TP 9
Recherche et tri dans un texte

Dans ce TP on s'intéresse à diverses manipulations sur du texte.
On rappelle que que les objets de type `str` (*string*), en français *chaines de caractères*, sont écrits entre guillements doubles `"` et sont traités en Python essentiellement de la même façon qu'une *liste* de caractères individuels (d'ailleurs, de nombreux langages de programmation ont un vrai type *caractère individuel* et les chaines ne sont réellement rien de plus que des listes de caractères). On écrira donc par exemple `s = "Texte"` et alors le premier caractère est `s[0]` (ici c'est `"T"`), le suivant est `s[1]` (ici `"e"`) etc et la longueur de la chaine est `len(s)` qui est ici 5.
Les fonctions que nous allons écrire utilisent essentiellement des boucles sur les indices pour parcourir la chaine, elles ont donc une structure similaire aux fonctions vues sur les listes. Il est interdit d'utiliser `in` ou bien `<` sur les chaines...

I Fonctions préliminaires

D'abord la première fonction ne sera pas utilisée tel quel mais est un exercice des plus classique à savoir faire par tous.

Exercice 1. Écrire la fonction `cherche_caractere(s, c)` qui prend en argument une chaine `s` et un caractère seul `c` et qui retourne le *premier indice* auquel le caractère `c` apparait dans `s`. Si le caractère n'apparait pas, on convient qu'elle retourne le nombre `-1` (on aurait d'autres choix possibles ; mais ceci est une convention pratique qui permet de ne pas provoquer d'erreur lors de l'exécution de la fonction).
Méthode : c'est une boucle sur les indices.

On doit par exemple avoir

```
>>> cherche_caractere("BCPST", "P")
2
>>> cherche_caractere("BCPST", "M")
-1
>>> cherche_caractere("Bonjour", "o")
1 # le premier "o" est à l'indice 1, même s'il y en a un autre à l'indice 4
```

Pour réviser les bases, on a aussi :

Exercice 2. Écrire la fonction `compte_caractere(s, c)` qui prend en argument une chaine `s` et un caractère seul `c` et qui retourne le nombre de fois que le caractère `c` apparait dans `s`. Ici il n'y a pas de problème si le caractère n'apparait pas : le nombre est simplement 0.
Méthode : on initialise une variable `n` au départ qui vaut 0 et compte dans une boucle le nombre de caractères.

On doit par exemple avoir :

```
>>> compte_caractere("Bonjour", "o")
2
```

Celle-ci par contre sera bien utile. On se donne un mot `m` qu'on veut chercher dans un texte `s`. Les deux sont en fait des variables de type chaine de caractère quelconques (rien ne dit que `s` est un texte et `m` est un mot au sens français), la seule contrainte pour que cela ait du sens est que `m` soit plus court que `s`. On se donne aussi un indice `i` qui sert de position dans la chaine `s`. On veut savoir si le mot `m` se retrouve dans `s` à partir de l'indice `i` précisément, c'est à dire si `m[0]` et bien égal à `s[i]` puis si `m[1]` est égal à `s[i+1]` puis `m[2]` est égal à `s[i+2]` et ainsi de suite.

Exercice 3. Écrire une fonction `test(s, m, i)` qui prend en argument une chaine `s` (le texte dans lequel on cherche), une chaine `m` (le mot que l'on veut chercher) et un indice `i` (une position dans la chaine `s`) et qui teste (c'est à dire, retourne `True` ou `False`) si le mot `m` apparait dans `s` à l'indice `i`.
Réfléchir d'abord au brouillon à la question suivante : si on cherche un mot de 2 lettres, alors si `i` est le dernier indice de `s` on est sûr que le mot n'est pas à partir de cette position. De même si on cherche un mot de 3 lettre alors `i` ne peut pas être dans les 2 derniers indices de `s`. Cela aide à fixer les bornes des boucles.

Le but est d'avoir par exemple :

```
>>> test("Bonjour", "jo", 3)
True
>>> test("Bonjour", "jo", 2)
False
```

II Recherche de mot

On en vient maintenant au cœur du problème.

Exercice 4. Écrire la fonction `cherche_mot(s, m)` qui prend en argument une chaine `s`, une chaine `m`, et qui cherche si `m` apparait dans `s`, en retournant l'indice `i` où `m` apparait (et `-1` si `m` n'apparait pas).
Méthode : avec une boucle, on cherche si le premier caractère de `m` apparait. Dès qu'on le trouve, on teste en utilisant la fonction précédente s'il s'agit du mot entier qui démarre à la position `i`.

