Crypter , Décrypter et Casser un code simple.

Un code très simple mais néanmoins encore utilisé lorsque la sécurité est peu critique[1] est le codage par OU exclusif avec une clé (mot de passe) constituée d'une chaîne de caractères.

On utilise la propriété de réversibilité d'un OU exclusif :

Si `code = clair ⊕ clé` alors `code ⊕ clé  = clair` (en python a⊕b est noté a^b)

(en effet `code⊕clé=(clair⊕clé)⊕clé=clair⊕(clé⊕clé)=clair⊕0=clair`)

La même opération est alors utilisée pour coder et décoder un message.

Par exemple, pour coder le message "May the force be with you" avec la clé "Yoda", on code chaque caractère du message avec un caractère de la clé en recommençant avec le premier caractère de la clé lorsque on atteint le dernier :

```
YodaYodaYodaYodaYodaYodaY
May the force be with you
```

Le message codé sera : 'Y'⊕'M' , 'o'⊕'a', 'd'⊕'y', 'a'⊕' ', 'Y'⊕'t', 'o'⊕'h','d'⊕'e','a'⊕' ',.....
Bien sûr, le message codée n'est pas constitué uniquement de caractères imprimables, mais ce n'est pas un problème car il n'est pas destiné à être lu tel quel. Cette méthode s'applique à n'importe quel type de fichier. Le décodage se fait en appliquant le même principe.

Le langage python distingue les caractères (type string), de la représentation mémoire de leur code (type byte) et de la valeur numérique associée (type int). Les calculs (et donc le ou exclusif) ne peut se faire que sur les entiers. Comme il faut manipuler les fichiers en mode 'b' pour travailler sur les octets en non sur les caractères, on va devoir passer du type byte ou au type int pour faire les calculs et repasser au type byte pour réécrire dans le fichier crypté (ou décrypté).

Pour simplifier sans diminuer la généralité, nous utiliserons des clés composées uniquement de caractères ASCII. Ainsi la longueur de la clé sera exactement égale à son nombre de caractères.

La première partie, consiste à coder des fichiers quelconques et les décoder en connaissant la clé. (très facile)

La deuxième partie montre qu'il est possible de retrouver la clé si un texte en français assez long a été codé avec une clé de longueur connue (difficulté moyenne).

La troisième partie montre que la connaissance de la longueur de la clé n'est pas nécessaire (difficile).

---

1 Ce code simple devient impossible à casser sur on utilise la méthode très contraignante du *masque jetable* : une clé aléatoire aussi longue que le message à crypter et qui ne doit jamais être réutilisée pour coder un autre message.

1. Crypter et décrypter un fichier quelconque

Du fait des transformations de type (caractère/byte/entier), la fonction de cryptage/décryptage est donnée :

```
def encode(filename,wFilename, cle ) :
    try :
        f= open(filename,'rb')
    except FileNotFoundError :
        print('Unable to open ',filename )
        sys.exit(-1)
    try :
        fw = open(wFilename,'wb')
    except FileNotFoundError :
        print('No such file or directory:',wFilename )
        sys.exit(-1)
    except PermissionError:
        print('Permission denied:',wFilename )
        sys.exit(-1)
    i=0
    while 1 :
        data = f.read(1)
        if data == b'':
            break
        data = int.from_bytes(data, byteorder='big')
        clex = cle[i%len(cle)]
        clex = clex.encode('latin-1')
        clex = int.from_bytes(clex, byteorder='big')
        code = clex^data
        code = code.to_bytes(1, byteorder='big')
        fw.write(code)
        i+=1
    f.close()
    fw.close()
```

- Vérifiez que vous savez coder et décoder un fichier quelconque avec cette fonction.

- Vérifiez que vous pouvez décoder le fichier **question1.enc** fourni qui a été codé avec la clé

'z}ed!/?fG/YX'

- Commentez cette fonction.

