Travaux pratiques - Listes, Piles, Files

Exercice 1

Voici une classe Python `ListeChainee` dont le code est incomplet.

- Un objet de la classe `ListeChainee` est représenté par sa tête et sa queue.
- La queue est également un objet de la classe `ListeChainee`.
- Pour créer un liste vide, il faut instancier un objet dont la tete et la queue sont égaux à `None`.

```
class ListeChainee:
    def __init__(self,tete,queue):
        self.tete = ...  #tete de la liste chainee (car)
        self.queue = ... # queue de la liste chainee (cdr), objet de la classe
           ListeChainee

    def getTete(self):
        return self.tete

    def getQueue(self):
        return ...

    def estVide(self): #une liste est consideree vide si la tete est None
        return self.tete == None

    def taille(self): #retourne la taille le liste
        if not(self.estVide()):
            cpt = ...
        else:
            cpt = 0
        tmp_liste = self.queue
        while tmp_liste.estVide() == False:
            cpt = ...
            tmp_liste = tmp_liste. ...
        return cpt

    def ajouteEnTete(self,new_tete):
        tmp = ListeChainee(self.tete,self.queue)
        self.queue = ...
        self.tete = ...

    def supprEnTete(self):
        if not(self.estVide()):
            tete = ...
            tmp = ListeChainee(self.queue.tete,self.queue.queue)
            self.queue = ...
            self.tete = ...
            return tete

    def __repr__(self):
        tmp_lst = self
        ch = ""
        while tmp_lst.estVide() == False:
            ch = ch + str(tmp_lst.tete) + "-> "
            tmp_lst = tmp_lst.queue
        ch = ch + "None"
        return ch
```

1. Compléter le code manquant (en pointillé) de la classe `ListeChainee`.

2. Nous disposons donc maintenant d'un nouvelle interface, basée sur la programmation objet, pour gérer des listes chainées.
   L'objectif est de traduire les instructions ci-desous (TD ex 1) avec cette nouvelle interface.
   ```
   L = listeVide()
   ajoutEnTete(10,L)
   ajoutEnTete(9,L)
   ajoutEnTete(7,L)
   t = supprEntete(L)
   ```
   (a) Sur papier, traduire chaque instruction, à l'aide de la nouvelle interface orientée objet.
   (b) Ecrire également l'instruction qui permet d'afficher le contenu de la liste `L`.
   (c) Ecrire un programme principal python reprenant ces instructions.
   (d) Vérifier que l'affichage de la liste correspond au résultat de l'exercice 1 du TD.
3. (a) Ecrire un programme principal python permettant de traduire les instructions suivantes :
   ```
   L2 = listeVide()
   L2 = cons(5, cons(4, cons(3, cons (2, cons(1, cons(0,L2))))))
   ```
   (b) Afficher le contenu de la liste `L2` et vérifier le résultat de l'exercice 1 du TD.
4. Pour chacune des instructions suivantes, écrire le programme python correspondant et vérifier que le résultat correscpond au résultat de l'exercice 1 du TD.

   - `car(L2)`
   - `cdr(L2)`
   - `car(cdr(L2))`
   - `cdr(cdr(cdr(L2)))`
   - `car(cdr(cdr(L2)))`
   - `cdr(cdr(cdr(cdr(cdr(cdr(L2))))))`

Exercice 2

Pour éviter à ses clients de faire la queue, une grande enseigne de distribution a mis en place sur son site web, une application. Elle permet à chaque client qui veut rentrer dans le magasin, de s'inscrire. Il est alors placé dans une file d'attente virtuelle. Le client sera prévenu lorsque ce sera à son tour de rentrer dans le magasin. Le code de cette application utilise un paradigme de programmation objet. Chaque client est modélisé par une chaîne de caractère. La file d'attente est modélisée par un objet d'une classe `File_Attente` dont le code incomplet vous est donné ci-dessous. L'implémentation de la file utilise le type `List` de python .

```
class File_Attente:
    def __init(self):
        self.file= []

    def enfiler(self, client):
        '''
           ajoute un client dans la file d'attente
        '''
        self.file.append(client)
```

Dans la partie programme principal, on crée un objet de la classe `File_Attente()` nommé `mammouth`, représennant la file d'attente du magain *Mammouth*.
On représente la liste des clients déja présent à l'entrée du magasin à l'aide d'une liste de chaîne de caractères.

```
#PROGRAMME PRINCIPAL
mamouth = ...
client = ["Mikael", "Caroline", "Jeanine", "Christine", "Guy"]
```

