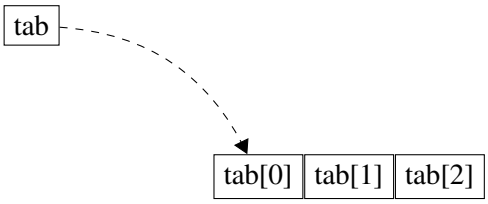
Spécialité NSI en terminale
Structures linéaires 1

Une structure linéaire sur un ensemble E est une suite d'éléments de E, c'est-à-dire une liste ordonnée d'éléments de E. Chaque élément a une place bien précise. Soit les éléments ont chacun un indice qui permet d'accéder directement à cette place, soit chaque place a un successeur ou (et) un prédécesseur et l'accès à cette place n'est plus direct. Dans le premier cas on parle de tableau, dans le second cas de liste chaînée. L'avantage du premier est la rapidité d'accès à un élément particulier, l'avantage du second est le dynamisme de la structure. En Python les objets de type list sont implémentés de manière à profiter des avantages des tableaux et de ceux des listes chaînées.

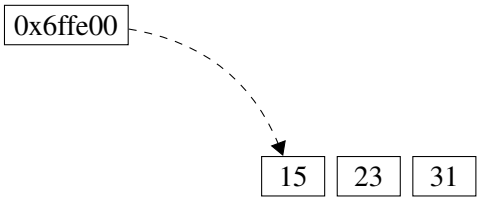
1 Principe d'un tableau

Si tous les éléments du tableau sont du même type, ils occupent tous la même taille, soit t, en mémoire. Il suffit alors de stocker l'adresse en mémoire du premier élément, soit a, et on accède à un élément d'indice k en calculant son adresse en mémoire par $a + k \times t$. Tous les éléments sont donc accessibles avec un coût constant, le temps de calcul de l'adresse et l'accès à cette adresse. La place du tableau en mémoire est réservée à la création, soit $n \times t$ si n est le nombre d'éléments. Une contrainte est l'impossibilité de remplacer un élément par un élément d'un autre type ou d'agrandir la taille du tableau. On ne sait pas ce qu'il y a en mémoire après la case contenant le dernier élément.

Le schéma ci-dessous présente la situation en mémoire :



Les valeurs placées en mémoire sont indiquées dans ce schéma :



Prenons par exemple le type `array` proposé par Python. Il ne s'agit pas de tableau au sens strict que l'on vient de décrire mais ce type, qui présente à peu près les mêmes caractéristiques que le type list, possède une contrainte importante des tableaux : le type des éléments est fixé à la création.

Tester le code Python suivant :

```
>>> import array
>>> t = array.array('b', [1, 2, 3])
>>> t[0] = 127
>>> t[0] = -128
```

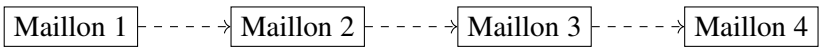
```
>>> t[0] = 128
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    t[0] = 128
OverflowError: signed char is greater than maximum
```

Le paramètre 'b' signifie que les éléments sont des entiers signés codés sur un octet. Donc un élément peut être n'importe quel entier de -128 à 127. C'est pourquoi l'instruction `t[0] = 128` provoque une erreur.

2 Principe des listes chaînées

Un élément d'une liste chaînée est stocké en mémoire avec l'adresse de l'élément suivant. Donc pour accéder au n-ième élément, il faut passer par les $n-1$ éléments qui le précèdent. Le coût d'accès à un élément est alors linéaire en fonction de la position de l'élément. Mais on peut ajouter des éléments au début, à l'intérieur ou à la fin de la liste, dont la taille peut donc varier. Pour ajouter un élément au début de la liste, pratiquement en temps constant, on place dans une case mémoire une donnée avec l'adresse du premier élément de la liste qui devient l'élément suivant de la donnée, donc le deuxième de la liste. La donnée est alors le premier élément de la liste.

Un élément, qu'on appelle un *maillon*, a une valeur et donne accès au maillon suivant.



En Python, il n'existe pas de type correspondant aux listes chaînées. On peut implémenter cette structure à l'aide du type tuple et cette méthode est proposée en exercice.

On peut aussi implémenter cette structure avec une classe.

Une manière de faire consiste à créer une classe `Maillon` puis une classe `Liste_Chaine`.

