

Structure de données : 4 Les listes chaînées

cours

Listes, piles, files : structures linéaires. Dictionnaires, index et clé.	Distinguer des structures par le jeu des méthodes qui les caractérisent. Choisir une structure de données adaptée à la situation à modéliser. Distinguer la recherche d'une valeur dans une liste et dans un dictionnaire.	On distingue les modes FIFO (<i>first in first out</i>) et LIFO (<i>last in first out</i>) des piles et des files.
--	--	--

FIGURE 1 – BO

Le type `Listen` Python est un terme impropre dans les autres langages de programmation : le terme approprié serait plutôt **tableau dynamique**. Cette structure de tableaux permet de stocker des séquences d'éléments mais n'est pas adaptée à toutes les opérations. Ainsi il est facile d'ajouter ou supprimer efficacement des éléments à la fin d'un tableau, autant ce type se prête mal à l'insertion d'un élément ou la suppression d'un élément à une autre position.

Nous allons étudier une structure de données, la **liste chaînée** qui apporte une meilleure solution au problème de *l'insertion* et la *suppression* au début de séquences d'éléments, mais qui sert aussi de brique de base à plusieurs structures dans les prochains chapitres.

1 Structure de liste chaînée

✎ **Définition** : Une **liste chaînée** est une structure de données pour représenter une séquence finie d'éléments. Les éléments sont chaînés entre eux permettant le passage d'un élément à l'élément suivant.

On l'observe dans l'exemple suivant où une liste chaînée contient trois éléments. Chaque élément de la liste est matérialisé par un **emplacement en mémoire** contenant :

- * sa valeur dans la case de gauche
- * l'adresse mémoire de la valeur suivante dans la case de droite.

Rq : le symbole '⊥' nommé "taquet vers le haut" représente ici `None`.

En Python, il est habituel d'utiliser une **classe** décrivant les cellules de la liste : chaque élément de la liste est un **objet** de cette classe. La classe `Cellule` contient deux attributs :

- **valeur** pour la valeur de l'élément

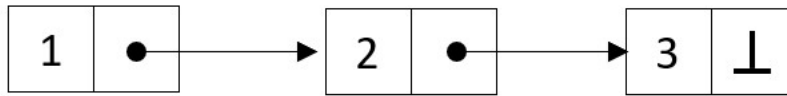


FIGURE 2 – exemple

- **suivante** pour chaîner l'objet suivant.

Voici le code Python présentant l'exemple :

```
class Cellule :
    '''cellule d'une liste s
    '''
    def __init__( self, v, s):
        self.valeur = v
        self.suivante = s

liste_ch = Cellule(1, Cellule(2, Cellule(3, None)))
```

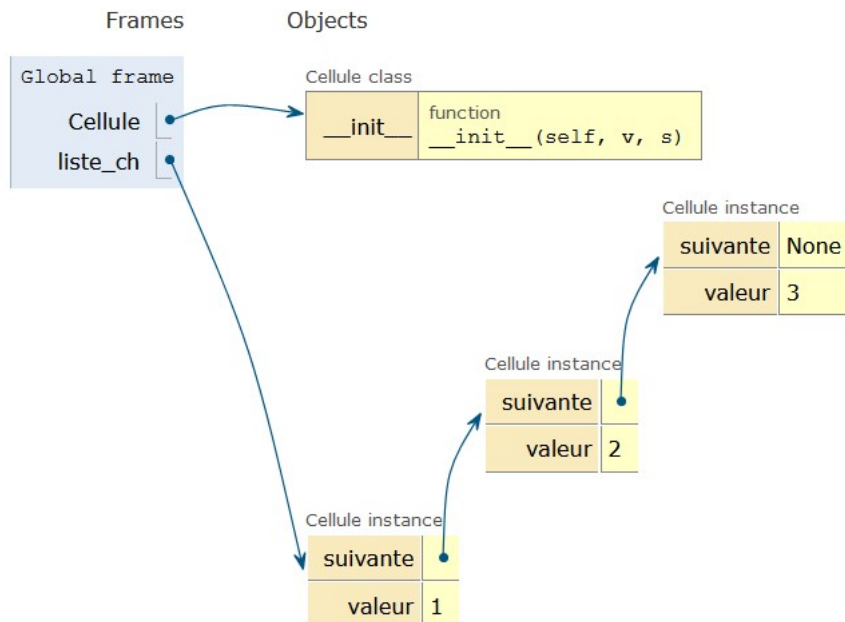


FIGURE 3 – liste_ch

Ainsi la valeur contenue dans la variable `liste_ch` est l'adresse mémoire de l'objet contenant la valeur 1,...

