I Quelques rappels sur les chaînes de caractères

1.1 Structure

Un chaîne est un itérable contenant uniquement des caractères (un caractère est l'équivalent d'une chaîne ne contenant qu'une seule lettre)

```
s = ''Une chaîne d'exemple''
len(s)  # le nombre de caractères, espaces compris
s[0]  # le premier caractère
```

À la différence des listes, on ne peut pas modifier les caractères contenus dans une chaîne.

```
s[0] = ''u''  # ceci doit renvoyer une erreur
```

1.2 Itérer sur une chaîne

Comme pour les listes, il y a deux manières d'accéder séquentiellement à tous les caractères d'une chaîne

```
for i in range(len(s)):
    # i est un entier, représentant un indice dans la chaîne.
    # i va prendre les valeurs 0, 1, 2...
    s[i]  # il s'agit du caractère d'indice i
```

ou alors

```
for c in s:
    # c est directement un caractère
    # Avec notre exemple de chaîne s, c va prendre comme valeurs
    # successives ''U'', ''n'', ''e'', '' ''...
```

II Anagrammes

Cadre de l'étude

- On appellera "mot" une chaîne de caractères uniquement composée des 26 lettres de l'alphabet en minuscules. Ainsi on interdit les espaces, accents, cédilles....
- Un **anagramme** d'un mot m un mot obtenu par permutation des lettres de m (on change l'ordre). Par exemple "carte", "tracé", "écart" sont des anagrammes. Rappelons que nous ne considérons pas les accents. Dans le code (sauf mention explicite du contraire), on considère que tous les mots sont sous la forme normalisée du point précédent.

Le but de ce TP est de pouvoir déterminer si deux mots donnés sont anagrammes l'un de l'autre, puis de trouver la liste de tous les anagrammes d'un mot donné.

2.1 Quelques outils

Vous disposez d'un fichier python **outils.py** fourni avec cet énoncé. Ce fichier ne doit PAS être modifié, on s'en servira comme d'une bibliothèque.
Nous n'aurons pas besoin de consulter directement ce fichier non plus. Comme lorsqu'on utilise numpy

Exercice 1
Une fonction contenue dans **outils.py** est la fonction

```
def nb_occurences(mot, lettre):
    c = 0
    for car in mot:
        if car == lettre:
            c = c + 1
    return c
```

1. Sans utiliser python, prévoir le résultat de `nb_occurences('alphabet', 'a')`.
2. Expliquer en une phrase la valeur de retour, dans le cas général.

3. Exprimer en fonction de n, la longueur de `mot`, le nombre de comparaison effectuées.

Exercice 2
On considère maintenant

```
def sont_anagrammes1(mot1, mot2):
    for lettre in mot1:
        nb_occ = nb_occurences(mot1, lettre)
        if nb_occurences(mot2, lettre) != nb_occ:
            return False
    return True
```

1. Il s'agit d'une implémentation d'un algorithme permettant de déterminer si deux mots sont anagrammes l'un de l'autre. Expliquer rapidement cet algorithme.
2. En supposant que `mot1` et `mot2` sont de même longueur n (sinon ils ne peuvent pas être anagrammes l'un de l'autre), exprimer dans le pire des cas le nombre de comparaisons effectuées lors de l'appel à `sont_anagrammes1(mot1, mot2)`

Ouvrez dans l'éditeur le fichier **td7.py**. On remarque qu'il commence par l'import de la bibliothèque outils

Exercice 3
Nous souhaitons maintenant évaluer le temps mis par l'exécution de la fonction étudiée à l'exercice précédent.

