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

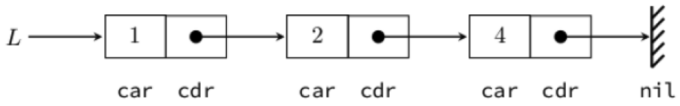
Voici les opérations qui peuvent être effectuées sur une liste :

- ▷ 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.

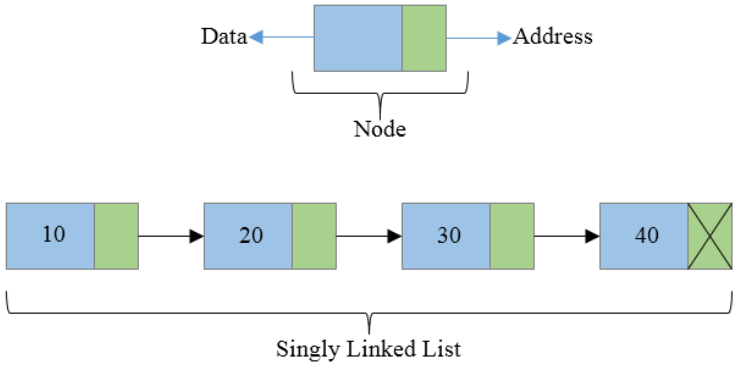
Chaque cellule possède :

— 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**.

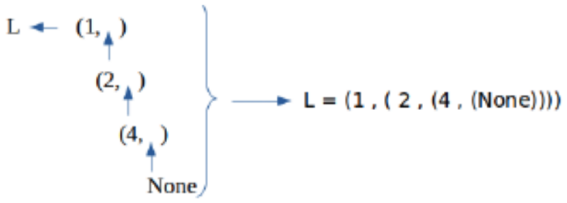


5 Implémentation

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 :

```
def vide():
    return None

def cons(x,L):
    return (x,L)

def ajouteEnTete(L,x):
    return cons(x,L)

def estVide(L):
    return L is None

def compte(L):
    if estVide(L):
        return 0
    else:
        return 1 + compte(L[1])
```

A faire 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`

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

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

```
class Liste:
    def __init__(self,c):
        self.cellule = c

    def estVide(self):
        return self.cellule is None

    def car(self):
        assert not(self.cellule is None), 'Liste vide'
        return self.cellule.car

    def cdr(self):
        assert not(self.cellule is None), 'Liste vide'
        return self.cellule.cdr
```

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

```
def cons(tete,queue):
    return Liste(Cellule(tete,queue))
```

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

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

A faire 2 : 1. Tester les instructions suivantes et noter les résultats obtenus :

```
print(L.estVide())
print(L.car())
print(L.cdr().car())
print(L.cdr().cdr().car())
```

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

A faire 3 : *Fonctions supplémentaires*

1. Ajouter les deux fonctions suivantes :

```
def longueurListe(L):
    n = 0
    while not(L.estVide()):
        n += 1
        L = L.cdr()
    return n
```

```
def listeElements(L):
    t = []
    while not(L.estVide()):
        t.append(L.car())
        L = L.cdr()
    return t
```

2. Que font-elles ?

3. Que produit l'instruction ci-dessous ?

```
L = cons(6,L)
```

4. Écrire une fonction `ajouteEnTete` qui prend en paramètre une liste et un nombre et qui renvoie une liste dans laquelle le nombre a été ajouté en entête.

5. Que produisent les instructions suivantes ?

```
x = L.car()
L = cons(L.cdr().car(),L.cdr().cdr())
```

6. Écrire une fonction `supprEnTete` qui prend en paramètre une liste et qui retourne son entête et la liste amputée de l'entête.

6 Exercices

Dans les exercices ci-après, on suppose que l'on utilise l'implémentation de liste chaînée suivante :

```
class Cellule:
    '''une cellule d'une liste chainee'''

    def __init__(self,v,s):
        self.valeur = v
        self.suivante = s
```

Exercice 1 : Ecrire une fonction `listeN(n)` qui reçoit en argument un entier `n`, supposé positif ou nul, et renvoie la liste des entiers 1,2,..., n dans cet ordre. Si n = 0, la liste renvoyée est vide.

Exercice 2 : Ecrire une fonction `affiche(n)` qui affiche tous les éléments de la liste `lst`, séparés par des espaces, suivis d'un retour chariot. L'écrire en récursif puis en itératif avec une boucle `while`.

Exercice 3 : Ecrire une fonction `occurences(x,lst)` qui renvoie le nombre d'occurences de la valeur `x` dans la liste `lst`. L'écrire en récursif puis en itératif avec une boucle `while`.

Exercice 4 : Ecrire une fonction `trouve(x,lst)` qui renvoie le rang de la première occurence de `x` dans la liste `lst`. L'écrire en récursif puis en itératif avec une boucle `while`.

Exercice 5 : Ecrire une fonction `identiques(l1,l2)` qui renvoie un booléen indiquant si les listes `l1` et `l2` sont identiques, c'est-à-dire contiennent exactement les mêmes éléments, dans le même ordre.

Exercice 6 : Ecrire une fonction récursive `inserer(x,lst)` qui prend en argument un entier `x` et une liste d'entiers `lst`, supposée triée par ordre croissant, et qui renvoie une nouvelle liste dans laquelle `x` a été inséré à sa place. Ainsi, insérer la valeur `3` dans la liste `[1,2,5,8]` renvoie la liste `[1,2,3,5,8]`.

Exercice 7 : En se servant de l'exercice précédent, écrire une fonction récursive `tri_par_insertion(lst)` qui prend en argument une liste d'entiers `lst` et renvoie une nouvelle liste, contenant les mêmes éléments et triée par ordre croissant.

Exercice 8 : Ecrire une fonction `liste_de_tableau(t)` qui renvoie une liste qui contient les éléments du tableau t, dans le même ordre. On suggère de l'écrire avec une boucle `for`.

Exercice 9 : Ecrire une fonction `derniere_cellule(lst)` qui renvoie la dernière cellule de la liste `lst`. On suppose la liste `lst` non vide.