TP Informatique 6 : Manipulation de chaînes de caractères

1 Préambule

Chargeons tout d'abord deux modules qui seront utiles dans ce TP :

```
import string
import timeit
```

Le premier module, `string`, contient des fonctions utiles pour manipuler les chaînes de caractères, et quelques autres raccourcis utiles. Le second module, `timeit`, contient des fonctions permettant de calculer le temps d'exécution d'une fonction (plus précisément qu'on a pu le faire précédemment avec le module `time`).

2 Mission 1 : identifier la langue d'un texte

Un message électronique a été intercepté sur la boîte au lettres d'un suspect, dont le début est reproduit ci-dessous. Malheureusement, il n'est manifestement pas écrit en français ou en anglais. On cherche à déterminer à quel traducteur il faudrait faire appel.

2.1 Charger le contenu du mail en mémoire

Le fichier `texte.dat` contient une seule ligne correspondant à l'intégralité du contenu dudit mail. Pour charger ce mail dans Python, on exécute les commandes suivantes (on reviendra ultérieurement plus en détail sur les méthodes pour lire et écrire dans un fichier) :

```
fichier = open("D:/TPChaines/texte.dat")
texte = fichier.readline()
fichier.close()
```

La première commande ouvre le fichier `texte.dat`. On peut alors faire référence à son contenu avec le nom `fichier`. La seconde commande prend la première (et unique ici) ligne du fichier et la range sous le nom `texte`. La dernière commande referme le fichier.

Faute de reconnaître la langue, il faudra procéder autrement. On connaît, pour une série de langues, les cinq lettres les plus courantes (par fréquence décroissante) :

- Anglais : etaoi
- Français : esait
- Espagnol : eaosr
- Portugais : aeosr
- Italien : eaion
- Allemand : enisr
- Suédois : eantr
- Turc : aeinr
- Danois : enati
- Polonais : aoien

Nous allons donc compter le nombre d'occurrence de chacune des lettres dans le texte pour déterminer les cinq plus courantes, et donc la langue probable du mail.

2.2 Détermination d'une fréquence

Comme vous l'avez déjà vu, il est possible de faire une boucle qui s'appliquera successivement à tous les caractères d'une chaîne ch très simplement en Python :

```
for caractere in ch :
    (code à exécuter sur chaque caractère)
```

1. Écrire une fonction **CompteCar(c, ch)** qui prend en argument un caractère c et une chaine ch et retourne le nombre d'occurrences de c dans la chaîne ch.

2. Écrire une fonction **CompteAlpha(ch)** qui prend en argument une chaîne ch et affiche le nombre d'occurrence de chacune des lettres (minuscules) de l'alphabet.

- On rappelle que l'instruction `print("a", ":", 5)` permet d'afficher `a : 5` à l'écran. On pourra se servir de ce genre d'instruction pour mettre en forme les résultats.
- le module `string` contient la chaîne de caractères `string.lowercase` qui contient la liste des lettres minuscules, ce qui pourra éventuellement vous être utile.

Pour obtenir les cinq caractères les plus fréquents, par fréquences décroissantes :

- on construit une liste contenant autant d'éléments qu'il y a de caractères dans `string.ascii_lowercase`, ces éléments étant des listes contenant le nombre d'occurence de chaque caractère et le caractère en question
  `L = [ [ 25, 'a' ], [ 1, 'b' ], [ 2, 'c' ] ... ]`
- puis on trie la liste avec `L.sort()` (le tri se fait par ordre lexicographique, donc sur le premier élément de chaque liste, et en cas d'égalité seulement, sur le second) et on ne conserve que les cinq premières listes

3. Écrire une fonction **CarPlusFrequents(ch)** qui prend en argument une chaîne de caractères ch et retourne les cinq caractères présents en plus grand nombre dans la chaîne et leur fréquence, en mettant en œuvre l'approche précédente.

