TP 2 – Rechercher et ordonner

Proposition de corrigé et commentaires

Objectifs de la séance :

- Intégrer les idées permettant d'automatiser
  - des recherches dans une liste
  - un rangement de liste selon un ordre
- Manipuler des boucles imbriquées

1 Comptage d'éléments dans un tableau

Réaliser une fonction `comptage` qui compte le nombre de fois que l'élément `x` est présent dans une liste `L`.
On pourra la tester en tapant dans la console d'exécution la ligne :
`comptage(5,[1,2,5,3,5,10,5,0,5])` ou la ligne :
`comptage('s','Pour qui sont ces serpents qui sifflent sur vos têtes ?')`

Combien d'instructions de base sont effectuées dans ce programme ?

Programme possible :

```
def comptage(x,L):
    nb=0
    for element in L:
        if element==x:
            nb+=1
    return nb
```

La boucle est réalisée autant de fois que la liste `L` contient d'éléments.

2 Recherche d'un mot dans un texte

Quelques rappels sur les chaînes de caractères
En python les chaînes de caractère sont des données de type `string`, elles sont délimitées par ' ou ". Elles se manipulent comme des listes.

`len(X)`	longueur de la chaîne X ; nombre de caractères dans la chaîne X
`X+Y`	concaténation des chaînes X et Y c'est-à-dire X suivie de Y
`X[i]`	le $(i+1)$-ème caractère de la chaîne X
`X[i:j]`	caractères d'indices i à j dans la chaîne (si i manque : depuis le début, si j manque, jusqu'à la fin)
`int("23")`	conversion de "23" en l'entier 23

Le problème de la recherche d'un mot `m` dans un texte `t` est fondamental dans de nombreux domaines : écriture de document, recherche de gène sur des chromosomes, recherche d'information via un moteur de recherche, etc ...
Une méthode assez naturelle de recherche consiste à comparer une à une les lettre du mot recherché et les lettres du texte `t` (en partant d'une position k que l'on fait varier)

				b	o	n	b	o	n			
							b	o	n	b	o	n
q	u	e	l	b	o	n	b	o	n	b	o	n

Dans la suite, les mots et les textes sont représentés par des chaînes de caractères (type `string`).

1. Que fait l'algorithme suivant ?

```
def Préfixe(m,t):
    # Entrée : deux chaines de caracteres avec len(m) <= len(t)
    n = len (m)
    for i in range(n):
        if m[i] != t[i]:
            return False
    return True
```

Cet algorithme est une fonction qui prend deux paramètres :
- une chaîne de caractères `m` ;
- une chaîne de caractère `t` (plus longue que la chaîne `m`).

- Il crée la variable `n` qui est la longueur de la chaîne `m`.
- Il lance une boucle dont le compteur `i` va de 0 à $n-1$.
- Dans la boucle, il teste si le *ième* caractère de `m` est différent du *ième* caractère de `t`.
- Si tel est le cas, la boucle s'arrête, le programme de la fonction s'arrête, et renvoie une donnée booléenne à la valeur `False`.
- Dans le cas contraire, la boucle se poursuit.
- Si la boucle arrive à son terme (si son compteur a pris autant de valeurs qu'il y a de caractères dans la chaîne `m`), la fonction retourne la valeur booléenne `True`

Au final, il teste si la chaîne `m` (comme "*mot*") est au début de la chaîne `t` (comme "*texte*"). D'où le nom de la fonction : `Préfixe`.

2. Expliquer en quoi l'indentation de la dernière ligne de ce programme est très importante.

Si l'instruction "`return True`" était indentée plus à gauche, elle ne ferait pas partie du code de la fonction.
Inversement, si elle était indentée plus à droite, elle serait dans la boucle `for`, ce qui veut dire que si le premier caractère de `m` est le même que celui de `t`, le programme de la fonction s'arrête et renvoie la valeur `True`, alors que le reste de `m` peut être différent de la suite de `t`.
Enfin, si elle est indentée encore plus à droite, c'est une instruction du `if` qui suit l'instruction `return False` qui fait sortir du programme, elle ne serait donc jamais exécutée.

3. Écrire une fonction `Présent(m,t)` qui décide si le mot `m` est présent dans le texte `t` à l'aide d'une boucle imbriquée dans une autre boucle.

