Chap. 5 — Listes chaînées

1 — Introduction

1.1 — Méthodes de base sur les tableaux

Les tableaux en Python possèdent 2 méthodes essentielles extrêmement efficaces : `pop` et `append`.

`append()`

La méthode `append` ajoute un élément à la fin d'un tableau

[1]:
```
nb_premiers = ['un', 'trois', 'cinq', 'sept', 'onze']

# ajoute le nombre 13 dans le tableau
nb_premiers.append('treize')
```

Rien ne s'affiche ? C'est normal car `append` est une **méthode** qui agit sur l'objet `nb_premiers`. L'objet a été modifié mais *on ne le voit pas*.

[2]:
```
# Affiche le contenu de la variable
# nb_premiers

print(nb_premiers)
```

```
['un', 'trois', 'cinq', 'sept', 'onze', 'treize']
```

Comme tu peux le voir, le tableau a bien été modifié !

ACTIVITÉ 1

Dans le bloc suivant, implémenter la fonction `pair(n: int) -> list` qui renvoie un tableau contenant les `n` premiers nombres pairs en commençant par 0

Exemple 1 : `print(pair(5))` devrait afficher `[0, 2, 4, 6, 8]`.

Exemple 2 : `tab_pairs = pair(10)` ne devrait rien afficher mais seulement affecter à `tab_pairs` le tableau des 10 premiers nombres premiers.

CORRECTION

[3]:
```
def pair(n):
    tab = []
    for i in range(n):
        tab.append(i * 2)
    return tab


print(pair(5))
```

```
[0, 2, 4, 6, 8]
```

`pop()`

La méthode `pop` fait deux choses :

- supprimer le dernier élément du tableau
- renvoyer l'élément supprimé

[4]:
```
nb_premiers.pop()
nb_premiers.pop()
nb_premiers.pop()
```

4]: `'sept'`

Normalement, le bloc précédent affiche `'sept'`.

En effet,

- le premier appel à la méthode `pop` supprime le dernier élément (`treize`) et le renvoie. Il n'y a pas d'affectation donc cette information se perd.
- le deuxième appel à la méthode `pop` supprime le dernier élément (`onze`) et le renvoie. Il n'y a pas d'affectation donc cette information se perd.

- le troisième appel à la méthode `pop` supprime le dernier élément (`sept`) et le renvoie. Il n'y a pas d'affectation donc cette information se perd. **Mais** comme on travaille dans un notebook, la dernière ligne de code est affichée si elle renvoie quelque chose (ce qui est le cas ici).

Si on en a besoin, on peut affecter le dernier élément supprimé à une variable pour le réutiliser plus tard.

[5]:
```
dernier = nb_premiers.pop()

texte = dernier * 5

print(texte)
```

```
cinqcinqcinqcinqcinq
```

ACTIVITÉ 2

Implémenter la fonction `vide(tab: list) -> None` qui vide un tableau élément par élément en commençant par la fin et en affichant à chaque fois l'élément supprimé.

Exemple 1 : `vide([2, 4, 6])` doit afficher 6, puis 4 puis 2.

Exemple 2 : le code suivant

```
annee = [1998, 2003, 2004, 2008, 2021]
vide(annee)
print(annee)
```

doit afficher 2021, puis 2008, puis 2004, puis 2003 puis 1998 puis `[]`. Le dernier affichage s'explique car le tableau `annee` est vide à la fin.

CORRECTION

[6]:
```
def vide(tab):
    n = len(tab)
    for i in range(n):
        print(tab.pop())
```

Exemple

[7]:
```
annee = [1998, 2003, 2004, 2008, 2021]
vide(annee)
print(annee)
```

```
2021
2008
2004
2003
1998
[]
```

1.2 — Ajouter un élément au début d'un tableau

Les tableaux de Python permettent par exemple d'insérer ou de supprimer efficacement des éléments à la fin d'un tableau, avec les opérations `append` et `pop`, mais se prêtent mal à l'insertion ou la suppression d'un élément à une autre position.