1. Exécuter complètement la feuille **td7.py** (dans le but de changer le répertoire courant) puis, dans la console

   ```
   help(outils.temps_evaluation)
   ```

2. On souhaite évaluer le temps moyen d'exécution de deux fonctions (puis d'une troisième dans la suite) déterminant si deux mots sont anagrammes l'un de l'autre, et constater l'évolution avec la longueur des mots testés.
   Le principe est le suivant : on fixe un nombre de répétition noté N ainsi que la longueur n des mots testés, puis on répète N fois
   - générer deux mots aléatoires de taille n grâce à `outils.chaine_alea` (voir l'aide dans la console)
   - calculer le temps d'exécution grâce à `outils.temps_evaluation`

   Le but est de calculer la moyenne, on doit donc sommer les temps obtenu puis diviser par N pour obtenir la valeur souhaitée.
   Compléter la fonction `temps_moyen` du fichier **td7.py**

Exercice 4
On souhaite obtenir un graphique de l'évolution du temps moyen en fonction de la taille pour les fonctions `sont_anagrammes1` et `sont_anagrammes2`.

1. La liste des abscisses (tailles) utilisées sera `[1, 21, 41, …, 201]`. Créer cette liste et la convertir en tableau numpy, dans une variable `X` de la console.
2. Créer une première liste `Y1` contenant les temps d'évaluation par la fonction `sont_anagrammes1` pour les tailles contenues dans `X` et un nombre de répétition égal à 2000, puis tracer un premier graphique.
3. Même chose pour `Y2` et `sont_anagrammes2`, et tracer sur un même graphique (on peut séparer deux commandes par un point virgule, sur une seule ligne).
4. La complexité temporelle de `sont_anagrammes2` semble linéaire, de la forme αn. Déterminer graphiquement α en superposant les courbes sur un même graphique et par tâtonnement. Sur ma machine, je trouve que $\alpha = \frac{1}{8000000}$ est convenable.

2.2 Une autre méthode

Exercice 5
Voici un deuxième algorithme, qui pourra être plus efficace que `sont_anagrammes1` :

- Commencer par convertir les deux mots en deux listes.
- Trier les listes (par ordre alphabétique)
- Vérifier si les listes triées sont égales ou non.

Pour la mise en place en python, nous aurons besoin de deux outils :

```
L.sort()  # trie la liste L en modifiant l'ordre de ses éléments
# cette fonction n'a pas de valeur de retour

L1 == L2  # teste si les deux listes L1 et L2 ont exactement les mêmes
```

```
# éléments et dans le même ordre

list('une chaine') # retourne une liste dont les éléments sont les caractères de la chaîne passée en argument.
```

Compléter la fonction `sont_anagrammes3` en implémentant cet algorithme (c'est à dire, écrire le code python qui correspond à l'algorithme proposé).

Exercice 6
Comparer les temps d'exécution avec `sont_anagrammes2` (en traçant un graphique, comme dans l'exercice précédent). Quelle semble être la meilleure méthode ?

III Application à la recherche d'anagrammes

La bibliothèque **outils** contient une variable globale `liste_mots`. Cette liste contient une liste de mots français modifiés pour respecter nos conventions (aucun accent, cédille ou autre caractère spécial). Nous allons maintenant chercher, pour un mot donné, la liste de tous ses anagrammes en français.

Exercice 7
Compléter la fonction `liste_anagrammes(mot, L)` qui prend un argument un mot et une liste de mots et retourne la liste de tous les anagrammes de `mot` trouvé dans la liste `L`. On utilisera au choix une des trois fonctions précédentes pour vérifier si deux sont sont anagrammes l'un de l'autre.
Une astuce pour accélérer les calculs : tester d'abord si les mots sont de même longueurs. S'il ne le sont pas, il n'y a aucune chance qu'ils soient anagrammes l'un de l'autre.

Exercice 8
Trouver les mots français (dans `outils.liste_mots`) qui sont anagrammes de votre prénom, votre nom ou tout autre mot vous passant par la tête. Le but est d'obtenir une liste la plus grande possible.

Exercice 9 (Pour aller plus loin)
Parmi les mots de `outils.liste_mots`, quelle est la longueur la plus représentée ? Créer la liste des mots de cette longueur puis trouver la liste maximale de mots anagrammes les uns des autres de cette longueur.