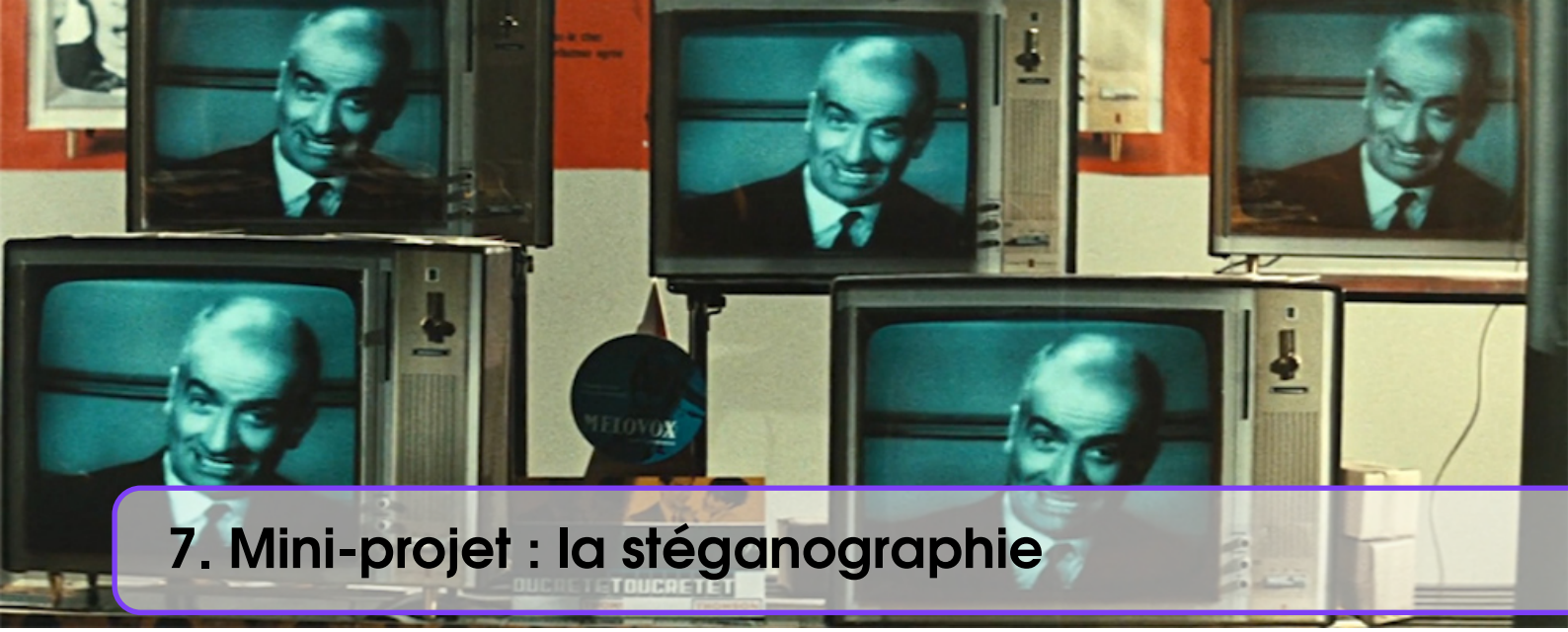


7. Mini-projet : la stéganographie

La **stéganographie**, du grec ancien *steganos* (étanche) et *graphé* (écriture), est l'art de dissimuler un message important dans un autre message anodin.
On peut par exemple masquer un message dans une image numérique, un fichier son, ... Il faudra bien évidement utiliser des fichiers numériques non compressés [1].
Nous allons dans ce mini-projet dissimuler une image numérique dans une autre, mais si vous voulez plus de détail sur cet art, vous pouvez regarder la page wikipédia correspondante :
https://fr.wikipedia.org/wiki/Stéganographie
ou encore cette page dédiée du site bibmath :
http://www.bibmath.net/crypto/index.php?action=affiche&quoi=stegano/index

