

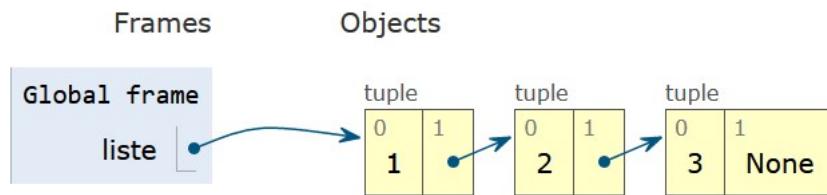
## 5.3

### Implémentation en Python des listes chainées

NSI TERMINALE - JB DUTHOIT

#### 5.3.1 Implémenter avec les tableaux ou tuples

On peut utiliser un tuple, et dans ce cas écrire `(1, (2, (3, None)))`.



On peut aussi encore un tableau à deux éléments, et dans ce cas écrire `[1, [2, [3, None]]]`.

Une liste chaînée peut être encore représentée par deux tableaux, l'un (contenu) contenant des valeurs et l'autre (suivant) contenant des indices. Le chaînage sera effectué de la façon suivante : l'élément suivant contenu[k] aura suivant[k] comme indice dans le tableau contenu.

	contenu	suivant
0	'2'	1
1	'3'	-1
2	'1'	0

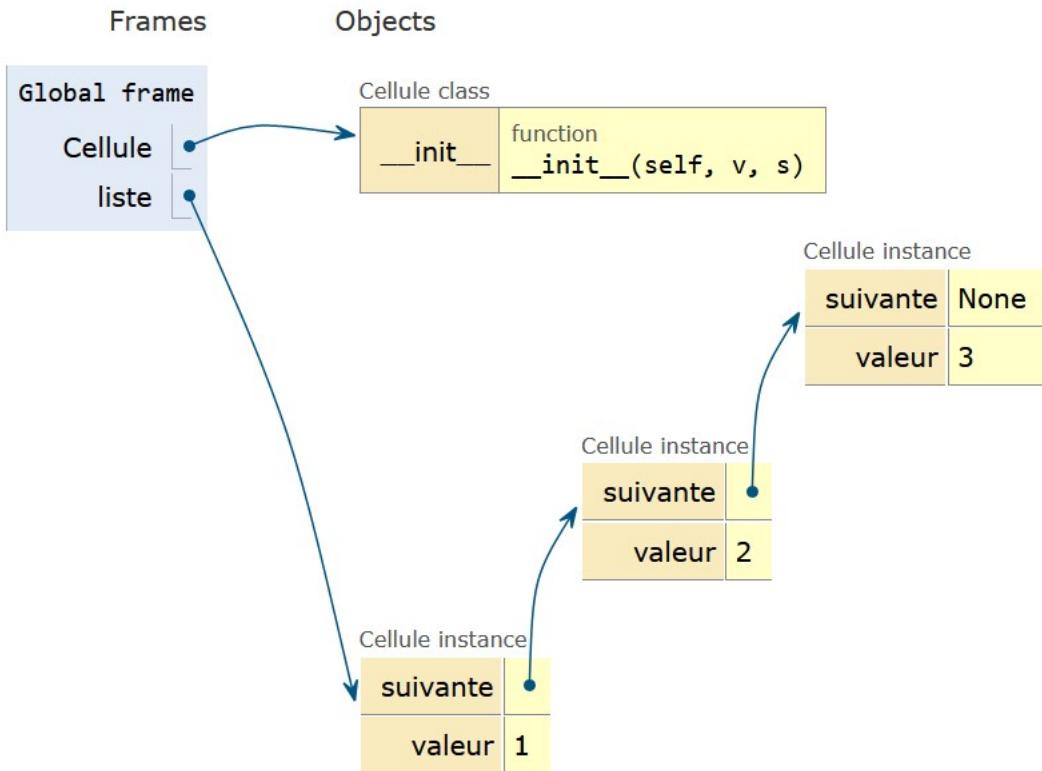
#### 5.3.2 Implémenter une liste chaînée avec les classes

Une façon d'utiliser des listes chaînées avec Python, est d'utiliser une classe :

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

Ainsi la liste 1,2,3 est possible avec l'instruction :

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



## Remarque

| None remplace  $\perp$ .

## Définition

Une liste chaînée est une structure de données pour représenter une séquence finie d'éléments. Chaque élément est contenu dans une cellule, qui fournit par ailleurs un moyen d'accéder à la cellule suivante. Les opérations sur les listes chaînées se programment sous la forme de parcours qui suivent ces liaisons, en utilisant une fonction récursive ou une boucle

### Exercice 5.110

Pour cet exercice, on considère la classe suivante, qui permet de fabriquer des listes chaînées :

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

On écrira `lst = None` lorsque `lst` est une liste vide.

Écrire une fonction `listeN(n)` qui reçoit un argument entier `n` et qui renvoie la liste chaînée des entiers 1,2,3,...,n dans cet ordre. Si `n = 0`, la liste renvoyée est vide.

