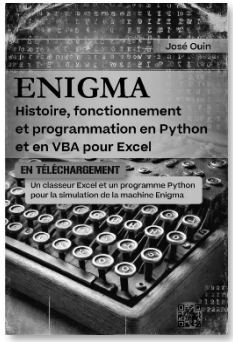
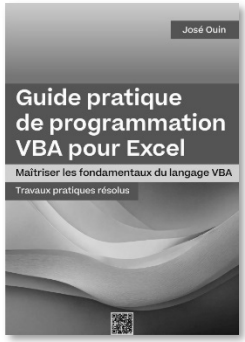
José Ouin
Ingénieur INSA Toulouse
Ancien élève de l'ENS Cachan
Professeur Agrégé de Génie civil
Professeur Agrégé de Mathématiques

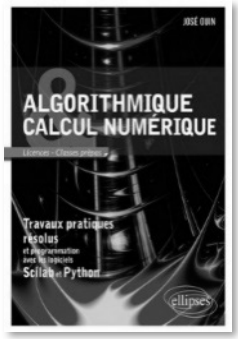
Cryptographie et stéganographie appliquées :

Concepts et programmation en Python

Du même auteur aux Editions Ellipses et sur Amazon







ISBN : 978-2-9593648-9-1

Avant-propos

Depuis toujours, la cryptographie et la stéganographie m'émerveillent par leur capacité à transformer et dissimuler l'information, comme si un voile de mystère enveloppait les messages, rendant leur contenu inaccessible à tous sauf aux initiés. Mon intérêt pour cet art remonte à l'enfance, lorsque je découvrais avec fascination les énigmes et les codes dans les livres d'aventure. Ces jeux de l'esprit m'ont conduit à explorer des concepts bien plus profonds et techniques à l'âge adulte.

Ce qui me passionne particulièrement dans la stéganographie, c'est son côté presque magique : le simple fait de cacher un message ou même un fichier entier dans une image banale me semble être une prouesse digne des meilleurs prestidigitateurs. Ce processus, qui repose à la fois sur des mathématiques élégantes et sur une maîtrise des outils numériques, m'a souvent donné l'impression de manipuler l'invisible.

Ce livre est né de cette passion et de l'envie de partager avec d'autres cet univers fascinant. Il ne s'agit pas seulement d'un manuel technique ou d'un guide pratique ; c'est aussi une invitation à plonger dans un monde où les bits et les pixels peuvent devenir des gardiens de secrets.

Que vous soyez curieux de comprendre comment fonctionne un chiffre de César, désireux de chiffrer des messages avec AES, ou intrigué par l'idée d'insérer un fichier dans une image sans que cela soit détectable, cet ouvrage est conçu pour vous guider pas à pas. J'ai voulu l'écrire comme un compagnon de route, qui vous aidera à maîtriser les bases tout en vous donnant envie d'explorer encore davantage.

En lisant ces pages, j'espère transmettre non seulement des connaissances, mais aussi une part de cet émerveillement qui m'habite à chaque fois que je découvre une nouvelle technique pour coder, crypter ou cacher l'information.

Bonne lecture et bienvenue dans l'univers fascinant des secrets numériques !

José Ouin 

- Un avis positif ?

Merci de prendre le temps de laisser votre évaluation (☆☆☆☆☆) sur la page Amazon de ce livre. Vos avis aident les autres lecteurs à mieux comprendre l'ouvrage.

- Un avis négatif, une question, une suggestion ou une remarque ?

N'hésitez pas à m'envoyer un message via le formulaire de contact de mon site Internet. Lien : `https://joseouin.fr/bandeaucontact`

Table des matières

2- Installation de Python et de l'éditeur PyScripter

Pour débuter avec Python, il est essentiel d'installer le langage sur votre machine ainsi qu'un éditeur de code adapté pour écrire et exécuter vos programmes. Voici les étapes pour installer Python et les recommandations d'éditeurs.

2-1. Installation de Python

2-1.1 Télécharger Python

- Rendez-vous sur le site officiel de Python https://www.python.org/downloads
- Accédez à cette section « Downloads » et choisissez la version appropriée pour votre système d'exploitation (Windows, macOS, ou Linux).
- Téléchargez la version 3.x.