4. En quelle langue le texte est-il vraisemblablement écrit ?

Remarque : dans le texte, il y a des majuscules qui ne sont pas comptées. Si l'on veut améliorer le décompte, il est utile de transformer les majuscules dans le texte en minuscules, ce que l'on peut faire avec la commande

```
texte = texte.lower()
```

3 Mission 2 : déterminer des contacts communs

Dans l'enquête en cours, deux suspects ont été identifiés, qui avaient chacun un téléphone portable. De ces deux téléphones portables ont été extraits deux listes de numéros qui ont été utilisés sur chacun des portables. On cherche à savoir si les deux suspects ont un contact en commun en comparant les deux listes de numéros.

3.1 Comparaison de deux chaines

Dans un premier temps, on souhaite écrire une fonction qui prend en entrée deux chaînes de caractères, et indique si les deux chaînes sont identiques en retournant `True`, ou différentes en retournant `False`. Comme dans le cours, on s'interdira, exceptionnellement, d'utiliser l'opérateur d'égalité fourni par Python afin de se familiariser avec la manipulation de chaînes de caractères.

Deux chaines sont différentes si l'une de ces conditions est vérifiée :
- elles n'ont pas la même longueur ;
- s'il existe un index i tel que la i-ème lettre de la première chaîne n'est pas identique à la i-ème lettre de la seconde chaîne.

Dans le cas contraire, les deux chaînes sont identiques.

1. À partir des conditions ci-dessus, écrire la fonction **Compare(ch1, ch2)** retournant un booléen indiquant si les chaînes sont identiques.

3.2 Charger un dictionnaire

Le fichier `mots.dat` contient l'ensemble des mots français (avec leurs déclinaisons et conjugaisons) de moins de 15 lettres. Nous allons ouvrir ce fichier et le lire, afin de mémoriser tous ces mots dans une liste appelée `mots`, grâce aux commandes suivantes :

```
fichier = open("D:/TPChaines/mots.dat")
mots = [ ligne.rstrip() for ligne in fichier ]
fichier.close()
```

Les commandes ressemblent à celles utilisées pour lire un fichier contenant une unique ligne. Mais ici, il y a un grand nombre de lignes : une pour chaque mot. La seconde commande permet donc de lire chacune de ces lignes d'un seul coup, et de les ranger dans une liste. Chaque ligne du fichier devient un élément de la liste sous la forme d'une chaîne de caractère.

La fonction `.rstrip()` appliquée à chaque ligne permet de retirer le caractère à la fin de chacune d'entre elles qui indique la fin de la ligne (vous pouvez essayer d'exécuter les trois commandes précédentes sans ce `.rstrip()` et voir la différence).

On peut ensuite regarder par exemple les cinq premiers mots de la liste (il n'est pas recommandé de tout afficher !) avec

```
mots[0:5]
```

La commande `len(mots)` nous indique qu'il y a 386264 éléments dans la liste, soit autant de mots (et déclinaisons) de moins de 15 lettres valides en français.

3.3 Recherche simple, vérificateur orthographique

2. Écrire une fonction **Verifie(ch)** qui prend en argument une chaîne de caractère `ch` et retourne `True` si la chaîne est présente dans la liste `mots` et `False` dans le cas contraire. On a écrit un vérificateur orthographique !

3. Combien faut-il de tests pour conclure qu'un mot n'est pas correct ?

Remarque : on peut tester la présence d'un élément `e` dans une liste `L` très simplement en Python avec le test `(e in L)`. On essaiera de s'en passer ici.

3.4 Recherche dichotomique

Il est possible de déterminer plus rapidement la présence d'une chaîne de caractères `ch` dans une liste `L` de chaînes par dichotomie si la liste est triée (ce qui est le cas de notre liste de mots, par ordre alphabétique), dont on rappelle le principe :

Tant que la liste `L` n'est pas vide, on prend un élément au milieu de la liste, que l'on compare à `ch`.
- s'ils sont identiques, alors on a trouvé la chaîne `ch` dans la liste ;
- si la chaîne `ch` se trouve avant l'élément au milieu de la liste, alors on ne conserve de la liste `L` que les éléments figurant avant l'élément que l'on vient d'extraire ;
- si la chaîne `ch` se trouve après l'élément au milieu de la liste, alors on ne conserve de la liste `L` que les éléments figurant après l'élément que l'on vient d'extraire.

