



L3 I

Python

TD n°5 et 6 : traitement d'images (partie n°1)

Manipulations de fichiers images (partie n° 1)

Objectifs : Savoir ouvrir et modifier des fichiers images à l'aide de Python.

Exercice N° 1 : Vérifier que votre installation contient toutes les bibliothèques nécessaires à ce TP.

1 Les images « en noir et blanc »

Exercice N° 2 : L'objectif de ce premier exercice est de comprendre comment un ordinateur « voit » et manipule des images que l'on appelle abusivement en noir et blanc, mais qui sont en réalité bien souvent des images en niveaux de gris.

Commencez par récupérer le fichier `noir_et_blanc.png` se situant sur la page web et enregistrez-le dans votre répertoire courant. Regardez l'image en double-cliquant dessus dans l'explorateur de fichiers. En zoomant sur l'image, on s'aperçoit qu'elle est constituée d'une multitude de pixels organisés en lignes (y) et colonnes (x). La couleur de chaque pixel est codée sur un octet (c'est-à-dire 8 bits), et représente donc une nuance de gris parmi 255 valeurs possibles. On peut voir cette valeur comme la luminosité du pixel : la valeur 0 code la couleur noire, la valeur 255 code la couleur blanche, et les valeurs intermédiaires codent des nuances de gris de plus en plus claires à mesure que la valeur augmente.



FIGURE 1 – Zoom à 150 % de l'image `noir_et_blanc.png`

Dans la bibliothèque `PIL.Image`, on dispose des fonctions

- `Image.show(image)` affiche l'image `image` (qui a été chargé au préalable à l'aide de la fonction `open`).

- `Image.getpixel((x,y))` récupère la valeur de la couleur du pixel situé en ligne `y` et colonne `x` de l'image `image`. À savoir que les indices des lignes et colonnes commencent à 0.
- `Image.putpixel((x,y),valeur)` remplace la valeur de la couleur du pixel situé en ligne `y` et colonne `x` de l'image `image` par `valeur`.

Exercice N° 3 :



Afficher l'image `noir_et_blanc` sous Python puis récupérer la valeur de la couleur du pixel en ligne 50 et colonne 25.

Exercice N° 4 :



Écrire une fonction `inversion(image)` qui prend en entrée une image et retourne l'image inversée¹. Pour ce faire, on pourra regarder ce que fait la fonction `image.size`. Tester cette fonction avec le fichier `noir_et_blanc.png`.

Exercice N° 5 :

Écrire une fonction `noir_et_blanc(image,s)` qui prend en entrée une image et un seuil `s` et qui transforme tous les pixels de valeur inférieure à `s` en 0 et tous les autres en 255. Tester cette fonction avec le fichier `noir_et_blanc.png`.



Exercice N° 6 :

Écrire une fonction `ligne(image,y,c)` qui trace une ligne de couleur de pixel égal à `c` sur la ligne `y` de l'image `image`. Tester cette fonction avec le fichier `noir_et_blanc.png`.



Exercice N° 7 :

Écrire une fonction `assombrir(image)` qui assombrit l'image `image` en divisant par 2 la valeur de chaque pixel. Tester cette fonction avec le fichier `noir_et_blanc.png`.



Exercice N° 8 :

Écrire une fonction `eclaircir(image)` qui éclaircit l'image `image` en multipliant par 2 la valeur de chaque pixel (en prenant soin de ne pas dépasser la valeur maximale de 255). Tester cette fonction avec le fichier `noir_et_blanc.png`.



2 Les images « en couleur »

Exercice N° 9 : Comme précédemment, regardons comment un ordinateur « voit » et manipule des images couleurs.

Commencez par récupérer le fichier `couleurs.png` se situant sur la page web et enregistrer-le dans votre répertoire courant. Regardez l'image en double-cliquant dessus dans l'explorateur de fichiers. En zoomant sur l'image, vous devriez voir qu'elle est constituée d'une multitude de pixels organisés

1. Par image inversée, on entend que les couleurs blanches sont remplacées par du noir et inversement.

en lignes (y) et colonnes (x), tout comme le sont les images en niveaux de gris.

Pour représenter assez finement pour l'œil humain un pixel en niveaux de gris, on n'a besoin que d'une seule valeur entre 0 et 255 (0 : noir, 255 : blanc). Dans une image couleur, on indique pour chaque pixel un mélange de 3 couleurs : rouge, vert, bleu. Chaque couleur est donnée par un nombre entre 0 et 255, on a donc besoin de 3 nombres (qu'on représente chacun sur un octet). Ce codage s'appelle RGB pour Red Green Blue et où pour chacune de ces couleurs

- la valeur 0 indique qu'on met 0% du « pot de cette couleur ».
- la valeur 255 indique qu'on met 100% du « pot de cette couleur ».

