Travaux Pratiques Chapitre 3 : Données structurées, Piles et Files

Tous les exercices à partir du II sont extraits de la banque de sujets pratiques.
Tous les codes python sont fournis en fichiers joints.

I. TP1 : Exemples du cours

1) Implémenter la liste chaînée avec toute son interface
   a. sous forme de tuples
   b. en POO (classe Cellule)
2) Implémenter une pile avec toute son interface
   a. Sous forme de tableau (list en python)
   b. En POO (classe Pile)
3) Implémenter une file avec toute son interface
   a. Sous forme de tableau (list en python)
   b. En POO (classe Pile)
   c. Sous forme de deux piles (en réutilisant la question 2)

II. TP 2 : Sujet 22-NSI-16, Exercice 2

Cet exercice utilise des piles qui seront représentées en Python par des listes (type `list`). On rappelle que l'expression `T1 = list(T)` fait une copie de `T` indépendante de `T`, que l'expression `x = T.pop()` enlève le sommet de la pile `T` et le place dans la variable `x` et, enfin, que l'expression `T.append(v)` place la valeur `v` au sommet de la pile `T`.

Compléter le code Python de la fonction `positif` ci-contre qui prend une pile T de nombres entiers en paramètre et qui renvoie la pile des entiers positifs dans le même ordre, sans modifier la variable `T`.

```
def positif(T):
    T2 = ...(T)
    T3 = ...
    while T2 != []:
        x = ...
        if ... >= 0:
            T3.append(...)
    T2 = []
    while T3 != ...:
        x = T3.pop()
        ...
    print('T = ',T)
    return T2
```

Exemple :

```
>>> positif([-1,0,5,-3,4,-6,10,9,-8 ])
T = [-1, 0, 5, -3, 4, -6, 10, 9, -8]
[0, 5, 4, 10, 9]
```

III. TP3 : Sujet 22-NSI-13, Exercice 2

On veut écrire une classe pour gérer une file à l'aide d'une liste chaînée. On dispose d'une classe `Maillon` permettant la création d'un maillon de la chaîne, celui-ci étant constitué d'une valeur et d'une référence au maillon suivant de la chaîne :

```
class Maillon :
    def __init__(self,v) :
        self.valeur = v
        self.suivant = None
```

Compléter la classe File suivante où l'attribut `dernier_file` contient le maillon correspondant à l'élément arrivé en dernier dans la file :

```
class File :
    def __init__(self) :
        self.dernier_file = None

    def enfile(self,element) :
        nouveau_maillon = Maillon(...)
        nouveau_maillon.suivant = self.dernier_file
        self.dernier_file = ...

    def est_vide(self) :
        return self.dernier_file == None

    def affiche(self) :
        maillon = self.dernier_file
        while maillon != ... :
            print(maillon.valeur)
            maillon = ...

    def defile(self) :
        if not self.est_vide() :
            if self.dernier_file.suivant == None :
                resultat = self.dernier_file.valeur
                self.dernier_file = None
                return resultat
            maillon = ...
            while maillon.suivant.suivant != None :
                maillon = maillon.suivant
            resultat = ...
            maillon.suivant = None
            return resultat
        return None
```

On pourra tester le fonctionnement de la classe en utilisant les commandes suivantes dans la console Python :

```
>>> F = File()
>>> F.est_vide()
True
>>> F.enfile(2)
>>> F.affiche()
2
>>> F.est_vide()
False
>>> F.enfile(5)
>>> F.enfile(7)
>>> F.affiche()
7 5 2
>>> F.defile()
2
>>> F.defile()
```

```
5
>>> F.affiche()
7
```

IV. TP4 : Sujet 23-NSI-08, Exercice 2

Nous avons l'habitude de noter les expressions arithmétiques avec des parenthèses comme par exemple : $(2 + 3) \times 5$.

Il existe une autre notation utilisée par certaines calculatrices, appelée notation postfixe, qui n'utilise pas de parenthèses. L'expression arithmétique précédente est alors obtenue en saisissant successivement 2, puis 3, puis l'opérateur +, puis 5, et enfin l'opérateur $\times$. On modélise cette saisie par le tableau `[2, 3, '+', 5, '*']`.
Autre exemple, la notation postfixe de $3 \times 2 + 5$ est modélisée par le tableau :
`[3, 2, '*', 5, '+']`.
D'une manière plus générale, la valeur associée à une expression arithmétique en notation postfixe est déterminée à l'aide d'une pile en parcourant l'expression arithmétique de gauche à droite de la façon suivante :

- si l'élément parcouru est un nombre, on le place au sommet de la pile ;
- si l'élément parcouru est un opérateur, on récupère les deux éléments situés au sommet de la pile et on leur applique l'opérateur. On place alors le résultat au sommet de la pile ;
- à la fin du parcours, il reste alors un seul élément dans la pile qui est le résultat de l'expression arithmétique.

