Chaînes de caractères (2) : algorithmes classiques

Le but de ce T.P. est de mettre en œuvre un algorithme classique sur les chaînes de caractères : la recherche d'un mot dans un texte. On examinera ensuite des applications de la manipulation de chaînes de caractères.
Dans tout le TP, on supposera que les chaînes de caractères manipulées ne sont pas vides.
Pour chaque exercice, les réponses aux questions devront être consignées dans un fichier **exX.py** où "**X**" sera remplacé par le numéro de l'exercice, qui sera enregistré dans un dossier **TP7** du répertoire **IPT** de votre session.

1 Comparaisons de chaînes de caractères

1.1 Lettres en commun

Exercice 1. Ecrire une fonction `lettres_communes` qui compare deux chaînes de caractères et renvoie l'indice de la première lettre pour laquelle elles diffèrent. En particulier, si les chaînes sont égales, la fonction `lettres_communes` renvoie la longueur des chaînes.

Exemples d'appels à la fonction :

```
>>> lettres_communes("algebre","algorithme")
3
>>> lettres_communes("algebre","algebre")
7
```

```
>>> lettres_communes("algebre","al")
2
```

1.2 Occurrences d'un motif dans un texte

Trouver les occurrences d'une chaîne de caractères (usuellement appelée motif) dans une autre chaîne de caractères (usuellement appelée texte) est une fonctionnalité attendue et implémentée dans les éditeurs de texte (OpenOffice Writer, Word...), elle est par ailleurs indispensable dans les moteurs de recherche (Google, Bing...). Dans d'autres domaines, comme par exemple l'analyse du génome, des algorithmes de recherche de chaînes de caractères sont encore incontournables.

Exercice 2. La fonction proposée ci-dessous, écrite telle quelle dans le fichier TP7_ex2.py, est destinée à recevoir en paramètre deux chaînes de caractères `texte` et `motif` et à renvoyer la position de la première occurrence de `motif` dans `texte`, et `-1` si `motif` n'est pas une sous-chaîne de `texte`. Que faut-il mettre à la place du point d'interrogation pour que cette fonction remplisse son rôle ?

```
def cherche(motif,texte) :
    """en entree : motif et texte sont des chaines de caracteres non vides
    en sortie : le premier indice auquel motif apparait dans texte
    ou -1 si texte ne contient pas motif
    """
    T = len(texte)
    M = len(motif)
    for i in range( ? ):
        if texte[i:i + M]==motif:
            return i
    return -1
```

Tester la proposition faite sur plusieurs cas particuliers.

Exemples d'appels à la fonction :

```
>>> cherche("Milou","Milou")
0
>>> cherche("Milou","Tintin")
-1
```

```
>>> cherche("tin","Tintin")
3
>>> cherche("i","Tintin")
1
```

Exercice 3. Ecrire une fonction `cherche_dernier` très largement inspirée de la fonction `cherche` précédente, qui, cette fois, doit renvoyer l'indice de la dernière occurrence de `motif` dans `texte` ou `-1` si le texte ne contient pas le motif.

Exemples d'appels à la fonction :

```
>>> cherche_dernier("Milou","Milou")
0
>>> cherche_ dernier("Milou","Tintin")
-1
```

```
>>> cherche_ dernier("tin","Tintin")
3
>>> cherche_ dernier("i","Tintin")
4
```

Trouver une méthode du module `str` susceptible de donner la même réponse.

Exercice 4. Reprendre la fonction `cherche` et la modifier pour qu'elle compte le nombre d'occurrences du motif dans le texte. On nommera `nombre` cette nouvelle fonction.
Trouver une méthode du module `str` la même opération.

1.3 Optimisation de la fonction de recherche

L'un des défauts de l'implémentation de la fonction `cherche` de l'exercice **2** est qu'elle crée de nombreuses chaînes `texte[i:i+M]`, ce qui occupe potentiellement une place importante en mémoire.

Exercice 5. **Q5.1.** En s'inspirant de la fonction `cherche`, écrire une fonction `cherche2` corrigeant ce défaut par l'utilisation de deux boucles imbriquées. Il s'agit spécifiquement de remplacer la comparaison `texte[i:i+M]==motif` par une boucle qui parcourra les caractères de la chaîne `motif`.

Q5.2. Modifier la fonction `cherche2` précédente en une fonction `cherche3` de sorte que la fonction `cherche3` affiche le nombre de comparaisons de deux caractères qu'elle a effectuées. La fonction `cherche3` renverra un tuple contenant deux éléments : le résultat de la recherche, comme précédemment avec la fonction `cherche2`, et le nombre de comparaisons.