2-1.2 Installer Python

- Lancez l'installateur téléchargé. Pour les utilisateurs de Windows, veillez à cocher la case « Add Python.exe to PATH » avant de continuer. Cela permet de rendre Python accessible depuis la ligne de commande.
- Suivez les instructions pour compléter l'installation. Sur Windows et macOS, l'installateur installe Python, ainsi que le gestionnaire de paquets **pip** (utilisé pour installer des bibliothèques supplémentaires).

La capture d'écran suivante montre les options à cocher lors de l'installation de Python sur Windows :

Logiciel d'installation de Python pour Windows

2-1.3 Gestionnaire de paquets pip

pip est installé par défaut avec Python et permet d'ajouter facilement des bibliothèques externes. Par exemple, pour installer une bibliothèque, vous pouvez utiliser la commande suivante :

```
pip install nom_de_la_bibliotheque
```

pip est essentiel pour installer des bibliothèques populaires comme NumPy, Scipy, Pandas, Matplotlib et TensorFlow.

2-2. Editeur PyScripter

PyScripter est un **IDE** (Environnement de Développement Intégré (en anglais, ***I**ntegrated **D**evelopment **E**nvironment*)) léger et gratuit pour Python, conçu pour les utilisateurs de Windows. Il offre une interface simple avec des fonctionnalités essentielles comme la coloration syntaxique, le débogage et une console interactive intégrée.

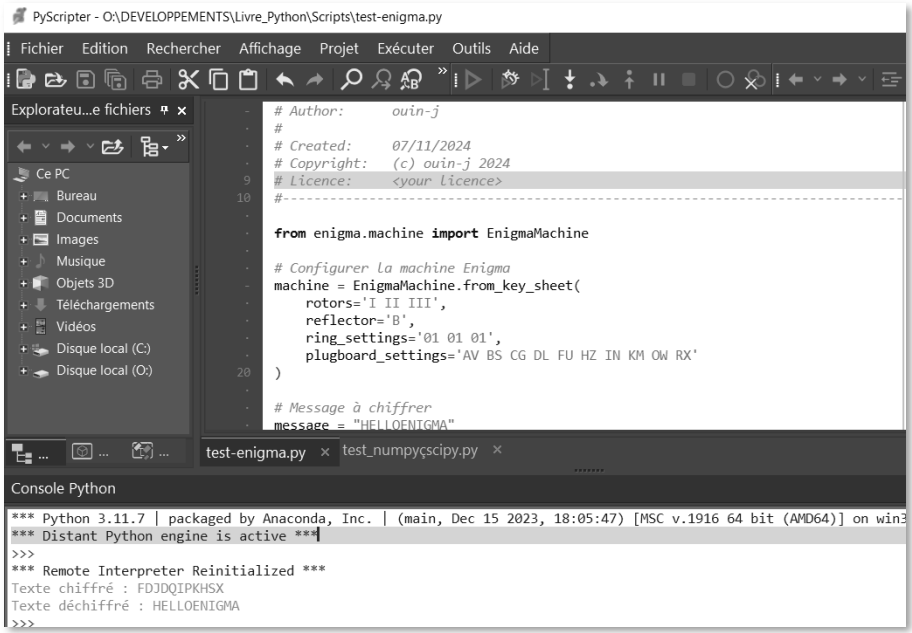
2-2.1 Étapes d'installation

- **Téléchargement :**
  Allez sur https://sourceforge.net/projects/pyscripter
  Cliquez sur « Download »

- **Installation :**
  Lancez l'installateur téléchargé et suivez les instructions.
  Lors de l'installation, sélectionnez la version Python à utiliser si plusieurs sont installées.