Si l'on vide la liste sans trouver l'élément, c'est qu'il n'était pas présent dans la liste.

4. Écrire une fonction **RechercheDichotomique(ch, L)** prenant en argument une chaîne `ch` et une liste de chaînes `L`, et qui renvoie `True` si l'élément figure dans la liste et `False` dans le cas contraire.

5. Écrire une nouvelle fonction **VerifieVite(ch)** qui prend en argument une chaîne de caractère `ch` et retourne `True` si la chaîne est présente dans la liste `mots` et `False` dans le cas contraire, mais cette fois-ci en utilisant **RechercheDichotomique(ch, L)** pour effectuer la recherche plus rapidement.

6. En sachant que $386264 < 2^{19}$, combien de tests faudra-t-il effectuer pour conclure qu'un mot est incorrect cette fois-ci ?

3.5 Recherche du contact commun

Les fichiers `contacts1.dat` et `contacts2.dat` contiennent chacun une liste de numéros de téléphone, un par ligne, triées par numéros croissants.

7. En s'inspirant des commandes précédentes, ouvrir les deux fichiers et construire deux listes `liste1` et `liste2` contenant chacune les listes triées de *chaînes de caractères* correspondant aux numéros appelés par chacun des deux téléphones.

8. Proposer une fonction, faisant appel à **RechercheDichotomique(ch, L)**, fournissant la liste des numéros de téléphone qui ont été appelés par les deux suspects.

4 Mission 3 : identifier un ADN

Sur les lieux du crime, on a pu récupérer 5 empreintes partielles d'ADN. On souhaite savoir si ces empreintes ADN figurent dans la base de données regroupant les empreintes ADN de personnes précédemment mises en cause.

L'ADN est une macro-molécule ressemblant à ruban torsadé, dans lequel se succèdent des bases appelées adénine, cytosine, guanine, thymine. On peut donc représenter un chromosome ou un morceau d'ADN comme une chaîne de caractères contenant uniquement les lettres A, C, G et T.

Pour savoir si un fragment d'ADN fait partie de l'ADN d'une personne, il suffit de regarder si la série de lettres A, C, G, T correspondant au fragment est présente quelque part dans la totalité de la signature ADN. Il s'agit donc ici de déterminer si une chaîne est présente dans une sous-chaîne.

Il se trouve qu'en Python, il est simple de savoir si une sous-chaîne `ch1` est incluse dans une chaîne plus longue `ch2`. Il suffit d'utiliser le test `(ch1 in ch2)`. Cela dit, le but de ce TP étant d'apprendre à manipuler les chaînes de caractères, cette fois encore, on fera dans la suite comme si cette comparaison, bien pratique, n'existe pas !

Pour déterminer la présence de la chaîne **ch1** à l'intérieur de la chaîne **ch2**, il est possible d'utiliser l'algorithme « naif » suivant :

```
Pour tout i entre 0 et len(ch2)-len(ch1) (inclus)
    si ch1[0]=ch2[i] ET ch1[1]=ch2[i+1] ET ... ET ch1[len(ch1)-1]=ch2[i+len(ch1)-1]
        alors la chaîne ch1 est présente dans la chaîne ch2 à partir du i-ème caractère
Sinon la chaîne ch1 n'est pas présente dans la chaîne ch2
```

1. Écrire une fonction **Contenue(ch1, ch2)** qui renvoie `True` si **ch1** est présente à l'intérieur de **ch2** et `False` dans le cas contraire.

Le fichier `baseADN.dat` contient 200 empreintes ADN, à raison d'une empreinte par ligne. Le fichier `fragments.dat` contient des fragments incomplets d'ADN provenant de 5 prélèvements (un par ligne) effectués sur les lieux du crime.

