La structure de données liste

Terminale NSI

2020/01/01

La structure de données liste

Notre objet d'étude aujourd'hui est la structure de données linéaire **liste**.

Les objectifs de ce travail sont : - de définir la structure de données liste via les méthodes qui la caractérise - de manipuler cette structure de données - d'appréhender la notion de mutabilité des listes - d'appréhender la complexité de la manipulation des listes - de comprendre que ce qui est appelé liste en Python n'est pas une liste au sens commun du terme

> De manière parallèle, nous en profiterons pour introduire quelques éléments de programmation objet, puisqu'il figure explicitement au programme de discuter de différents paradigmes de programmation. Nous reviendrons sur cette notion dans une prochaine séance du DIU. Lorsque les notions décrites abordent spécifiquement la programmation objet, le paragraphe sera formaté comme celui-ci.

La structure de données

Vous connaissez déjà la structure de liste puisque vous l'avez largement utilisée dans les programmes Python que vous avez pu écrire précédemment. Vous avez créé des listes, ajouté des éléments, accédé à sa longueur, accédé à un élément, etc.

Néanmoins vous ne vous êtes jamais interrogés sur ce qu'était la liste en tant que structure de données.

Qu'est ce qu'une liste ?

Nous savons tous intuitivement ce qu'est une liste. Une liste est une collection finie d'éléments qui se suivent. C'est donc une structure de données *linéaire*.

Une liste peut contenir un nombre quelconque d'éléments y compris nul (la liste vide).

Nous allons essayer de dégager une définition plus formalisée de ce qu'est une liste afin d'en dégager les opérations primitives et une structure essentielle qui nous permettra d'en donner des implantations.

Prenons une liste comme par exemple '$\ell_1 = [3, 1, 4]$'. C'est une liste à trois éléments (ou de longueur trois) dont le premier est '3', le deuxième '1', et le dernier '4'.

Une façon de décrire cette liste consiste à dire que

- la liste 'ℓ_1' possède un premier élément '3' qu'on nommera élément de *tête*,
- et que vient après cet élément de tête la liste '$\ell_2 = [1, 4]$' des éléments qui suivent, liste qu'on nommera *reste*.

Ce qu'on vient de dire de la liste 'ℓ_1' peut être répété pour la liste 'ℓ_2' qui est donc constituée :

- d'un élément de *tête* : '1',
- et d'un *reste* : '$\ell_3 = [4]$'.

À nouveau on peut répéter le même discours pour la liste 'ℓ_3' qui est donc constituée :

- d'un élément de *tête* : '4',
- et d'un *reste* : '$\ell_4 = []$'.

La liste 'ℓ_4' étant vide, elle ne possède pas d'élement de tête, et ne peut donc pas être décomposée comme nous venons de le faire à trois reprises.

Si on convient d'utiliser la notation '(x, ℓ)' pour désigner le couple constitué de l'élément 'x' de tête, et du reste 'ℓ' d'une liste, on peut alors écrire :

```
\ell_1 = (3, (1, (4, [])))
```

On conçoit aisément que ce qui vient d'être fait pour notre exemple de liste 'ℓ_1' peut être reproduit pour n'importe quelle liste.

On peut conclure cette approche en donnant une définition abstraite et formelle des listes d'éléments appartenant tous à un ensemble 'E'.

Une *liste* d'éléments d'un ensemble 'E' est

- soit la liste vide
- soit un couple '(x, ℓ)' constitué d'un élément '$x \in E$' et d'une liste 'ℓ' d'éléments de 'E'.

Il ressort de cette définition que les listes peuvent être vues comme des structures de données récursives.

Primitives sur les listes

En nous appuyant sur la définition formelle qui vient d'être établie, nous sommes en mesure de dégager les opérations primitives qui suivent.

Constructeur D'après la définition, une liste est * soit la liste vide, * soit un couple constitué de l'élément de tête suivi de la liste des éléments qui suivent.

Le constructeur de liste doit donc permettre de produire soit une liste vide et pour cela aucun argument n'est nécessaire, soit une liste à partir de deux arguments.

