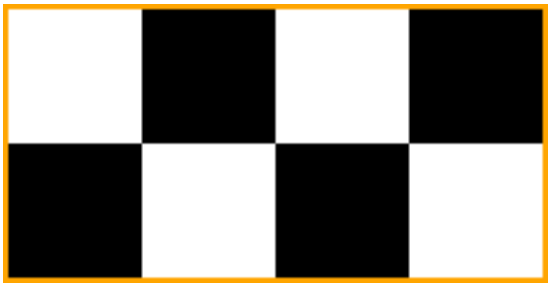
Images et tableaux à plusieurs dimensions

Thème types construits

1 Représentation d'une image bitmap

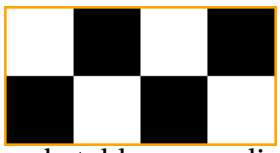
Point de cours 1

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**

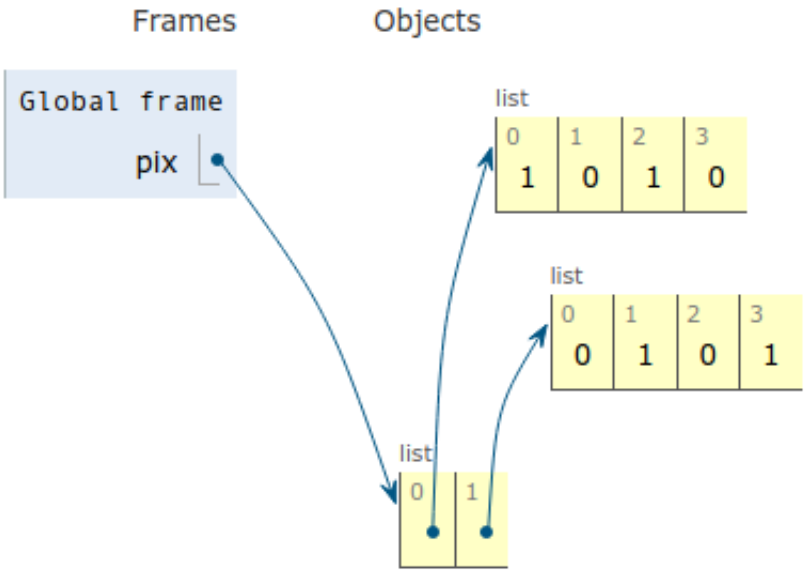


En `Python`, une matrice de pixels peut être représentée comme un tableau de tableaux, ou liste de listes dans la nomenclature `Python`. On utilise l'opérateur crochet une fois pour accéder à une ligne par son index et deux fois pour accéder à un pixel par ses index de ligne puis de colonne.

```
>>> 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[1][3] #pixel en 3eme ligne et 4eme 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.



Méthode *Outils de traitement d'image*

Dans le fichier `Images-Tableaux2d-Eleves-Partie1.py` (lien Capytale) on fournit trois fonctions outils pour manipuler des images :

- `dimensions` prend en paramètre un tableau à deux dimensions et renvoie le couple (nombre de colonnes, nombre de lignes) soit (largeur, hauteur) s'il s'agit d'une matrice de pixels ;
- `matrice_to_image` transforme un tableau à deux dimensions représentant une matrice de pixels en une représentation binaire conforme au format renvoyé par la fonction `Image` du module `PIL` ;
- `image_to_matrice` n'est pas tout à fait la réciproque de la précédente, elle prend en paramètre un chemin vers un fichier image et renvoie un tableau à deux dimensions représentant la matrice de pixels de l'image.

Ces fonctions utilisent les modules `PIL`, `numpy` et `matplotlib`.

Exercice 1

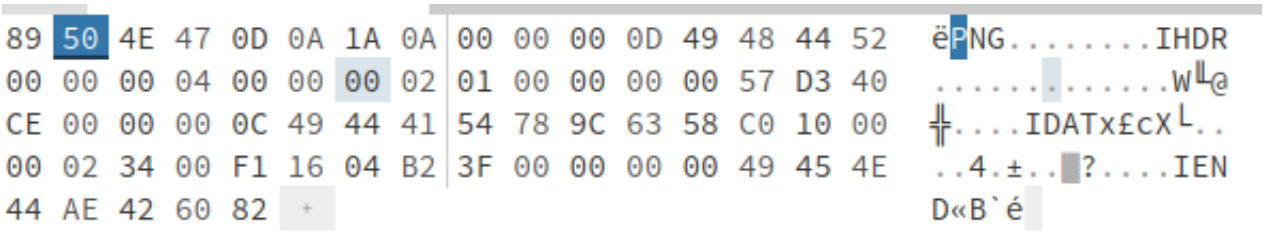
1. Récupérer l'archive `Images-Tableaux2d-materiel.zip`, la copier dans un répertoire pertinent et la déballer.
2. Ouvrir le fichier `Images-Tableaux2d-Eleves-Partie1.py` dans un IDE `Python` et le renommer éventuellement.

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 3 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 tests unitaires ci-dessous doivent être vérifiés :

