleur2,
    couleur3):
    #on crée une matrice vide de bonnes dimensions
    pix = matrice_vide(ncol, nlig, 'RGB')
    tiers_colonne = ncol // 3
    deux_tiers_colonne = 2 * tiers_colonne
    for x in range(ncol): #boucle sur les colonnes
        for y in range(nlig): #boucle sur les lignes
            if   x < tiers_colonne:
                pix[y][x] = couleur1
```

```
            #à  compléter !
    return pix
```

3. Écrire une fonction `transpose(pix, mode)` qui échange les lignes et les colonnes de la matrice de pixels passée en paramètre. La matrice de pixels `tpix` retournée est telle que la valeur `tpix[y][x]` est celle de `pix[x][y]`.

   On utilisera la fonction `matrice_vide` et la fonction `dimensions` qui est fournie :

```
def dimensions(pix):
    "Retourne les dimensions (Largeur, Hauteur) de la matrice pix"
    return len(pix[0]), len(pix)
```

   La série d'assertions ci-dessous doit être vérifiée par la fonction `transpose` :

```
assert transpose([[0]],'L') == [[0]]
assert transpose([[1,2],[4,5]], 'L') == [[1,4],[2,5]]
assert transpose([[1,2,3],[4,5,6]],'L') == [[1,4],[2,5],[3,6]]
assert transpose([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]],'RGB') ==
    [[[1,2,3],[7,8,9]],[[4,5,6],[10,11,12]]]
```

4. En déduire l'écriture d'une fonction `drapeau_3bandes_horizontales`, telle que cette expression génère le drapeau de la Hollande :

```
matrice_to_image(drapeau_3bandes_horizontales(3,6,[255,0,0],
    [255,255,255],[0,0,255]),mode='RGB', fichier='drapeau-hollande.
    png', res = 100)
```

Exercice 7

L'opérateur modulo % permet de calculer le reste de la division euclidienne d'un entier `a` par un entier `b` avec la syntaxe `a % b`. On peut ainsi évaluer la parité d'un entier.

```
>>> 7 % 2
1
>>> 8 % 2
0
```

Dans cet exercice, on travaille sur des images binaires avec deux couleurs possibles noir (0) ou blanc (1).

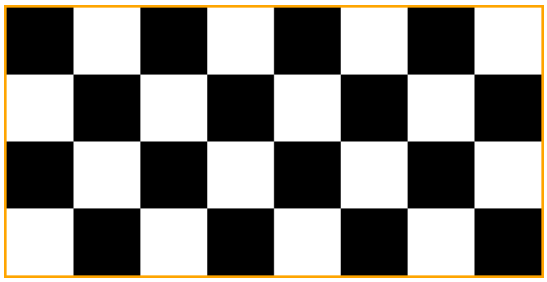
1. Compléter la fonction `barres_horizontales` pour qu'elle retourne la matrice de pixels d'une image de dimensions `ncol` × `nlig` avec alternance de lignes noires (index pair) ou blanches (index impair).

```
def barres_horizontales(nlig, ncol):
    #on crée une matrice vide de bonnes dimensions
    pix = matrice_vide(ncol, nlig, '1')
    for x in range(ncol): #boucle sur les colonnes
        for y in range(nlig): #boucle sur les lignes
            "à  compléter"
    return pix
```

L'expression `matrice_to_image(barres_horizontales(4, 5), mode='1', res = 50)` génère cette image (sans le cadre orange) :

2. Écrire une fonction `damier(nlig, ncol)` qui retourne la matrice de pixels permettant de générer un damier de cases blanches et noires avec `nlig` lignes et `ncol` colonnes.

L'expression `matrice_to_image(damier(4,8), mode='1', res = 50)` doit générer l'image ci-dessous (sans le cadre orange) :



Exercice 8 *Filtre pixel par pixel*

La fonction `applique_filtre` permet d'appliquer une fonction (ou filtre) à chaque pixel d'une matrice de pixels et de retourner une nouvelle matrice de mêmes dimensions avec les pixels transformés. Le paramètre `mode` désigne le mode de l'image transformée qui peut être différent de celui de l'image source.

```
def applique_filtre(pix, filtre, mode):
    ncol, nlig = dimensions(pix)
    pix_but = matrice_vide(ncol, nlig, mode)
    for x in range(ncol): #boucle sur les colonnes
        for y in range(nlig): #boucle sur les lignes
            pix_but[y][x] = filtre(pix[y][x])
    return pix_but

def filtre_negatif_gris(pixel):
    """Filtre négatif pour image en niveaux de gris"""
    return 255 - pixel
```

1. Avec la fonction `filtre_negatif_gris`, on peut générer le négatif d'une image en niveaux de gris comme `exemple_gris.png` qui est fournie. Exécuter la séquence d'instructions ci-dessous :

```
>>> exemple_gris = image_to_matrice('exemple_gris.png')
>>> exemple_gris_negatif = applique_filtre(exemple_gris,
    filtre_negatif_gris, 'L')
```

