

Coloriage magique :

Les enfants aiment faire des coloriages, c'est bien connu. Il existe une famille de coloriages appelée coloriages magiques qui permettent aux enfants de colorier une image suivant un codage (un code correspondant à une couleur). L'objectif de ce projet est de créer un coloriage magique à partir d'une image en couleur (en pratique, ce sera un dessin coloré assez simple où les contours des zones colorées sont bien nets).

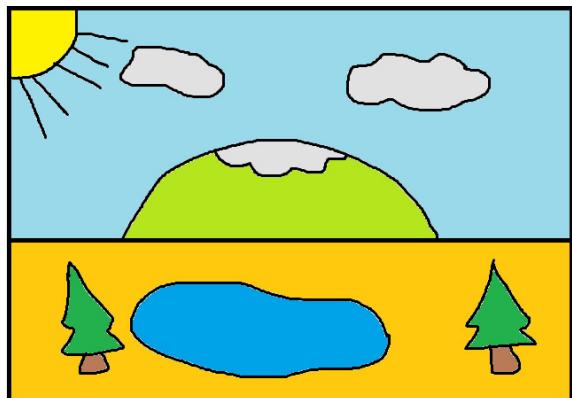


Image d'origine

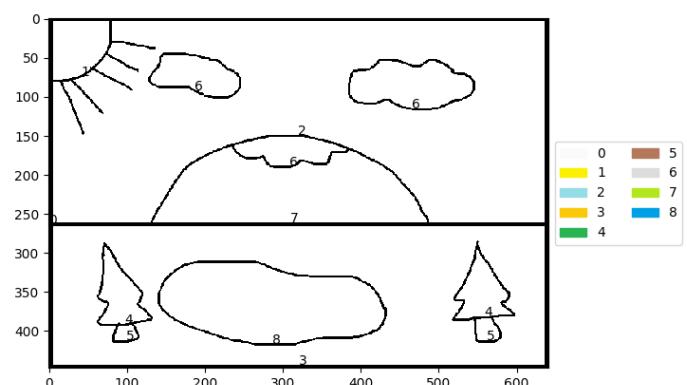


Image traitée

Illustration sur un exemple :

Préliminaires

Utilisation du module `pillow`.

`pillow` est un module permettant d'effectuer du traitement d'image avec Python. Nous ne nous servirons que de commandes de base.

Télécharger le module `pillow` en tapant : `conda install pillow` dans la console.

Initiation à Pillow : (contenu de cette page emprunté à Pierre Dieumegard)

Les images en noir et blanc ou en nuances de gris peuvent être considérées comme des matrices à 2 dimensions (largeur, hauteur) où chaque élément est la luminosité d'un pixel. Les images en couleur sont des matrices à 3 dimensions, constituées de 3 couches des couleurs respectives rouge, vert et bleu. On importe le module `pillow` consacré aux images par `import PIL.Image as Image`.

Ce module contient les fonctions :

`Image.open` pour ouvrir un fichier image

`img=Image.open("monfichier.png")`; on peut utiliser divers formats d'images : png, bmp, gif, jpg, tif... Ensuite, l'objet `img` a diverses méthodes utilisables.

`img.show()` pour afficher l'image dans le programme par défaut.

`img.putpixel((x,y),couleur)` met le point (x,y) à la couleur indiquée. Le point de coordonnées (0,0) est en haut à gauche. Si l'image est en noir et blanc, ou en gris, la couleur correspond à une valeur numérique entre 0 et 255 ; si l'image est en couleur, il faut mettre un triplet de valeurs numériques (rouge,vert,bleu). `img.getpixel((x,y))` donne la couleur du pixel (x,y) ; si c'est en noir et blanc ou nuances de gris, c'est une valeur numérique, si c'est une image en couleurs, c'est un triplet de valeurs numériques.