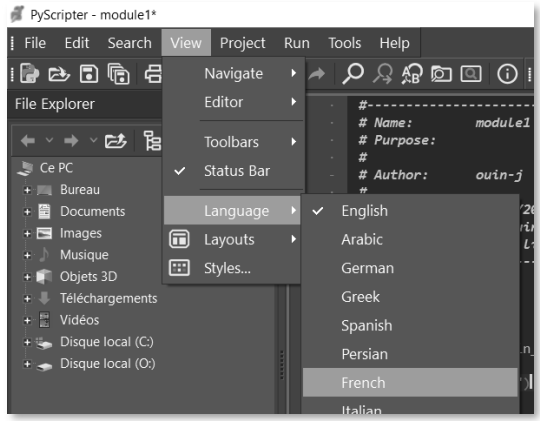
- **Configuration :**
  Utilisez la console interactive intégrée ou créez un nouveau script pour écrire et exécuter votre code (« Ctrl + F9 » pour exécuter).

- **Fonctionnalités principales**
  Coloration syntaxique et débogage : Points d'arrêt, suivi du code ligne par ligne.
  Console interactive : Pour tester des commandes rapidement.
  Explorateur de variables : Permet d'observer les valeurs des variables en temps réel.



Interface du logiciel PyScripter

Pour les menus en français, sélectionner « French » depuis le menu suivant : « View/Language/French »



Menu pour obtenir l'interface en langue Française

Script pour l'opérateur ET :

```
# Définir deux nombres binaires sur 8 bits
a = int('10111010', 2)  # 186 en décimal
b = int('11010101', 2)  # 213 en décimal

# Appliquer l'opérateur ET
resultat = a & b

# Afficher les résultats en 8 bits
print(f"Nombre 1 (binaire) : {bin(a)}")
print(f"Nombre 2 (binaire) : {bin(b)}")
print(f"Résultat (ET bit à bit) : {bin(resultat)}")
```

On obtient dans la console Python de l'éditeur PyScripter :

```
*** Remote Interpreter Reinitialized ***
Nombre 1 (binaire) : 10111010
Nombre 2 (binaire) : 11010101
Résultat (ET bit à bit) : 10010000
>>>
```

Remarque : Compatibilité de l'opérateur "ET" avec les entiers et les nombres binaires

L'opérateur "ET" (**&**) en Python peut être appliqué aussi bien sur des entiers décimaux (type **`int`**) que sur des nombres binaires. Cette flexibilité provient du fait que les nombres binaires en Python sont simplement une représentation d'entiers dans une autre base.

Exemple d'illustration :

```
# Même opération avec des représentations différentes
a_decimal = 186  # En décimal
b_decimal = 213  # En décimal

a_binary = 0b10111010  # En binaire (équivalent à 186)
b_binary = 0b11010101  # En binaire (équivalent à 213)

# Appliquer ET sur des entiers décimaux
resultat_decimal = a_decimal & b_decimal
print(f"Résultat (décimal) : {resultat_decimal} (binaire :
{bin(resultat_decimal)})")

# Appliquer ET sur des nombres binaires
resultat_binary = a_binary & b_binary
print(f"Résultat (binaire) : {bin(resultat_binary)}")
```

On obtient dans la console Python de l'éditeur PyScripter :

```
*** Remote Interpreter Reinitialized ***
Résultat (décimal) : 144 (binaire : 0b10010000)
Résultat (binaire) : 0b10010000
>>>
```

Script pour l'opérateur OU :

```
# Même opération avec des représentations différentes
a_decimal = 186  # En décimal
b_decimal = 213  # En décimal

a_binary = 0b10111010  # En binaire (équivalent à 186)
b_binary = 0b11010101  # En binaire (équivalent à 213)

# Appliquer OU sur des entiers décimaux
resultat_decimal = a_decimal | b_decimal
print(f"Résultat (décimal) : {resultat_decimal} (binaire :
{bin(resultat_decimal)})")

# Appliquer OU sur des nombres binaires
resultat_binary = a_binary | b_binary
print(f"Résultat (binaire) : {bin(resultat_binary)}")
```

On obtient dans la console Python de l'éditeur PyScripter :

```
*** Remote Interpreter Reinitialized ***
Résultat (décimal) : 255 (binaire : 0b11111111)
Résultat (binaire) : 0b11111111
>>>
```