Créer en Python les classes `Maillon` et `Liste_Chaine`.

```
class Maillon:
    def __init__(self, valeur = None, suivant = None):
        self.val = valeur
        self.suiv = suivant

    def __str__(self): # méthode pour l'affichage
        if self.val is not None:
            return str(self.val) + " - " + str(self.suiv)
        else:
            return str(self.val)

class Liste_Chaine:
    def __init__(self, premier = None):
        self.prem = Maillon(premier)

    def ajoute(self, valeur): # ajout d'un élément en fin de liste
        m = Maillon(valeur) # création d'un maillon
        if self.prem.val is None: # si la liste est vide
            self.prem = m
```

```
        else:
            p = self.prem
            while p.suiv is not None: # recherche du dernier élément
                p = p.suiv
            p.suiv = m # m est le dernier élément

    def __str__(self):
        return str(self.prem)
```

Effectuer des tests :

```
>>> maliste = Liste_Chaine()
>>> maliste.ajoute("Z")
>>> maliste.ajoute("O")
>>> maliste.ajoute("R")
>>> print(maliste)
Z - O - R - None
```

On peut compléter le code de la classe `Liste_Chaine` avec une fonction `tete` qui renvoie la tête de la liste et une fonction `queue` renvoyant la liste obtenue après le retrait du premier élément.

```
class Liste_Chaine:
    ...

    def tete(self):
        if self.prem is not None:
            return self.prem.val

    def queue(self):
        if self.prem.val is None:
            return None
        else:
            liste = Liste_Chaine()
            liste.prem = self.prem.suiv
            return liste
```

Une fonction `insere1` qui insère un élément en tête de liste se code assez simplement.

```
class Liste_Chaine:
    ...

    def insere1(self, valeur):
        m = Maillon(valeur)
        if self.prem.val is None: # si la liste est vide
            self.prem = m
        else:
```

```
            p = self.prem # p est le premier élément actuel
            self.prem = m # m devient le premier élément
            m.suiv = p # p devient le suivant de m
```

3 Listes

Le type list en Python présente de nombreux avantages des tableaux et des listes chaînées, comme l'accès à un élément en coût constant et la possibilité de modifier la taille de la liste.

Les objets de type list, que nous appelons des listes, sont étudiés en classe de première. Dans une liste on peut insérer ou supprimer un élément à n'importe quel endroit, au début, à la fin mais aussi à l'intérieur. On peut accéder à tous les éléments en temps constant, parcourir toute la liste du début à la fin, supprimer une partie d'une liste ou concaténer deux listes. Il est nécessaire de maîtriser le cours de première et connaître les différentes méthodes de création d'une liste, de copie d'une liste, d'accès à un ou plusieurs éléments, de suppression ou d'ajout d'éléments.

Pour comprendre un fonctionnement parfois surprenant, il est nécessaire, au moins de manière schématique, de comprendre comment sont implémentées les listes.

Nous allons examiner ce qui se passe avec une instruction comme `liste = [300, 301, 302]`. La fonction `id` renvoie l'identifiant d'un objet que nous pouvons assimiler à une adresse en mémoire.

```
>>> liste = [300, 301, 302]
>>> id(liste)
2166984596552
```

À l'adresse 2166984596552 se trouvent diverses informations dont le nombre d'éléments et les adresses de ces éléments.

Nous pouvons obtenir l'identifiant ou l'adresse de chaque élément de la liste :

```
>>> id(liste[0])
2166986905360
>>> id(liste[1])
2166986906896
```

Le nombre 300 se trouve à l'adresse 2166986905360.

Nous avons donc le schéma suivant :

nom	$\longleftrightarrow$	adresse
liste	$\longleftrightarrow$	2166984596552

adresse	nombre d'éléments	adresses des éléments		
2166984596552 :	3	2166986905360	2166986906896	2166986906864

adresse	élément
2166986905360 :	300
2166986906896 :	301
2166986906864 :	302

Que se passe-t-il si nous exécutons l'instruction `liste[0] = [400, 401]` ?

La liste `[400, 401]` est créée avec une adresse, 2166987724552 dans l'exemple, qui remplace l'adresse du nombre 300. L'adresse de la liste n'a pas changé ni celles des deux autres éléments.

```
>>> liste[0] = [400, 401]
>>> id(liste)
2166984596552
>>> id(liste[0])
2166987724552
```

Nous disons que la liste a été modifiée en place.

Le schéma est alors le suivant, avec l'adresse modifiée de `liste[0]` en gras :

nom	⟷	adresse
liste	⟷	2166984596552

adresse	nombre d'éléments	adresses des éléments		
2166984596552 :	3	**2166987724552**	2166986906896	2166986906864

adresse	nombre d'éléments	adresses des éléments	
2166987724552 :	2	2166987533584	2166987533616

adresse	élément
2166987533584 :	400
2166987533616 :	401

adresse	élément
2166986906896 :	301
2166986906864 :	302

Avec une instruction comme `liste.append(x)`, l'adresse de `liste` n'est pas modifiée. C'est le contenu se trouvant à cette adresse qui est modifié. Le nombre d'éléments passe à 4 et l'adresse de ce quatrième élément est ajoutée.