```
>>> matrice_to_image(exemple_gris_negatif, fichier='
    exemple_gris_negatif.png', mode='L', res=1)
```

exemple_negatif.png

exemple_gris_negatif.png

2. Écrire une fonction `filtre_negatif_rgb` qui associe à la valeur d'un pixel en RGB son négatif.

L'assertion ci-dessous doit être vérifiée :

```
assert filtre_negatif_rgb([255,0,100]) == [0,255,155]
```

Tester cette fonction sur l'image fournie `cardinal.jpg` représentant le « Cardinal Fernando Niño de Guevara (1541–1609) » peint par *El Greco* Metropolitan Museum of New York licence CC0 1.0

```
>>> cardinal = image_to_matrice('cardinal.jpg')
>>> cardinal_negatif = applique_filtre(cardinal, filtre_negatif_rgb,
     'RGB')
>>> matrice_to_image(cardinal_negatif, fichier='cardinal-negatif.png
    ', mode='RGB',res=1)
```

Original

Négatif

3. Écrire une fonction `fonction_seuil(val, seuil, vmin, vmax)` qui retourne `vmin` si `val <= seuil` et `vmax` sinon.

On donne la fonction `filtre_seuillage_gris` qui retourne une fonction de filtre permettant d'associer à un pixel en nuance de gris soit 0 (noir) s'il est foncé par rapport au seuil, soit 255 (blanc) s'il est clair.

Cette méthode permet de fixer un paramètre dans une fonction à deux paramètres.

```
def filtre_seuillage_gris(seuil):
    def f(pixel):
        return fonction_seuil(pixel, seuil, 0, 255)
    return f
```

L'image en nuances de gris `lena.png` est fournie. Exécuter la séquence d'instructions ci-dessous, on peut séparer, selon un seuil, les pixels d'une image en deux catégories : les clairs et les foncés.

```
>>> lena = image_to_matrice('lena.png')
>>> lena_seuil = applique_filtre(lena, filtre_seuillage_gris(100), '
    L')
>>> matrice_to_image(lena_seuil, fichier = 'lena-seuil.png', mode =
    'L', res = 1)
```

Original

Avec un seuillage de 100

4. **a.** Écrire une fonction `filtre_rouge` qui prend en paramètre un pixel RGB, sous forme de liste `[r,g,b]` de trois entiers entre 0 et 255, et retourne sa projection sur la composante rouge `[r,0,0]`.

Les assertions suivantes doivent être vérifiées.

```
assert filtre_rouge([255,0,0]) == [255,0,0]
assert filtre_rouge([255,255,0]) == [255,0,0]
assert filtre_rouge([255,255,255]) == [255,0,0]
assert filtre_rouge([0,255,0]) == [0,0,0]
```

Exécuter la séquence d'instructions ci-dessous, on peut ainsi extraire l'image constituée de toutes les composantes rouges des pixels d'une image RGB

```
>>> cardinal_rouge = applique_filtre(cardinal, filtre_rouge, '
    RGB')
>>> matrice_to_image(cardinal_rouge, mode ='RGB', res= 1,
    fichier='cardinal-rouge.png')
```

Original

Composante rouge

b. Sur le modèle de la fonction `filtre_seuillage_gris`, écrire une fonction `filtre_composante_rg` qui retourne la fonction de filtre de composante pour un pixel en RGB correspondant à l'index de la composante passé en paramètre.

La fonction doit vérifier les assertions suivantes :

```
assert filtre_composante_rgb(0)([255,200,100]) == [255,0,0]
assert filtre_composante_rgb(1)([255,200,100]) == [0,200,0]
assert filtre_composante_rgb(2)([255,200,100]) == [0,0,100]
```

Extraire les composantes bleue et verte de l'image `cardinal.jpg` et les enregistrer sur le disque.

5. **a.** Compléter la fonction de filtre de pixel `filtre_monochrome` qui retourne la moyenne pondérée des composantes d'un pixel RGB par les coefficients `[0.299,0.587,0.114]` qui sont habituellement utilisés pour convertir une image RGB en nuances de gris.

```
def filtre_monochrome(pixel_rgb):
    """Retourne la moyenne pondérée des composantes
    d'un pixel RGB par les coefs [0.299,0.587,0.114]"""
    coef = [0.299,0.587,0.114]
    pixel_gris = 0
    somme_coef = 0
    "à  compléter"
```

Les assertions suivantes doivent être vérifiées :

```
assert filtre_monochrome([255,100,200]) == 157
assert filtre_monochrome([200,255,100]) == 220
assert filtre_monochrome([100,200,255]) == 176
```

b. Utiliser la fonction `filtre_monochrome` pour générer et enregistrer sur le disque la conversion en nuances de gris de l'image `cypres.jpg` représentant un tableau peint par Vincent Van Gogh, Metropolitan Museum of New York licence CC0 1.0.

Original en couleur

Nuances de gris

Table des matières

1 Représentation d'une image bitmap 1

2 Tableaux à 2 ou n dimensions 3

3 Traitement d'image 8