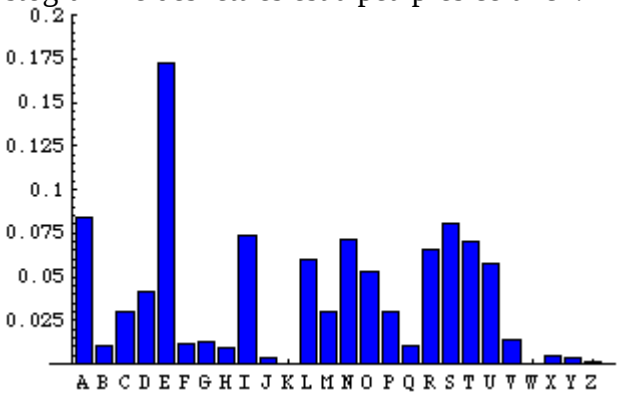
2. Décrypter un fichier texte sans connaître la clé mais en connaissant sa longueur

Lorsque l'on a des connaissances préalables sur le contenu du message, on peut le décrypter

sans posséder la clé. Ici nous faisons l'hypothèse que le message est un texte écrit en français.

De plus nous connaissons la longueur de la clé. Dans ce cas une analyse de fréquence sur un texte assez long peut permettre de retrouver la clé.

Dans un texte français l'histogramme des lettres est à peu près celui ci :



On voit que la lettre E est nettement plus présente que les autres lettres. Dans un texte informatique, c'est le caractère espace qui est le plus présent car il apparaît entre chaque mot (mais souvent dans les messages chiffrés les mots n'étaient sont pas séparés).

Si nous connaissons la longueur n de la clé, nous savons alors qu'un caractère sur n est codé avec le même caractère (inconnu pour l'instant):

```
xyztxyztxyztxyztxyztxyztxyztxyztxyztxyztxyztxyzt
May the force be with you and beware Darkvador
```

Pour trouver le premier caractère de la clé on extrait dans message codé un caractère sur n en partant du premier. On alors

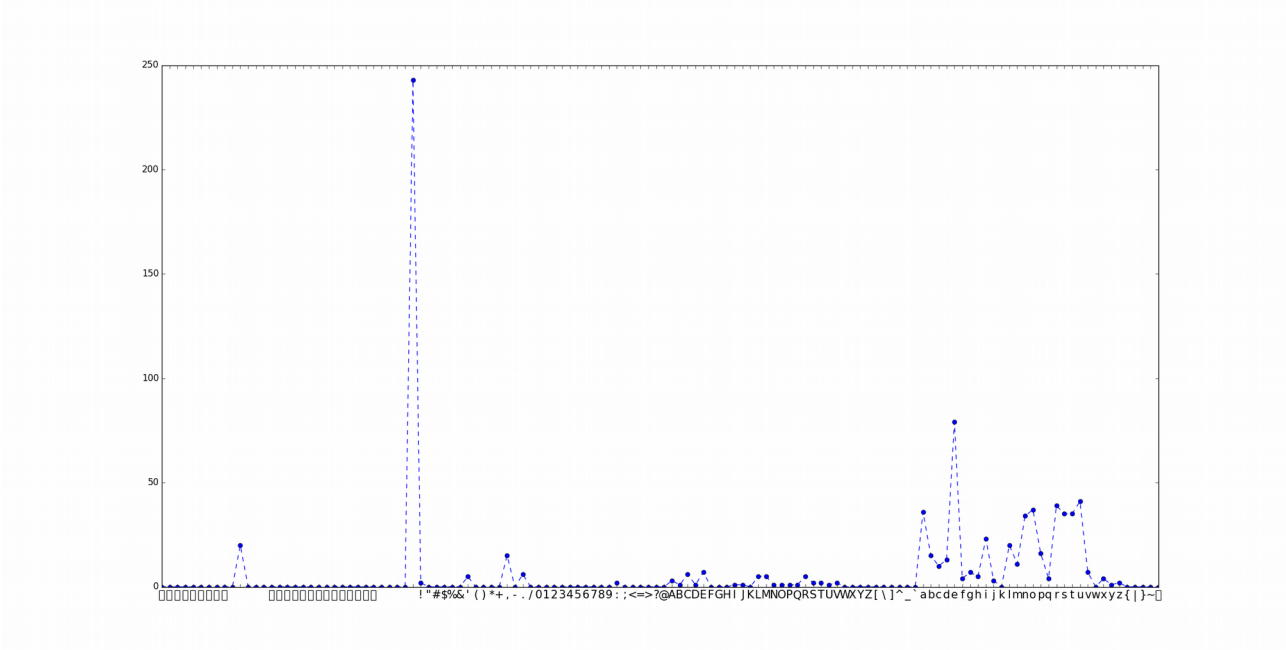
[x⊕M, x⊕t,x⊕f,x⊕e,x⊕' ',x⊕h,x⊕u,x⊕d,x⊕w,x⊕' ',x⊕k,x⊕o,…..]

