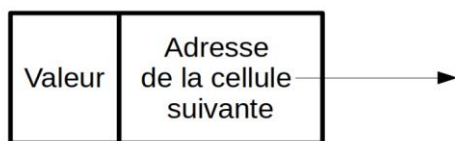


Autres propositions d'implémentation pour une liste chaînée

Que cela soit pour les listes, les piles ou les files, ces structures sont définies par un ensemble d'éléments ordonnés et reliés entre eux. Pour faciliter les implantations de ces structures en utilisant la programmation orientée objet, nous allons introduire une classe Cellule qui permet de définir un objet contenant deux attributs :

- un attribut valeur définissant la valeur contenue dans cette "cellule" (nombre, texte, un autre objet de nature quelconque...) ;
- un attribut suivant définissant l'adresse d'un autre objet Cellule qui "suit" cet objet (qui est l'objet suivant dans l'ordre de la structure).



La structure de base pour les implantations objet : la cellule

"""

Implantation de la classe "Cellule"

Elle sera utilisée pour les listes, les piles et les files.

"""

class Cellule :

def __init__(self, valeur = None, suivant = None) :

"""

Paramètres

valeur : type quelconque

Description : Une valeur stockée dans la cellule

suivant : Un autre objet de type "Cellule"

Description : La cellule qui "suit" cette cellule selon l'ordre défini par la structure.

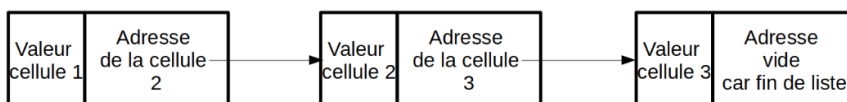
Crée une cellule avec une valeur et l'adresse de la cellule qui la suit.

"""

self.valeur = valeur

self.suivant = suivant

Implémentation utilisant la programmation orientée objet



Une liste chaînée à trois maillons

Une proposition très minimaliste d'implantation peut être :

#On importe la classe "Cellule" que l'on a défini précédemment (et qui doit être dans le même dossier)

```
from Cellule import Cellule
```

```
class Liste :
```

```
    def __init__(self) :
```

```
        """
```

```
        Crée une liste vide.
```

```
        L'attribut "head" est un objet Cellule qui définit la cellule
        en tête de la liste (premier élément de la liste)
```

```
        """
```

```
        self.head = None
```

```
    def estVide(self) :
```

```
        """
```

```
        Renvoie ``True`` si la liste est vide et ``False`` sinon.
```

```
        """
```

```
        return self.head == None
```

```
    def insererEnTete(self, element) :
```

```
        """
```

```
        Paramètres
```

```
        -----
```

```
        element : N'importe quel type
```

```
        Description : L'élément à ajouter en tête de la liste
```

```
        -----
```

```
        Ajoute un élément en tête de liste.
```

```
        """
```

```
        nouvelle_cellule = Cellule(element, self.head)
```

```
        self.head = nouvelle_cellule
```

```
    def tete(self) :
```

```
        """
```

```
        Renvoie la valeur de l'élément en tête de liste.
```

```
        """
```

```
        if not(self.estVide()) :
```

```
            return self.head.valeur
```

```
    def queue(self) :
```

```
        """
```

```
        Renvoie la liste privée de son premier élément (queue de la liste)
```

```
        """
```

```
        subList = None
```

```
        if not(self.estVide()) :
```

```
            subList = Liste()
```

```
            subList.head = self.head.suivant
```

```
        return subList
```

Celle-ci peut être complétée par des méthodes implantant de nouvelles possibilités décrites succinctement dans la ressource « Types abstraits de données - Présentation ». Par exemple :

- renvoyer la longueur de la liste ;
- accéder au élément d'une liste ;
- ajouter un élément à la fin de la liste ;
- rechercher un élément dans une liste en renvoyant "Vrai" si l'élément est présent, "Faux" sinon.

Implémentez donc les méthodes suivantes dans la classe Liste :

```
def __len__(self):
def __getitem__(self, position)
def ajouterFin(self, element)
def __contains__(self, element)
```

Implémentation avec la récursivité et la programmation orientée objet

Implémentez ces deux méthodes récursives dans la classe Cellule :

```
def nombreCellules(self):
    """
    Retourne le nombre de cellules accessibles à partir de cette cellule.
    On compte la cellule + ses voisins.
    """

def element(self, distance):
    """
    Paramètres
    -----
    distance : Nombre entier positif
    Description : Distance de l'élément auquel on veut accéder
    -----
    Retourne l'élément se trouvant à une distance spécifiée de la cellule
    La distance 0 donne donc la valeur de la case courante, 1 celle de son
    voisin immédiat, etc.
    """
```

Ce qui permet de considérablement raccourcir le code des 2 méthodes suivantes dans la classe Liste :

- `def __len__(self) :`
- `def __getitem__(self, position)`

Re-implémentez ces deux méthodes en prenant en compte les deux nouvelles méthodes de cellule (`nombreCellules` et `element`).