Les listes

Une liste est une structure de données permettant de regrouper des données. Elle est donc composée de données, et offre un moyen de passer à la donnée suivante. Les listes sont très souvent implémentées sous forme d'une chaîne de valeurs, chacune pointant vers la suivante (on parle alors de liste chaînée).

1 - Présentation

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").

Comme comme nous l'avons déjà vu, la structure list de Python réalise en fait l'implémentation du type abstrait de données «tableau dynamique» et doit ici être laissée de côté, malgré l'utilisation du même vocabulaire. La liste permet de stocker des données et d'y accéder directement. C'est un type abstrait de données:

- linéaire : les données sont stockées dans une structure unidimensionnelle;
- indexé : chaque donnée est associée à une valeur;
- ordonné : les données sont présentées les unes après les autres.

2 - Le type abstrait de données

Une liste est une collection finie de données. On appelle «tête» le premier élément de la liste et «queue» la liste privée de son premier élément. Il est seulement possible d'ajouter et de lire une donnée en tête de la liste .

Une liste L est composée de 2 parties:

- Sa tête (souvent noté *car (Content of Address Register)*), qui correspond au dernier élément ajouté à la liste. Peut être retenu par : Contenu Autre que le Reste
- Sa queue (souvent noté *cdr (Content of Decrement Register)*) qui correspond au reste de la liste. Peut être retenu par : Contenu Du Reste

5 primitives permettent de définir le type abstrait de données:

- listeCree(), qui crée une liste vide;
- listeAjout(liste, element), qui ajoute un élément en tête de liste; ces 2 première primitives peuvent parfois se regrouper en une seule;
- listeTete(liste), qui renvoie la valeur de l'élément en tête de liste;
- listeQueue(liste), qui renvoie la liste privée de son premier élément ;
- listeEstVide(liste), qui renvoie vrai si la liste est vide, faux sinon.

On peut constater que le type abstrait de données est non mutable. L'intérêt de ce choix a déjà été développé dans la ressource «Types mutables et problèmes associés» dédiée au programme de première.

3 - Remarques

De nombreux autres algorithmes sur les listes peuvent être ajoutés en fonction des besoins

et peuvent faire l'objet d'exercices en fonction des besoins des activités proposées:

- renvoyer une liste dont l'ordre des éléments a été inversé;
- renvoyer la longueur, c'est-à-dire le nombre des éléments de la liste;
- accéder au N^{ème} élément d'une liste;
- rechercher un élément dans une liste en renvoyant au choix:
  - "Vrai" si l'élément s'y trouve, «Faux» sinon;
  - la position de la première occurrence de l'élément recherché;
  - la liste des positions de toutes ses occurrences ;
- renvoyer une nouvelle liste dont le N^{ème} élément a été supprimé;
- renvoyer une nouvelle liste correspondant à la liste d'origine à laquelle un élément a été ajouté à la fin;
- toute primitive pouvant correspondre à un besoin spécifique lié notamment à un projet ou à un exercice

D'autre part, des types abstraits de données plus élaborés peuvent être construits à partir du type initial, que l'on nomme alors «liste chaînée»:

- listes doublement chaînées permettant de se déplacer vers la queue, mais aussi vers la tête;
- listes circulaires dont le premier élément suit le dernier;
- listes circulaires doublement chainées combinant les deux propriétés précédentes

La fonction *cons* 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)))

Exemples :

Voici une série d'instructions (les instructions ci-dessous s'enchaînent):

- L=vide() => on a créé une liste vide
- estVide(L) => renvoie vrai
- ajoutEnTete(3,L) => La liste L contient maintenant l'élément 3
- estVide(L) => renvoie faux
- ajoutEnTete(5,L) => la tête de la liste L correspond à 5, la queue contient l'élément 3
- ajoutEnTete(8,L) => la tête de la liste L correspond à 8, la queue contient les éléments 3 et 5
- t = supprEnTete(L) => la variable t vaut 8, la tête de L correspond à 5 et la queue contient l'élément 3
- L1 = vide()
- L2 = cons(8, cons(5, cons(3, L1))) => La tête de L2 correspond à 8 et la queue contient les éléments 3 et 5

Une 1° implémentation de la structure liste:

```
def vide():
        return None

def cons(x, L):
```

```
        return x, L

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

    def supprime_en_tete(L):
        return L[0], L[1]

    def est_vide(L):
        return L is None

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

Exo:

Vérifier le bon fonctionnement de cette implémentation en exécutant ces instructions:

- L1 = vide()
- afficher est_vide(L1)
- L2 = cons(3, cons(2, cons(1, cons(0, L1))))
- afficher est_vide(L2)
- afficher compte(L2)
- L2 = ajouter_en_tete(L2, 4)
- afficher compte(L2)
- t, L2 = supprimer_en_tete(L2)
- afficher t
- afficher compte(L2)
- t, L2 = supprimer_en_tete
- afficher t
- afficher compte(L2)

Une 2° implémentation de la structure liste:

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

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

        def est_vide(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

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

Ainsi, pour créer une liste il suffit de faire:

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

Exos:

- Testez les instructions suivantes:
  - print(L.est_vide())
  - print(L.car())
  - print(L.cdr().car())
  - print(L.cdr().cdr().car())
- Ecrire l'instruction qui permet d'afficher le dernier élément de la liste

- Rajoutez les 2 fonctions suivantes:

```
def longueur_liste(self, L):
    n = 0
    while not(L.est_vide()):
        n += 1
        L = L.cdr()
    return n

def liste_elements(self, L):
    t = []
    while not(L.est_vide()):
        t.append(L.car())
        L = L.cdr()
    return t
```

- Que font-elles ?

- Que produit l'instruction: L = nil.cons(6, L) ?

- Ecrire une fonction ajouter_en_tete qui prend en paramètre une liste et un nombre et qui renvoie une liste dans laquelle le nombre à été ajouté en entête

- Que produisent les instructions:
  - x = L.car()

- L = L.cons(L.cdr().car(), L.cdr().cdr())

- Ecrire une fonction supprimer_en_tete qui prend en paramètre une liste et qui retourne son entête et la liste amputée de l'entête

Lire les fichiers:

- 2.0 – Cellule.py
- 2.1.1 – liste fonctions.py
- 2.1.2 – liste objet.py
- 2.1.3 – liste objet recursive.py

Pour aller plus loin...

Sources :

- Eduscol
- https://pixees.fr/informatiquelycee/n_site/nsi_term_structDo_liste.html
- https://www.da-code.fr/car-cdr-cadr/
- Prépabac Tle générale spécialité NSI, édition Hatier
- https://isn-icn-ljm.pagesperso-orange.fr/NSI-TLE/co/section_chapitre2.html