```
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 tests unitaires ci-dessous doivent être vérifiés :

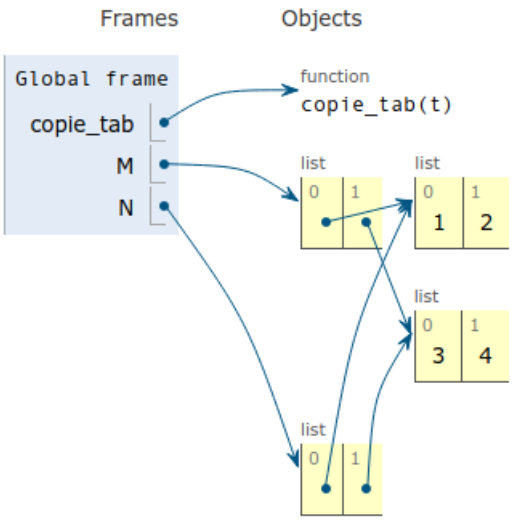
```
assert moyenne_tab2d([[-1,-2],[-2,-3,-0.5]]) == -1.7
assert moyenne_tab2d([[1,2],[float('inf'),10]]) == float('inf')
assert moyenne_tab2d([[1,2],[8,0]]) == 2.75
assert moyenne_tab2d([[8, float('-inf')],[]]) == float('-inf')
```

Méthode *Copie d'un tableau à plusieurs dimensions*

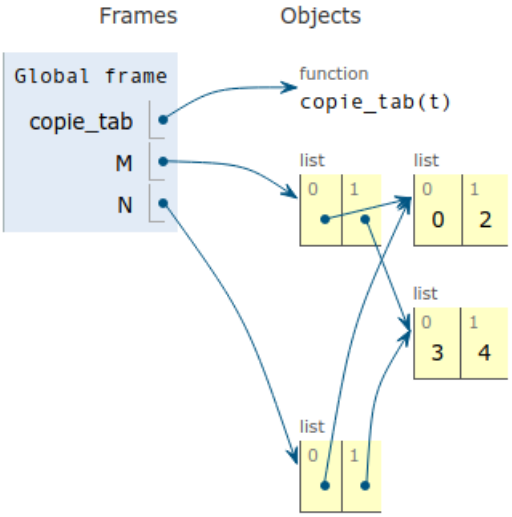
La valeur d'un tableau étant une référence vers une séquence de données en mémoire, un tableau à deux dimensions est une référence vers une séquence de références, ce qui nécessite un soin particulier pour déréférencer les tableaux imbriqués lors d'une opération de copie.
La fonction `copie_tab` ci-dessous 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

3.1 Création d'images par manipulation de pixels

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.