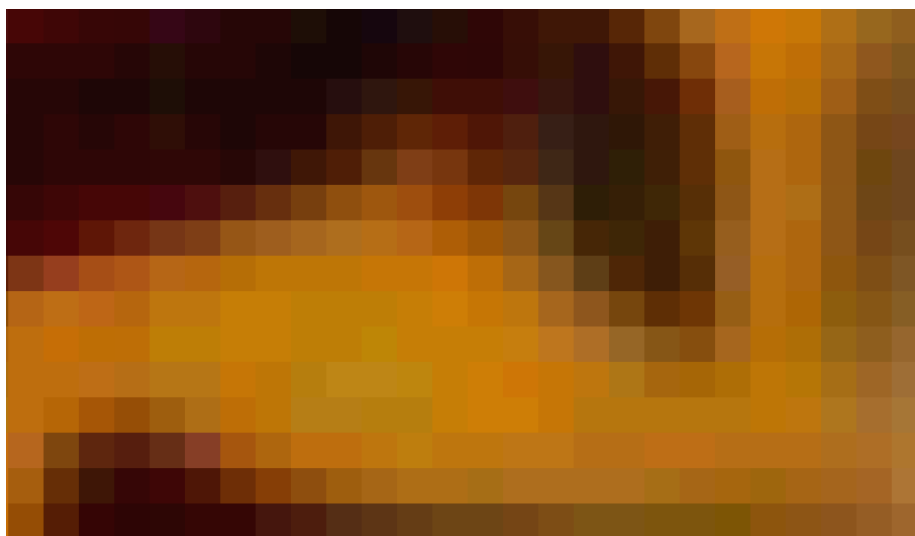








FIGURE 2 – Zoom à 150 % de l'image couleurs.png

Voici quelques exemples pour bien comprendre le codage des couleurs :

couleur	R	G	B	Octet 1	Octet 2	Octet 3
	0	0	0	00000000	00000000	00000000
	255	255	255	11111111	11111111	11111111
	255	0	0	11111111	00000000	00000000
	0	255	0	00000000	11111111	00000000
	0	0	255	00000000	00000000	11111111
	132	122	191	10000100	01111010	10111111

Dans la bibliothèque PIL. Image, on dispose des fonctions

- `image.getpixel((x,y))` retourne le tuple (rouge,vert,bleu) qui récupère la valeur des 3 couleurs R,G,B du pixel situé en ligne y et colonne x de l'image image.
- `image.putpixel((x,y),(rouge,vert,bleu))` remplace la valeur de la couleur du pixel situé en ligne y et colonne x par rouge,vert,bleu de l'image image.

Exercice N° 10 :



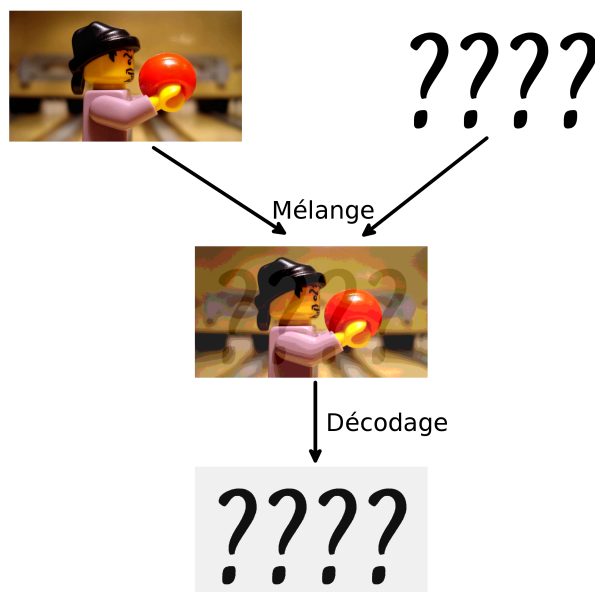
Écrire une fonction `no_rouge(image)` qui prend en entrée une image image et qui supprime toute la couleur rouge de chaque pixel.

Tester cette fonction avec le fichier couleurs.png.

L'image couleurs.png que vous avez chargée contient une autre image cachée que vous allez essayer de découvrir.

2.1 Principe pour cacher une image dans une autre

L'image a été cachée par mes soins en changeant légèrement la couleur de certains pixels de l'image originale. Tout se passe comme si l'image cachée était « sous » l'image visible.