Sélecteurs Les listes non vides possèdent une tête et un reste. Il nous faut les sélecteurs pour accéder à ces deux composantes.

Prédicat Un prédicat testant la vacuité d'une liste peut s'avérer utile dans bien des circonstances.

Une classe implémentant les listes

Quelques éléments de programmation objet

En programmation objet le code est centré, comme son nom l'indique, sur la notion d'*objet* qui *instancie* une *classe*. Une *classe* est un concept dont l'*objet* est une réalisation. Par exemple une classe peut être le concept de rectangle dont une réalisation est un rectangle de 2×3cm de côté.

Une classe possède un certain nombre d'attributs qui permettent de définir les propriétés du concept (par exemple un rectangle possède une longueur et une largeur). Ces attributs sont des variables. Une classe possède également des méthodes qui sont des fonctions agissant sur un objet (par exemple une méthode `draw` pour la classe Rectangle, qui permettra de dessiner une instance d'un rectangle). Une méthode s'applique bien à un objet particulier et non à la classe : on peut dessiner une instance d'un rectangle mais on ne peut pas dessiner la notion de rectangle.

En Python, dans le code d'une classe, toutes les méthodes ont un premier paramètre obligatoire : `self`. Il s'agit de l'objet lui-même, celui sur lequel on agit (le rectangle qu'on est en train de dessiner pour poursuivre l'exemple précédent).

D'autre part une méthode spéciale, appelée le constructeur, permet *d'instancier* un objet, c'est-à-dire d'en créer un. En Python ce constructeur s'appelle obligatoirement `__init__` (notez bien la présence de deux caractères `_` entourant le nom `init`).

La notation permettant d'accéder aux attributs ou aux méthodes d'un objet est spécifique. C'est la *notation pointée*. Elle s'écrit avec l'identifiant de l'objet suivi d'un point suivi de l'identifiant de l'attribut

ou de la méthode. En Python vous avez déjà utilisé cette notation : par exemple pour trier une liste `l`, vous avez écrit `l.sort()` ce qui consiste à appeler la méthode `sort` sur l'objet `l` de la classe `list`.

La classe `List`

On propose une implantation des listes en Python dans le module `list.py` (que vous devez récupérer) sous la forme d'une classe. Nous avons fait le choix de représenter une liste en suivant la définition donnée ci-dessus.

C'est l'occasion de jeter un œil à la manière dont la classe est implantée. Le fichier contient en fait deux classes. L'une est juste une exception définie uniquement pour la classe Liste. La seconde classe est l'implantation de notre classe Liste. Dans les méthodes de la classe, et uniquement dans celles-ci, on peut accéder aux attributs de la classe. En l'occurrence il n'y a qu'un seul attribut `__cell` qui correspond à un couple '(x, ℓ)', suivant les explications ci-dessus. L'attribut est initialisé dans le constructeur (`__init__`). Le constructeur peut fonctionner de deux manières : soit sans argument pour créer une liste vide, soit avec deux arguments pour ajouter un élément en tête à une liste déjà existante. Dans les méthodes, l'accès à cet attribut se fait *via* la variable spéciale `self` (déclarée comme premier paramètre de la méthode) qui représente l'objet sur lequel la méthode est appelée. De même pour appeler une méthode de la classe au sein d'une autre méthode (par exemple l'appel à `is_empty` dans `head`).

Remarque

Voici quelques relations qu'on peut établir à partir de ces opérations primitives.

- pour toute liste `l` et tout élément `x`, on a `List(x, l).tail() == l` et `List(x, l).head() == x`,
- et pour toute liste non vide, on a `List(l.head(), l.tail()) == l`.

