EXERCICE 2 (6 points)

Cet exercice porte sur la programmation, la programmation orientée objet et les structures de données linéaires.

Cet exercice est composé de 2 parties indépendantes.

Dans cet exercice, on appelle parenthèses les couples de caractères `()`, `{}` et `[]`. Pour chaque couple de parenthèses, la première parenthèse est appelée la parenthèse ouvrante du couple et la seconde est appelée la parenthèse fermante du couple.

On dit qu'une expression (chaine de caractères) est bien parenthésée si

- chaque parenthèse ouvrante correspond à une parenthèse fermante de même type ;
- les expressions comprises entre parenthèses sont des expressions bien parenthésées.

Par exemple, l'expression `'tab[2*(i + 4)] - tab[3]'` est bien parenthésée.

En revanche, l'expression `'tab[2*(i + 4] - tab[3)'` n'est pas bien parenthésée, car la première parenthèse fermante `]` devrait correspondre à la dernière parenthèse ouvrante `(`.

1. Déterminer si l'expression `'[2*(i+1)-3) for i in range(3, 10)]'` est bien parenthésée. *Justifier votre réponse*

Partie A

On peut observer que, si les parenthèses vont par couple, une expression bien parenthésée contient autant de parenthèses ouvrantes que de parenthèses fermantes.

On se propose d'écrire une fonction qui vérifie si une chaine de caractères est bien parenthésée.

2. Écrire une fonction `compte_ouvrante` qui prend en paramètre une chaine de caractères `txt` et qui renvoie le nombre de parenthèses ouvrantes qu'il contient.

3. Écrire une fonction `compte_fermante` qui prend en paramètre une chaine de caractères `txt` et qui renvoie le nombre de parenthèses fermantes qu'il contient.

4. En utilisant les deux fonctions précédentes, écrire une fonction `bon_compte` qui prend en paramètre une chaine de caractères `txt` et qui renvoie `True` si `txt` a autant de parenthèses ouvrantes que parenthèses fermantes et `False` sinon.

5. Donner un exemple de chaine de caractères pour laquelle `bon_compte` renvoie `True` alors qu'elle n'est pas bien parenthésée.

Partie B

Comme l'algorithme précédent n'est pas suffisant, on se propose d'implémenter un algorithme utilisant une structure linéaire de pile.

On se propose d'écrire une classe `Pile` qui implémente la structure de pile.

```
1 class Pile:
2     def __init__(self):
3         self.contenu = []
4
5     def est_vide(self):
6         return len(self.contenu) == 0
7
8     def empiler(self, elt):
9         ...
10
11    def depiler(self):
12        if self.est_vide():
13            return "La pile est vide."
14        return ...
```

6. Compléter les lignes 9 et 14 du code précédent pour que la méthode `empiler` permette d'empiler un élément `elt` dans une pile et que la méthode `depiler` permette de dépiler une pile en renvoyant l'élément dépilé. *Vous n'écrirez que le code des deux méthodes.*

Un algorithme permettant de vérifier si une expression est bien parenthésée consiste à

- créer une pile vide `p` ;
- parcourir l'expression en testant chaque caractère :
  - si c'est une parenthèse ouvrante, on l'empile dans `p` ;
  - si c'est une parenthèse fermante, on dépile `p` ;
    - si les deux caractères correspondent à un couple de parenthèses, on continue le parcours,
    - sinon l'expression n'est pas bien parenthésée ;
  - sinon on continue le parcours.

Si l'expression a été entièrement parcourue, on teste la pile ; l'expression est bien parenthésée si la pile est vide.

7. Déterminer le nombre de comparaisons effectuées si on applique l'algorithme précédent à la chaine de caractères `'tab[2*(i + 4)] - tab[3]'`. En déduire le nombre maximum de comparaisons effectuées si on applique l'algorithme précédent à une chaine de caractères de taille *n* (attention aux comparaisons de la classe pile).

8. Écrire une fonction `est_bien_parenthesee` qui prend en paramètre une chaine de caractères et qui implémente l'algorithme précédent.