`img.size` est une variable (largeur,hauteur)

`img.new(mode,(largeur,hauteur),color=couleur)` crée une nouvelle image. Pour le mode, "1" est le noir et blanc (1 bit par pixel), "L" est en nuances de gris, "RGB" est en couleurs (rouge, vert, bleu). "couleur" est la couleur de fond, soit une valeur numérique, soit un triplet de valeurs numériques (rouge,vert,bleu)

- convertir une image en un nouveau mode par la méthode `convert(mode_nouveau)`

- sauvegarder une image par `save("nomfichier.bmp",format="bmp")`, ou d'autres formats.

Le plus simple pour afficher une image est `img.show()`, mais on peut aussi faire le lien avec d'autres modules, numpy,

`matplotlib.pyplot`, etc. En particulier, en utilisant pyplot par un `import matplotlib.pyplot as plt`, l'instruction `plt.imshow(img)` suivi de `plt.show()` affiche l'image `img`. Pour notre usage, on affichera les images en utilisant cette dernière démarche (c'est-à-dire via `matplotlib`).

Un exemple : On change tous les pixels noirs d'une image couleur en pixels blancs, et on affiche le résultat.

Rappel : Différences entre listes et tuples :

- les listes sont déclarées par des crochets `ml=[1,2,3]`, et leurs éléments sont modifiables.
- les tuples sont déclarés par des parenthèses `mt=(1,2,3)` et leurs éléments ne sont pas modifiables. Ici, on utilisera des doublets (x,y) pour indiquer la position d'un point, et des triplets (r,v,b) pour indiquer la couleur d'un point.

```

1
2 # Chargement de l'image
3 im1=Image.open(r"C:\Users\blanc\Desktop\image.jpg")
4
5 def Nouvelle(img):
6     nbx, nby = img.size
7     # on parcourt chaque pixel:
8     for x in range(nbx):
9         for y in range(nby):
10            px = img.getpixel((x,y)) # saisie d'un pixel
11            if px == (0,0,0): # si le pixel est noir,
12                img.putpixel((x,y),(255,255,255)) # on le remplace par un blanc
13    # On affiche le résultat
14    plt.imshow(img)
15    plt.show()
16
17 # Appel de la fonction avec im1:
18 Nouvelle(im1)

```

Ce projet s'articulera autour de quatre tâches essentielles :

- ① Opération de seuillage pour accentuer les contours.
- ② Effacer chaque zone colorée du dessin avec mémorisation de la couleur standard la plus proche de la couleur d'origine (pour renseigner la légende).
- ③ Affichage du codage de la couleur au centre de la zone (il faudra bien définir ce qu'on appelle le "centre" de la zone).
- ④ Affichage de la légende des codes couleurs.

Cet exercice permet d'introduire une notion de "distance" d'une couleur à une autre.

Exercice 1 Détection des couleurs standard les plus proches des couleurs d'origine.

1) On rappelle que la distance euclidienne entre deux points $M : (x, y, z)$ et $M' : (x', y', z')$ de l'espace est donnée par la formule :

$$d(M, M') = \sqrt{(x - x')^2 + (y - y')^2 + (z - z')^2}$$

Écrire une fonction Python `distance` prenant en entrée deux triplets (x, y, z) et (x', y', z') et donnant en sortie le carré de la distance séparant les deux points correspondants.

2) a) Nous avons listé dans un fichier texte 139 triplets RGB correspondant à une première palette de couleurs standard.

Ce fichier est nommé `rgb_couleurs.txt` et est fourni [ci-joint](#).

De même, nous avons listé dans un [fichier texte](#) 17576 triplets RGB correspondant à une seconde palette de couleurs standard, plus importante que la précédente.