Manipulation des listes

Dans un fichier autre que `list.py` ou directement dans l'interpréteur, s'exercer à manipuler la structure de liste telle qu'implantée ici :

```
# Appelle le constructeur de la liste
l = List()
# Notons que bien que le constructeur ne contienne pas de `return`
# l'appel au constructeur renvoie bien l'objet construit.

# On vérifie que la liste est vide
l.is_empty()

# Crée une liste non vide en mettant 1 en tête et la liste précédente (vide) à la suite
l = List(1, l)
# On vérifie qu'il n'y a qu'un seul élément
l.tail().is_empty()

# On crée la liste (1, (2, (3, ())), qu'on écrit plus couramment (1, 2, 3)
L = List(1, List(2, List(3, List())))
```

Visualisation de la structure de donnée construite

- Suivez le lien suivant pour exécuter la création de la liste (1.(2.3())) sous PythonTutor et observer la structure récursive construite.

Construire des fonctions avancées avec les primitives

Nous allons maintenant voir que les méthodes primitives définies sur les `List` sont suffisantes pour réaliser des opérations plus complexes.

Pour ce faire, nous allons créer une classe `Extendedlist` qui permettra d'enrichir la classe `List` d'autres méthodes. La classe `Extendedlist` héritera de la classe `List`. A gros trait, hériter cela veut dire créer un nouveau type en réutilisant ce qui a été déjà réalisé dans la définition d'autres types.

Dans le cas présent on veut étendre un type pour permettre d'ajouter de nouvelles fonctionnalités.

Faire cela est facile en Python, comme vous pouvez le voir ci-dessous, il suffit d'indiquer dans la déclaration de classe, entre parenthèses, la classe de laquelle on hérite.

```
from list import *

class Extendedlist(List):

    pass
```

Une fois cela fait on peut créer une `Extendedlist` comme on le faisait avec les `List` et utiliser les méthodes déjà définies :

```
>>> l = Extendedlist(1,Extendedlist(2,Extendedlist()))
>>> l.head()
1
>>> print(l)
(1.(2.()))
```

Il suffit ensuite d'ajouter des méthodes.

- Récupérer le fichier extendedlist.py qui sera à compléter.

Parcours de liste

On propose maintenant d'écrire un certain nombre de méthodes qui réalisent des parcours de listes. La documentation et les doctests de ces méthodes sont accessibles dans le fichier `extendedlist.py`.

Notez que vous n'aurez pas d'autre choix que d'utiliser le constructeur et les méthodes `head`, `tail` et `is_empty` pour écrire les méthodes ci-dessous. En effet l'accès à l'attribut `__cell` de la classe `List` est interdit au sein de la classe `Extendedlist`.

- Écrire le code de la méthode `length` qui calcule de manière itérative la longueur de la liste.
- Écrire le code de la méthode `get` qui permet d'accéder de manière itérative à l'élément en position `i` dans la liste.
- Écrire le code d'un prédicat `search` qui retourne vrai si un élément donné en paramètre est dans la liste. Le calcul sera fait de manière récursive.
- Écrire le code d'une méthode `toString` retournant une représentation sous forme de chaîne de caractères de la liste. Le calcul sera fait de manière récursive.
- Écrire le code d'une méthode `toPythonList` qui construit une liste Python équivalente à la liste.

Toutes ces méthodes auraient aussi bien pu s'écrire itérativement que récursivement. Le concept de liste ayant été défini récursivement, chacune des méthodes peut s'écrire récursivement sans difficulté théorique.

Création de nouvelles listes

On propose maintenant d'écrire deux méthodes qui vont créer une nouvelle liste à partir d'une liste donnée.

- Écrire le code de la méthode récursive `sortedInsert` qui s'applique sur une liste triée et prend un élément. Elle doit renvoyer une liste, copie de la liste d'origine, dans laquelle l'élément a été inséré de manière à ce que la liste résultante soit encore triée.
- Écrire le code de la méthode récursive `reverse` qui s'applique à une liste et renvoie une nouvelle liste dont les éléments sont dans l'ordre inverse de la liste de départ.

Changer de représentation des listes

La classe `List` implante les listes en considérant des couples dont le premier élément est l'objet à stocker dans la liste et le second est le reste de la liste.

Jamais nous n'avons eu besoin de connaître cette implémentation pour écrire `Extendedlist`, il a suffit de connaître les primitives de manipulation des listes : constructeur, sélecteurs et prédicats.