En effet, les éléments d'un tableau étant contigus et ordonnés en mémoire, insérer un élément dans une séquence demande de déplacer tous les éléments qui le suivent pour lui laisser une place.

Si par exemple on veut insérer une valeur v à la pemière position d'un tableau

1	1	2	3	5	8	13

il faut d'une façon ou d'une autre construire le nouveau tableau

v	1	1	2	3	5	8	13

REMARQUE

Cette opération est cependant très coûteuse, car elle déplace tous les éléments du tableau d'une case vers la droite après avoir agrandi le tableau.

En effet, avec une telle opération :

1. On commence donc par agrandir le tableau, en ajoutant un nouvel élément à la fin avec `append`.

1	1	2	3	5	8	13	None

2. Puis on décale tous les éléments d'une case vers la droite, en prenant soin de commencer par le dernier et de terminer par le premier.

1	1	1	2	3	5	8	13

3. Enfin, on écrit la valeur v dans la première case du tableau.

v	1	1	2	3	5	8	13

ACTIVITÉ 3

Utiliser la description de l'algorithme en 3 étapes ci-dessus pour implémenter la fonction `entete(tab: list, val: int) -> None` **qui insère en début de tableau le nombre entier** `val`.

Exemples et tests :

```
>>> tab_impairs = [3, 5, 7, 9]
>>> entete(tab_impairs, 1)
>>>
[1, 3, 5, 7, 9]
```

CORRECTION

[8]:
```
def entete(tab, val):
    """
    Procédure qui modifie tab
    en ajoutant val en tête du tableau.
    """
    # étape 1
    tab.append(None)

    # étape 2
    n = len(tab)
    for i in range(n-1, 0, -1):
        # parcours de n-1 (inclus) à 0 (exclus)
        # en soustrayant 1 à chaque tour de boucle
        temp = tab[i-1]
        tab[i-1] = tab[i]
        tab[i] = temp

    # étape 3
    tab[0] = val
```

Exemple

[9]:
```
tab_impairs = [3, 5, 7, 9]
entete(tab_impairs, 1)
print(tab_impairs)
```

```
[1, 3, 5, 7, 9]
```

En Python, la méthode `insert(index:int, val)` est équivalente à `entete` si on définit `index` à `0`.
Ainsi, `tab.insert(0, 42)` est équivalent à `entete(tab, 42)`.

REMARQUE

Mais comme on l'a dit plus haut, que l'on utilise `entete` ou `insert` on a réalisé au total un nombre d'opérations proportionnel à la taille du tableau. Si par exemple le tableau contient un million d'éléments, on fera un million d'opérations pour ajouter un premier élément. En outre, supprimer le premier élément serait tout aussi coûteux, pour les mêmes raisons.

Dans ce chapitre nous étudions une structure de données, la **liste chaînée**, qui d'une part apporte une meilleure solution au problème de l'insertion et de la suppression au début d'une séquence d'éléments, et d'autre part servira de brique de base à plusieurs autres structures dans les prochains chapitres.

ACTIVITÉ 4

Implémenter à l'aide de la méthode `pop` une fonction `supprime(tab: list)` qui supprime le premier élément du tableau `tab`.

Exemple : Le code suivant

```
tab_impairs = [3, 5, 7, 9]
supprime(tab_impairs)
print(tab_impairs)
```

doit afficher `[5, 7, 9]`

CORRECTION

[26]:
```
def supprime(tab):
    """
    Procédure qui supprime le premier élément d'un tableau
    en utilisant la méthode pop

    Args:
        tab (list): tableau à modifier
    """
    n = len(tab)
    for i in range(n-1):
        # parcours le tableau de façon ascendante
        # en s'arrêtant une cellule avant la fin
        temp = tab[i+1]
        tab[i+1] = tab[i]
        tab[i] = temp

    # modification du tableau
    tab.pop()
```