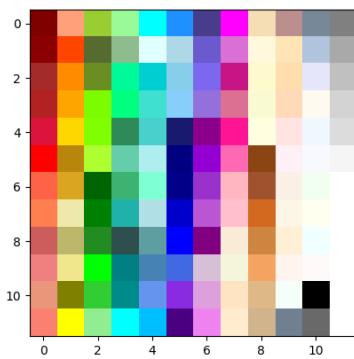
Recopier le code ci-dessous (en adaptant le chemin complet menant aux fichiers texte sur votre ordinateur) :

```

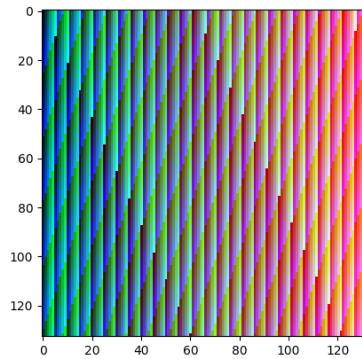
1
2 CodesRGB1 = []
3 with open (r"C:\Users\blanc\Desktop\DocuTextes\rgb_couleurs.txt",'rt') as f:
4     compteur = 0
5     for line in f:
6         CodesRGB1.append(eval(line))
7
8 CodesRGB2 = []
9 with open (r"C:\Users\blanc\Desktop\DocuTextes\rgb_couleursBis.txt",'rt') as f:
10    compteur = 0
11    for line in f:
12        CodesRGB2.append(eval(line))
13
14 CodesRGB = [CodesRGB1,CodesRGB2]

```

Dans ce code, `CodesRGB1` fait référence à la première palette et `CodesRGB2` fait référence à la seconde. Pour identifier sans ambiguïté chaque couleur d'un dessin de type, nous l'identifierons à la couleur standard qui lui est la plus proche, ce qui facilitera le codage et la légende du coloriage (on utilisera pour cela une des deux palettes). On a stocké ces deux palettes dans une liste `CodesRGB` (les couleurs de ces palettes sont représentées ci-dessous).



Palette 1



Palette 2

b) Écrire une fonction Python prenant en entrée un triplet de couleurs RGB et donnant en sortie la couleur standard qui lui est la plus proche au sens de la distance euclidienne. La sortie sera donc le triplet correspondant à cette couleur standard.

Cette fonction prendra aussi en entrée un indice (0 ou 1) correspondant à l'indice de la palette que l'on choisit dans la liste `CodesRGB`.

3) Cette question n'est pas directement utile au projet mais permet néanmoins de mettre en application la fonction précédente.

Écrire une fonction `NouvImage` prenant en entrée une image et affichant en sortie l'image où on a remplacé les couleurs originales par les couleurs standard les plus proches de la première palette (celle correspondant à `CodesRGB1`). Tester cette fonction avec des images assez petites, comme celle-ci ou celle-là.

Exercice 2 : Seuillage

Dans cet exercice, nous allons accentuer les contours du dessin à partir duquel on va créer le coloriage magique. L'idée va consister à remplacer les couleurs très sombres du dessin en noir pur (code RGB : (0,0,0)) pour mieux les identifier par la suite.

Écrire une fonction **Seuillage** qui prend en entrée une image et un entier naturel $m \geq 1$. Cette fonction doit transformer en pixel noir pur chaque pixel de l'image dont le code (r, g, b) vérifie $r^2 + g^2 + b^2 < m^2$. Elle donnera l'image ainsi traitée en sortie.

En pratique, un choix judicieux d'une valeur de m pour effectuer le seuillage sur des images de type "coloriage" sera une valeur située entre 150 et 160.

Exercice 3 : Effacement et mémorisation des zones de couleurs.

Dans cet exercice (essentiel au projet), l'objectif va être de coder la procédure suivante :

Pour tout pixel du dessin qui n'est ni noir pur, ni blanc pur, effacer la zone du dessin de ce pixel, calculer une "moyenne" des couleurs de cette zone et déterminer les extrémités gauche et droite de cette zone.

Par "effacer la zone du dessin de ce pixel", on entend : remplacer les pixels de cette zone par des pixels blancs.

Une zone de couleur est délimitée par des contours de couleur noir pur (contours accentués grâce à la fonction **Seuillage** de l'exercice précédent.).

Illustration sur un exemple complet (où on a placé également les codages) :



On gardera trace dans une matrice d'un marquage de tous les pixels d'une zone qui ont déjà été effacés. Si l'image est de taille (n, p) , la matrice correspondante **Mat** sera de taille (p, n) (la position (x, y) d'un pixel correspondra au coefficient d'indice (y, x) dans la matrice) et cette matrice sera initialisée à 0.