Si le message est assez long, dans cette suite le x est le plus souvent codé avec l'espace vu que c'est le caractère le plus fréquent de message clair. Il suffit donc de trouver le caractère le plus fréquent et de faire un xor avec pour retrouver la valeur de x.
On procède alors de manière similaire pour trouver les autres caractères de la clé.

- Écrire une fonction qui retourne l'histogramme d'un octet sur n d'un fichier.
L'histogramme est simplement un tableau de 256 cases (une pour chaque octet possible) dont la valeur de chaque case est le nombre de fois que l'octet correspondant apparaît. (Attention l'indice d'une liste doit être un entier et non un octet).
Par exemple `hystogram(filename, begin=0, step=1)`

`begin` est l'indice de départ, `step` est le pas. Il suffit de lire le fichier octet par octet et d'incrémenter la case correspondant à l'octet lu. Vous pouvez utiliser `seek()` pour vous déplacer dans un fichier, voir la documentation python pour faire des déplacement relatifs ou absolus.

- Tracer l'histogramme.



Sur un texte clair, vous devriez obtenir une courbe similaire (ici uniquement sur les octets ASCII 0-128). On voit nettement le maximum correspondant au caractère espace (valeur 32), ensuite le pic du e (valeur 101). Les occurrences des autres lettres minuscules s'écartent parfois de la loi générale sur un texte court.
Faire des essais sur des textes courts pour bien vérifier que votre histogramme est exact.

- Retrouver la clé déjà connue du fichier **question1.enc** .
- Retrouver la clé du fichier **question2.enc** fourni dont la clé est de longueur 37.
- Décoder le fichier **question2.enc**.

3. Décrypter un fichier sans connaissance de la clé

Il faut d'abord déterminer la longueur de la clé pour se ramener au cas précédent. Nous allons utiliser la méthode de Kasiski (1863).

L'idée est que des séquences identiques dans le message clair cryptées avec la même partie de la clé donnent des séquences identiques dans le message crypté. De telles séquences existent dans un texte en langage clair : mots répétés, terminaisons en ons, ent, mment, ….

Il pourrait aussi y avoir par hasard des séquences différentes du message clair qui donnent des séquences identiques lorsqu'elles sont crypté avec des portions différentes de la clé mais si nous trouvons des séquences assez longues cette probabilité reste faible.

Pour la compréhension de ce qui suit nous allons utiliser un codage par décalage (type césar) la clé est une suite de nombre donnant le décalage (modulo 26). La méthode pourra s'appliquer directement au codage par ou exclusif.