2. Charger ces deux fichiers dans deux listes de chaînes de caractères, appelée par exemple `baseADN` et `fragments`.

3. Essayer grâce à la fonction **Contenue(ch1, ch2)** de trouver une correspondance entre les signatures ADN et les fragments. On essaiera de déterminer les numéros des lignes du fichier ADN dont les signature correspondent à un ou plusieurs prélèvements.

5 Mission 4 : décrypter un message

Un message entre trafiquants a été intercepté. Il est reproduit dans le fichier `message.dat`. Malheureusement, il est « crypté ». Le principe du cryptage est de remplacer les caractères constituant le message par d'autres caractères, en utilisant un ensemble de règles, de façon à rendre le message illisible pour quinconque ne connaît pas les règles en question.

L'une des plus anciennes façon de crypter un message est le code *de César*. Il consiste à remplacer chaque caractère par le suivant dans l'alphabet. Par exemple, « `MPSI` » devient « `NQTJ` ». Le `Z` est, quant à lui, remplacé par un `A`.

Une version un peu plus évoluée consiste à décaler les lettres non pas d'un rang dans l'alphabet, mais de n rangs. Par exemple, si $n = 5$, `A` devient `F`, `C` devient `H` et `X` devient `C`. Et « `MPSI` » devient donc « `RUXN` ».

Le code de César porte ce nom car César l'aurait sensément utilisé (avec un décalage de trois, et en travaillant avec l'alphabet grec, inconnu des gaulois), même s'il est possible qu'il n'ait pas été le premier à le faire. On ne connaît pas avec certitude la fiabilité qu'avait un tel cryptage à cette époque (les premières études sérieuses de cryptanalyse sont à porter au crédit des arabes au IX^e siècle, mais cela ne signifie pas que les messages codés dans l'antiquité étaient impossibles à déchiffrer par tâtonnement).

1. Écrire une fonction **Decale(ch, n)** qui prend en argument une chaîne de caractères `ch` et un entier `n` et crée une nouvelle chaîne dans laquelle les caractères sont ceux de `ch`, décalés de `n` rang si ce sont des lettres majuscules ou minuscules.

On rappelle que la fonction Python `chr(c)` prend en argument un caractère et retourne son point Unicode, et que la fonction `ord(n)` réalise l'opération inverse. Les majuscules ont pour points unicodes, dans l'ordre, les entiers 65 (pour A) à 90 (pour Z), et les minuscules les points 97 (pour a) à 122 (pour z).

2. Si le message est crypté avec un décalage de n, comment peut-on le décrypter ?

3. Décrypter le message fourni.

4. Voyez-vous une façon de « deviner » la valeur de n sans essayer les 25 décalages possibles, sachant que le message est écrit en français ?

C'est le principe de base de la cryptanalyse. Pour corriger ce défaut, le code de Vigenère utilise un décalage qui change à chaque lettre. Si la « clé » (la liste des décalages) est aussi

longue que le message et n'est utilisée qu'une seule fois, le message est impossible à déchiffrer sans la connaissance de la clé.

6 Mission 5 : casser un mot de passe

Des documents importants pour l'enquête ont été enregistrés dans un ordinateur. Malheureusement, il est nécessaire d'avoir un login et un mot de passe pour se connecter et récupérer les données. Il a été possible de récupérer les logins, qui sont `adupont` et `bdurand`. Malheureusement, on ne dispose pas des mots de passe, il va falloir les « casser ».

On commencera par charger le module permettant de se connecter aux comptes avec la commande :

```
exec(open("D:/TPChaines/log.py").read())
```

Pour essayer de se connecter au compte, on exécute la fonction

```
Login("monlogin", "monmotdepasse")
```

Lorsque le couple login/mot de passe est incorrect, cette fonction retourne **`False`** et n'affiche rien. Si le couple login/mot de passe est correct, elle affiche **`True`** et donne accès aux données.