Remarque sur une structure de donnée fondamentale : Les Piles

Dans l'implémentation, on utilisera la notion de "Pile". Une "Pile" est une structure de donnée permettant de stocker un certain nombre d'éléments puis de les récupérer plus tard.

Les éléments sont extraits de la pile selon la règle "dernier entré, premier sorti" : un élément stocké dans une pile en est toujours extrait avant les éléments qui y étaient depuis longtemps.

Ceci correspond bien sûr à la notion de pile d'objet dans le monde réel : on considère que l'on empile les éléments les uns sur les autres, donc qu'il faut retirer les éléments du dessus avant de pouvoir accéder à ceux du dessous (un peu comme pour une pile d'assiette : on retire toujours de la pile la dernière assiette, au sommet de la pile).

Avec une pile, on effectue essentiellement deux opérations : empiler une valeur au sommet de la pile ou dépiler la valeur située au sommet.

Implantation :

- Pour initialiser une pile, on peut créer une liste vide :

```
1 Pile = []
```

- Pour empiler une valeur, il suffit d'utiliser la méthode `append()` :

```
1 Pile.append(valeur)
```

- Pour dépiler la valeur située au sommet, il suffit d'utiliser la méthode `pop()` (qui permet aussi de récupérer cette valeur) :

```
1 valeur = Pile.pop()
```

- Pour afficher la valeur se trouvant au sommet de la pile :

```
1 print(Pile[-1])
```

- Pour afficher le nombre d'éléments de la pile :

```
1 print(len(Pile))
```

Dans cet exercice, nous allons nous attacher à programmer la procédure suivante :

```

        Marquage = 0
        Initialisation de Mat
(Marq) : Pour tout pixel de l'image ni noir, ni blanc
        Marquage = Marquage + 1
        Effacer et marquer la zone de ce pixel

```

La tâche principale : **Effacer et marquer la zone de ce pixel** va en fait se décomposer en plusieurs sous-tâches :

blanchir et marquer la zone, calculer la moyenne des couleurs de la zone, déterminer les extrémités gauche et droite de la zone (ce qui sera utile ultérieurement pour le placement du codage). On utilisera pour cela la structure de pile précédemment présentée (les éléments de cette pile seront des coordonnées de pixels) et, si **Coord=(x, y)** est la position du pixel, cette tâche pourra se résumer ainsi (en omettant les instructions correspondant au calcul de la moyenne des couleurs et aux calculs des limites de la zone) :

Effacer_et_marquer_la_zone_de_ce_pixel(Marquage,Coord)	
1	Initialiser Pile comme la liste [Coord]
2	Blanchir et marquer le pixel à la position Coord
3	Tant que Pile n'est pas vide
4	Dépiler l'élément (u, v) en fin de Pile
5	Pour tout pixel voisin de (u, v) ni noir, ni blanc
6	Blanchir et marquer le pixel en (u, v)
7	Empiler le pixel en (u, v) dans Pile
8	Fin(Pour)
9	Fin(Tant que)

1) Cette question a pour objectif d'effacer tous les pixels voisins d'un pixel donné (faisant partie de la zone du pixel) ce qui correspond aux lignes 5, 6, 7 et 8 du pseudo-code précédent.

On considère qu'un pixel situé en (x, y) a 4 pixels voisins au maximum :

	$x, y - 1$	
$x - 1, y$	x, y	$x + 1, y$
	$x, y + 1$	

Pour accéder à chacun de ces voisins, on utilisera une matrice **Dep** à 4 lignes indiquant les directions à suivre pour aller de la position (x, y) à l'une des 4 positions possibles des pixels voisins :

Dep = np.array([[0,-1], [-1,0], [1,0], [0,1]]).

Ainsi, si **Pos** est la matrice **Pos=np.array([x,y])**, la position du k-ième voisin sera donnée par exemple par **Pos+Dep[k]** (on rappelle que **Dep[k]** est la k-ième ligne de **Dep**). Par exemple, pour k=1, **Pos+Dep[1]** représentera la position $[x,y]+[0,-1]=[x,y-1]$. On peut aussi itérer directement sur les lignes de **Dep** si la valeur de k nous est inutile.

