

# Chapitre 4 - Les listes

## Objectifs :

- ▷ 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.

## 1 Introduction

### Historique

Le langage de programmation *Lisp* (inventé par *John McCarthy* en 1958) a été un des premiers langages de programmation à introduire cette notion de liste (*Lisp* signifie *list processing*).

*Lisp* qui fait partie de la plus ancienne famille de langages de programmation à la fois impératifs et fonctionnels, est devenu dans les années 1970-80 le langage de choix pour la recherche en intelligence artificielle. Ils sont aujourd'hui utilisés dans de nombreux domaines, de la programmation Web à la finance.

*Python* abuse du terme liste qu'il utilise pour ce qui sont des **tableaux dynamiques** munis de méthodes d'accès typiques des listes. Nous nous intéressons ici à ce que les informaticiens appellent vraiment des **listes**.

## 2 Définition

### A retenir !

**Une liste est une structure de données permettant de regrouper des données.**

Elle est composée de 2 parties :

- ▷ sa **tête** notée *car* (qui signifie *content of address register*), qui correspond au premier élément ajouté à la liste.
- ▷ sa **queue** notée *cdr* (qui signifie *content of decrement register*), qui correspond au reste de la liste.

## 3 Opération sur les listes

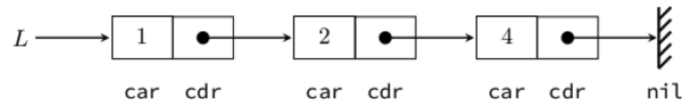
Voici les opérations qui peuvent être effectuées sur une liste :

- ▷ créer une liste vide (souvent notée `nil`)
- ▷ tester si une liste est vide
- ▷ ajouter un élément en tête de liste
- ▷ supprimer la tête d'une liste et renvoyer cette tête
- ▷ compter le nombre d'éléments présents dans une liste
- ▷ un constructeur, historiquement appelé `cons`, qui permet d'obtenir une nouvelle liste à partir d'une liste et d'un élément (`L1 = cons(x,L)`)

Il est possible "d'enchaîner" les `cons` et d'obtenir ce genre de structure : `cons(x, cons(y, cons(z,L)))`

## 4 Représentation graphique

Voici la représentation schématique d'une liste comportant, dans cet ordre, les trois éléments 1, 2 et 4 :



La liste est constituée de trois **cellules**, représentées ici par des rectangles.

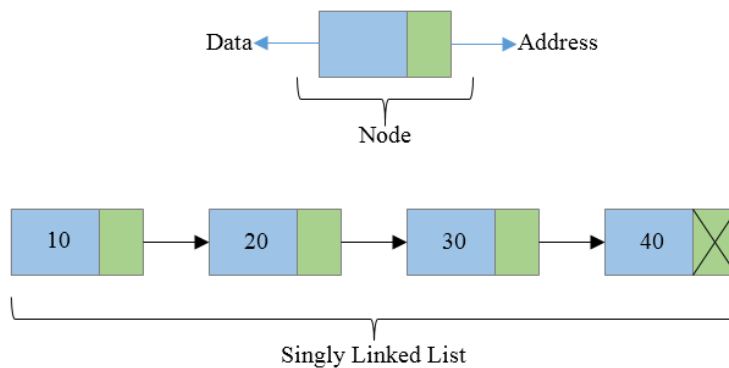
Chaque cellule possède :

- une première partie nommée **car** , qui contient l'entier correspondant
- une deuxième partie, nommée **cdr** , qui contient un lien vers la cellule suivante.

On a donc affaire à une chaîne de cellules qu'on appelle plus couramment une **liste chaînée**.

Le champ **cdr** de la dernière cellule renvoie vers une liste vide, conventionnellement représentée par un hachurage. On notera que le premier champ d'une cellule contient ici un entier (car on considère une liste d'entiers) alors que le deuxième est une liste, c'est-à-dire un lien vers une autre cellule.