Exemple

[27]:
```
tab_impairs = [3, 5, 7, 9]
supprime(tab_impairs)
print(tab_impairs)
```

```
[5, 7, 9]
```

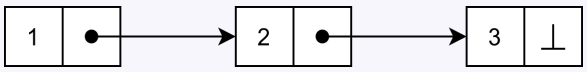
2 — Structure de liste chaînée

Une **liste chaînée** permet avant tout de réprésenter une liste dont les éléments sont chaînés entres eux, permettant ainsi le passage d'un élément à l'élément suivant.

Ainsi, chaque élément est stocké dans un bloc mémoire que l'on pourra appeler **maillon**. Ce maillon possède deux informations : la `valeur` stockée et l'adresse mémoire du maillon `suivant`.

Exemple

Par exemple on a illustré une liste contenant trois éléments, respectivement 1, 2 et 3.



Chaque élément de la liste est matérialisé par un emplacement en mémoire contenant d'une part sa valeur (dans la case de gauche) et d'autre part l'adresse mémoire de la valeur suivante (dans la case de droite).

Dans le cas du dernier élément, qui ne possède pas de valeur suivante, on utilise une valeur spéciale désignée ici par le symbole ⊥ et marquant la fin de la liste

Une façon traditionnelle de représenter une liste chaînée en Python consiste à utiliser une classe décrivant les maillons de la liste, de sorte que **chaque élément de la liste est matérialisé par un objet de cette classe**.

Implémenter la classe `Maillon` possédant deux attributs : `valeur` pour la valeur de l'élément (l'entier, dans notre exemple) ; et `suivant` pour le maillon suivant de la liste.

Exemple

Exemple 1 : pour affecter à `m2` le maillon contenant la valeur `3` et n'ayant aucun maillon suivant on écrira l'instruction `m2 = Maillon(3, None)`.

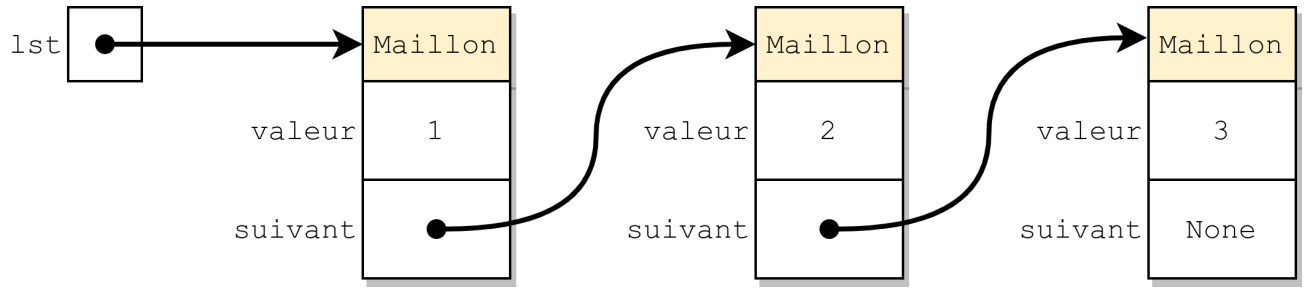
Exemple

Exemple 2 : Pour construire une liste, on applique le constructeur autant de fois qu'il y a d'éléments dans la liste. Ainsi pour construire la liste 1, 2, 3 du schéma que l'on stocke dans la variable `lst` on écrira : `lst = Maillon(1, Maillon(2, Maillon(3, None)))`.