Avec la clé 3421 le message `'sde`**`abc`**`dqsdfgh`**`abc`**`djklmn`**`abc`**`xxfctv`**`abc`**`zrtff`**`abc`**`yde'`

devient `'vhg`**`beg`**`frvhhhkeddgnmmprccfbzgfxx`**`beg`**`bswjh`**`beg`**`aeh'`

parce que la même portion de la clé s'est retrouvée en correspondance avec la séquence **abc** du message "clair" :

`'sde`**`abc`**`dqsdfgh`**`abc`**`djklmn`**`abc`**`xxfctv`**`abc`**`zrtff`**`abc`**`yde'`

`342`**`134`**`213421342134213421342134213421342`**`134`**`21342`**`134`**`213`

donc la clé se répète un nombre entier de fois entre chaque décalage. Les séquences **beg** débutent au positions 3, 31 et 39. Soit un écart de 28 positions puis de 8 positions.

La clé est donc multiple de 28 et aussi de 8. 28=7x4 , 8 = 4x2 donc la clé est de longueur 4 !

Ici dans ce cas simple on trouve directement la bonne longueur. Mais on pourrait trouver un multiple de la vraie longueur. En recommençant avec plusieurs séquences on finit par trouver la longueur minimale qui est en fait le pgcd (plus grand diviseur commun) des intervalles entre toutes les répétions.

Résumons la méthode :

- rechercher des séquences identiques dans le message crypté et noter leurs positions.

- calculer les écarts entre ces positions

- calculer le pgcd des écarts.

On vous demande de coder cette méthode en python.

Retrouvez la longueur de la clé du fichier de la question 2

Retrouvez la longueur de la clé du fichier **question3.enc** et décodez le.

Quelques indications :

Commencer par écrire une fonction qui recherche des répétions d'une taille donnée et retourne leurs positions. C'est le plus difficile. Il faut être méthodique et bien tester chaque étape.

`def repetitions(s , le=3)` où s est le message crypté, le la longueur des répétions à chercher.

On peut utiliser la méthode find() de python. Pour faire des essais vous pouvez travailler avec des chaînes de caractères. Le code obtenu fonctionnera de manière identique avec le type byte.

Un exemple d'algorithme (volontairement décrit d'une manière assez éloignée du code) :

début ← 0

Tant que la fin de s n'est pas atteinte :

prendre une tranche test de longueur le commençant à la position début de s

p ← début

(1) rechercher test dans s en partant de la position p+le (find())

(2) si test est trouvée mémoriser test et sa position

(3) p ← après la fin de la chaîne test trouvée

recommencer (1),(2) ,(3) tant que test existe dans s

décaler le début de test de 1

(Nb : ne pas faire (1) (2) (3) si on retombe sur une séquence déjà trouvée)

Avec l'exemple précédent `s='vhg`**`beg`**`frvhhhkeddgnmmprccfbzgfxx`**`beg`**`bswjh`**`beg`**`aeh'`

pour une recherche de longueur 3, on commence par chercher vhg qu'on ne trouve pas puis hgb puis gbe puis beg qu'on trouve en position 3 puis 31 puis 39. Comme on ne trouve plus beg on recherche alors egf, puis gfr jusqu'à la fin de s.

Pour mémoriser les séquences trouvées et les positions, vous pouvez utiliser deux listes : une liste de séquences et une liste de listes de position. ['beg', 'xyz',….] et [[3,31,39], [40, 48, 96, 104],…..]

Vous pouvez également utiliser un dictionnaire dans lequel les clés sont les séquences et les valeurs sont les listes de positions { 'beg' : [3,31,39], 'xyz' : [40, 48, 96, 104], …} (voir le cours)

Tester votre fonction sur des chaînes connues pour être bien certains qu'elle donne des résultats corrects.

Ensuite il suffit de calculer les écarts en soustrayant la première position aux autres valeurs de la liste.

Pour calculer le pgcd vous pouvez utiliser numpy : `np.gcd.reduce([28,8,56,96]) → 4`

Si vous obtenez la valeur 1 c'est que certaines répétitions sont des coïncidences fortuites, il faut les éliminer.

En général pour un fichier assez long il n'est pas nécessaire d'exploiter toutes les répétions. Les plus longues ont moins de chance d'être fortuites.