Avec l'exemple précédent on voit avoir

```
>>> cherche_mot("Bonjour", "jo")
3
```

Tester sur des exemples plus compliqués !
En modifiant un peu la fonction précédente, et en s'inspirant des fonctions préliminaires, il n'est pas plus difficile d'écrire :

Exercice 5. Écrire une fonction `compte_mot(s, m)` qui prend en argument une chaine `s` et un mot `m` et compte combien de fois le mot `m` apparait dans `s`.

En fait il se peut que le mot `m` se recoupe avec lui-même, comme dans cet exemple :

```
>>> compte_mot("aaaa", "aa")
3 # en effet, c'est soit sur les indices 0-1 soit 1-2 soit 2-3
```

Cela ne pose pas de problème particulier à programmer, c'est même plus simple...

III Comparaisons

On s'intéresse maintenant au problème de la comparaison de deux mots « comme dans le dictionnaire ». Plus formellemennt cela s'appelle *l'ordre lexicographique* : étant donnés deux mots `s`, `t`, on dit que `s` est *inférieur* à `t` si : ou bien le premier caractère de `s` vient avant celui de `t`, ou bien ils sont égaux et alors on compare de même les seconds caractères, et ainsi de suite.

Remarque 1. Formellement, on peut définir un ordre sur $\mathbb{R}^2$ (ou sur $\mathbb{C}$) de la façon suivante :

$$\forall (a,b),(c,d)\in\mathbb{R}^2,\quad (a,b)\leqslant(c,d)\quad\iff\quad\begin{cases}a<c\\ \text{ou}\\ a=c\quad\text{et}\quad b\leqslant d\end{cases}\tag{1}$$

Cet ordre est bien reflexif, anti-symétrique, transitif (exercice !) mais, même sur $\mathbb{C}$, il n'a pas de bonnes propriétés algébriques par rapport à l'addition et la multiplication...

Tous les caractères utilisables sont représentés dans l'ordinateur chacun par un nombre. Cela signifie qu'il existe des tables de caractères donnant tous les caractères utilisables, classés dans l'orde.
On accède au nombre représentant un caractère en Python par la fonction `ord()` :

```
>>> ord("a")
97
>>> ord("b")
98
>>> ord("?")
63
```

L'inverse est la fonction `chr()` :

```
>>> chr(65)
"A"
>>> chr(201)
"É"
```

Testez !
Pour comparer si un caractère `c` vient avant un caractère `d` on pourra donc simplement écrire la condition `ord(c) <= ord(d)` (on fera attention qu'on prend toutes les inégalités au sens large).

Exercice 6. Écrire une fonction `inferieur(s, t)` qui prend en argument deux chaines `s`, `t` et qui teste (c'est à dire, renvoie `True` ou `False`) si la chaine `s` vient avant la chaine `t` dans l'ordre lexicographique.
Méthode : attention car les mots `s` et `t` n'ont pas forcément même longueur, du coup quand doit s'arrêter la boucle ? Et quelle condition dans la boucle permet d'être sûr que l'on en sort ? Dans le pire des cas, les deux mots sont égaux sur leur début ; mais alors un mot plus court vient avant le même mot rallongé, par exemple `"prépa"` vient avant `"préparatoire"`.

On doit par exemple avoir :

```
>>> inferieur("BioA", "BioB")
True
```

Remarque 2 (optionnel). Un problème qui se pose éventuellement est que les majuscules ont un code de caractère inférieur aux minuscules, on a donc par exemple

```
>>> ord("B")
66
>>> ord("a")
97
>>> ord("B") < ord("a")
True # peu satisfaisant
```

Pour écrire une comparaison qui ne tient pas compte des majuscules, on peut convertir un caractère `c` un minuscule **avant** de tester la comparasion, avec la méthode `c.lower()` :

```
>>> c = "B"
>>> c.lower()
'b'
>>> ord(c.lower()) < ord("a")
False
```

Maintenant on reprend le TP précédent. Le but est de trier une liste de mots dans l'ordre du dictionnaire. Formellement il suffit de faire une seule chose... reprendre les fonctions de tri de liste de nombres mais remplacer les signes `<` par notre fonction `inferieur()`.

Exercice 7. Écrire une fonction `tri_mots(L)` qui prend en argument une *liste* de *chaines de caractères* et qui trie la liste `L` selon l'ordre lexicographique, en s'inspirant de l'une des fonctions de tri du TP précédent.

En fait on peut écrire plusieurs fonctions : `tri_mots_bulle`, `tri_mots_selection` et `tri_mots_insertion`.
On doit par exemple avoir :

```
>>> tri_mots(["injection", "surjection", "antécédent", "bijection", "application"])
["antécédent", "application", "bijection", "injection", "surjection"]
```