Si nous changeons l'implémentation de la liste tout en gardant les mêmes primitives alors `Extendedlist` restera fonctionnel.

- Récupérer le module otherlist.py qui implémente une liste avec un dictionnaire à deux champs.
- Remplacer `from list import *` par `from otherlist import *` dans `extendedlist.py`.
- Lancer les doctests sur `extendedlist.py`.

Listes mutables

Les listes que nous avons utilisées jusqu'à maintenant sont *non mutables*. Cela signifie qu'on ne dispose pas de la possibilité ni de changer la valeur d'un élément de la liste, ni de modifier la liste en elle-même, par exemple en y ajoutant des éléments.

Dans la classe `OtherList` on a ajouté deux mutateurs (setter en anglais) : `set_tail` permet de modifier le reste de la liste, et `set_head` permet de modifier la tête de la liste (la valeur de l'élément en tête de liste).

- Observer en suivant ce lien sur PythonTutor ce qui se passe au niveau de la structure de données lorsqu'on exécute le code suivant :

```
>>> l1 = List(1,List(2,List()))
>>> l2 = List(3,List(4,List()))
>>> l1.set_tail(l2)
```

- Que va afficher `print(l1)` ? `print(l2)` ?

Vous observerez que le chaînage des éléments de la liste a changé, et comme la liste dont la tête était 2 n'est plus référencée, alors elle disparaît de l'affichage. Remarquez que la liste vide correspondant à `l1.tail().tail()` disparaît aussi. Si on avait stocké `l1.tail()`dans une variable, alors ces deux éléments n'auraient pas disparu.

- Ajouter la méthode `append` à la classe `Extendedlist` qui ajoute un élément passé en paramètre en queue de la liste sur laquelle la méthode est invoquée. La liste doit contenir au moins un élément.

```
def append (self, e):
    '''
    Append element e at the end of the list.
    Raises Listerror if the list is empty.

    >>> l = Extendedlist()
    >>> l.append(1)
    Traceback (most recent call last):
    ...
    otherlist.ListError
    >>> l = Extendedlist(1,Extendedlist())
    >>> l.append(2)
    >>> l
    (1.(2.()))
    >>> l.append(3)
    >>> l
    (1.(2.(3.())))
    '''
```

- Quels seront les affichages de la séquence d'instructions suivante :

```
>>> l1 = Extendedlist(1,Extendedlist(2,Extendedlist()))
>>> l2 = l1.tail()
>>> l1.length()
>>> l2.length()
>>> l1.append(3)
>>> l1.length()
>>> l2.length()
```

Itérer sur des listes plutôt qu'accéder au i-ème élément

Nous allons procéder à une expérience pour comparer le temps d'exécution des deux fonctions ci-dessous:

```
def somme_des_elements_a(l):
    s = 0
    for i in range(len(l)):
        s += l[i]
    return s

def somme_des_elements_b(l):
    s = 0
    for e in l:
        s += e
    return s
```

La différence est simplement la manière dont nous accédons aux éléments successifs : - d'un côté en accédant aux éléments par leur indice - d'un autre côté en itérant sur les éléments

Utilisation des fonctions spéciales de Python

Les fonctions précédentes ne peuvent pas être utilisées directement sur nos `Extendedlist`. Afin de pouvoir garder la même écriture qu'avec les listes Python, nous allons ajouter quelques méthodes spéciales de Python à notre classe `Extendedlist`: - la méthode `__str__` permet d'obtenir une représentation de l'objet sous forme de chaîne de caractères - la méthode `__len__` permet d'appliquer la fonction `len` à un objet - la méthode `__getitem__` permet l'écriture avec `[i]` pour obtenir l'élément en position `i` - les méthodes `__next__` et `__iter__` permettent l'écriture de boucle `for elt in`