Nous venons de définir les contours de la structure de liste. Celle-ci est un concept ("une vue de l'esprit") de ce qu'est une liste. On dit que c'est un un **type abstrait de données**.

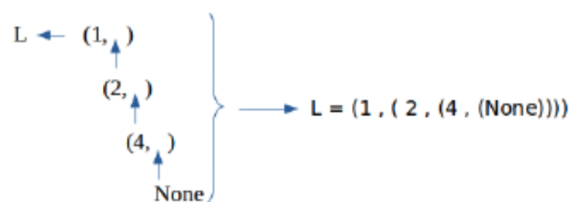


## 5 Implémentation

Pour implémenter cette structure de liste chaînée, peu importe le langage utilisé, il faut que l'implémentation permette de retrouver les fonctions qui ont été définies pour le type abstrait.

### 5.1 Implémentation n°1

En utilisant des *tuples* pour implémenter la structure de liste.



Ce qui donne les fonctions *Python* suivantes :

```
1 def vide():
2     return None
3
4 def cons(x,L):
5     return (x,L)
6
7 def ajouteEnTete(L,x):
8     return cons(x,L)
9
10 def estVide(L):
11     return L is None
12
13 def compte(L):
14     if estVide(L):
15         return 0
16     else:
17         return 1 + compte(L[1])
```

### Exercice 1 : Implémentation en tuples

1. Vérifier le bon fonctionnement de cette implémentation en exécutant les instructions ci-dessous et en notant l'affichage obtenu.

```
▷ nil = vide()
▷ print(estVide(nil))
▷ L = cons(5,cons(4,cons(3,cons(2,cons(1,cons(0,nil)))))
▷ print(L)
▷ print(estVide(L))
▷ print(compte(L))
▷ L = ajouteEnTete(L,6)
▷ print(compte(L))
▷ print(L)
```

2. Ecrire une fonction `supprEnTete(L)` qui supprime l'en-tête de la liste passée en paramètre :

3. Tester si la fonction que vous venez d'écrire fonctionne en essayant les instructions ci-dessous et en notant l'affichage obtenu.

```
▷ L = supprEnTete(L)
▷ print(L)
▷ print(compte(L))
```

4. Qu'en concluez-vous ?

## 5.2 Implémentation n°2

Maintenant, on va implémenter les listes chaînées en utilisant la *programmation orientée objet* à l'aide de deux classes :

▷ la classe `Cellule` qui crée un objet (instance) cellule avec les attributs `car` et `cdr`

```
1 class Cellule:
2     def __init__(self, tete, queue):
3         self.car = tete
4         self.cdr = queue
```

▷ la classe `Liste` dont les instances (objets) seront de type `Cellule`, avec quelques méthodes :

```
1 class Liste:
2     def __init__(self, c):
3         self.cellule = c
4
5     def estVide(self):
6         return self.cellule is None
7
8     def car(self):
9         assert not(self.cellule is None), 'Liste vide'
10        return self.cellule.car
11
12    def cdr(self):
13        assert not(self.cellule is None), 'Liste vide'
14        return self.cellule.cdr
```

▷ La fonction `cons` qui permet de construire la liste :

```
1 def cons_tete(queue):
2     return Liste(Cellule(tete, queue))
```

Ainsi pour créer une la liste `L = [5, 4, 3, 2, 1, 0]`, il suffit d'exécuter les instructions :

```
1 nil = Liste(None)
2 L = cons(5, cons(4, cons(3, cons(2, cons(1, cons(0, nil))))))
```

### Exercice 2 : Implémentation en classes

1. Tester les instructions suivantes et noter les résultats obtenus :

```
1 print(L.estVide())
2 print(L.car())
3 print(L.cdr().car())
4 print(L.cdr().cdr().car())
```

2. Écrire l'instruction qui permet d'afficher le dernier élément de la liste.