Écrire une fonction Python **Efface** prenant en entrée la position **Pos** d'un pixel (sous forme de matrice),

un entier **Marquage** $\geqslant 1$ et réalisant la procédure ci-dessous :

Efface(Pos, Marquage)

Pour tout pixel **PosBis** voisin de **Pos** qui n'est ni noir, ni blanc

Empiler **PosBis** dans **Pile**

Blanchir le pixel **PosBis**

Marquer par **Marquage** le coefficient de **Mat** correspondant à **PosBis**

Ici **Pile** est la pile de coordonnées (sous forme de matrices) qui sera déclarée comme une variable globale de la fonction.

De même, l'image à traiter **Img** et la matrice **Mat** seront déclarées comme variables globales.

```

1 def Efface(Pos, Marquage):
2     global Pile, Img, Mat
3     # ... corps de la fonction ...

```

2) Programmer la fonction **Effacer_et_marquer_la_zone_de_ce_pixel** (déclarer globales les variables **Pile**, **Img** et **Mat**).

On appellera la fonction de la question précédente pour ce qui concerne les lignes 5,6,7 et 8 du pseudo-code.

3) On va compléter un peu les fonctions précédentes. On veut que la procédure **Effacer_et_marquer_la_zone_de_ce_pixel** calcule également une "moyenne" des couleurs de la zone, et détermine la couleur standard la plus proche de cette moyenne.

On veut également que cette procédure détermine les limites gauche et droite de la zone pour pouvoir ensuite placer correctement le codage couleur de cette zone.

Si $(r_0, g_0, b_0), (r_1, g_1, b_1), \dots, (r_n, g_n, b_n)$ sont les codes RGB des pixels de la zone, la moyenne **Moy** des couleurs de ces pixels sera définie comme l'isobarycentre des points correspondant à ces triplets, c'est-à-dire :

$$\text{Moy} = \left(\frac{1}{n} \sum_{i=0}^n r_i, \frac{1}{n} \sum_{i=0}^n g_i, \frac{1}{n} \sum_{i=0}^n b_i \right).$$

Pour calculer les limites gauche et droite de la zone, il faudra calculer le minimum **Min** des x et le maximum **Max** des x pour chaque position (x, y) de pixels de la zone.

On définira enfin une variable **compteur** donnant le nombre de pixels de la zone effacée.

Compléter la fonction **Efface** et **Effacer_et_marquer_la_zone_de_ce_pixel** pour que cette dernière donne comme variables de sortie **Moy**, **Min**, **Max** et **compteur** (dans la fonction, ces variables seront mises à jour chaque fois qu'un couple (u, v) est extrait de la pile).

On créera une variable **Somme_pixel** qui devra avoir pour valeur $(\sum_{i=0}^n r_i, \sum_{i=0}^n g_i, \sum_{i=0}^n b_i)$ (sous forme de matrice). Cette variable pourra être déclarée globale dans les deux fonctions (en plus de **Pile**, **Img** et **Mat**).

4) Programmer enfin en Python la procédure (*Marq*). On notera également que la variable **Marquage** numérote les zones à colorier du dessin et sa valeur finale sera donc le nombre de zones à colorier.

Exercice 4 : Exploitation des données

Dans cet exercice, on va exploiter les données calculées dans les fonctions de l'exercice précédent, afin de créer le coloriage magique.

1) Rajouter l'initialisation d'une liste `Les_Couleurs` juste avant la procédure (*Marq*). Cette liste sera initialisée comme une liste vide.