Ci-dessous une solution

```
def Present(m,t):
    # Entree : deux chaÃ®nes de caracteres avec len(m) <= len(t)
    n = len (m)
    N = len (t)
    print(n,N)
    for i in range(N-n): # premiÃ¨re lettre du texte comparÃ©e
        presence=True # passera Ã  False si un caractÃ¨re diffÃ¨re
        for j in range(n): # lettres du mot m
# dÃ©bug
#             print(i,j,m[j],t[i+j])
#             input()
            if m[j] != t[i+j]:
                presence=False
        if presence==True: # indentation dÃ©cisive !
            return True
    return False
```

Remarque : Le choix de deux boucles séquentielles peut sembler plus simple, mais réalise nombre de contrôles inutiles. En effet, dès qu'un caractère de `m` est différent d'un caractère de `t`, on peut sortir de la boucle.
Il existe une façon de quitter une boucle, c'est d'utiliser l'instruction `break`. Sous la ligne 13 on peut insérer cette instruction. .

4. Proposer d'écrire la fonction `Présent(m,t)` de façon plus simple à l'aide des fonctions de chaînes de caractère.

Plusieurs idées peuvent être mises en oeuvre.

Méthode 1 :
Appel à la fonction `Préfixe` déjà réalisée et grignotage à gauche de la chaîne de caractères `t`.

```
def Present(m,t):
```

```
    # Entree : deux chaines de caracteres avec len(m) <= len(t)
    n = len (m)
    N = len (t)
    for i in range(N-n):
        if Prefixe(m,t) == True :
        # Qui se code aussi avantageusement par :
        # if Prefixe(m,t) :
            return True
        t=t[1:] # Grignottage
    return False
```

Méthode 2 : Exploitation de la fonction d'extraction de chaîne de caractère, la plus directe.

```
def Present(m,t):
    # Entree : deux chaines de caracteres avec len(m) <= len(t)
    n = len (m)
    N = len (t)
    for i in range(N-n):
        if m == t[i:i+n]:
            return True
    return False
```

Remarque : La fonction `Préfixe` *pouvait s'écrire plus simplement :*

```
def Prefixe(m,t):
    # Entree : deux chaines de caracteres avec len(m) <= len(t)
    if m == t[0:len(m)-1]:
        return False
    else:
        return True
```

5. Combien d'instructions de base sont effectuées lorsque l'on utilise `Present(m,t)` ?
La solution initiale avec boucles imbriquées réalise nécessairement $k \times n$ passages dans la boucle intérieure, k étant le rang de la première lettre où commence le mot s'il est présent (sinon $k = N - n$).
Par contre avec l'instruction `break` qui force à sortir de la boucle intérieure dès que deux caractères sont différents réduit le nombre de passages dans la boucle. Puisque dans le cas général les caractères sont différents, le nombre de passages dans la boucle peut chuter jusqu'à $k + n$ en cas de présence du mot dans le texte, et $N - n$ en cas d'absence.

6. Si on souhaite minimiser le nombre d'instructions effectuées, vaut il mieux utiliser des boucles `for` ou des boucles `while` ?
Les boucles `for` et `while` ont un caractère d'équivalence en ce sens que chacune peut être écrite à l'aide de l'autre :

L'instruction : `for i in range(n):` est équivalente à :

```
i=0
while i<n :
    [instructions]
    i=i+1
```

L'instruction : `while condition` est équivalente à (presque équivalente, le "beaucoup" est infini avec `while`) :

```
for i in range beaucoup
    if condition == False:
        break     # Instruction qui permet de sortir d'une boucle for
    [instructions]
```

En bonne programmation, on réservera l'instruction `for` pour les boucles dont on connaît à priori le nombre de répétitions (ce n'est pas le cas dans les fonctions créées ci-dessus :-/). L'instruction `while` est utilisée quand la sortie de boucle est conditionnée à un test.

Réécrivons la fonction `Présent` à l'aide d'une boucle `while` :

3 Le tri bulle

1. Que fait le code suivant ? On pourra commencer par faire un exemple

```
def Mystere(L,k):
    # Entree : une liste d'entiers L et un indice k < len(L)
    for j in range (k):
        if L[j +1] < L[j]:
            L[j], L[j+1] = L[j+1], L[j]
    return L
```

2. En déduire un algorithme de tri par ordre croissant de listes d'entiers, que vous implémenterez en Python (cet algorithme de tri s'appelle s'appelle le tri bulle, « bubble sort » en anglais).