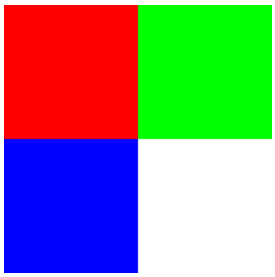
  Expression pour générer l'image en niveaux de gris ci-contre :

```
>>> 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 :

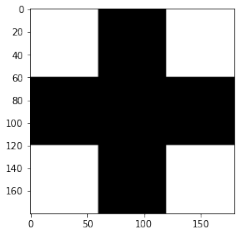
```
>>> pix = [
           [[255,0,0],[0,255,0]],
           [[0,0,255],[255,255,255]]
         ]
>>> matrice_to_image(pix,mode = 'RGB',
fichier='exempleRGB.png',res=100)
```



Exercice 4

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

1. Quelle expression permet de générer l'image binaire (mode `'1'`) ci-dessous avec le paramètre `res = 60` ?



2. 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):
    """Paramètre : couleur un tableau de 3 entiers entre 0 et 255
    Valeur renvoyée : Image au format PIL représentant une croix sur
         fonds
    blanc de la couleur passée en paramètre
    """
    blanc = [255,255,255]
    croix = ..............
    im = matrice_to_image(croix, mode = 'RGB', res = 10, fichier='
        croix.png')
    return im
```

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

Exercice 5

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) et on utilisera la fonction `matrice_nulle` définie dans les outils fournis.

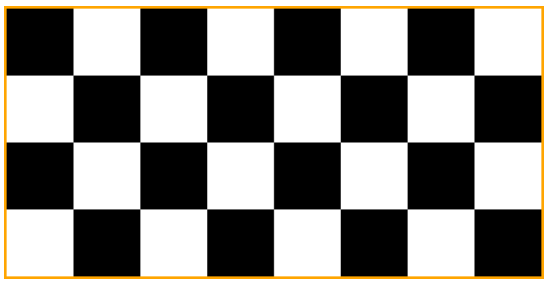
1. Compléter la fonction `barres_horizontales` pour qu'elle renvoie la matrice de pixels d'une image de dimensions `ncol` × `nlig` avec alternance de lignes noires (index pair) ou blanches (index impair).

```
def barres_horizontales(nlig, ncol):
    """Spécification à compléter"""
    #on crée une matrice nulle de bonnes dimensions
    pix = matrice_nulle(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 renvoie 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) :

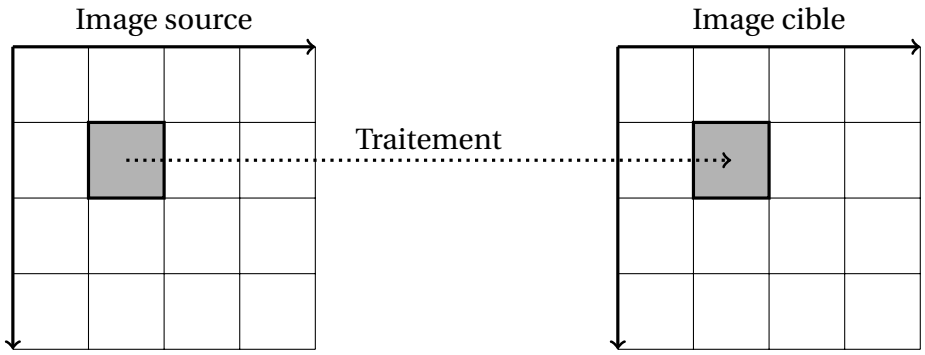


3.2 Traitement d'image par filtre de pixel

Méthode *Filtre de pixel*

Dans un premier temps, on étudie des traitements d'image où chaque pixel de *l'image cible* est une fonction d'un pixel de *l'image source*. Un cas simple est celui où les pixels source et cible sont à la même positon dans des images de même dimension. Il suffit alors de créer une image cible de mêmes dimensions et de la parcourir en attribuant à chaque pixel l'image du pixel source en même position.

La fonction de traitement (ou *filtre*) doit renvoyer une valeur de pixel compatible : un *entier* pour une image binaire ou en niveaux de gris ou un triplet d'entiers pour une image RGB.



Méthode *Annotations de type*

Par la suite on utilise les **annotations de type** et en particulier le type `List` qu'il faut importer du module `typing`.

- un tableau d'entiers est du type `List[int]`;
- un tableau de tableaux d'entiers (comme une matrice de pixels en mode `'1'` ou `'L'`) est du type `List[List[int]]`;
- un tableau de tableaux de tableaux d'entiers (comme une matrice de pixels en mode `'RGB'`) est du type `List[List[List[int]]]`;

Exercice 6 *Luminance et mélange d'images en niveaux de gris*

La luminance est une grandeur correspondant à la sensation visuelle de luminosité d'une surface.
Source : Wikipedia

On fournit dans `Images-Tableaux2d-Eleves-Partie1.py` une fonction calculant la luminance d'un pixel en mode RGB à partir des coefficients de pondération recommandés par l'Union Internationale des Télécommunications.

