Listes chaînées

1 Les tableaux

La structure de tableau permet de stocker des séquences d'éléments de manière contigus et ordonnés en mémoire.

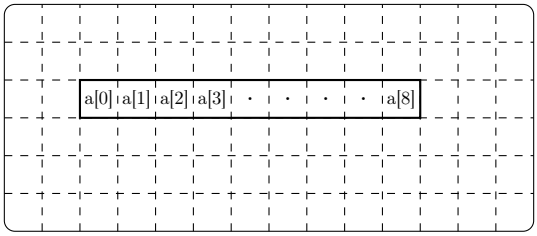
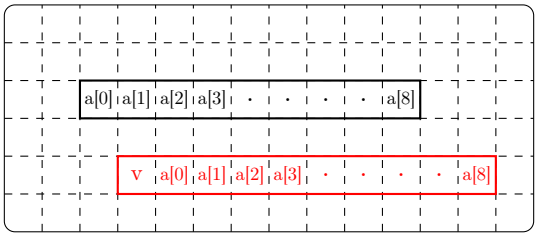


Tableau `a` de 9 éléments stocké en mémoire.

Accéder à un élément d'un tableau ce fait alors à coût constant, il suffit de connaître l'adresse du premier élément et la taille d'un élément. Par contre, le tableau se prête mal à l'ajout d'un élément.

Si l'on reprend notre tableau précédent et qu'on aimerais insérer la valeur `v` à la première position, la case adjacente à `a[0]` étant très probablement occupée par une autre donnée, il faut réserver de la place ailleurs en mémoire, recopier tout le tableau avec la valeur `v` en première position.

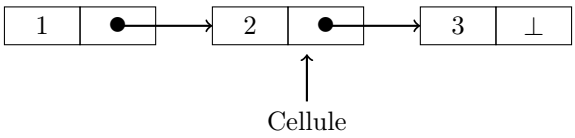


Au total, on a réalisé un nombres d'opérations proportionnel à la taille du tableau.

Nous allons étudier dans ce chapitre une structure de données, la *liste chaînée*, qui apporte une meilleur slution au problème de l'insertion au début d'une séquence mais qui nous servira également de brique de base à plusieurs autres structures.

2 Structure de liste chaînée

Une *liste chaînée* représente une liste finie de valeurs où les éléments sont chaînés eutre eux. Chaque élément est stocké dans un bloc alloué quelque part en mémoire appelé maillon ou *cellule*, et est accompagné d'une deuxième information : l'adresse mémoire de la cellule suivante.



Liste chaînée contenant les éléments 1, 2 et 3.

Une façon de représenter une liste chaînée en Python consiste à utiliser une classe `Cellule`.

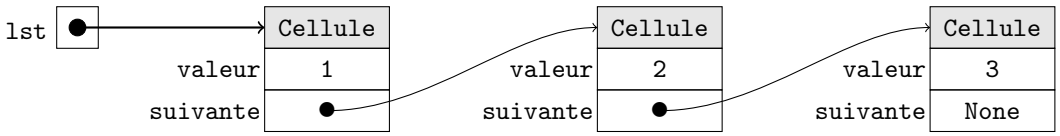
```
class Cellule:
    '''Une cellule d une liste chainee'''
    def __init__(self, v, s):
        self.valeur = v
        self.suivante = s
```

Pour la dernière cellule, on donne la valeur `None` à l'attribut `suivante`.

Pour construire une liste il suffit d'appliquer le constructeur de la classe `Cellule` autant de fois qu'il y a d'éléments dans la liste. Ainsi, l'instruction

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

construit la liste 1, 2, 3 donnée en exemple et la stocke dans une variable `lst`. Plus précisément, on a créé trois objets de la classe `Cellule`, que l'on peut visualiser comme suit.



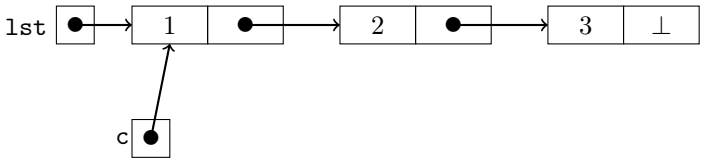
3 Opérations sur les listes chaînées

3.1 Longueur d'une liste

Pour calculer la longueur d'une liste, on parcours la liste de la première à la dernère cellule en suivant les liens qui relient ces cellules entre elles.

On commence par se donner deux variables : une variable `c` contenant la cellule courante du parcours et une variable `l` contenant la longeur du parcours déjà réalisé.

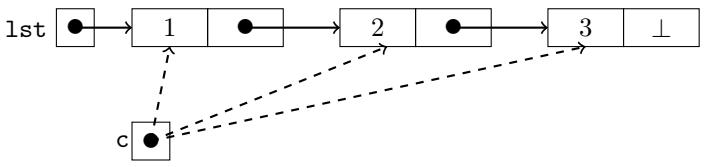
```
def longueur(lst):
    '''renvoie la longeur de la liste lst'''
    c = lst
    l = 0
```



La variable `c` va parcourir à l'aide d'une boucle `while` la liste, en incrémentant `l` à chaque itération.

```
    while c is not None:
        l += 1
        c = c.suivante
```

On s'arrête lorsque `c` est `None`, ce qui correspond bien à la fin de la liste.



