TP D'INFORMATIQUE 2
Listes et boucles

1 Listes

Une **liste** est est un objet composé de plusieurs sous-objets, chacun associé à un indice :

```
def somme(L):              #prend en argument une liste L
  s = 0
  n = len(L)               #longueur de L
  for i in range(n):       #les indices de L sont entre 0 et n-1
    s = s + L[i]           #L[i] est le terme d indice i dans L
  return s

print(somme([3,6,1]))      #Test sur un exemple de liste
```

On peut également utiliser une boucle `for` pour parcourir directement les éléments d'une liste, plutôt que ses indices :

```
def somme2(L):
  s = 0
  for x in L:
    s = s + x
  return s
```

1. Écrire une fonction `contient` prenant en argument une liste `L` et un élément `x` et testant (en renvoyant un booléen) si `x` est présent dans `L` ou non.
2. Écrire une fonction `positions` prenant en argument une liste `L` et un élément `x`, et renvoyant la liste des indices où `x` apparaît dans `L`.

   On pourra utiliser `R = []` pour initialiser une liste vide et `R.append(i)` pour ajouter l'élément `i` à la fin de la liste `R`.
3. Écrire une fonction `maximum` prenant en argument une liste non vide de nombres `L`, et renvoyant son élément maximum.

   On testera en particulier cette fonction sur une liste de nombres négatifs.
4. Écrire une fonction `positions_maximum` prenant en argument une liste non vide (d'objets comparables) et renvoyant la liste des positions de son maximum.

   Donner une version utilisant les fonctions précédentes, et une version ne faisant qu'un seul parcours de la liste.
5. On dit qu'une fonction est de **complexité constante** si le nombre d'opérations qu'elle effectue ne dépend pas de la taille de son entrée, et de **complexité linéaire** si ce nombre d'opérations est proportionnel à la taille de son entrée.

   Par exemple, dans le meilleur cas (où `x` est au début de `L`), la fonction `contient` est de complexité constante. Dans le pire cas (où `x` n'est pas dans `L`), `contient` est de complexité linéaire.

   Déterminer la complexité des fonctions `positions`, `maximum` et `positions_maximum`.

2 Boucles imbriquées

On parle de **boucles imbriquées** lorsqu'une boucle (`for` ou `while`) se trouve dans le corps d'une autre boucle. À chaque itération de la boucle extérieure, la boucle intérieure est alors exécutée intégralement :

```
def f(n):
  r = 0
  for i in range(n):
    for j in range(n):
      r = r + j
  return r
```

1. Déterminer le résultat de l'appel `f(4)`.
2. Combien d'additions sont effectuées lors d'un appel `f(n)`, en fonction de n ?
3. En exploitant une simplification mathématique, modifier la fonction `f` pour calculer le même résultat en utilisant moins d'opérations arithmétiques lors d'un appel.

4. Écrire une fonction `somme_double` prenant en argument deux entiers positifs `n` et `m` et renvoyant

$$\sum_{i=0}^{n-1}\sum_{j=0}^{m-1}\ln(i+j^2+1)$$

On utilisera la fonction `log`, correspondant au logarithme naturel ln, qu'il faut importer en précédant la fonction de la ligne

```
from math import log
```

Vérifier que `somme_double(3,5)` renvoie `26.17033192699271`.

5. Combien de fois la fonction `log` est-elle appelée à chaque appel de `somme_double`, en fonction de `n` et `m` ?
6. Écrire une fonction `minmax` prenant en argument une liste de listes de nombres, et renvoyant le minimum des maximums de ces listes.
Par exemple, `minmax([[4, 1, 7, 2], [3, 1, 8], [5, 6]])` doit renvoyer `6`.
La fonction `minmax` pourra faire appel à la fonction `maximum`. Cela reste une situation de boucles imbriquées, dans laquelle la boucle intérieure est celle de la fonction `maximum`.
7. Combien de comparaisons sont effectuées lors d'un appel à `minmax` ?

3 Recherche d'un facteur dans un texte

On s'intéresse dans cette partie au problème de la recherche d'un mot dans un texte.

1. Écrire une fonction `facteur_position` prenant en argument une chaîne de caractères `texte`, une chaîne de caractères `mot` et un indice `i`, et testant si `mot` apparaît dans `texte` à partir de l'indice `i`. On supposera que `i` vérifie la condition `i + len(mot) <= len(texte)` (qu'on appelle une **précondition** de la fonction). On parcourera une chaîne de caractères de la même façon qu'une liste, chaque caractère étant un élément (y compris les espaces).
Par exemple :
`facteur_position('hello world', 'world', 7)` ne respecte pas la précondition, la fonction peut faire n'importe quoi
`facteur_position('hello world', 'world', 6)` doit renvoyer `True`
`facteur_position('hello world', 'world', 5)` doit renvoyer `False`
2. En utilisant la fonction précédente, écrire une fonction `facteur` prenant en argument une chaîne de caractères `texte` et une chaîne de caractères `mot`, et testant si `mot` apparaît dans `texte`.
3. Écrire une fonction `liste_occurrences` prenant en argument une chaîne de caractères `texte` et une chaîne de caractères `mot`, et renvoyant la liste des indices où commencent chaque occurrence de `mot` dans `texte`.
Ainsi :
`liste_occurrences('blablabla', 'bla')` doit renvoyer `[0,3,6]`.
`liste_occurrences('blablablabla', 'blabla')` doit renvoyer `[0,3,6]`.
4. Combien de comparaisons de caractères sont effectuées au maximum lors d'un appel de la fonction précédente, en fonction de `n` la longueur de `texte` et `m` la longueur de `mot` ?
5. En Python, on peut obtenir la **tranche**, c'est-à-dire la sous-chaîne, de l'indice `i` inclus à l'indice j exclu de la chaîne `s` en écrivant `s[i:j]`. Écrire des variantes de `facteur` et `liste_occurrences` utilisant les tranches plutôt que la fonction `facteur_position`.

4 Recherche des deux valeurs d'une liste les plus proches

Dans cette partie, on s'intéresse à déterminer les deux éléments les plus proches dans une liste de nombres, au sens de la valeur absolue.

1. Écrire une fonction `valeurs_plus_proches` prenant en argument une liste de nombres `L`, qu'on supposera de longueur au moins 2, et renvoyant les deux valeurs les plus proches.
Ainsi, `valeurs_plus_proches([-2, 3, 8, 5, -9])` doit renvoyer `(3, 5)`.
2. Combien de fois la fonction `abs` est-elle appelée à chaque appel de `valeurs_plus_proches`, en fonction de `n` la longueur de la liste argument ?
Remarque : D'après le calcul précédent, le temps de calcul de la fonction a un terme dominant en n^2. On dit qu'elle est de **complexité quadratique**.