Script pour l'opérateur XOR :

```
# Même opération avec des représentations différentes
a_decimal = 186  # En décimal
b_decimal = 213  # En décimal

a_binary = 0b10111010  # En binaire (équivalent à 186)
b_binary = 0b11010101  # En binaire (équivalent à 213)

# Appliquer XOR sur des entiers décimaux
resultat_decimal = a_decimal ^ b_decimal
print(f"Résultat (décimal) : {resultat_decimal} (binaire :
{bin(resultat_decimal)})")
```

Exemple Python : Vérifier une congruence modulaire

```
def est_congruent(a, b, n):
    return (a - b) % n == 0

# Exemple d'utilisation
print(est_congruent(10, 4, 3))
# True, car 10 - 4 = 6, et 6 est divisible par 3

print(est_congruent(10, 5, 3))
# False, car 10 – 5 = 5, et 5 n'est pas divisible par 3
```

➔ Le plus grand commun diviseur (PGCD)

Le $\mathrm{PGCD}(a, b)$ de deux nombres a et b est le plus grand entier qui divise à la fois a et b. Il est utilisé dans les algorithmes comme RSA pour vérifier la coprimalité de deux nombres.

La coprimalité désigne le fait que deux entiers a et b n'ont aucun diviseur commun autre que 1 c'est-à-dire si $\mathrm{PGCD}(a, b) = 1$.

Exemple Python : Calculer le PGCD avec l'algorithme d'Euclide

```
def pgcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Exemple d'utilisation
print(pgcd(60, 48))  # Résultat : 12
print(pgcd(101, 103))  # Résultat : 1 (coprimes)
```

Utilisation de la fonction intégrée `math.gcd`

```
import math

# Exemple d'utilisation
print(math.gcd(60, 48))  # Résultat : 12
print(math.gcd(101, 103))  # Résultat : 1
```

➔ **Applications combinées**

Les nombres premiers, la congruence modulaire et le PGCD se combinent souvent dans des algorithmes cryptographiques. Par exemple, pour le chiffrement RSA :

- On choisit deux grands nombres premiers p et q.
- On calcule $n = p \times q$.
- On choisit un exposant e qui est coprimé avec $\phi(n) = (p-1)(q-1)$
- On utilise la puissance modulaire pour le chiffrement et le déchiffrement.

Exemple numérique simpliste :

Pour $p = 11$ et $q = 5$, on obtient : $\phi(n) = \phi(55) = (11-1)(5-1) = 40$

Exemple Python : Trouver un nombre e coprimé avec 40

```
def pgcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def est_coprime(a, b):
    return pgcd(a, b) == 1

# Exemple d'utilisation
phi_n = 40
for e in range(2, phi_n):
    if est_coprime(e, phi_n):
        print(f"{e} est coprimé avec {phi_n}")
```

On obtient dans la console Python de l'éditeur PyScripter :

```
*** Remote Interpreter Reinitialized ***
3 est coprimé avec 40
7 est coprimé avec 40
9 est coprimé avec 40
11 est coprimé avec 40
13 est coprimé avec 40
17 est coprimé avec 40
19 est coprimé avec 40
21 est coprimé avec 40
23 est coprimé avec 40
27 est coprimé avec 40
29 est coprimé avec 40
31 est coprimé avec 40
33 est coprimé avec 40
37 est coprimé avec 40
39 est coprimé avec 40
>>>
```

On obtient dans la console Python de l'éditeur PyScripter :

```
*** Remote Interpreter Reinitialized ***
Taille du fichier : 275478 octets
Dimensions : 303x302
>>>
```

➔ Le format PNG (Portable Network Graphics)

Le PNG est un format d'image compressé sans perte, ce qui signifie que la qualité visuelle est préservée. Contrairement au BMP, il est plus complexe, mais il reste adapté pour la stéganographie :

- La compression sans perte permet de modifier les bits de poids faible sans altérer les données après décompression.
- Il inclut également des métadonnées, qui peuvent être exploitées pour insérer des informations supplémentaires.

Structure d'un fichier PNG :

- En-tête : Décrit le type de fichier et les dimensions de l'image.
- Chunks (blocs) : Contiennent les données de l'image, comme les pixels, mais aussi des informations supplémentaires (métadonnées, palette de couleurs).