[11]:
```
from doctest import testmod

class Maillon:
    """
    Une classe pour représenter le maillon d'une liste

    Attributs
    ---------
    valeur : type
        valeur contenue dans le maillon
    suivant : maillon
        maillon suivant ou None si pas de maillon
    """

    def __init__(self, valeur, suivant):
        """Constructeur de classe

        Args:
            valeur (type): valeur stockée dans le maillon
            suivant (maillon): maillon suivant ou None

        Exemples et tests:
        >>> l3 = Maillon(3, None)
        >>> assert (l3.valeur == 3)
        >>> assert (l3.suivant == None)

        >>> l2 = Maillon(2, l3)
        >>> assert (l2.valeur == 2)
        >>> assert (l2.suivant.valeur == 3)

        >>> l1 = Maillon(1, l2)
        >>> assert (l1.valeur == 1)
        >>> assert (l1.suivant.valeur == 2)
        >>> assert (l1.suivant.suivant.valeur == 3)
        >>> assert (l1.suivant.suivant.suivant == None)

        >>> l1.suivant.suivant.suivant.valeur
        Traceback (most recent call last):
```

```
        AttributeError: 'NoneType' object has no attribute 'valeur'
        """
        self.valeur = valeur
        self.suivant = suivant


lst = Maillon(1, Maillon(2, Maillon(3, None)))

testmod()
```

1]: TestResults(failed=0, attempted=12)



La valeur contenue dans la variable lst est l'adresse mémoire de l'objet contenant la valeur 1,

Définition récursive d'une liste chaînée

Comme on le voit, une liste est soit la valeur `None`, soit un objet de la classe `Maillon` dont l'attribut `suivant` contient une liste. C'est là une **définition récursive** de la notion de liste.

REMARQUE

Représentation alternatives

On peut représenter une liste chaînée par autre chose qu'un classe :

- avec des couples : `lst = (1, (2, (3, None)))`
- avec des tableaux à deux éléments : `lst = [1, [2, [3, None]]]`

— avec des champs nommés (dictionnaires)

REMARQUE

Variantes des listes chaînées

Il existe de nombreuses variantes des listes chaînées :

— listes cycliques (le dernier élément est lié au premier)
— listes doublement chaînées où chaque élément est lié à l'élément suivant **et** à l'élément précédent dans la liste
— listes cycliques doublement chaînées
— etc.

3 — Opérations sur les listes

Comme on l'a vu dans la partie 2, on se munit pour la suite d'une classe `Maillon` possède deux attributs : `valeur` et `suivant`.

[1]:
```
from doctest import testmod
```

[2]:
```
class Maillon:
    """
    Une classe pour représenter le maillon d'une liste

    Attributs
    ---------
    valeur : type
        valeur contenue dans le maillon
    suivant : maillon
        maillon suivant ou None si pas de maillon
    """

    def __init__(self, valeur, suivant):
        """Constructeur de classe

        Args:
            valeur (type): valeur stockée dans le maillon
            suivant (maillon): maillon suivant ou None

        Exemples et tests:
        >>> l3 = Maillon(3, None)
        >>> assert (l3.valeur == 3)
        >>> assert (l3.suivant == None)

        >>> l2 = Maillon(2, l3)
        >>> assert (l2.valeur == 2)
        >>> assert (l2.suivant.valeur == 3)

        >>> l1 = Maillon(1, l2)
        >>> assert (l1.valeur == 1)
        >>> assert (l1.suivant.valeur == 2)
        >>> assert (l1.suivant.suivant.valeur == 3)
        >>> assert (l1.suivant.suivant.suivant == None)

        >>> l1.suivant.suivant.suivant.valeur
        Traceback (most recent call last):
        AttributeError: 'NoneType' object has no attribute 'valeur'
        """
        self.valeur = valeur
        self.suivant = suivant


testmod()
```

2]: `TestResults(failed=0, attempted=12)`

3.1 — Longueur d'une liste

Par une fonction récursive L'objectif est d'implémenter une fonction récursive `longueur` qui reçoit en argument une liste `lst` et renvoie sa longueur.

Il faut distinguer le **cas de base** (c'est-à-dire une liste vide ne contenant aucun maillon) et le **cas récursif** c'est-à-dire une liste contenant au moins un maillon.