... qui lui-même contient dans son attribut `suivante` l'adresse mémoire de l'objet contenant la valeur 2, ...

... qui enfin contient dans son attribut `suivante` l'adresse mémoire de l'objet contenant la valeur 3. Ce dernier contient la valeur `None` marquant la fin de la liste.

🎓 Comme on le voit, une liste chaînée est soit la valeur `None`, soit un objet de la classe `Cellule` dont l'attribut `suivante` contient une liste. C'est là une **définition récursive** de la notion de liste.

2 Opérations sur les listes

2.1 Longueur d'une liste

Afin de connaître le nombre de cellules, il faut *parcourir* la liste chaînée. On peut réaliser ce parcours par une fonction récursive ou avec une boucle.

2.1.1 Fonction récursive

Dans le cas de base, la liste ne contient aucune cellule ; dans le cas général, la liste contient au moins une cellule et on calcule le reste de la liste avec la longueur de la liste `liste_ch.suivante` que l'on calcule récursivement.

```
def longueur(liste_ch) :  
    '''renvoie la longueur de la liste  
    '''  
    # 'is' est à préférer à '==' car peut avoir été redéfini dans une classe  
    if liste_ch is None :  
        return 0  
    else :  
        return 1 + longueur(liste_ch.suivante)
```

La fonction se termine puisque le nombre de cellules de la liste passée en argument décroît strictement à chaque appel.

2.1.2 Version itérative : la boucle

On utilise une boucle non bornée 'POUR' ; on utilise ainsi un accumulateur (n) et on choisit une condition qui évolue pour qu'on puisse en sortir. On réduit ainsi la chaîne à chaque boucle en plaçant dans la liste chaînée c qui est réduite au fur et à mesure d'une cellule :


```
def longueur(liste_ch) :  
    '''renvoie la longueur de la liste  
    '''  
    n = 0 # accumulateur  
    c = liste_ch  
    while c is not None :  
        n = n + 1  
        c = c.suivante # on affecte à 'c' la liste chaînée à partir de la valeur suivante  
    return n
```

2.2 N-ième élément

On prend comme convention que le premier élément est désigné par $n = 0$ et n doit être positif.

On effectue la version avec une fonction récursive.

Notre liste chaînée ne peut être vide si on souhaite accéder au n-ième élément : ceci un **invariant de boucle**. Si la liste est vide ou $n < 0$, on doit renvoyer une erreur : c'est une **levée d'exception**.

 Méthode **Levée d'exception** : On peut lever explicitement une exception en appelant le mot-clé **raise** suivi d'une instance d'une classe.

```
if not (invariant.....) :
    raise Nom_Erreur("commentaire")
```

Par exemple en utilisant une exception native :

```
def dire_bonjour(nom):
    if not nom:
        raise ValueError("nom vide")
```

Dans notre cas, l'erreur serait celle d'une erreur d'indice `IndexError` .

Le cas de base correspond l'indice $n = 0$: on retourne la valeur de la cellule ; le cas général correspond à un appel récursif de la liste diminuée d'une cellule.

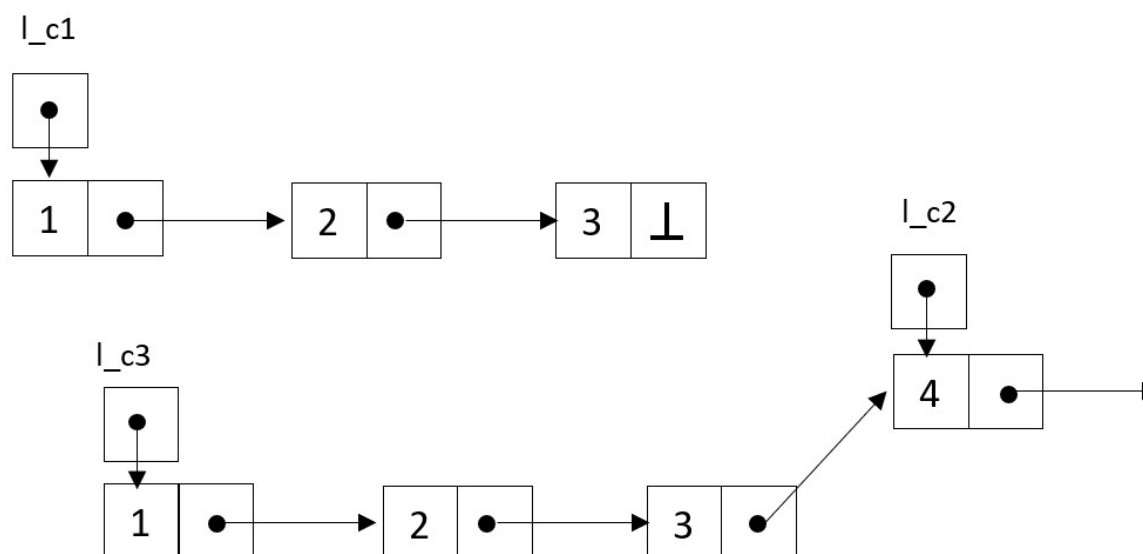
```
def n_ieme_elt(n, liste_ch) :
    ''' renvoie le n-ième élément de la liste chaîne
    '''
    if liste_ch is None or n<0 :
        raise IndexError("Indice invalide")
    if n==0 :
        return liste_ch.valeur
    else:
        return n_ieme_elt(n-1, liste_ch.suivante)
```