7.1 Image numérique

Quand vous regardez une image numérique sur un écran, cette image est définie par un ensemble de petits points lumineux que l'on appelle pixels. Chacun de ces pixels a une couleur qui est obtenu par la synthèse soustractive d'une lumière rouge, une lumière verte, une lumière bleu. Une image numérique peut dont être vue comme une table (donc un tableau de tableau) contenant dans chaque case la définition d'un pixel, autrement dit un triplet contenant l'intensité lumineuse de rouge, l'intensité lumineuse de vert, l'intensité lumineuse de bleu. On parle de **triplet RVB** pour Rouge, Vert, Bleu (en anglais *RGB* pour Red Green Blue).
Une image 24 bits a une définition de 8 bits (1 octet) pour le rouge, 8 bits (1 octet) pour le vert et enfin 8 bits (1 octet) pour le bleu, toujours dans cet ordre.
Les triplets RVB nous permettent, en mélangeant les couleurs d'obtenir $256 \times 256 \times 256 = 16\,777\,216$ couleurs différentes.
Par exemple,

- le triplet (255,0,0) est le rouge
- le triplet (0,255,0) est le vert
- le triplet (0,0,255) est le bleu
- le triplet (0,0,0) est le noir
- le triplet (255,255,255) est le blanc
- tout triplet (g,g,g) avec g un nombre entre 1 et 254 est du gris, plus le nombre est grand, plus le gris est clair
- le triplet (0,255,255) est le jaune
- ...

1. Si vous ne voyez pas encore pourquoi, j'espère que ce travail vous permettra de comprendre pourquoi.

7.2 Le module PIL pour Python

Pour lire, modifier, afficher une image, nous avons besoin d'un module qui n'est pas installé de base dans la distribution Python. Il faut donc commencer par l'installer avec la commande suivante à écrire dans la console de votre environnement de développement (ou IDE pour *Integrated Development Environment*) qu'il soit Pyzo, Spyder, Thonny, ...

```
pip install pillow
```

Une fois l'installation faite, nous pouvons commencer à écrire notre premier programme. Il faut évidemment que l'image se place dans le même dossier que votre programme.

```
from PIL import Image

#on charge l'image numérique
photo = Image.open('cache-cache.bmp')

#on affiche l'image
photo.show()

#on récupère les dimensions de l'image
largeur , hauteur = photo.size
print(largeur, hauteur)

#on ferme le fichier
photo.close()
```

exercice 7.1 — PIL.

1. Les dimensions de la photo `cache-cache.bmp` sont :
   largeur .....
   hauteur .....
2. Le nombre total de pixels de cette photo est donc de :
   .....
3. Le poids de cette photo, en octets est donc de :
   .....
4. Le poids de cette photo, en kilooctets est donc de :
   .....

■

Nous pouvons récupérer la composante en Rouge, Vert et Bleu de chaque pixel.

```
#récupérer la composante RVB du pixel de coordonnées (120,200)
rouge , vert , bleu = photo.getpixel((190 , 220))
print(rouge, vert, bleu)
```

exercice 7.2 — PIL.

1. Quelle est la composante en Rouge, Vert et Bleu du pixel de coordonnées (120,200) ?
   .....
2. Quelle est la couleur de ce pixel ?
   .....