Récupérer le code ci-dessous et l'ajouter au code de la classe `Extendedlist`

```
    def __str__ (self):
        return self.toString()

    def __getitem__ (self,i):
        return self.get(i)

    def __len__ (self):
        return self.length()

    def __next__ (self):
        try:
            v = self.__iter.head()
            self.__iter = self.__iter.tail()
            return v
        except:
            raise StopIteration

    def __iter__ (self):
        """
        Implantation très sommaire d'un itérateur. Ne permet pas d'itérer
        sur la même liste dans une boucle imbriquée.
        """
        self.__iter = self
        return self
```

Expérimenter

- Écrire un programme Python, utilisant les listes de type `Extendedlist` et des listes Python, et qui calcule le temps d'exécution des deux fonctions sommes pour :
  - des listes contenant 500, 1000, 1500, 2500, 3000 entiers

  - avec des listes de type `Extendedlist`
  - avec des listes de Python

On en profitera pour calculer également le rapport de temps des deux fonctions somme pour les deux types de listes.

On rappelle qu'on peut utiliser le module `timeit` pour calculer le temps mis par une fonction. Le paramètre `number` de la fonction `timeit` permet de demander la répétition de l'expérience un certain nombre de fois. Pour une expérimentation qui fasse du sens, il faut bien sûr que ce nombre soit plus grand que 1. On remarquera que c'est le temps total qui est retourné par `timeit`, pas le temps moyen.

```
d = timeit.timeit(lambda: somme_des_elements_a(l), number = 1)
```

- comparer les temps d'exécution en utilisant les listes natives de Python et notre implantation

Pour observer l'évolution des temps d'exécution, vous pouvez tracer un graphique de ces temps, en utilisant `pylab` (comme vous l'avez déjà fait pour Wator). Contrairement à Wator, il pourrait être utile de tracer un graphique avec une échelle logarithmique sur l'axe des ordonnées, ce qui nécessitera d'utiliser `pylab.semilogy(data_x, data_y)` au lieu de `pylab.plot(data_x, data_y)`.

- comparer les temps d'exécution des deux fonctions pour notre implantation, pouvait-on s'attendre au résultat observé ?
- comparer les temps d'exécution des deux fonctions pour les listes natives de Python, pouvait-on s'attendre au résultat observé ?

Conclusion

En réalité, la structure de liste en Python n'est pas celle que nous croyons. Ce n'est pas la structure de données communément appelée liste. Celle-ci ressemble plus à un tableau qu'à une liste.

Quelles différences ?

Dans un tableau les éléments sont alloués de manière consécutive en mémoire, ce qui fait que la complexité d'accès au i-ème élément est attendue en temps constant : un simple calcul d'adresse permet d'y accéder. L'inconvénient est que le tableau ne peut être redimensionné. Alors que dans la liste, les éléments sont alloués à chaque fois qu'un élément est ajouté en tête, ce qui fait qu'ils ne sont pas nécessairement alloués consécutivement. Il est nécessaire de parcourir la liste pour accéder au i-ème.

En Python la structure est plus complexe. Pour garantir une bonne efficacité la liste Python utilise des tableaux tout en permettant un redimensionnement (i.e. l'ajout d'un nouvel élément). Pour comprendre comment cela fonctionne on pourra se plonger dans le code C de l'implantation de Python ou bien lire ceci (an anglais).

Vous aurez peut-être noté également que l'itération est plus rapide avec les « listes » natives de Python plutôt qu'avec nos Extendedlist. Cela peut sembler étrange puisque dans les deux cas l'opération est en temps constant. La différence tient aux accès mémoire. Deux cases consécutives dans un tableau sont consécutives en mémoire. Récupérer la case suivante est donc très peu coûteux car une partie, voire la totalité, du tableau a été placée dans la mémoire cache. À l'inverse, pour accéder à l'élément suivant d'une `Extendedlist`, il est nécessaire d'accéder à une autre `ExtendedList` qui peut (ou pas) être consécutive en mémoire. Avec les `ExtendedList` il est possible qu'il y ait besoin d'aller chercher l'objet dans la mémoire centrale, et non dans la mémoire cache du processeur, ce qui est une opération bien plus coûteuse. Cela explique les différences de temps observées entre l'itération sur des `Extendedlist` et avec des « listes » natives Python.