```
from typing import List

def luminance(rgb:List[int])->int:
    """Paramètre : un tableau de 3 entiers [r,g,b]
    Précondition : 0 <= r <= 255 et 0 <= g <= 255 et 0 <= b <= 255
    Valeur renvoyée : un entier
    Postcondition : renvoie la luminance 0.299*r+0.587*g+0.114*b"""
    r, g, b = rgb
    assert 0 <= r <= 255 and 0 <= g <= 255 and 0 <= b <= 255
    return int(0.299*r + 0.587*g + 0.114*b)
```

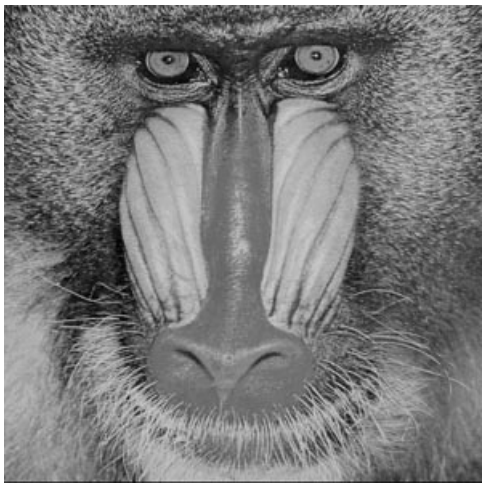
1. Compléter la fonction ci-dessous qui prend en paramètre une matrice de pixels en RGB et renvoie la matrice de pixels en niveaux de gris obtenue avec la luminance de chaque pixel source.

   On utilisera encore les fonctions `dimensions` et `matrice_nulle`.

```
def matrice_rgb_to_gris(pix:List[List[List[int]]])->List[List[int]]:
    """Précondition: pix une matrice de pixels rgb
    Valeur renvoyée: une matrice de pixels en niveaux de gris
    Postcondition : renvoie une matrice des luminances des pixels
        sources"""
    ncol, nlig = dimensions(pix)
    pix_but = matrice_nulle(ncol, nlig)
    #à  compléter
    return pix_but

#test
mandrill_rgb = image_to_matrice('mandrill.png')
```

```
mandrill_gris = matrice_rgb_to_gris(mandrill_rgb)
matrice_to_image(mandrill_gris, mode = 'L', fichier='mandrill_gris.
    png', res=1)
```



2. Compléter la fonction `melange_matrice_gris` fournie dans `Images-Tableaux2d-Eleves-Partie1.py` pour qu'elle mélange deux matrices de pixels en niveaux de gris avec des coefficients `coef` et `1 - coef`.

Tester en transformant progressivement l'image `mandrill_gris.png` en `darwin_gris.png`!

Cet exercice est tiré de l'excellent ouvrage Python Programming de Robert Sedgewick, disponible au CDI.