■

Nous pouvons modifier la composante en Rouge, Vert et Bleu de chaque pixel.

```
#modifier la couleur du pixel de coordonnées (120,200) pour lui donner la couleur (r,v,b)
photo.putpixel((120,200),(r,v,b))
```

exercice 7.3 — PIL. Colorier le pixel de coordonnées (120,200) en blanc. ■

Enfin, nous pouvons créer une nouvelle image en mode RVB et nous pouvons la sauvegarder.

```
#créer une nouvelle image , toute blanche
nouvelle_image = Image.new("RGB", (largeur,hauteur),"white")

#sauvegarder une image
nouvelle_image.save ("nom de l'image à sauvegarder")
```

exercice 7.4 — PIL. Programmer une fonction qui prend en argument deux entiers `larg` et `haut` et une chaîne de caractères `nom`. Cette fonction devra créer et sauvegarder une image de dimension `larg` et `haut`.
Cette image sera au format `BMP` et sera nommer `nom`.
Enfin tous les pixels auront des couleurs aléatoires [a]. ■

a. On pourra utiliser la fonction `randint(a,b)` du module `random` qui renvoie un nombre entier aléatoire entre `a` et `b` inclus.

7.3 Bits de poids fort, bits de poids faible

Petit rappel sur le codage binaire. Un nombre entier entre 0 et 255 peut-être codé à l'aide d'un octet sous la forme d'une suite de 0 et/ou de 1. Chacun de ses 0 ou 1 correspond à une puissance de 2. Soit on la prend (1) soit on la laisse (0).
Par exemple avec le nombre 204 :

Les puissances	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
Les bits	1	1	0	0	1	1	0	0	$1\times2^7+1\times2^6+1\times2^3+1\times2^2=128+64+8+4=204$

Plus un bit est à gauche, plus il correspond à un grand nombre (128, 64, ...), on les appelle les **bits de poids fort**.
Plus un bit est à droite, plus il correspond à un petit nombre (4, 2, 1, ...), on les appelle les **bits de poids faible**.
Si lors d'un codage ou d'une transmission, on fait une erreur sur un bit de poids faible, cette erreur ne sera pas trop importante.
Ainsi, si j'écris 1100 1010, mon résultat est de 202. Ce qui est assez proche de 204.
Par contre, si j'écris 1010 1100, mon résultat est de 172. Mon résultat est très éloigné de 204.

Le principe de la stéganographie numérique est de dire que les bits de poids faible son sans importance. On peut les enlever. En effet, si je décide de supprimer les 3 bits de poids faible correspondants aux puissances 2^2, 2^1 et 2^0 je vais commettre au maximum une erreur de $2^2+2^1+2^0=7$. J'obtiens alors 3 bits libres dans lesquels je vais pouvoir dissimuler mon message !

exercice 7.5 — Masque. Quelle est l'opération binaire qui permet de transformer tout nombre entier compris entre 0 et 255 en remplaçant ses 3 bits de poids faible par des 0.
Pour mémoire les opérateurs binaires de bases sont :
AND : `&` OR : | XOR : ^ (ou exclusif) NOT : ~
Par exemple `7 & 12` renverra le résultat 4. Pourquoi ? ? ?

.....
.....
.....
.....
.....
.....
.....
.....

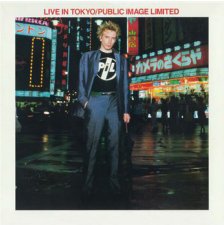
■

exercice 7.6 — Masque. Écrire une fonction `masque(nb,bt)` qui prend en arguments deux entiers `nb` et `bt`.
`nb` est un entier entre 0 et 255.
`bt` est entier entre 0 et 8.
La fonction transforme le nombre `nb` en remplaçant ses `bt` bits de poids faible par des 0. ■

exercice 7.7 — Image Vs bits de poids faible. Appliquer cette fonction pour mettre des 0 aux bits de poids faible des composantes de rouge, vert et bleu de chaque pixels de l'image `cache-cache.bmp`.
Déterminer quel est le nombre de bits de poids faible à supprimer pour ne pas perdre trop en qualité. ■

7.4 Cacher l'image dans l'image

Nous allons maintenant cacher l'image `PIL.bmp` [2] dans l'image `cache-cache2.bmp` qui est une légère modification [3] de l'image d'origine pour que les dimensions entre les deux images correspondent.