Il suffit alors de retourner `l`.

```
def longueur(lst):
    '''renvoie la longeur de la liste lst'''
    c = lst
    l = 0
    while c is not None:
        l += 1
        c = c.suivante
    return l
```

Une autre solution consiste à parcourir la liste de manière récursive. En effet, si la liste est vide sa longueur est 0.

```
    if lst is None:
        return 0
```

Sinon, il faut renvoyer 1 (pour la case en cours de parcours) plus la longeur du reste de la liste.

```
    else:
        return 1 + longueur(lst.suivant)
```

```
def longueur(lst):
    '''renvoie la longeur de la liste lst'''
    if lst is None:
        return 0
    else:
        return 1 + longueur(lst.suivant)
```

Complexité : La complexité du calcul de la longueur est directement proportionnelle à la longueur elle même. Ainsi pour une liste `lst` de mille cellules, `longueur(lst)` va effectuer mille tests, mille additions et deux milles affectations dans sa version itérative.

3.2 N-ième élément d'une liste

Comme pour les tableaux, nous allons prendre comme convetion que le premier élément à pour indice 0. On cherche à écrire une fonction de la forme suivante.

```
def nieme_element(n, lst):
    '''renvoie le n-ieme element de la liste lst
    les elements sont numerotes a partir de 0'''
    return lst.valeur
```

Comme pour la fonction `longueur`, nous avons le choix d'écrire la fonction `nieme_element` comme une fonction récursive ou itérative. Nous faisons ici le choix d'une fonction récursive ; l'exercice 3 propose d'écrire cette fonction avec une boucle.

La condition d'arrêt s'obtient en considérant le cas où `n = 0`, le premier élément de la liste est alors renvoyé.

```
    if n == 0:
        return lst.valeur
```

Sinon, il faut continuer la recherche dans le reste de la liste. On passe à l'élément suivant en diminuant `n` de 1.

```
    else:
        return nieme_element(n-1,lst.suivante)
```

Reste encore à gérer le cas où la liste est vide, dans ce cas l'indice donné est invalide et on lève une exception `IndexError`.

```
    if lst is None:
        raise IndexError("indice invalide")
```

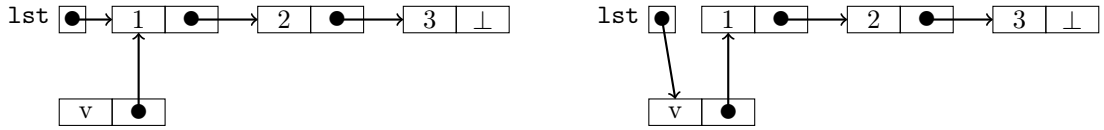
```
def nieme_element(n, lst):
    '''renvoie le n-ieme element de la liste lst
    les elements sont numerotes a partir de 0'''
    if lst is None:
        raise IndexError("indice invalide")
    if n == 0:
        return lst.valeur
    else:
        return nieme_element(n-1,lst.suivante)
```

4 Modifier une liste chaînée

Nous allons programmer une fonction qui permet d'ajouter un élément en tête de liste.

```
def ajouter(v ,lst):
    '''ajoute la valeur v en tete de la liste lst'''
```

On aimerait ajouter la valeur `v` en tête de liste. Pour cela on créé une nouvelle cellule qui a pour valeur `v` et qui a pour cellule suivante la cellule pointé par `lst`. Cette nouvelle cellule devient ensuite notre nouvelle tête.




```
    lst = Cellule(v, lst)
```

```
def ajouter(v ,lst):
    '''ajoute la valeur v en tete de la liste lst'''
    lst = Cellule(v, lst)
```

L'ajout d'un élément en tête d'une liste chaînée se fait à coût constant.

5 Encapsulation dans un objet

Pour terminer ce chapitre, nous allons définir une classe `Liste`. Cette classe possède un unique attribut `tete` qui correspond à la tête de la liste.

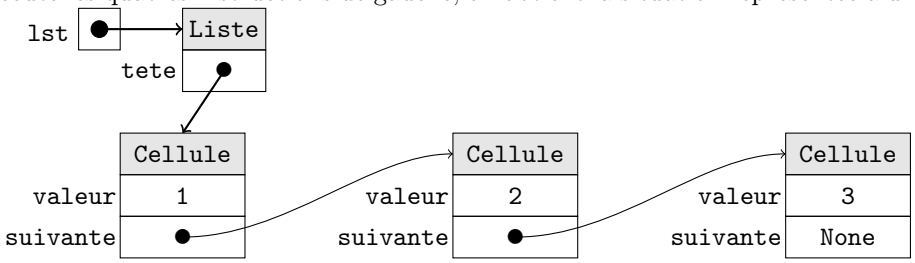
```
class Liste:
    '''une liste chainee'''
    def __init__(self):
        self.tete = None
```

Pour ajouter des éléments en tête de liste, on peut utiliser la fonction `ajouter` que l'on place dans une méthode. On peut garder le même nom pour la méthode, ou le modifier si cela peut prêter à confusion.

```
    def ajoute(self, v):
        ajouter(v, self.tete)
```

Si par exemple on exécute les quatres instructions de gauche, on obtient la situation représentée à droite :

```
lst = Liste()
lst.ajoute(3)
lst.ajoute(2)
lst.ajoute(1)
```



Les fonctions `longueur` et `nieme_element` vont nous permettre de définir les méthodes `__len__` et `__getitem__`.

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

    def __getitem__(self, i):
        return nieme_element(i, self.tete)
```

On pourra alors écrire comme pour un tableau `lst[i]` et `len(lst)`.

Beaucoup d'autres méthodes sont encore possibles, comme celles proposées dans les exercices. Voici une première class `List`.

```
1  class Liste:
2      '''une liste chainee'''
3      def __init__(self):
4          self.tete = None
5
6      def ajoute(self, v):
7          ajouter(v, self.tete)
8
9      def __len__(self):
10         return longueur(self.tete)
11
12     def __getitem__(self, i):
13         return nieme_element(i, self.tete)
14
15     def est_vide(self):
16         return self.tete is None
```

Exercice : Ajouter une méthode `__setitem__(self, i)`.