2) On va alors compléter (*Marq*) de la sorte (cf dernières lignes) :

	Marquage = 0
1	Initialisation de Mat
2	Pour tout pixel de l'image ni noir, ni blanc
3	Marquage = Marquage + 1
(<i>Marq</i>) :	Effacer et marquer la zone de ce pixel
4	Si la zone possède au moins nbPix_Mini pixels
5	Calculer la couleur standard la plus proche de celle de la zone
6	Mettre à jour la liste <code>Les_Couleurs</code>
7	Placer le codage dans la zone
8	
9	Fin(pour)

Dans cette nouvelle version `nbPix_Mini` correspond au nombre minimal de pixels d'une zone qu'on autorise pour réellement prendre en compte cette zone (en effet, l'image peut présenter de minuscules zones de couleurs dûes à un défaut de celle-ci et qui devront être ignorées.).

a) En utilisant la fonction du 2) b) de l'exercice 1 (avec l'indice de la deuxième palette `CodesRGB2`), ajouter le code Python correspondant aux lignes 5 et 6 dans (*Marq*). On pourra utiliser la variable de sortie `Moy` de la fonction `Effacer_et_marquer_la_zone_de_ce_pixel` et on stockera le triplet RGB de la couleur standard correspondant dans une variable `Couleur`.

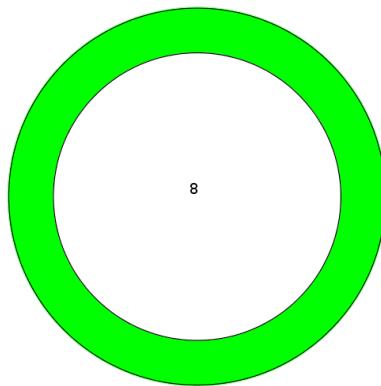
b) Écrire une fonction Python (en dehors de la procédure (*Marq*)) prenant en entrée une liste et un objet ; cette fonction doit intégrer cet objet à la fin de cette liste s'il n'y figure pas déjà. Cette fonction devra donner en sortie l'indice de la liste où se trouve alors l'objet.

Appeler cette fonction pour coder la partie correspondant à la ligne 7 de (*Marq*) :

`Mettre à jour la liste Les_Couleurs.`

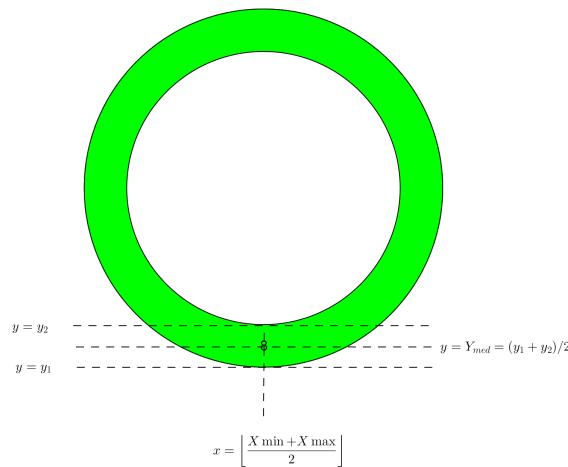
Il faudra placer à la fin de la liste `Les_Couleurs` le code RGB de la couleur standard correspondant à celle de la zone si celle-ci ne figure pas déjà dans `Les_Couleurs`. On récupérera la position de ce triplet de la liste `Les_Couleurs` dans une variable `Code` (pour le code de la couleur).

c) La ligne 8 de la nouvelle version de (*Marq*) a pour objectif de placer convenablement le code couleur dans la zone qui vient d'être traitée. Une difficulté ici est de définir à quel endroit de la zone on va placer ce code (il faut, bien sûr, que ce code soit situé dans la zone concernée). Une première idée naturelle serait de prendre le point correspondant à la "moyenne" des coordonnées (x, y) des points de la zone (autrement dit l'isobarycentre des points de la zone) comme endroit où placer le codage. Cependant, cette idée ne serait pas appropriée pour des zones "creuses" comme celle ci-dessous (avec son codage - ici 8 - placé en l'isobarycentre) :



Forme creuse avec codage mal placé

Une meilleure idée consiste à choisir de placer le codage au point d'abscisse $\left\lfloor \frac{X_{\min}+X_{\max}}{2} \right\rfloor$ (où X_{\min} et X_{\max} sont respectivement l'abscisse minimale et l'abscisse maximale des points de la zone) et d'ordonnée Y_{med} où Y_{med} serait l'ordonnée médiane des points de la zone d'abscisse $\left\lfloor \frac{X_{\min}+X_{\max}}{2} \right\rfloor$.