Voici comment nous allons procéder. Pour chacune des images, pour chaque pixels, on récupère la composante en rouge, vert et bleu, puis pour chaque composante, on va supprimer des bits de poids faibles. Par exemple, nous allons supprimer 5 bits pour l'image `PIL.bmp` et 3 bits pour l'image `cache-cache2.bmp`. Le choix du nombre de bits n'est pas pris au hasard, comme vous allez le constater, il faut que le nombre total soit de 8. Ici $5+3=8$ donc tout va bien.

2. PIL, pour Public Image Limited est un groupe post punk de Johnny « Rotten » Lydon ex Sex Pistols. Vous pouvez écouter leur tube `This is not a love song` : https://www.youtube.com/watch?v=Az_GCJnXAI0.
3. Cette modification a été faite à la main avec le logiciel GIMP, mais elle aurait très bien pu être automatisée à l'aide d'un programme écrit en Python.

Par exemple, on peut avoir :

L'image à cacher
Composante Rouge d'un pixel de `PIL.bmp` :

1	0	1	1	1	0	1	1

L'image conteneur
Composante Rouge d'un pixel de `cache-cache2.bmp` :

0	0	1	1	0	1	0	1

1. On supprime les 5 bits de poids faible

1	0	1	0	0	0	0	0

On ne regarde que les bits de poids fort

1	0	1

2. On décale les bits de poids forts pour qu'ils deviennent des bits de poids faibles

					1	0	1

3. On supprime les 3 bits de poids faible

0	0	1	1	0	0	0	0

On ne garde que les 5 bits de poids fort

0	0	1	1	0

4. A l'aide d'un simple opérateur binaire (lequel ?) on complète les vides des bits de poids faible de notre image `cache-cache2.bmp` par les nouveaux bits de poids faible de `PIL.bmp`

0	0	1	1	0	1	0	1

Et voilà ! Notre image `PIL.bmp` est cachée dans l'image `cache-cache2.bmp`. La seule difficulté reste dans le décalage des bits vers la droite. c'est ce que fait l'opérateur `»`.
Ainsi : `149 » 2` renvoie 37 car 149 s'écrit 1001 0101 en binaire et si on décale de 2 bits vers la droite on obtient le nombre binaire 1001 01*01* soit 0010 0101 qui correspond bien au nombre décimale 37.

exercice 7.8 — Stéganographie. Écrire une fonction `stegano(support,cache,bt)` répondant à la spécification suivante :
Entrées :
`support` chaîne de caractères donnant le nom de l'image avec son extension, qui cachera l'image
`cache` chaîne de caractères donnant le nom de l'image à cacher
`bt` le nombre de bits de poids faible modifiée dans l'image support
Pré-requis : Les deux images `support` et `cache` sont de mêmes dimensions.
Sortie : Aucune
Rôle : l'image finale contiendra sur les bits de poids fort l'image `support` et sur les bits de poids faible l'image `cache` ■

exercice 7.9 L'image `cache-cache-surprise.bmp` contient une image cachée. La retrouverez-vous ? ■

7.5 Corrections

■ **Correction 7.1 — PIL.**
1. Les dimensions de la photo `cache-cache.bmp` sont :
   largeur 461 pixels
   hauteur 380 pixels
2. Le nombre total de pixels de cette photo est donc de :
   $461 \times 380 = 175\ 180$ pixels
3. Le poids de cette photo, en octets est donc de :
   Chaque pixel à un poids de 3 octets (1 octet pour le rouge, 1 octet pour le vert, 1 octet pour bleu).
   $175\ 180 \times 3 = 525\ 540$ octets
4. Le poids de cette photo, en kilooctets est donc de :
   Attention ! Piège !
   1 ko = $2^10 = 1\ 024$ octets
   $525\ 540 \div 1\ 024 \approx 513$ soit 513 ko.
   Une image au format `BMP` n'étant pas compressée, vérification faite, l'image fait bien 513 ko, par contre sa version compressée au format `JPG` ne fait plus que 23 ko ... mais avec ce type de format nous ne pourrons pas manipuler les pixels...