1. pour le cas de base, il faut renvoyer `0` car c'est une liste de longueur nulle ;
2. pour le cas récursif, il faut renvoyer la somme de `1` (pour le premier maillon) avec la longueur de la liste `lst.suivant` (que l'on calcule récursivement)

Implémenter la fonction récursive `longueur(lst) -> int` qui renvoie la longueur de la liste `lst`.

Exemple 1 : l'instruction `print( longueur(Maillon(42, None)) )` doit afficher `1` car cette liste ne contient qu'un seul maillon. :

```
>>> print( longueur(Maillon(42, None)) )
1
```

Exemple 2 : l'instruction `print( longueur(None) )` doit afficher `0` car c'est la liste vide.

```
>>> print( longueur(None) )
0
```

Exemple 3 : l'instruction `print(longueur( Maillon(1, Maillon(2, Maillon(3, None))) ))` doit afficher `3`.

```
>>> print(longueur( Maillon(1, Maillon(2, Maillon(3, None)))
3
```

CORRECTION

[3]:
```
def longueur(lst):
    """ longueur d'une liste chaînée

    Exemples et tests:
    >>> lst = Maillon(42, None)
    >>> assert longueur(lst) == 1

    >>> lst = None
    >>> assert longueur(lst) == 0

    >>> lst = Maillon(1, Maillon(2, Maillon(3, None)))
    >>> assert longueur(lst) == 3
    """
    if lst == None:
        return 0
    return 1 + longueur(lst.suivant)


testmod()

print(longueur(Maillon(42, None)) )
print(longueur(None) )
print(longueur( Maillon(1, Maillon(2, Maillon(3, None))) ))
```

```
1
0
3
```

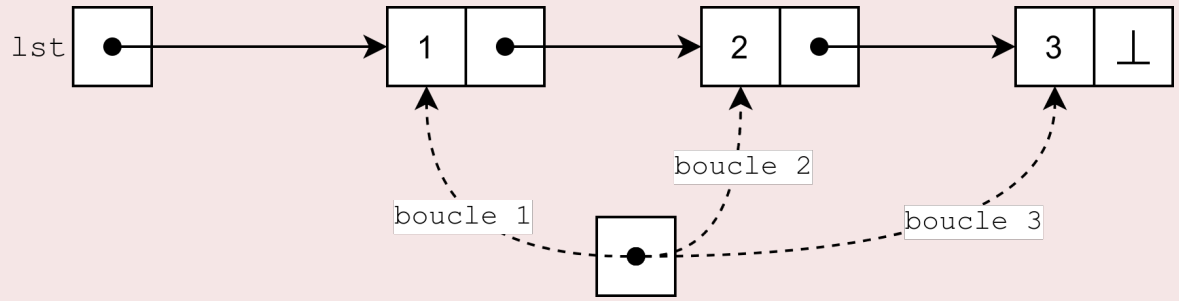
REMARQUE

Complexité temporelle. Il est clair que la complexité du calcul de la longueur est directement proportionnelle à la longueur elle-même, puisqu'on réalise un nombre constant d'opérations pour chaque maillon de la liste. Ainsi, pour une liste `lst` de mille maillons, longueur(lst) va effectuer mille tests, mille appels récursifs et mille additions dans sa version récursive.

Par une fonction itérative L'objectif est maintenant d'implémenter une version **itérative** de la fonction `longueur` qui reçoit en argument une liste `lst` et renvoie sa longueur.

Idée de l'algorithme. Définir :

- une variable accumulateur qui stocke la longueur de la liste parcourue qui vaut initialement 0
- une variable contenant le maillon courant qui vaut initialement `lst` (car lst est une liste et une liste pointe vers le premier maillon



Puis tant que le maillon courant n'est pas `None`, il faut incrémenter l'accumulateur de 1 et mettre à jour le maillon courant avec le maillon suivant. Lorsque le boucle s'arrête, c'est que le maillon courant est `None` et donc tous les maillons ont été visités. Il faut alors renvoyer l'accumulateur qui contient le nombre de maillons visités, qui est égal à la longueur de la liste.