Exemples d'appels à la fonction :

```
>>> cherche3("a","b")
(-1, 1)
>>> cherche3("a","a")
(0, 1)
```

```
>>> cherche3("abcd","abcabcabcd")
(6,16)
```

Q5.3. Donner l'expression du nombre de comparaisons à effectuer en fonction des longueurs respectives, n et p, du texte et du motif cherché, dans le meilleur cas, puis dans le pire cas.

Exercice 6. **[Bonus]** A l'instar de ce qui a été fait au **5.1**, modifier les fonctions `cherche_dernier` et `nombre`, en deux fonctions `cherche_dernier2` et `nombre2`, en évitant la comparaison `texte[i:i+M]==motif`.

1.4 Substitution

Exercice 7. **Q7.1.** Ecrire une fonction `subst` qui prend en argument trois chaînes de caractères, `old`, `new` et `texte`, et qui renvoie une chaîne de caractères correspondant à la chaîne de caractères `texte` dans laquelle toutes les occurrences du motif `old` auront été remplacées par le motif `new`. Cette fonction correspond au chercher-remplacer des éditeurs de texte (Ctrl+h dans IDLE). On pourra s'inspirer des fonctions `cherche` ou `cherche2` et utiliser une ou plusieurs boucles.

Exemples d'appels à la fonction :

```
>>> subst("d","t","Dupond")
'Dupont'
>>> subst('Romains',"Bretons","Ils sont fous, ces Romains !")
'Ils sont fous, ces Bretons !'
>>> subst("ennemis","amis","Les ennemis de mes ennemis sont mes amis")
'Les amis de mes amis sont mes amis'
```

Q7.2. Utiliser la fonction `subst`**, éventuellement plusieurs fois,** pour transformer la chaîne

```
"Il faut manger pour vivre et non pas vivre pour manger."
```

en la chaîne :

```
"Il faut vivre pour manger et non pas manger pour vivre."
```

Q7.3. Trouver une fonction intégrée réalisant la même opération que la fonction **subst**.

1.5 [Bonus] Jokers

Les jokers (wildcard characters) sont des caractères pouvant remplacer un ou plusieurs caractères lors de la recherche d'un motif. Ce mécanisme est utilisé dans les shell unix, dans les bases de données SQL, *etc.* Dans cet exercice, nous nous limiterons au caractère de substitution `?` qui remplace un unique caractère. A titre d'exemple, les mots `"toto"`, `"titi"` et `"tata"`" correspondent tous les trois au motif `"t?t?"`. De tels motifs utilisés à fin de recherche de chaînes de caractères ayant des spécificités données, sont appelées des **expressions régulières**, ou **expressions rationnelles**, et il existe en Python un module spécifique, nommé `re`, pour les manipuler (https://docs.python.org/3.1/library/re.html).

Exercice 8. Ecrire une fonction `joker` prenant en paramètre une chaîne `texte` et une chaîne `motif` comportant, une ou plusieurs fois, le caractère `?` et qui teste si la chaîne `texte` contient une sous-chaîne correspondant à `motif` (on supposera que le caractère `"?"` n'est pas présent dans `texte`).

Exemples d'appels à la fonction :

```
>>> joker("?i","riri")
True
>>> joker("i?i","fifi")
True
>>> joker("?i","loulou")
False
```

2 [Bonus] Interlude : le jeu du pendu

Dans un répertoire du dossier Classe_LYC_nomdelaclasse, se trouvent les fichiers nécessaires pour cet exercice. L'ensemble de ces cinq fichiers doit être déplacé vers le répertoire de travail du jour, **TP7**. Le fichier de démonstration, **TP7_ex9_demo.pyc**, peut être lancé par un double clic. C'est une version jouable correspondant à ce que vous devez obtenir. Le jeu s'appuie sur des fonctions de dessin et d'affichage, situées dans le fichier **TP7_ex9_affichage_dessin.py** (écrit à l'aide du module **turtle**). Ce fichier ne doit pas être modifié. Les mots à deviner sont écrits dans le fichier **TP7_ex9_liste_mots.txt**. Le script principal, **TP7_ex9_principal.py**, ne doit pas être modifié, mais il fait appel à une fonction, `traitement_proposition`, définie dans le fichier **TP7_ex9_traitement_proposition.py**. Telle quelle la fonction « ne fait rien », mais on peut néanmoins jouer, sans que les propositions soient prises en compte. Le travail demandé consiste à réécrire cette fonction, de sorte que les propositions de lettres soient prises en compte et que le jeu devienne jouable.