■

■ **Correction 7.2 — PIL.**
1. Quelle est la composante en Rouge, Vert et Bleu du pixel de coordonnées (120,200) ?
   Rouge : 213, Vert : 187, Bleu : 126
2. Quelle est la couleur de ce pixel ?
   Un gris rose

■

■ **Correction 7.3 — PIL.** Colorier le pixel de coordonnées (120,200) en blanc.

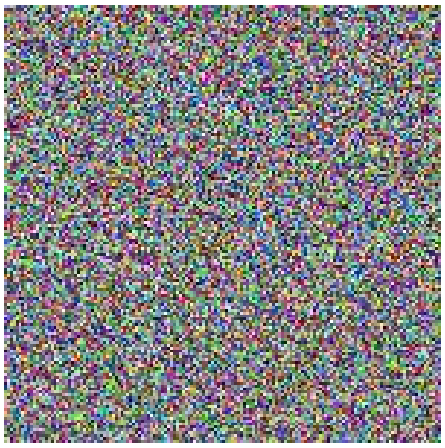
```
1 photo.putpixel((120,200),(255,255,255))
```

■

■ **Correction 7.4 — PIL.** Programmer une fonction qui prend en argument deux entiers `larg` et `haut` et une chaîne de caractères `nom`. Cette fonction devra créer et sauvegarder une image de dimension `larg` et `haut`.
Cette image sera au format `BMP` et sera nommer `nom`.
Enfin tous les pixels auront des couleurs aléatoires [4].

Voici le résultat obtenu avec la commande :

```
nouv_image(150,150,"essais")
```



```
from PIL import Image
from random import randint

def nouv_image(larg,haut,nom):
    '''
    Entrées : larg et haut sont deux entiers
    Sortie : aucune
    Rôle : création et affichage d'une image de dim (lar,haut)
    Tous les pixels étant choisi au hasard
    '''

    #création de l'image
    image = Image.new("RGB",(larg,haut),"white")

    #pixels aléatoires sur la nouvelle image
    for x in range(larg):
        for y in range(haut):
            rouge = randint(0,255)
            vert = randint(0,255)
            bleu = randint(0,255)
            image.putpixel((x,y),(rouge,vert,bleu))

    #afficahe de l'image
    image.show()

    #sauvegarde de l'image
    image.save(nom+".bmp")

    #on ferme le fichier
    image.close()
```

■

■ **Correction 7.5 — Masque.** Quelle est l'opération binaire qui permet de transformer tout nombre entier compris entre 0 et 255 en remplaçant ses 3 bits de poids faible par des 0.
Pour mémoire les opérateurs binaires de bases sont :
AND : `&` OR : `|` XOR : ^ (ou exclusif) NOT : ~
Par exemple `7 & 12` renverra le résultat 4. Pourquoi ? ? ?
`7 & 12` en effet, en binaire 7 s'écrit 0000 0111 et 12 s'écrit 0000 1100. On fait ensuite le ET sur chaque bit : 0000 0111 & 0000 1100 = 0000 0100 autrement dit 4.
Pour le masque, il faut faire l'opération nb & not 7.
not 7 donne le contraire binaire de 7, soit 1111 1000.
Si le nombre est par exemple 204, soit en binaire 1100 1100, 204 & not 7 se calcule ainsi :
1100 1100 & not 0000 0111 = 1100 1100 & 1111 1000 = 1100 1000 autrement dit 200.

4. On pourra utiliser la fonction `randint(a,b)` du module `random` qui renvoie un nombre entier aléatoire entre `a` et `b` inclus.

