

Documentation programme crypt.py :

```
import os
from random import randint
import sys

def key(rang):
    for _ in range(rang):
        yield randint(1, 255)
```

Il faut importer les modules os random et sys pour le bon fonctionnement du programme

La fonction key est celle qui génère la suite d'octet aléatoire selon une certaine longueur indiquée par le paramètre rang.

J'ai fait le choix de mot clé yield dans le but de retourner un générateur pour gagner en performance.

```
def main():
```

```
    main()
```

L'ensemble du reste du programme est dans une fonction main appelée en fin du fichier dans le but de gagner en performance.

```
sys.stdout.write("Veuillez donner le nom du fichier à chiffrer" + "\n")
file_to_crypt = sys.stdin.readline().replace("\n", "")
try:
    binary_file_to_crypt = open(file_to_crypt, "rb")
except:
    sys.stdout.write("Impossible de trouver le fichier spécifier en entrée" + "\n")
else:
```

Par la suite j'utilise les fonctions sys.stdout.write pour écrire plus vite dans le terminal et sys.stdin.readline() pour lire les entrées utilisateurs.

Comparer aux fonctions de base print et input les retours à la ligne ne sont pas gérés donc il faut rajouter '\n'. Et a contrario le retirer lorsque on lit les entrées utilisateurs.

La construction en try et except permet de vérifier l'existence du fichier à ouvrir. Si la condition est satisfaite alors le programme continue après le else et ouvre le fichier en lecture binaire.

```

sys.stdout.write("Veuillez donner le nom du fichier de sortie chiffré" + "\n")
file_to_crypt_out = sys.stdin.readline().replace("\n", "")
with open(file_to_crypt_out, "wb") as binary_file_to_crypt_out:
    with open("masque_jetable.txt", "wb") as secret_key:
        taille_fichier_to_crypt = os.stat(file_to_crypt).st_size

```

Par la suite je choisis d'opter pour la construction « with open() as » qui permet notamment de s'assurer que les fichiers seront bien fermés à la fin de l'exécution. Dans la fonction open qui appartient à la librairie standard deux paramètres sont renseignés : d'abord le nom du fichier et s'il n'est pas trouvable alors il est créé, puis le mode de lecture ici « wb » qui signifie une écriture en binaire une fois de plus pour la rapidité. L'instruction « os.stat(file_to_crypt).st_size » permet de récupérer la taille du fichier à chiffrer en octet.


```

for random_octet in key(taille_fichier_to_crypt):
    octet_to_encrypt = ord(binary_file_to_crypt.read(1))
    secret_key.write(bytes(str(random_octet) + " ", 'utf-8'))
    binary_file_to_crypt_out.write(bytes([random_octet ^ octet_to_encrypt]))

```

Cette boucle bornée permet de parcourir le générateur formé par « key(taille_fichier_to_crypt) ». L'instruction « key(taille_fichier_to_crypt) » permet de retourner un générateur et le processus de boucle se répétera selon la taille du fichier à chiffrer. L'instruction « binary_file_to_crypt.read(1) » permet de lire le fichier un octet par un octet puis on applique la fonction ord qui retourne le code ASCII correspondant au caractère en train d'être traité.

L'instruction « secret_key.write(bytes(str(random_octet) + " ", 'utf-8')) » permet d'écrire le chiffre aléatoire dans le fichier du masque jetable. Il est nécessaire d'utiliser la fonction bytes() et de préciser un encodage car le fichier est ouvert en mode binaire. L'information pour pouvoir être traitée plus tard pour le déchiffrement se décompose en : premièrement le chiffre aléatoire puis un espace afin de bien pouvoir distinguer chaque nombre et sa position, qui correspond au caractère auquel il est associé. Un exemple ci-dessous de masque jetable :

 masque_jetable.txt - Bloc-notes

Fichier Edition Format Affichage Aide

197 73 139 74 131 19 171 107 207 236 1 184

On peut se poser la question pourquoi ne pas tous laisser en binaire car l'opération XOR symbolisée par le caractère « ^ » n'est pas possible entre type bytes en python.

```

binary_file_to_crypt.close()
sys.stdout.write("Programme terminé" + "\n")

```

Ces dernières lignes représentent la fin du processus où le fichier source non chiffré est fermé puis on affiche un message pour signaler à l'utilisateur la fin d'exécution du script.

Documentation programme decrypt.py :

```
import sys

def main():
    sys.stdout.write("Veuillez donner le nom du fichier à déchiffrer" + "\n")
    file_to_encrypt = sys.stdin.readline().replace("\n", "")
    try:
        binary_file_to_encrypt = open(file_to_encrypt, "rb")
        secret_key = open("masque_jetable.txt", "rb")
    except:
        sys.stdout.write("Impossible de trouver le fichier à déchiffrer ou le masque jetable" + "\n")
    else:
```

On importe le module sys puis on teste la présence du fichier renseigné par l'utilisateur et du masque jetable.

```
sys.stdout.write("Veuillez donner le nom de sortie du fichier déchiffré" + "\n")
fichier_decrypt = sys.stdin.readline().replace("\n", "")
with open(fichier_decrypt, "wb") as fichier_decrypt_binary:
    liste_octet = tuple(map(int, secret_key.readline().split()))
```

L' instruction «tuple(map(int, secret_key.readline().split()))» peut être décomposer en plusieurs points :

- D'abord secret_key.readline().split() permet de lire la ligne où toute la chaîne d'octet du masque jetable est présente puis de séparer les différents éléments dans une liste avec pour séparateur un espace grâce à la méthode .split().
- Puis la fonction map permet de convertir toutes les valeurs de la liste en int dans le but futur de réaliser l'opération XOR.
- Enfin tuple() permet de réduire la taille en mémoire

```
for octet in liste_octet:
    octet_to_decrypt = ord(binary_file_to_encrypt.read(1))
    fichier_decrypt_binary.write(bytes([octet_to_decrypt ^ octet]))
```

On effectue la même méthode que crypt sauf qu'au lieu de parcourir un générateur ici on parcourt un tuple ; aussi on écrit dans le nouveau fichier de sortie.

```
binary_file_to_encrypt.close()  
sys.stdout.write("Programme terminé" + "\n")
```

Enfin on ferme le fichier source qui était chiffré puis on affiche un message de fin d'exécution du script.