Exercice 9. **Q9.1.** Editer le script principal et, en s'aidant de la version de démonstration, expliquer le rôle des différentes parties du programme, et le rôle de chacune des variables « principales ».

La fonction `traitement_proposition` prend en argument une lettre, la lettre proposée par le joueur, et deux chaînes de caractères, `mot`, correspondant au mot à trouver, et `trouvees` correspondant aux lettres déjà trouvées, disposées dans une chaîne de caractères de même longueur que le mot à trouver, et dans laquelle les lettres déjà trouvées sont apparentes, tandis que les lettres non encore découvertes sont remplacées par des astérisques. Cette fonction doit renvoyer, sous la forme d'un tuple, une chaîne `trouvees_nouvelle` et un booléen, `succes`. La chaîne `trouvees_nouvelle` est une copie de la chaîne `trouvees` en entrée, à ceci près que toutes les occurrences dans le mot à trouver de la lettre proposée sont remplacées par cette lettre si celle-ci est présente dans le mot sans avoir déjà été proposée.

Q9.2. Modifier dans le script **TP7_ex9_traitement_proposition.py**, la fonction `traitement_proposition` de sorte que les propositions de lettres soient prises en compte.

Exemples d'appels à la fonction :

```
>>> traitement_proposition("a","python","p**h**")
('p--h--', False)
>>> traitement_proposition("y","python","p**h**")
('py-h--', True)
>>> traitement_proposition("y","python","py*h**")
('py-h--', False)
```

3 Traitement d'expressions mathématiques

Exercice 10. Mini-Calculatrice Python

Ecrire un script **ex10.py** demandant à l'utilisateur de saisir une somme d'entiers positifs et affichant en sortie la valeur de cette somme.

Exemple d'exécution du script ex10.py :

```
>>>
Saisir une somme d'entiers positifs :
155+5+40
Valeur de la somme :  200
```

L'expression, saisie à l'aide de la fonction `input`, aura de ce fait le type `str` et devra être parcourue et analysée afin de pouvoir en effectuer le calcul et d'en renvoyer la valeur. On pourra travailler à partir du script suivant, en le complétant :

```
expr=input("Saisir une somme d'entiers positifs :\n")
valeur=0#la variable valeur s'actualisera au fur et à mesure du calcul,
        #de gauche à droite, de la somme à calculer
nombre=""#la variable nombre est une chaîne de caractères
i=0
while i<len(expr):#on parcourt la chaîne saisie par l'utilisateur
    if expr[i]!="+":
        ...
    else:
        ...
    i+=1
valeur+=int(nombre)
print("Valeur de la somme : ",valeur)
```

Exercice 11. Evaluation d'un polynôme

Ecrire un script **ex11.py** demandant à l'utilisateur de saisir une expression polynomiale d'une variable x, à coefficients entiers, et une valeur entière a, et renvoyant la valeur de cette expression pour la valeur a de la variable x. Là encore, l'expression, saisie à l'aide de la fonction `input` aura le type `str` et devra être décomposée et analysée afin de pouvoir renvoyer la valeur attendue.

Le script comportera la définition d'une fonction `analyse_expression` prenant en argument une expression polynomiale en x, de type `str`, et renvoyant la liste des monômes présents dans l'expression, chacun sous la forme d'une liste de trois entiers correspondant au signe du coefficient du monôme, à la valeur du coefficient et enfin au degré du monôme, de sorte qu'hormis cette fonction, le programme principal pourra être écrit sous la forme suivante :

```
expr=input("Saisir une expression polynomiale en une variable x, \
à coefficients entiers :\n")
a=int(input("Saisir une valeur de la variable x :\n"))
L=analyse_expression(expr)
valeur=0
for m in L:
    valeur+=m[0]*m[1]*a**m[2]
print("Valeur du polynôme",expr,"pour x="+str(a)+" :  ",valeur)
```

La fonction `analyse_expression` devra gérer les monômes sous la forme générique « coefficient »`*x**`« exposant », mais aussi les formes particulières, comme dans les exemples suivants :

Exemples d'appels à la fonction `analyse_expression` :

```
>>> analyse_expression("2+3*x**2")
[[1, 2, 0], [1, 3, 2]]
>>> analyse_expression("x+x**2")
[[1, 1, 1], [1, 1, 2]]
```

```
>>> analyse_expression("-2*x**3-x+1")
[[-1, 2, 3], [-1, 1, 1], [1, 1, 0]]
>>> analyse_expression("-x+5-4+x**3-4*x")
[[-1, 1, 1], [1, 5, 0], [-1, 4, 0], [1, 1,
3], [-1, 4, 1]]
```