Avantage pour la stéganographie :

- Très utilisé dans la communication numérique.
- Les blocs de métadonnées peuvent être un bon endroit pour cacher des informations discrètement.

Le script Python ci-après donne un exemple pour examiner la structure d'un fichier PNG. Ce script identifie les différents blocs (ou chunks) qui composent un fichier PNG et affiche leurs noms et tailles. Le fichier « exemple.png » est disponible en téléchargement.

On obtient dans la console Python de l'éditeur PyScripter :

```
*** Remote Interpreter Reinitialized ***
Signature PNG valide détectée.

Structure des blocs (chunks) :

Chunk : IHDR - Taille : 13 octets
Chunk : pHYs - Taille : 9 octets
Chunk : tEXt - Taille : 25 octets
Chunk : IDAT - Taille : 32673 octets
Chunk : IEND - Taille : 0 octets

Fin de fichier PNG.
>>>
```

Signature PNG :
Le fichier PNG commence toujours par une signature spécifique de 8 octets : `\x89PNG\r\n\x1a\n`. Le script vérifie que cette signature est présente pour s'assurer qu'il s'agit bien d'un fichier PNG.

Description des chunks courants :

- IHDR : Contient des informations sur l'image (largeur, hauteur, profondeur de couleur, etc.).
- IDAT : Contient les données compressées de l'image.
- IEND : Indique la fin du fichier PNG.
- tEXt : Métadonnées textuelles
- pHYs : Métadonnées supplémentaires.

Ce script est une introduction simple pour comprendre la structure d'un fichier PNG. Une fois cette base maîtrisée, on peut explorer la manière de modifier les données des chunks (comme IDAT) pour dissimuler des informations en utilisant la stéganographie. Pour ce faire, la bibliothèque **pillow** s'avère particulièrement utile, permettant de manipuler et d'intégrer des données directement dans les pixels ou les métadonnées des images PNG.

```
                # Modifier les LSB des trois composantes (R, V, B)
                for i in range(3):  # R, V, B
                    if bit_index < len(message_bits):
                        pixel[i] = (pixel[i] & ~1) |
int(message_bits[bit_index])
                        bit_index += 1

                pixels[x, y] = tuple(pixel)  # Réassigner le pixel
modifié

    # Sauvegarder l'image avec le message caché
    image.save(output_path)
    print(f"Message caché avec succès dans {output_path}")
    print(f"Nombre total de bits écrits : {bit_index}")

# Exemple d'utilisation
cacher_message("image_originale.png", "Bonjour, ceci est un test",
"image_modifiee.png")
```

On obtient dans la console Python de l'éditeur PyScripter :

```
*** Remote Interpreter Reinitialized ***
Message caché avec succès dans image_modifiee.png
Nombre total de bits écrits : 224
>>>
```

Voici les deux images PNG :

image_originale.png

image_modifiee.png

Conversion en mode RGB :
Si l'image n'est pas en mode RGB, elle est convertie avec image.convert("RGB") pour s'assurer que chaque pixel est un tuple contenant trois valeurs (Rouge, Vert, Bleu).

➔ Pourquoi AES est-il sécurisé ?
AES repose sur des opérations mathématiques complexes qui rendent extrêmement difficile le décryptage sans connaître la clé. Ces opérations incluent des substitutions, des permutations et des transformations linéaires répétées plusieurs fois (appelées tours). Le nombre de tours dépend de la taille de la clé :

- 10 tours pour une clé de 128 bits
- 12 tours pour une clé de 192 bits
- 14 tours pour une clé de 256 bits

➔ Mode d'opération CBC (Cipher Block Chaining)
Un mode d'opération spécifie comment les blocs de données sont chiffrés et liés entre eux. AES peut fonctionner en plusieurs modes, chacun ayant ses spécificités et ses cas d'usage (CBC, GCM, ec.). On étudie ici le mode le plus courant, le CBC (Cipher Block Chaining) :

