Numérique et science informatique
Classe de terminale 2024-25
Contrôle n° 3
Jeudi 9 janvier 2025

Exercice 1 (

Partie A : Listes chaînées

Dans un fichier Python on a implémenté une structure de données « liste chaînée » à l'aide d'une classe `Liste_chainee`. Pour chaque méthode de cette classe, on a écrit ci-dessous les en-têtes et la chaîne de documentation (*docstring*) sans indiquer le code d'implémentation. On rappelle que l'indice du premier élément d'une liste chaînée est 0.

```
class Liste_chainee :
    """Les objets de la classe Liste_chainee
     sont mutables."""

    def __init__(self):
    """crée une liste chaînée vide"""
```

```
    def est_vide(self):
    """renvoie True si la liste chaînée est vide,
    False sinon. """

    def taille(self) :
    """renvoie le nombre d'élément dans la liste chaînée"""

    def inserer(self, n, val) :
    """insère un nouvel élément contenant la valeur val
    en une position d'indice n"""

    def lire(self, n) :
    """renvoie la valeur de l'élément d'indice n"""
```

1. Ecrire une suite d'instruction qui permet d'associer à la variable `maListe` la liste chaînée que l'on peut représenter de la façon suivante :

7 -> 19 -> 23 -> 41 -> None

2. Ecrire l'instruction qui insère un élément de valeur `35` juste après l'élément de valeur `23` dans la liste chaînée référencée par la variable `maListe`.

3. Ecrire l'instruction qui place dans la variable `nb` la valeur de l'élément d'indice 3.

4. Ecrire le code d'une fonction `inverser(lc)` qui prend en argument une liste chaînée `lc` et qui renvoie une nouvelle liste chaînée contenant les mêmes éléments dans l'ordre inverse sans modifier la liste chainée `lc`.

Partie B : **Files**

Dans un fichier Python, on a implémenté la structure de données « file » dans un style de programmation fonctionnelle. On a donc créé les fonctions suivantes :

- `creer_file()` qui renvoie une file vide.
- `est_vide(f)` qui prend pour argument une file et qui renvoie `True` si la file est vide, `False` sinon.
- `enfiler(f, val)` qui enfile un élément de valeur `val` dans la file `f`.
- `defiler(f)` qui défile un élément de la file non vide `f` et renvoie la valeur de cet élément.

5. On a écrit la suite d'instruction suivante :

```
maFile = creer_file()
for k in range(5) :
    enfiler(maFile, 3*k+2)
nb1 = defiler(maFile)
nb2 = defiler(maFile)
```

Indiquer le contenu des variables `maFile`, `nb1` et `nb2` après exécution de ce code. On représentera la file référencée par `maFile` horizontalement en plaçant à gauche le premier élément et à droite le dernier élément.

6. Ecrire une fonction `vider(f)` qui

* prend comme argument une file `f`,

* et renvoie un tableau Python (de type `list`) contenant la valeur de tous les éléments de la file `f`).

A l'issue de l'exécution de cette fonction, la file `f` **doit être vide**.

Exercice 2

Cet exercice est composé de 2 parties indépendantes.

Dans cet exercice, on appelle « couples de 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

- à chaque parenthèse ouvrante correspond une parenthèse fermante de même type ;
- les expressions comprises entre parenthèses correspondantes 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 deuxième 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.

On suppose que l'on a aussi écrit de façon similaire une fonction `compte_fermante` qui prend en paramètre une chaine de caractères `txt` et qui renvoie le nombre de parenthèses fermantes qu'il contient.

3. En utilisant les deux fonctions `compte_ouvrante` et `compte_fermante`, 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.

4. On peut observer que la condition testée par la fonction `bon_compte` n'est pas suffisante pour qu'une expression soit bien parenthésée. Donner un exemple de chaine de caractères pour laquelle `bon_compte` renvoie `True` bien qu'elle ne soit pas bien parenthésée.

Partie B

Comme la méthode précédente n'est pas suffisante, 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.

```
class Pile:

    def __init__(self):
        self.contenu = []

    def est_vide(self):
        return len(self.contenu) == 0

    def empiler(self, elt):
         ...

    def depiler(self):
        assert not self.est_vide(), "depiler : la pile est vide."
        return ...
```

5. a. Indiquer l'expression de 4 lettres qui permet de décrire le comportement d'une structure linéaire de pile et expliquer sa signification en une phrase.

b. Dans l'implémentation de la classe `Pile` ci-dessus, on utilise des tableaux Python (de type `list`). On décide d'empiler en plaçant chaque élément nouveau à la fin du tableau. A quelle extrémité de ce tableau doit-on alors retirer un élément lorsque l'on dépile une pile non vide ?

c. Recopier et compléter le code précèdent des méthodes `empiler` et `depiler` 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,
    - si la pile `p` est vide l'expression n'est pas bien parenthésée
    - on dépile `p` ;
    - si les deux caractères ne forment pas un couple de parenthèses, l'expression n'est pas bien parenthésée ; sinon on continue le parcours
  * 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.

6. a. Déterminer le nombre de tests effectués si on applique l'algorithme précèdent à la chaine de caractères `'tab[2*(i + 4)] - tab[3]'`.

b. En déduire le nombre maximum de tests effectués si on applique l'algorithme précèdent à une chaine de caractères de taille `n` (attention aux comparaisons de la classe `Pile`).

7. a. Recopier et compléter le code de la **fonction** `bien_parenthesee` qui prend en paramètre une chaine de caractères et qui implémente l'algorithme précèdent.

```
def bien_parenthesee(expr) :
    parentheses = {'(' : ')', '[': ']', '{': '}' }
    p = ...
    for car in ... :
        if car in parentheses.keys() :
            p....
        elif car in parentheses.values() :
            if p.est_vide() :
                return ...
            car2 = p....
            if parentheses[...] != ... :
                return ...
    return p....
```

b. Écrire quelques lignes d'un programme principal qui teste la fonction `bien_parenthesee` sur les expressions `'tab[5*(i + 4)] - tab[3]'` et `'tab[2*(i + 4] - tab[3)'`.

c. En supposant que cette fonction est correctement écrite, indiquer l'état de la pile `p` au moment où l'on sort de la fonction et ce qui s'affichera dans la console dans chacun de ces deux cas.