```
from typing import List

def melange_pixel_gris(p1:int, p2:int, coef:float)->int:
    """Précondition : 0<=p1<= 255 et 0 <= p2 <= 255 et 0 <= coef <=
        1
    Postcondition : renvoie int(p1 * coef + (1-coef) * p2)"""
    assert 0 <= p1 <= 255 and 0 <= p2 <= 255 and 0 <= coef <= 1
    return int(p1 * coef + (1-coef) * p2)

def melange_matrice_gris(pix1:List[List[int]], pix2:List[List[int]],
      coef:float)->List[List[int]]:
    """Précondition : pix1 et pix2 deux matrices de pixels en
        niveaux de gris de mêmes dimensions et 0 <= coef <= 1"""
    #à  compléter

def melange_progressif(pix1:List[List[int]], pix2:List[List[int]], n
     :int)->None:
    """Affiche le mélange progressif des images représentées par
        pix1 et pix2"""
    for k in range(n + 1):
        im = matrice_to_image(melange_matrice_gris(pix1, pix2, k/n),
             mode='L', fichier=f"melange{k}.png")
        im.show()

#test
```

```
darwin = image_to_matrice("darwin_gris.png")
mandrill = image_to_matrice("mandrill_gris.png")
melange_progressif(darwin, mandrill, 10)
```

3.3 Changement d'échelle

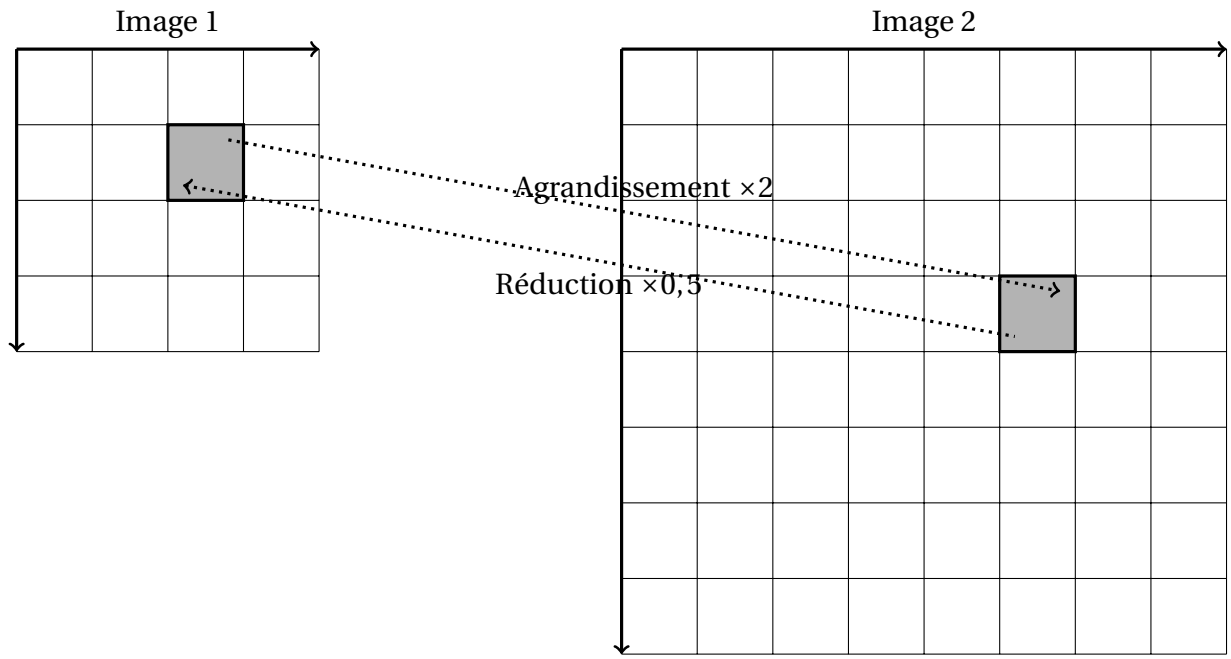
Méthode *Agrandissement / Réduction*

On considère une *image source* de largeur L et de hauteur H et un entier $k > 0$:

- ☞ Un agrandissement de coefficient k de l'*image source* est une image cible de dimensions $(k \times L, k \times H)$ dont le pixel de coordonnées (x, y) prend la valeur du pixel en $(x/k, y/k)$ dans l'*image source*.
- ☞ Une réduction de coefficient $1/k$ de l'*image source* est une image cible de dimensions $(L/k, H/k)$ dont le pixel de coordonnées (x, y) prend la valeur du pixel en $(k \times x, k \times y) = (x/(1/k), y/(1/k))$ dans l'*image source*.