1. Compléter la classe `File_Attente` afin d'ajouter les méthodes suivantes :
   - `estVide()` qui renvoie `True` si l'objet de la classe `File_Attente` ne comporte aucun élément, `False` sinon
   - `taille()` qui renvoi le nombre d'éléments de la File
   - `affichage()` qui affiche la liste des clients de la file d'attente.
   - `defiler()` qui retire le premier élément de la file et qui renvoie son contenu.
2. Compléter le programme principal afin créer le scénario suivant :
   - Les clients déjà présents à l'entrée du magasin sont mis en file d'attente (dans le même ordre que dans la liste de départ)
   - On affiche sur un écran lumineux la taille de la file d'attente.
   - On affiche sur un écran lumineux la liste des clients en attente.
   - Un nouveau client, Jean, arrive à l'entrée du magasin.
   - On affiche sur un écran lumineux la liste des clients en attente.
   - Une place se libère dans le magasin.
   - On affiche sur un écran lumineux le client entrant et la liste des clients en attente.
   - Une autre place se libère dans le magasin.
   - On affiche sur un écran lumineux le client entrant et la liste des clients en attente.
   - Un nouveau client, Claude, arrive à l'entrée du magasin.
   - On affiche sur un écran lumineux la liste des clients en attente.
   - Le magasin s'est vidé inexplicablement, toute la file d'attente peut donc entrer dans le magasin.

Exercice 3 **Implémentation d'une calculatrice polonaise inversée en Python**

L'objectif de ce TP est de vous familiariser avec les piles en Python en implémentant une calculatrice qui évalue des expressions écrites en **notation polonaise inversée (RPN - Reverse Polish Notation)**.
La notation polonaise inversée (ou postfixée) est une manière d'écrire des expressions mathématiques sans utiliser de parenthèses pour définir l'ordre des opérations.
Les opérateurs suivent leurs opérandes ; par exemple, l'expression infixée (3 + 4) * 2 s'écrit en RPN comme 3 4 + 2 *.

1. Ecrivez (sur papier) en RPN les expressions suivantes et puis donnez le résultat :
   - 3 * 5 - 7 * 1
   - 2 / (5 + 3) - 1
   - (5 * 3) + (7 / 2)
2. Pour implémenter une calculatrice RPN, nous allons utiliser la structure de pile. Une implémentaton orienté objet vous est donnée ci-dessous :

```
class Pile:
    def __init__(self):
        self.elements = []  # Liste pour stocker les elements de la pile

    def is_empty(self):
        """ Verifie si la pile est vide """

    def push(self, valeur):
        """ Ajoute une valeur au sommet de la pile """

    def pop(self):
        """ Retire et renvoie la valeur au sommet de la pile """

    def sommet(self):
        """ Renvoie la valeur au sommet de la pile sans la retirer """
```

Complétez la classe et testez votre classe avec des tests simples.

3. Voici le principe de l'algorithme de la calculatrice RPN :

   (a) Créez une pile vide.
   (b) Parcourez chaque élément de l'expression RPN.
   (c) Si l'élément est un opérande (un nombre), poussez-le sur la pile.
   (d) Si l'élément est un opérateur, dépilez les deux derniers opérandes de la pile, appliquez l'opérateur, et poussez le résultat sur la pile.
   (e) À la fin de l'expression, le sommet de la pile contient le résultat.

   (a) Ecrire un jeu de test avec les expressions de la question 1.
   Notez que tant que votre fonction ne sera pas implémentée, vos tests seront KO, ce qui est normal. C'est le principe du Test Driven Design (TDD), où l'on écrit d'abord le jeu de test. Vous pouvez testez votre fonction étape par étape en cours d'implémentation

   (b) Complétez le code de la fonction `evaluer_rpn(expression)` qui prend en entrée une chaîne de caractères représentant une expression RPN et qui renvoie le résultat de l'évaluation de cette expression.
   Votre fonction devra vérifier que l'on ne divise pas par 0. Une erreur sera indiquée dans ce cas.
   Cette première version prend en charge les 4 opérations usuelles uniquement (addition, soustraction, multiplication et division)

```
def evaluer_expression_rpn(expression):
    pile = Pile()
    tokens = expression.split()
        for token in tokens:
        if token.isdigit():
             #A COMPLETER
        else:
             #A COMPLETER
    return pile.pop()
```

   (c) Adaptez votre fonction pour prendre en charge les fonctions puissance et racine carrée, afin d'évaluer les expressions suivantes :

   - *5 2 ^*
   - *3 4 + 2 * 9 sqrt +*