ACTIVITÉ 2

Implémenter la fonction itérative `longueur(lst) -> int` qui renvoie la longueur de la liste `lst`.

Exemple 1 : l'instruction `print( longueur(Maillon(42, None)) )` doit afficher `1` car cette liste ne contient qu'un seul maillon.

```
>>> print( longueur(Maillon(42, None)) )
1
```

Exemple 2 : l'instruction `print( longueur(None) )` doit afficher `0` car

c'est la liste vide.

```
>>> print( longueur(None) )
0
```

Exemple 3 : l'instruction `print(longueur( Maillon(1, Maillon(2, Maillon(3, None))) ))` *doit afficher* `3`.

```
>>> print(longueur( Maillon(1, Maillon(2, Maillon(3, None)))
3
```

CORRECTION

[4]:
```
def longueur(lst):
    """ longueur d'une liste chaînée

    Exemples et tests:
    >>> lst = Maillon(42, None)
    >>> assert longueur(lst) == 1

    >>> lst = None
    >>> assert longueur(lst) == 0

    >>> lst = Maillon(1, Maillon(2, Maillon(3, None)))
    >>> assert longueur(lst) == 3
    """
    longueur_actuelle = 0
    maillon_actuel = lst

    while maillon_actuel is not None:
        longueur_actuelle = longueur_actuelle + 1
        maillon_actuel = maillon_actuel.suivant

    return longueur_actuelle


testmod()


print(longueur(Maillon(42, None)) )
print(longueur(None) )
print(longueur( Maillon(1, Maillon(2, Maillon(3, None))) ))
```

```
1
0
3
```

3.2 — Nième élément d'une liste

Comme pour la fonction précédente, on peut implémenter une version itérative et une version récursive de la fonction demandée...

ACTIVITÉ 3

Implémenter une version itérative de la fonction `nieme_element(n, lst)` qui renvoie le n-ième élément d'une liste chaînée. Évidement on prend par convention que le premier élément est désigné par $n = 0$.

Exemple 1 : `print( nieme_element(1, Maillon(42, None)) )` affiche `IndexError` car la liste chaînée n'a qu'un seul maillon (à l'indice 0) et donc pas de maillons à l'indice 1.

```
>>> print( nieme_element(1, Maillon(42, None)) )
Traceback (most recent call last):
IndexError: index out of range
```

Exemple 2 : `print( nieme_element(1, Maillon(1, Maillon(2, Maillon(3, None)))) )`affiche `2` car le maillon d'indice `1` contient la valeur `2`.

```
>>> print( nieme_element(1, Maillon(1, Maillon(2, Maillon(3
2
```

CORRECTION

[5]:
```
def nieme_element(n, lst):
    """ nieme element d'une liste chaînée
        version itérative

    Exemples et tests:
    >>> lst = Maillon(1, Maillon(2, Maillon(3, None)))
    >>> assert nieme_element(0, lst) == 1
    >>> assert nieme_element(1, lst) == 2
    >>> assert nieme_element(2, lst) == 3

    >>> nieme_element(3, lst)
    Traceback (most recent call last):
    IndexError: index out of range
    """
    if n >= longueur(lst):
        raise IndexError('index out of range')

    maillon_actuel = lst
    for _ in range(n):
        maillon_actuel = maillon_actuel.suivant
    return maillon_actuel.valeur


testmod()
```

[5]:
```
TestResults(failed=0, attempted=23)
```

ACTIVITÉ 4

Implémenter une version récursive de la fonction `nieme_element(n, lst)` qui renvoie le n-ième élément d'une liste chaînée. Évidement on prend par convention que le premier élément est désigné par $n = 0$.