En fait, on mélange les pixels des deux images originales, en privilégiant l'image 1 pour qu'on ne remarque pas que l'image 2 est cachée dedans. On utilise le fait que modifier légèrement la couleur des pixels d'une image est difficile à discerner pour l'œil humain. Par exemple, le mot

a b r a c a d a b r a

ne contient pas deux lettres de même couleur, ce qui est difficile à voir. Les couleurs employées sont trop proches les unes des autres pour pouvoir être distinguées. En résumé, modifier légèrement les pixels d'une image ne se voit pas facilement.

On utilise le fait que l'œil ne distingue pas les faibles différences de couleur pour cacher une image dans une autre. Pour simplifier, voyons ce qui se passe sur une seule composante de couleur, disons le rouge (même si en fait, on appliquera le même traitement aux trois couleurs).

Prenons 2 images de même dimension. Pour chacune des coordonnées (x, y) , on va mélanger une partie du pixel (x, y) de l'image 1 avec le pixel (x, y) de l'image 2. L'image obtenue sera très proche de l'image 1, mais l'image 2 sera « cachée » à l'intérieur.

Imaginons que l'on veut mélanger un pixel dans l'image 1 dont l'octet rouge vaut 190, c'est-à-dire

10111110

avec l'octet de l'image 2 qui vaut 121, c'est-à-dire

01111001.

L'idée est de recombinaison les 5 chiffres les plus significatifs du premier pixel avec les 3 chiffres les plus significatifs de l'image 2. Les chiffres significatifs sont les plus importants, ceux qui se trouvent à gauche. Au contraire, les unités ne sont pas très importantes pour nous. De nos 2 octets d'origine

10111 110 et 011 11001

on ne garde que les chiffres les plus significatifs, en mettant ceux de la première image en tête. Ce mélange des deux octets d'origine produit l'octet suivant :

10111 011.

On remarque que l'on a peu modifié la valeur par rapport à celle de l'image 1, parce que l'on a gardé les 5 bits les plus significatifs. Sur notre exemple, la différence est seulement de 3 : la nouvelle valeur est 187, ce qui donne un rouge **comme ce texte**, au lieu de 190 dans l'image originale, ce qui donne un rouge **comme celui-ci**, qui est très proche à l'œil nu. Au pire, on transforme les 3 derniers bits de 000 à 111, ou vice-versa, ce qui fait une différence de 7, au plus. En résumé, la fusion des 2 images ressemble beaucoup à la première image.

Qu'a-t-on gagné ? Ce que l'on a gagné, c'est que à partir de l'octet que l'on a créé dans l'image transformée

10111 011,

on n'a pas beaucoup modifié la première image, et on peut retrouver une couleur proche de celle du pixel de la seconde image grâce aux 3 derniers bits, 011. On sait donc que la valeur du pixel original de la seconde image se situe entre 011 00000 et 011 11111.

L'erreur est au maximum de 32. Pour reconstruire approximativement le pixel original, on peut compléter la valeur de façon intermédiaire, en ajoutant 16. À partir d'un pixel de l'image modifiée, on reconstruit donc l'approximation du pixel de l'image cachée :

01110000

Ce nombre vaut 112 au lieu de 121 dans l'image 2 qu'on voulait cacher. On a donc fait une erreur de 9 qui reste acceptable. Au pire, on fera une erreur de 16. Ça reste suffisant pour retrouver l'image cachée, même si la perte de précision la dégradera.

2.2 Comment récupérer l'image cachée ?

Exercice N° 11 :



Écrire une fonction `valeur_derniers_bits(n)` qui, à partir d'une valeur `n` codée sur un octet, retourne la valeur correspondant aux 3 derniers bits.

Par exemple, si on transmet 187 à cette fonction, dont le code en binaire est 10111011, elle devra garder le code des 3 derniers bits, c'est-à-dire 011 et retourner 3.

Exercice N° 12 :



Écrire une fonction `décaler(m)` qui, à partir d'un entier `m` dont le code se représente sur 3 bits, retourne la valeur correspondant à ces 3 bits complétés par 10000.

Par exemple, si on transmet 3 à cette fonction, dont le code binaire est 011, elle retournera 112, codé par 01110000.

Exercice N° 13 :

Écrire une fonction `decoder(image)` qui décode l'image `image`.



Tester cette fonction avec le fichier `couleurs.png`.

Exercice N° 14 :



Écrire une fonction `cache(image_illusion, image_cacher)` qui cache l'image `image_cacher` (qui a été chargé au préalable à l'aide de la fonction `open`) dans l'image `image_illusion` (qui a été chargé au préalable à l'aide de la fonction `open`).