### Exercice 5.111

Pour cet exercice, on considère la classe suivante, qui permet de fabriquer des listes chaînées :

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

On écrira `lst = None` lorsque `lst` est une liste vide.

Écrire une fonction `longueur(lst)` qui reçoit pour argument une liste chaînée `lst` et qui retourne la longueur de la liste chaînée.

1. L'écrire avec une boucle While
2. L'écrire comme fonction récursive

### Exercice 5.112

Pour cet exercice, on considère la classe suivante, qui permet de fabriquer des listes chaînées :

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

On écrira `lst = None` lorsque `lst` est une liste vide.

Écrire une fonction `affiche_liste(lst)` qui reçoit en argument une liste chaînée `lst` et qui affiche, en utilisant `print`, tous les éléments de la liste `lst`, séparés par des espaces, suivies d'un retour chariot.

1. L'écrire avec une boucle While
2. L'écrire comme fonction récursive

### Exercice 5.113

Pour cet exercice, on considère la classe suivante, qui permet de fabriquer des listes chaînées :

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

On écrira `lst = None` lorsque `lst` est une liste vide.

Écrire une fonction `n_ième_element(lst, n)` de paramètres `n` et `lst` une liste chaînée, et qui renvoie le `n`-ième élément de la liste

1. L'écrire de façon récursive
2. L'écrire avec une boucle While

### Exercice 5.114

Pour cet exercice, on considère la classe suivante, qui permet de fabriquer des listes chaînées :

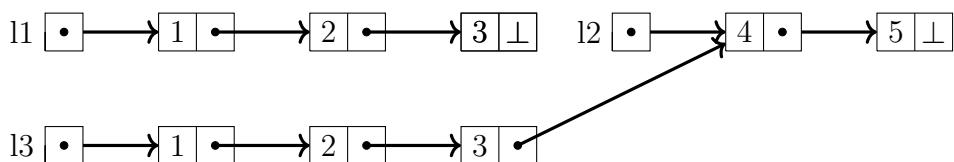
```
class Cellule:
    '''Une cellule d'une liste chaînée'''
    def __init__(self, v, s):
        self.valeur = v
        self.suivante = s
```

On écrira `lst = None` lorsque `lst` est une liste vide.

On souhaite maintenant mettre bout à bout deux listes. On appelle cela une **concaténation**. Écrire une fonction concaténer qui reçoit deux listes en arguments et renvoie une troisième liste contenant la concaténation.

```
def concatener(l1,l2):
    ''' concatène l1 et l2 ,
    sous la forme d'une nouvelle liste
    si l1 est vide, alors on renvoie l2
    si l2 est vide, alors on renvoie l1
    sinon la concaténation est obtenue en concaténant la tête de l1 et la concaténation du reste de
    l1=Cellule(1,Cellule(2,Ceellule(3,None)))
    l2= Cellule(4,Cellule(5,None))
    l3 = concatener(l1,l2)
    '''


```



### Remarque

On voit que les cellules de `l1` ont été dupliquées tandis que les cellules de `l2` partagées. Les cellules `l2` permettent de constituer la liste `l2` et la fin de la liste `l3`. Une alternative consistrait à dupliquer également les cellules de `l2`, mais ceci n'est pas forcément nécessaire.

#### Exercice 5.115

Pour cet exercice, on considère la classe suivante, qui permet de fabriquer des listes chaînées :

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

On écrira `lst = None` lorsque `lst` est une liste vide.

Écrire une fonction `derniere_cellule(lst)` qui renvoie la dernière cellule de la liste `lst`.

#### Exercice 5.116

Pour cet exercice, on considère la classe suivante, qui permet de fabriquer des listes chaînées :

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

On écrira `lst = None` lorsque `lst` est une liste vide.

Écrire une fonction `liste_de_tableau(t)` qui renvoie une liste qui contient les éléments du tableau `t`, dans le même ordre. On pourra utiliser une boucle `for`.

### 5.3.3 Opérations avec les listes chaînées

#### Longueur de la liste

```
def longueur(l):
    if lst is None:
        return 0
    else:
        return 1 + longueur(l.suivante)
```

Complexité : la complexité du calcul de la longueur est directement proportionnelle à la longueur elle-même, puisqu'on réalise un nombre constant d'opérations pour chaque cellule de la liste.

Ainsi, pour une liste l de mille cellules, longueur(l) va effectuer mille tests, mille appels récursifs et mille additions.

#### Renvoyer le n-ème élément d'une liste

```
def nieme_element(n, l):
    if l is None:
        raise IndexError("indice invalide")
    if n == 0:
        return l.valeur
    else:
        return nieme_element(n-1,l.suivante)
```

#### Concaténer deux listes

```
def concatener(l1, l2):
    """concatène les listes l1 et l2,
    sous la forme d'une nouvelle liste"""
    if l1 is None:
        return l2
    else:
        return Cellule(l1.valeur, concatener(l1.suivante,l2))
```