Dans le cadre de cet exercice, on se limitera aux opérations $\times$ et + .
Pour cet exercice, on dispose d'une classe `Pile` qui implémente les méthodes de base sur la structure de pile.

Compléter le script de la fonction `eval_expression` qui reçoit en paramètre une liste python représentant la notation postfixe d'une expression arithmétique et qui renvoie sa valeur associée.

```
class Pile:
    """
    Classe définissant une structure de pile.
    """
    def __init__(self):
        self.contenu = []

    def est_vide(self):
        """
        Renvoie le booléen True si la pile est vide, False sinon.
        """
        return self.contenu == []

    def empiler(self, v):
        """
        Place l'élément v au sommet de la pile.
        """
        self.contenu.append(v)

    def depiler(self):
        """
        Retire et renvoie l'élément placé au sommet de la pile,
        si la pile n'est pas vide.
        """
        if not self.est_vide():
            return self.contenu.pop()
```

```
def eval_expression(tab):
    p = Pile()
    for ... in tab:
        if element != '+' ... element != '*':
            p.empiler(...)
        else:
            if element == ...:
                resultat = p.depiler() + ...
            else:
                resultat = ...
            p.empiler(...)
    return ...
```

Exemple :

```
>>> eval_expression([2, 3, '+', 5, '*'])
25
```

V. TP5 : Sujet 23-NSI-10, Exercice 2

On dispose de chaînes de caractères contenant uniquement des parenthèses ouvrantes et fermantes.
Un parenthésage est correct si :

- le nombre de parenthèses ouvrantes de la chaîne est égal au nombre de parenthèses fermantes.
- en parcourant la chaîne de gauche à droite, le nombre de parenthèses déjà ouvertes doit être, à tout moment, supérieur ou égal au nombre de parenthèses déjà fermées.

Ainsi, `"(((()))(()))"` est un parenthésage correct.
Les parenthésages `"())(()"` et `"(())(()"` sont, eux, incorrects.
On dispose du code de la classe `Pile` ci-contre.

```
class Pile:
    """
    Classe définissant une pile
    """
    def __init__(self):
        self.valeurs = []

    def est_vide(self):
        """
        Renvoie True si la pile est vide, False sinon
        """
        return self.valeurs == []

    def empiler(self, c):
        """
        Place l'élément c au sommet de la pile
        """
        self.valeurs.append(c)

    def depiler(self):
        """
        Supprime l'élément placé au sommet de la pile, à
condition
        qu'elle soit non vide
        """
        if self.est_vide() == False:
            self.valeurs.pop()
```

On souhaite programmer une fonction *parenthesage* qui prend en paramètre une chaîne de caractères *ch* formée de parenthèses et renvoie *True* si la chaîne *ch* est bien parenthésée et False sinon.
Cette fonction utilise une pile et suit le principe suivant : en parcourant la chaîne de gauche à droite, si on trouve une parenthèse ouvrante, on l'empile au sommet de la pile et si on trouve une parenthèse fermante, on dépile (si possible) la parenthèse ouvrante stockée au sommet de la pile. La chaîne est alors bien parenthésée si, à la fin du parcours, la pile est vide.
Elle est, par contre, mal parenthésée :

- si dans le parcours, on trouve une parenthèse fermante, alors que la pile est vide ;
- ou si, à la fin du parcours, la pile n'est pas vide.

Compléter le code de la fonction *parenthesage* et le tester.

```
def parenthesage(ch):
    """
    Renvoie True si la chaîne ch est bien parenthésée
    et False sinon
    """
    p = Pile()
    for c in ch:
        if c == ...:
            p.empiler(c)
        elif c == ...:
            if p.est_vide():
                return ...
            else:
                ...
    return p.est_vide()
```

Exemples :

```
>>> parenthesage("(((()))(()))")
True
>>> parenthesage("())(()")
False
>>> parenthesage("(())(()")
False
```