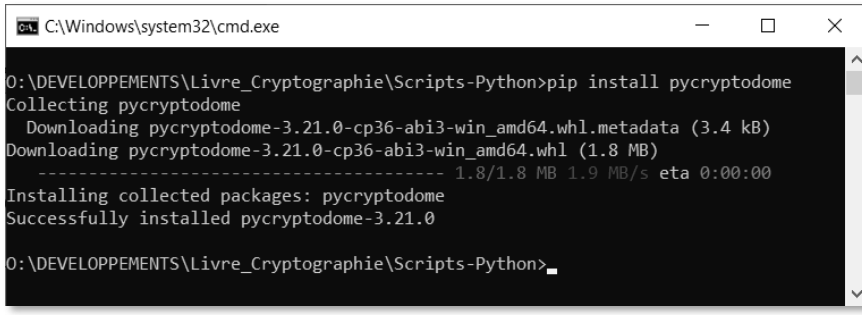
- Chaque bloc de texte en clair est combiné avec le bloc chiffré précédent à l'aide d'une opération XOR (eXclusive OR). **Le premier bloc** est combiné avec un **vecteur d'initialisation** (IV), qui est une **donnée aléatoire**.
- Ce mode garantit que deux blocs identiques de texte en clair ne donneront pas les mêmes blocs chiffrés, ce qui renforce la sécurité.

Avantage : Relativement simple à implémenter et efficace.
Inconvénient : Le déchiffrement nécessite l'accès à tous les blocs précédents, ce qui peut poser problème en cas d'erreur dans une partie des données.

8-3.3 Implémentation en Python

➔ Installation de la bibliothèque pycryptodome
Pour installer la bibliothèque, on exécute la commande :

```
pip install pycryptodome
```



Script Python : chiffrement et déchiffrement avec AES (texte chiffré en Base64)

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
import base64

# Génération d'une clé de 16 octets (128 bits) et d'un vecteur
d'initialisation (IV)
clé = get_random_bytes(16)  # Clé secrète (partagée entre les deux
parties)
iv = get_random_bytes(16)   # Vecteur d'initialisation (IV)

# Texte à chiffrer
texte_en_clair = "Message confidentiel"

# Fonction pour ajouter un padding au texte
def ajouter_padding(texte, taille_bloc):
    return texte + (taille_bloc - len(texte) % taille_bloc) *
chr(taille_bloc - len(texte) % taille_bloc)

# Fonction pour retirer le padding
def retirer_padding(texte):
    return texte[:-ord(texte[-1])]

# Ajout du padding au texte en clair pour correspondre à la taille
des blocs AES
texte_en_clair_paddé = ajouter_padding(texte_en_clair,
AES.block_size)

# Chiffrement
cipher = AES.new(clé, AES.MODE_CBC, iv)
texte_chiffré = cipher.encrypt(texte_en_clair_paddé.encode())

# Encodage du texte chiffré en Base64 pour une meilleure lisibilité
texte_chiffré_base64 = base64.b64encode(texte_chiffré).decode()

print("Texte chiffré (Base64) :", texte_chiffré_base64)
```

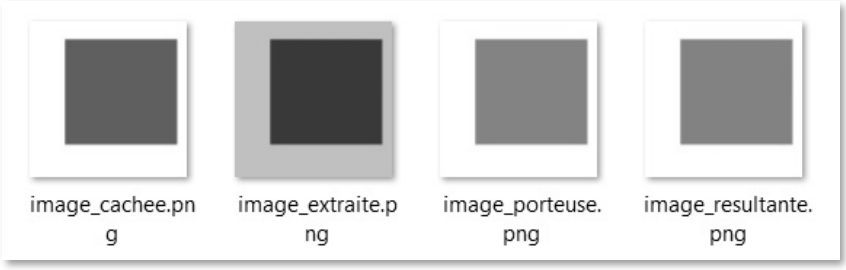
```
if __name__ == '__main__':
    # Paramètres des fichiers
    fichier_image_porteuse = "image_porteuse.png"
    fichier_image_cachee = "image_cachee.png"
    fichier_image_resultante = "image_resultante.png"
    fichier_image_extraite = "image_extraite.png"

    # Traitement des images
    traiter_images(fichier_image_porteuse, fichier_image_cachee,
fichier_image_resultante, fichier_image_extraite)
```

On obtient dans la console Python de l'éditeur PyScripter :

