Listes (Cours)

S2 - Structures de données

⚠ Attention !

En Première, nous avons utilisé le type de données `list` de Python pour représenter des **tableaux** de d'éléments de même type. Le vocabulaire propre à Python peut induire en erreur et amener à penser que le type "liste" est déjà connu. La structure `list` de Python réalise en fait l'implémentation du type abstrait de données "tableau dynamique" et doit être laissée de côté, malgré l'utilisation du même vocabulaire.
Cela ne nous empêchera pas d'implémenter le type abstrait de données liste en utilisant des structures de type `list` en Python.

1. Du point de vue utilisateur : interface

Définition

Une liste est une structure de données qui permet de stocker des données et d'y accéder directement.
C'est un type abstrait de données :

- linéaire : les données sont stockées dans une structure unidimensionnelle ;
- indexé : chaque donnée est associée à une valeur ;
- ordonné : les données sont présentées les unes après les autres.

Une liste est une collection finie de données. On appelle **tête** le premier élément de la liste et **queue** la liste privée de son premier élément. Il est seulement possible d'ajouter et de lire une donnée en tête de la liste.

L'**interface** minimale permettant de définir le type abstrait de données "liste" comporte cinq fonctions, qui sont appelées **primitives** :

- `creer()`, qui crée une liste vide ;
- `ajouter(element, liste)`, qui ajoute un élément en tête de liste ; ces deux première primitives peuvent parfois se regrouper en une seule ;
- `tete(liste)`, qui renvoie la valeur de l'élément en tête de liste ;
- `queue(liste)`, qui renvoie la liste privée de son premier élément ;
- `est_vide(liste)`, qui renvoie vrai si la liste est vide, faux sinon.

Ce type abstrait de données est non mutable (il n'y a pas de primitive permettant de modifier la valeur d'un élément de la liste).

Remarque : on peut selon les besoins ajouter d'autres fonctions permettant par exemple de renvoyer la longueur d'une liste, de rechercher un élément ou d'accéder au ième élément …

Exemple

Supposons implémenté le type abstrait **liste**. Nous disposons d'une interface composée des cinq primitives décrites ci-dessus. On exécute le code suivant ligne par ligne :

```
1 L = creer()
2 est_vide(L)
3 L1 = ajouter(12, L)
4 est_vide(L1)
5 L1 = ajouter(15, L1)
6 L1 = ajouter(1, ajouter(11,L1))
7 tete(L1)
8 L2 = queue(L1)
```

- La ligne 1 créée une liste vide `L` ;
- La ligne 2 affiche `True` car la liste `L` est vide ;
- Après exécution de la ligne 3, la liste `L1` contient l'élément unique 12 ;
- La ligne 4 affiche `False` car la liste `L1` n'est pas vide ;
- Après exécution de la ligne 5, la liste `L1` contient les éléments 12 et 15 ;
- La ligne 6 montre que l'on peut **composer** les ajouts pour ajouter en une seule fois plusieurs éléments. Après exécution de la ligne 6, la liste `L1` contient les éléments 12, 15, 11 et 1 ;
- La ligne 7 affiche 1 : c'est la tête de la liste (il s'agit du dernier élément ajouté) ;
- La ligne 8 définit une liste `L2` égale à la queue de la liste `L1`. `L2` contient donc les éléments 12, 15 et 11.

2. Du point de vue concepteur : implémentation(s)

Nous allons implémenter le type abstrait "liste" en Python de deux façons différentes.

Implémentation avec des tuples

Nous allons ici utiliser des tuples et la programmation fonctionnelle (rappel : fonctions pures, pas d'affectations, pas de boucles).

```
""" Implémentation du type abstrait "liste" avec des tuples"""


def creer() -> tuple:
    """Retourne une liste vide"""
    return ()


def ajouter(element: all, liste: tuple) -> tuple:
    """Retourne la liste avec l'élément ajouté en tête de liste"""
    return (element, liste)


def est_vide(liste: tuple) -> bool:
    """Retourne True si la liste est vide et False sinon"""
    return liste == ()


def tete(liste: tuple) -> all:
    """Retourne la tête de la liste"""
    assert not est_vide(liste), "Erreur : liste vide"
```

```
    return liste[0]


def queue(liste: tuple) -> tuple:
    """Retourne la queue de la liste"""
    assert not est_vide(liste), "Erreur : liste vide"
    return liste[1]


def longueur(liste: tuple) -> int:
    """Retourne le nombre d'éléments de la liste"""
    if est_vide(liste):
        return 0
    else:
        return 1 + longueur(liste[1])
```

Avec cette représentation une liste est toujours un tuple à deux éléments dont le premier est la tête de la liste (le dernier élément ajouté) et le deuxième est la queue (c'est donc une liste).

$$L = (1, (11, (15, (12, ()))))$$

On remarquera que la fonction `longueur` est codée sans boucle, mais de façon récursive, afin de correspondre au **paradigme fonctionnel**.

Implémentation en POO

Conformément au programme, on se limite à une version **naïve** de la POO. On pourra à titre d'exercice reprendre cette implémentation en respectant les règles plus strictes édictées dans les compléments de cours sur la POO.

On définit ci-dessous une **liste chaînée** : chaque chaînon est constitué de l'élément qui fait partie de la liste et de la référence à l'élément suivant. C'est la classe `Chainon` qui implémente cette structure. L'objet `Liste` est défini à partir de son premier élément (tête) et ses primitives sont définies sous forme de **méthodes**.

```
"""Implémentation du type abstrait liste en POO"""


class Chainon:
    def __init__(self, element=None, suivant=None):
        """element est la valeur du chainon et suivant est le chainon qui suit"""
        self.element = element
        self.suivant = suivant


class Liste:
    def __init__(self):
        """Crée une liste vide"""
        self.head = Chainon()

    def est_vide(self) -> bool:
        """Retourne True si la liste est vide et False sinon"""
```

```
        return self.head.element is None

    def ajouter(self, element):
        """Ajoute element en tête de la liste"""
        self.head = Chainon(element, self.head)

    def tete(self):
        """Retourne la valeur de la tête de la liste"""
        return self.head.element

    def queue(self):
        """Retourne la queue de la liste, c.-à-d. la liste privée de sa tête"""
        new_liste = Liste()
        new_liste.head = self.head.suivant
        return new_liste

    def longueur(self):
        """Retourne la longueur de la liste"""
        long = 0
        chainon = self.head
        while chainon.element is not None:
            chainon = chainon.suivant
            long = long + 1
        return long
```

On peut améliorer l'implémentation en redéfinissant les méthodes spéciales `__len__` (qui permettra de taper `len(L)` au lieu de `L.longueur()`) et `__str__` (qui permettra d'utiliser l'instruction `print(L)`).

```
    def __len__(self):
        return self.longueur()

    def __str__(self):
        rep = ""
        chainon = self.head
        while chainon.element is not None:
            rep = rep + str(chainon.element) + " --> "
            chainon = chainon.suivant
        return rep[:-4]
```

```
>>> L=Liste()
>>> L.ajouter(11)
>>> L.ajouter(12)
>>> L.ajouter(13)
>>> print(L)
13 --> 12 --> 11
```