Voir TP pour la version itérative

2.3 Concaténation de deux listes

Le but est ici d'associer deux listes pour n'en faire qu'une seule. Ainsi à partir de deux listes passées en arguments, la fonction `concatener` n'en renvoie qu'une seule.

Nous allons procéder récursivement sur la structure de la première liste : si celle-ci est vide, la concaténation est identique à la liste 2. Sinon le premier élément de la concaténation de la liste 1 et le reste est obtenu **récurivement** en concaténant le reste de la liste 1 avec la liste 2 au moyen d'une instance de classe `Cellule` :



Soit l'exemple suivant :

```
l_c1= Cellule(1, Cellule(2, Cellule(3, None)))
l_c2= Cellule(4, Cellule(5, None))
```

Visualiser les étapes entre 25 et 54 pour comprendre l'exécution : [Python Tutor](#)

On observe bien que les listes ne sont pas modifiées : les éléments de la liste 1 sont copiés et ceux de

la liste 2 sont partagés.

```
def concatener(l_c1 , l_c2):
    if l_c1 is None :
        return l_c2
    else :
        return Cellule(l_c1.valeur , concatener(l_c1.suivante , l_c2))
```

2.4 Renverser une liste

Créons la fonction `renverser` qui à partir d'une liste du type 1, 2, 3 renvoie 3, 2, 1. On utilise une fois encore la récursité :

- le cas de base : la liste est vide : on renvoie la liste vide
- le cas récursif : le premier élément doit devenir le dernier ; il faut donc renverser la queue de la liste et concaténer à la fin avec le premier élément.

```
def renverse(liste_ch) :
    if liste_ch is None :
        return liste_ch
    else :
        return concatener(renverse(liste_ch.suivante), Cellule(liste_ch.valeur , None))
```

Python Tutor

3 Modification d'une liste

Il est possible de modifier la valeur d'un attribut a posteriori avec des affectations. Il faudra éviter cette démarche car cela peut causer des événements indésirables. **Voir TP**

Par exemple pour modifier noter liste 1, 2, 3 en 1, 4, 3 :

```
l_c.suivante.valeur 4
```

4 Encapsulation

Afin de réutiliser les chapitres précédents, nous allons créer une nouvelle classe permettant de créer une liste chaînée.

Celle-ci possède :

- un attribut `tete` qui contient une liste chaînée : `tete` car l'attribut contient la tête de la liste si celle-ci n'est pas vide ; on l'initie avec `None` ;
- une méthode `est_vide(self)` qui renvoie un booléen ;
- une méthode `ajoute(self, x)` pour ajouter un élément en tête de liste avec la classe `Cellule` ;

```
class Liste:
    def __init__(self):
        self.tete = None

    def est_vide(self):
```

```

    return self.tete is None

def ajoute(self, x):
    self.tete = Cellule(x, self.tete)

```

On peut aussi reformuler les fonctions *longueur* (que l'on nommera **len** comme le mot réservé dans Python), *nieme_element* (**__getitem__**), *concatener* (**__add__**) et *renverse* (**reverse**).

```

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

def __getitem__(self, n)::
    return nieme_element(n, self.tete)

def __add__(self, l_c):
    '''attention on renvoie une nouvelle liste
    '''
    n_l_c = Liste()
    n_l_c.tete = concatener(self.tete, l_c.tete)
    return n_l_c

def reverse(self) :
    self.tete = renverser(self.tete)

```

Sources :

* NSI Terminale Nathan les vrais exos S Pasquet M Leopoldoff