■

■ **Correction 7.6 — Masque.** Écrire une fonction `masque(nb,bt)` qui prend en arguments deux entiers `nb` et `bt`.
`nb` est un entier entre 0 et 255.
`bt` est entier entre 0 et 8.
La fonction transforme le nombre `nb` en remplaçant ses `bt` bits de poids faible par des 0.

```
def masque(nb,bt) :
    '''
    Entrées :
        nb est un nombre entier entre 0 et 255
        bt est un nombre enter entre 0 et 8
    Sortie : un nb entier
    Rôle : on supprime les bt bits de poids faible à l'aide d'un masque
    '''

    mask = 0
    for i in range(bt,8):
        mask = mask + 2**i
    return nb & mask
```

■

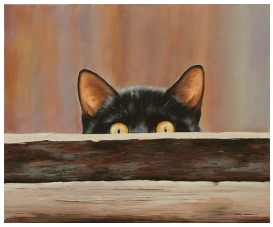
■ **Correction 7.7 — Image Vs bits de poids faible.** Appliquer cette fonction pour mettre des 0 aux bits de poids faible des composantes de rouge, vert et bleu de chaque pixels de l'image `cache-cache.bmp`.
Déterminer quel est le nombre de bits de poids faible à supprimer pour ne pas perdre trop en qualité.

```
from PIL import Image

def masque(nb,bt) :
    '''
    Entrées :
        nb est un nombre entier entre 0 et 255
        bt est un nombre enter entre 0 et 8
    Sortie : un nb entier
    Rôle : on supprime les bt bits de poids faible à l'aide d'un masque
    '''
    mask = 0
    for i in range(bt,8):
        mask = mask + 2**i
    return nb & mask


def image_faible(nom,bt):
    '''
    Entrées :
        nom est le nom de l'image avec son extension
        bt est un nombre entier entre 0 et 8
    Sortie : aucune
    Rôle : on supprime les bt bits de poids faible de chaque
    composante rouge, vert, bleu de chaque pixel
    '''
    photo = Image.open(nom)
    largeur , hauteur = photo.size
    for x in range(largeur):
        for y in range(hauteur):
            rouge , vert , bleu = photo.getpixel((x,y))
            #on supprime les bits de poids faible
            rouge = masque(rouge,bt)
            vert = masque(vert,bt)
            bleu = masque(bleu,bt)
```

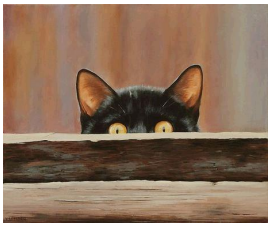
```
35              #on réinjecte la nouvelle composante
36              photo.putpixel((x,y),(rouge,vert,bleu))
37      #on sauvegarde le résultat
38      photo.save('poids_faible'+str(bt)+'.jpg')
39      #on affiche le résultat
40      photo.show()
41      #on ferme fichier
42      photo.close()
43
44 ##############################################################
45 #On fait les images pour des poids faibles allant de 0 à 8.
46 ##############################################################
47
48 for bt in range(0,9):
49      image_faible('cache-cache.bmp',bt)
```



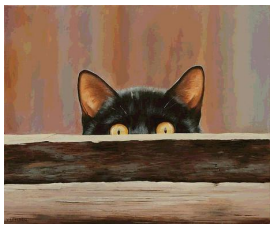
0 bit de poids faible à 0

1 bit de poids faible à 0

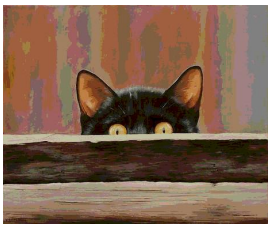
2 bits de poids faible à 0



3 bit de poids faible à 0



4 bits de poids faible à 0



5 bits de poids faible à 0

6 bits de poids faible à 0

7 bits de poids faible à 0

■

■ **Correction 7.8 — Stéganographie.** Écrire une fonction `stegano(support,cache,bt)` répondant à la spécification suivante :
Entrées :