```
*** Remote Interpreter Reinitialized ***
L'image cachée a été intégrée dans l'image porteuse.
L'image cachée a été extraite.
>>>
```

Capture d'écran de l'explorateur de fichiers :



➔ Détails du script :

1/ Chargement des images :

- Les deux images sont chargées au format `PIL.Image` puis converties en tableaux numpy pour manipuler leurs pixels.

2/ Redimensionnement :

- Si les images ne sont pas de tailles compatibles, l'image cachée est redimensionnée pour s'adapter à la taille de l'image porteuse.

Capture d'écran de l'explorateur de fichiers :
La transparence des images a été conservée.

➔ Détails du script :
1/ Mode RGBA :

- Les images sont converties en mode RGBA pour inclure la transparence. Cela permet de traiter les images ayant un canal alpha en plus des couches rouge, verte et bleue.
- Si l'image n'a pas de transparence, le mode RGBA est tout de même utilisé pour uniformiser le traitement.

Redimensionnement :

- L'image à cacher est redimensionnée pour s'adapter à la taille de l'image porteuse à l'aide de la méthode `.resize()`

Encodage et extraction :

- Le canal alpha est traité de la même manière que les canaux RGB.
- Les bits de poids fort de l'image cachée (2 bits par canal) sont insérés dans les bits de poids faible de l'image porteuse.
- Lors de l'extraction, les bits de poids faible de l'image résultante sont récupérés et décalés pour reconstituer les pixels de l'image cachée.

Sauvegarde des images :

- Les images résultantes et extraites sont sauvegardées avec leur transparence intacte grâce au mode RGBA.

```
int(bits_donnees[index_bit])
                    pixels_porteuse[i, j, k] = pixel_val
                    index_bit += 1

    # Sauvegarder l'image résultante
    image_resultante = Image.fromarray(pixels_porteuse, 'RGBA')
    image_resultante.save(fichier_image_resultante)
    print("Le fichier a été caché dans l'image et sauvegardé dans :",
fichier_image_resultante)


# Exemple d'utilisation
if __name__ == "__main__":
    # Variables d'entrée
    fichier_image_porteuse = "image_porteuse_fichier.png"
    fichier_a_cacher = "fichier_a_cacher.txt"
    fichier_image_resultante = "image_resultante_crypte.png"
    cle_xor = "maCleSecrete"  # Clé pour le chiffrement XOR

    # Cacher un fichier dans une image
    cacher_fichier_dans_image(fichier_image_porteuse,
fichier_a_cacher, fichier_image_resultante, cle_xor)
```

On obtient dans la console Python de l'éditeur PyScripter :

```
*** Remote Interpreter Reinitialized ***
Le fichier a été caché dans l'image et sauvegardé dans : image_result
ante_crypte.png
>>>
```

Capture d'écran de l'explorateur de fichiers :

```
    # Sauvegarder l'image résultante
    image_resultante = Image.fromarray(pixels_porteuse, 'RGBA')
    image_resultante.save(fichier_image_resultante)
    print("Le fichier a été chiffré avec AES et caché dans l'image
:", fichier_image_resultante)

if __name__ == "__main__":
    # Variables d'entrée
    fichier_image_porteuse = "image_porteuse_fichier.png"
    fichier_a_cacher = "fichier_a_cacher.txt"
    fichier_image_resultante = "image_resultante_chiffree_AES.png"
    cle_secrete = "maCleSecreteAES"  # Clé utilisée pour AES

    # Générer la clé AES à partir de la clé secrète
    cle_aes = generer_cle_aes(cle_secrete)

    # Cacher un fichier dans une image
    cacher_fichier_dans_image(fichier_image_porteuse,
fichier_a_cacher, fichier_image_resultante, cle_aes)
```

On obtient dans la console Python de l'éditeur PyScripter :

```
*** Remote Interpreter Reinitialized ***
Le fichier a été chiffré avec AES et caché dans l'image : image_resul
tante_chiffree_AES.png
>>>
```

Capture d'écran de l'explorateur de fichiers :