Il en est de même avec une instruction comme `liste += [x]`. Par contre, avec une instruction comme `liste = liste + [x]`, l'adresse de `liste` est modifiée, puisque c'est une nouvelle liste qui est créée.

Ce comportement peut produire des effets indésirables, des *effets de bord*.

Par exemple, considérons l'exemple suivant :

```
>>> def double1(u):
        u = u + u
        return u

>>> def double2(u):
```

```
        u += u
        return u

>>> liste = [0]
>>> double1(liste)
[0, 0]
>>> liste
[0]
>>> double2(liste)
[0, 0]
>>> liste
[0, 0]
```

Si on revient à la représentation d'une liste en mémoire, ce comportement s'explique tout à fait clairement. Dans le premier cas, une variable locale `u` est créée à une adresse différente de celle de la liste passée en paramètre. Dans le second cas, aucune nouvelle liste n'est créée. La modification est effectuée sur la liste dont l'adresse est passée en paramètre.

Une liste en Python peut être utilisée pour construire d'autres types d'objets. Une liste a deux extrémités, un début et une fin, et permet l'insertion ou la suppression d'un élément à n'importe quel endroit, extrémités ou intérieur. On peut alors définir différents types suivant les restrictions appliquées, les manipulations autorisées.

4 Dictionnaires et n-uplets nommés

Ce sont des structures linéaires qui sont particulièrement utilisées dans la gestion des données en table. Les dictionnaires et les n-uplets sont étudiés en classe de première. On utilise les dictionnaires à différentes occasions et en particulier dans la partie sur les graphes. Les n-uplets nommés ou tuples nommés combinent des propriétés des dictionnaires et des tuples. On dispose en Python de la structure `namedtuple`, une sous-classe de la classe `tuple`, dans le module `collections`.

Un exemple d'utilisation à tester :

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(3, 4)
>>> p[0]
3
>>> p.x
3
>>> dic = {'x': 7, 'y': 5}
>>> q = Point(**dic)
>>> q
Point(x=7, y=5)
```

5 Exercices

Exercice 1

Reprendre les classes `Maillon` et `Liste_Chaine`. Compléter la classe `Liste_Chaine` avec une méthode `concat` qui permet de concaténer une deuxième liste à une liste initiale.

Par exemple, si la première liste `liste1` contient les élément 13, 15, 17 et la seconde liste `liste2` contient les éléments "A", "B", "C", après l'instruction `liste1.concat(liste2)`, la liste `liste1` contient les éléments 13, 15, 17, "A", "B", "C".

S'inspirer de la méthode `ajoute` dans la classe `Liste_Chaine`. Il faut accéder au dernier élément de la première liste.

Exercice 2

1. Écrire les classes `Maillon` et `Liste_Chaine` dans un fichier nommé `liste_chaine.py`.
   Écrire dans un second fichier une fonction qui prend en paramètre une liste de type list et renvoie une liste chaînée. Le module (fichier) `liste_chaine` est importé dans ce second fichier avec l'instruction `import liste_chaine as lc`.
2. Écrire une fonction qui prend en paramètre une liste chaînée et renvoie une liste de type list.

Pour la question 2, utiliser les fonctions `tete` et `queue`.

Exercice 3

On définit en Python une liste chaînée à l'aide d'un tuple.

```
def lst_ch(liste):
    lc = None
    for i in range(len(liste)):
        lc = (liste[-1-i], lc)
    return lc

# ou bien en récursif
def lst_ch(liste):
    if liste != []:
        return (liste[0], lst_ch(liste[1:]))

maliste = lst_ch([2, 8, 3, 5, 4])
```

1. Écrire une fonction `tete` qui renvoie la tête de la liste, c'est-à-dire le premier élément de la liste si elle est non vide et une fonction `queue` qui renvoie la queue de la liste, c'est-à-dire la liste sans le premier élément.
2. Écrire une fonction `ajoute` qui prend en paramètres un élément `val` et une liste, ajoute cet élément au début de la liste et renvoie la nouvelle liste.
3. On numérote les éléments de la liste chaînée à partir de 0 pour le premier élément. Écrire une fonction `element_n` qui prend en paramètres un entier `n` et une liste, et renvoie la valeur de l'élément portant le numéro `n`.
4. Écrire une fonction `longueur` qui prend en paramètre une liste, et renvoie sa longueur.

Pour les questions 3 et 4, penser à la récursivité.