`support` chaîne de caractères donnant le nom de l'image avec son extension, qui cachera l'image
`cache` chaîne de caractères donnant le nom de l'image à cacher
`bt` le nombre de bits de poids faible modifiée dans l'image support

Pré-requis : Les deux images `support` et `cache` sont de mêmes dimensions.
Sortie : Aucune
Rôle : l'image finale contiendra sur les bits de poids fort l'image `support` et sur les bits de poids faible l'image `cache`

```
from PIL import Image

def masque(nb,bt) :
    '''
    Entrées :
        nb est un nombre entier entre 0 et 255
        bt est un nombre enter entre 0 et 8
    Sortie : un nb entier
    Rôle : on supprime les bt bits de poids faible à l'aide d'un masque
    '''
    mask = 0
    for i in range(bt,8):
        mask = mask + 2**i
    return nb & mask

def stegano(support,cache,bt):
    '''
    Entrées :
        support : chaîne de caractères donnant le nom de l'image
            avec son extension, qui cachera l'image
        cache : chaîne de caractères donnant le nom de l'image à cacher
        bt : le nombre de bits de poids faible à modifier dans l'image support
    Pré-requis : les 2 images sont de mêmes dimensions
    Sortie : Aucune
    Rôle : l'image finale contiendra sur les bits de poids fort l'image support
        et sur les bits de poids faible l'image cache
    '''
    supp = Image.open(support)
    cach = Image.open(cache)
    largeur , hauteur = supp.size
    #création de l'image
    finale = Image.new("RGB",(largeur,hauteur),"white")
    #traitement
    for x in range(largeur):
        for y in range(hauteur):
            #composante de l'image support
            r_sup, v_sup, b_sup = supp.getpixel((x,y))
            #on supprime les bits de poids faible
            r_sup = masque(r_sup,bt)
            v_sup = masque(v_sup,bt)
            b_sup = masque(b_sup,bt)

            #composante de l'image à cacher
            r_cac, v_cac, b_cac = cach.getpixel((x,y))
            #on décale les bits de poids forts vers la droite
            r_cac = r_cac >> (8-bt)
            v_cac = v_cac >> (8-bt)
            b_cac = b_cac >> (8-bt)

            #on complète les poids faibles du support
            #par les anciens poids forts de l'image à cacher
```

```
52              rouge = r_sup | r_cac
53              vert = v_sup | v_cac
54              bleu = b_sup | b_cac
55
56              #on injecte le mélange dans l'image finale
57              finale.putpixel((x,y),(rouge,vert,bleu))
58
59      #on sauvegarde le résultat
60      finale.save('stegano_'+str(bt)+'bits.bmp')
61      #on affiche le résultat
62      finale.show()
63      #on ferme les fichiers
64      supp.close()
65      cach.close()
66      finale.close()
67
68  ############################################################
69  #On fait les images pour des poids faibles allant de 0 à 8.
70  ############################################################
71
72  for bt in range(0,9):
73      stegano('cache-cache2.bmp','PIL.bmp',bt)
```

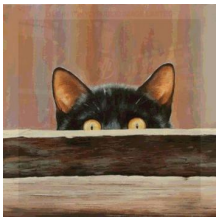
Voici les 9 résultats que nous pouvons obtenir en fonction du nombre de bits de poids faible touchés :

0 bit de poids faible modifié

1 bit de poids faible modifié

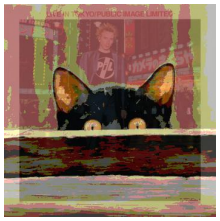
2 bits de poids faible modifiés

3 bits de poids faible modifiés



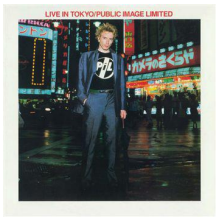
4 bits de poids faible modifiés

5 bits de poids faible modifiés



6 bits de poids faible modifiés

7 bits de poids faible modifiés



8 bits de poids faible modifiés

■