Forme creuse avec codage bien placé

C'est ce que nous allons faire maintenant.

En utilisant les variables de sortie **Min** et **Max** de la fonction **Effacer_et_marquer_la_zone_de_ce_pixel** de la ligne 4 de (*Marq*), et en utilisant le tableau **Mat** qui numérote les zones, afficher convenablement la valeur de **Code** dans la zone (autrement dit : programmer la ligne 8 de (*Marq*)). Écrire la procédure (*Marq*) sous forme d'une fonction prenant **nbPix_Mini** comme variable d'entrée et déclarant globales les variables **Img** et **Mat** (mais n'ayant aucune variable de sortie).

Aide : un pixel de coordonnées (x, y) appartient à la k -ième zone ssi le coefficient (y, x) de **Mat** vaut k . On ne confondra pas k (valeur de Marquage) avec la valeur de **Code** (codage de la zone à colorier). Pour afficher du texte, on utilisera la syntaxe

`plt.text(x,y,txt,fontsize=taille_du_texte)` comme dans l'exemple ci-dessous :

```

1 """
2 affiche "hello" en partant du point de coordonnées (10,65)
3 avec une taille de caractères égale à 5.
4 """
5
6 plt.text(10,65,"hello",fontsize=5)

```

Les coordonnées de (x, y) dans `plt.text(x,y,texte,fontsize=taille_du_texte)` font référence au coin inférieur gauche de la première lettre du texte.

Exercice 5 : Mise en place de la légende et finalisation

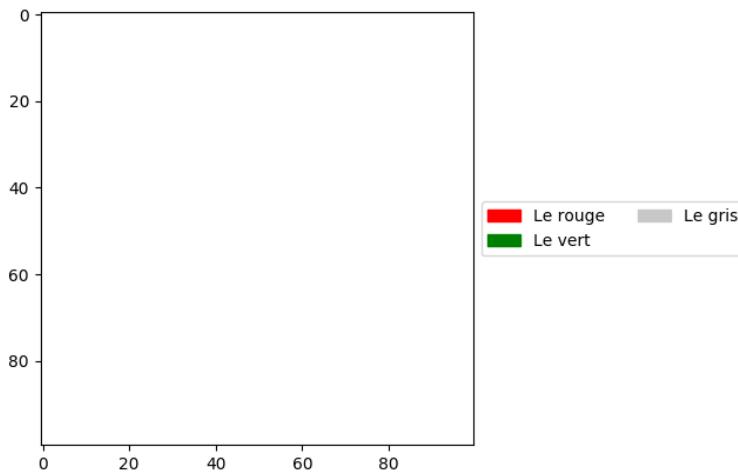
1) À l'aide de la liste `Les_Couleurs` et de la variable `Code`, créer enfin la légende du dessin à colorier. Pour vous aider, voici un exemple :

```

1 ######Pour la légende
2 import matplotlib.patches as mpatches
3 import matplotlib.pyplot as plt
4 from PIL import Image
5
6
7 # Création d'une image vierge ("blanche")
8 im = Image.new("RGB", (100,100), color="white")
9
10 # Définition des couleurs de la légende
11 red_patch = mpatches.Patch(color='red', label='Le rouge')
12 green_patch = mpatches.Patch(color='green', label='Le vert')
13 grey_patch = mpatches.Patch(color=(200/255,200/255,200/255), label='Le gris')
14
15 # Liste des couleurs de la légende
16 Leg = [red_patch, green_patch, grey_patch]
17
18 # Placement de la légende centrée sur la droite de la fenêtre (sur 2 colonnes)
19 plt.legend(handles=Leg, loc='center left', bbox_to_anchor=[1,0.5], ncol = 2 )
20
21 plt.imshow(im)
22 plt.show()

```

et le résultat :



Une légende

Précision : on veut que la légende soit formée d'éléments de la forme **Couleur : CodeCouleur** (voir l'exemple donné en préambule du projet).