6.1 Attaque brute par dictionnaire

Une première méthode pour « casser » un mot de passe est d'utiliser un dictionnaire contenant tous les mots de passe courants. Diverses études montrent que la plupart des personnes utilisent en effet de très mauvais mots de passes, trop faciles à deviner.

Ainsi, une étude a montré que les 25 mots de passes les plus courants en 2014 (d'après ceux rendus publics suite à des failles de sécurité), et donc aisément cassés, étaient :

- 123456
- password
- 12345
- 12345678
- qwerty
- 123456789
- 1234
- baseball
- dragon
- football
- 1234567
- monkey
- letmein
- abc123
- 111111
- mustang
- access
- shadow
- master
- michael
- superman
- 696969
- 123123
- batman
- trustno1

Si vous reconnaissez le votre, il est temps d'en changer !

On va essayer de voir si les suspects ont été suffisamment imprudents pour utiliser un mot français comme mot de passe.

1. Essayer d'accéder au compte "adupont" en essayant les mots du dictionnaire que l'on a précédemment chargés comme mots de passe (c'est ce que l'on appelle une attaque dictionnaire). Pourquoi est-il dangereux d'utiliser des mots du dictionnaire, des noms ou des dates de naissance comme mot de passe ?

2. Essayer d'accéder au second compte, "bdurand". Cela fonctionne-t-il ?

6.2 Attaque par analyse du temps de réponse

Nous allons envisager une autre méthode pour « deviner » le second mot de passe.

3. En se servant de la fonction **Compare(ch1, ch2)** qui a été écrite plus tôt, écrire une fonction **TestMdP(ch)** qui renvoie **`True`** si la chaîne fournie est "azerty" et **`False`** dans le cas contraire.

On va s'intéresser au temps nécessaire à la fonction compare pour répondre **`True`** ou **`False`**. Pour déterminer précisément le temps d'exécution de la fonction **testMdP(ch)** lorsqu'on lui fournit la chaîne `'qsdfgh'`, on peut utiliser la commande :

```
timeit.Timer("TestMdP('qsdfgh')", "from __main__ import TestMdP").timeit(100)
```

Cette commande permet d'exécuter 100 fois la commande `TestMdP('qsdfgh')` et retourne le temps qu'il a fallu pour cette exécution. Elle utilise le module `timeit` que l'on a chargé tantôt avec la commande

```
import timeit
```

4. Déterminer le temps qu'il faut à la fonction pour répondre **`False`** si l'on utilise les chaînes suivantes comme argument :

- qwerty
- passwd
- azimut
- azures
- azertx

5. En se basant sur les résultats de la question précédente, imaginer une méthode permettant de trouver le mot de passe vérouillant le compte "bdurand". Pour limiter les calculs, on admettra qu'il est composé de six caractères, tous des minuscules sans accent.

Avant que vous ne vous inquiétiez pour la sécurité de vos comptes, les programmes bien conçus demandant un mot de passe attendent toujours un temps identique avant de répondre. Mais cela montre qu'écrire une fonction qui vérifie un mot de passe est un exercice particulièrement délicat.

Pire, il faut aussi que le processeur fasse autant de calculs que le mot de passe soit bon ou faux. Il ne suffit pas d'attendre sans rien faire car le logiciel malveillant peut essayer de savoir si l'ordinateur calcule ou attends sans rien faire. Même sans accès à la machine, le

courant consommé, les vibrations... beaucoup de choses peuvent renseigner quelqu'un de malveillant sur le fonctionnement d'un programme.

On a même montré récemment qu'un simple téléphone, en écoutant le bruit produit par un ordinateur, était capable de savoir s'il travaillait ou attendait sans rien faire, et même partiellement déterminer ce qu'il était en train de calculer :

```
http://www.cs.tau.ac.il/\textasciitilde tromer/acoustic/
```

Une application malveillante sur votre téléphone peut aussi, en utilisant le gyroscope, déterminer en partie ce que vous tapez sur votre téléphone, voire approximativement ce que vous tapez sur un clavier posé sur la même table que le téléphone !