L'agrandissement ou la réduction ne diffèrent que par le coefficient et peuvent donc être implémentés par une même fonction de *changement d'échelle*.

Une réduction est un échantillonnage avec perte d'information de l'image source et un agrandissement ne permet pas d'obtenir plus d'informations ce qui se traduit dans les deux cas par une *pixellisation*.



Exercice 7 *Changement d'échelle*

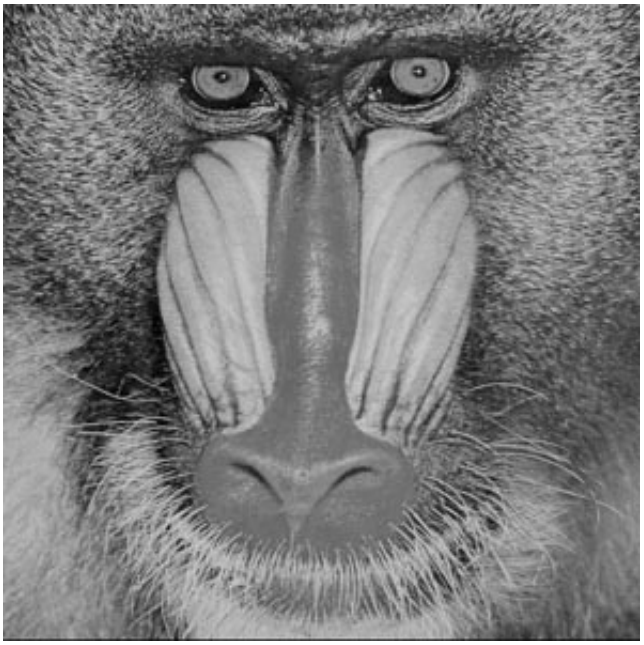
Écrire une fonction `changement_echelle` renvoyant le changement d'échelle d'une matrice de pixels et répondant à la spécification ci-dessous :

```
from typing import List
```

```
def changement_echelle(pix:List[List[int]], coef:float)->List[List[int
    ]]:
    """Précondition: pix une matrice de pixels rgb
    Valeur renvoyée: une matrice de pixels en niveaux de gris
    Postcondition : renvoie une matrice de l'image obtenue par changement
         d'échelle
    de coefficient coef"""
    ncol, nlig = dimensions(pix)
    ncol_but, nlig_but = int(ncol * coef), int(nlig * coef)
    #à  compléter

#code client
mandrill_gris = image_to_matrice('mandrill_gris.png')
mandrill_gris_quart = changement_echelle(mandrill_gris, 0.25)
im_quart = matrice_to_image(mandrill_gris_quart, mode = 'L', fichier='
    mandrill_gris_quart.png', res=1)
im_quart.show()
```

mandrill_gris.png



mandrill_gris_quart.png

Table des matières

1 Représentation d'une image bitmap **1**

2 Tableaux à 2 ou n dimensions **3**

3 Traitement d'image **8**

3.1 Création d'images par manipulation de pixels 8

3.2 Traitement d'image par filtre de pixel 11

3.3 Changement d'échelle 14