Dans le code ci-dessus, `color` est égal à un triplet de réels appartenant à $[0, 1]$ (au lieu d'un triplet d'entiers appartenant à $\llbracket 0, 255 \rrbracket$). Pour passer du triplet RGB (r, g, b) avec $(r, g, b) \in \llbracket 0, 255 \rrbracket^3$ au triplet RGB (r', g', b') avec $(r, g, b) \in [0, 1]^3$, on utilisera la formule $(r', g', b') = \left(\frac{r}{255}, \frac{g}{255}, \frac{b}{255}\right)$ (cf définition de `grey_patch` dans le code ci-dessus).

La définition de la légende complètera la fin de la fonction (*Marq*).

2) Proposer enfin à l'utilisateur une interface (sous forme de fonction) prenant en entrée le chemin complet du fichier de l'image à traiter et lui proposant en boucle quatre options pour cette image :

- ➊ Utiliser la fonction `Seuillage` (et redéfinir `Img` comme la variable de sortie de `Seuillage`),
- ➋ Utiliser la fonction `NouvImage` de la question 3) de l'exercice 1.
- ➌ Appliquer la fonction (*Marq*). L'image obtenue pourra éventuellement être sauvegardée.
- ➍ Afficher l'image et quitter (utiliser `plt.imshow(Img)` et `plt.show()`).

Si la fonction `Seuillage` ou la fonction (*Marq*) est utilisée, il faudra aussi demander à l'utilisateur le paramètre à fournir à la fonction.

Rappel :

- Enregistrer une chaîne de caractères fournie par l'utilisateur. Exemple :

```

1
2 pn = input("Quel est votre prénom?")

```

- Enregistrer un nombre entier fourni par l'utilisateur. Exemple :

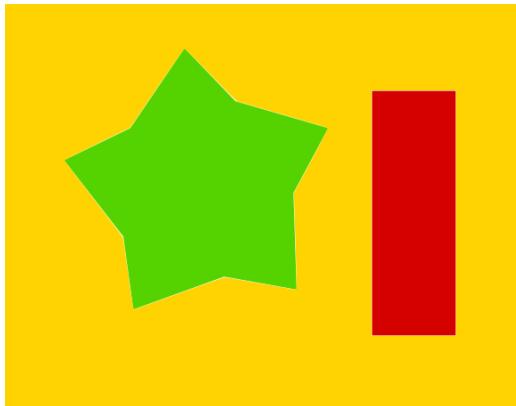
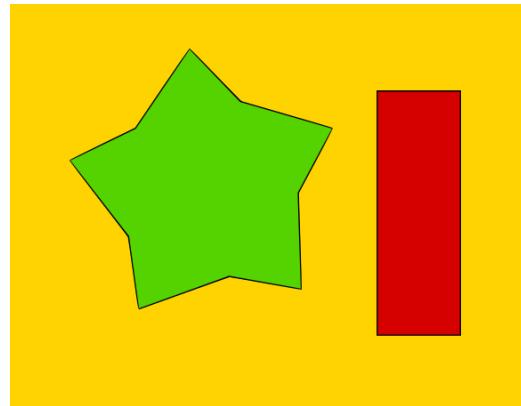
```

1
2 age = int(input("Quel est votre âge?"))

```

Exercice 6 : Gestion d'un dessin sans contour

L'objectif de cet exercice est de gérer des dessins sans contour noir et d'écrire un programme permettant de rajouter automatiquement ce contour (et pouvoir donc lui appliquer le traitement précédent).

**Sans contour****Avec contour**

Écrire une fonction **Contour** qui prend en entrée une image et un réel $r > 0$. Cette fonction doit créer une nouvelle image de cette manière :

Dans l'image passée en entrée, on regarde chaque pixel,

- si la moyenne des distances des couleurs des 4 pixels voisins à celle du pixel central est supérieure au seuil r , alors on colorie en noir les 5 pixels dans la nouvelle image (le pixel et ses 4 voisins),
- sinon, on place un pixel de même couleur dans la nouvelle image.