Exemple 1 : `print( nieme_element(1, Maillon(42, None)) )` affiche `IndexError` car la liste chaînée n'a qu'un seul maillon (à l'indice 0) et donc pas de maillons à l'indice 1.

```
>>> print( nieme_element(1, Maillon(42, None)) )
IndexError
```

Exemple 2 : `print( nieme_element.(1, Maillon(1, Maillon(2,`

Maillon(3, None))))*affiche 2 car le maillon d'indice 1 contient la valeur 2.*

```
>>> print( nieme_element.(1, Maillon(1, Maillon(2, Maillon
2
```

CORRECTION

[6]:
```
def nieme_element(n, lst):
    """ nieme element d'une liste chaînée
        version itérative

    Exemples et tests:
    >>> lst = Maillon(1, Maillon(2, Maillon(3, None)))
    >>> assert nieme_element(0, lst) == 1
    >>> assert nieme_element(1, lst) == 2
    >>> assert nieme_element(2, lst) == 3

    >>> assert nieme_element(3, lst) == 3
    Traceback (most recent call last):
    IndexError: index out of range
    """
    if n >= longueur(lst):
        raise IndexError('index out of range')

    if n == 0:
        return lst.valeur

    return nieme_element(n - 1, lst.suivant)


testmod()
```

[6]:
```
TestResults(failed=0, attempted=23)
```

3.3 — Concaténation de deux listes

Considérons maintenant l'opération consistant à mettre bout à bout les éléments de deux listes données. On appelle cela la **concaténation** de deux listes. Ainsi, si la première liste contient 1,2,3 et la seconde 4,5 alors le résultat de la concaténation est la liste 1,2,3,4,5.

Nous choisissons d'écrire la concaténation sous la forme d'une fonction

`concatener(l1, l2)` qui reçoit deux listes en arguments et renvoie une troisième liste contenant la concaténation.

L'algorithme **récursif** est très simple :

- si la liste `l1` est vide, la concaténation est identique à `l2` et il suffit de renvoyer `l2`
- sinon, le premier élément de la concaténation est le premier élément de `l1` et le reste de la concaténation est obtenu récursivement en concaténant le reste de `l1` avec `l2`.

Implémenter la version récursive de la fonction `concatener` qui prend deux listes `l1` et `l2` en argument et renvoie la concaténation des deux listes.

Exemples et tests :

```
>>> l1 = Maillon(1, Maillon(2, Maillon(3, None)))
>>> l2 = Maillon(4, Maillon(5, None))
>>> l3 = concatener(l1, l2)
>>> assert nieme_element(0, l3) == 1
>>> assert nieme_element(1, l3) == 2
>>> assert nieme_element(2, l3) == 3
>>> assert nieme_element(3, l3) == 4
>>> assert nieme_element(4, l3) == 5

>>> nieme_element(5, l3)
Traceback (most recent call last):
IndexError: index out of range
```

CORRECTION

[7]:
```
def concatener(start_lst, end_lst):
    """ concaténer deux liste de façon récursive

    Exemples et tests:
    >>> l1 = Maillon(1, Maillon(2, Maillon(3, None)))
    >>> l2 = Maillon(4, Maillon(5, None))
    >>> l3 = concatener(l1, l2)
    >>> assert nieme_element(0, l3) == 1
    >>> assert nieme_element(1, l3) == 2
    >>> assert nieme_element(2, l3) == 3
    >>> assert nieme_element(3, l3) == 4
    >>> assert nieme_element(4, l3) == 5

    >>> nieme_element(5, l3)
    Traceback (most recent call last):
    IndexError: index out of range

    """
    if longueur(start_lst) == 0:
        return end_lst

    premier_element = Maillon(start_lst.valeur, None)
    reste = concatener(start_lst.suivant, end_lst)
    premier_element.suivant = reste
    return premier_element


testmod()
```

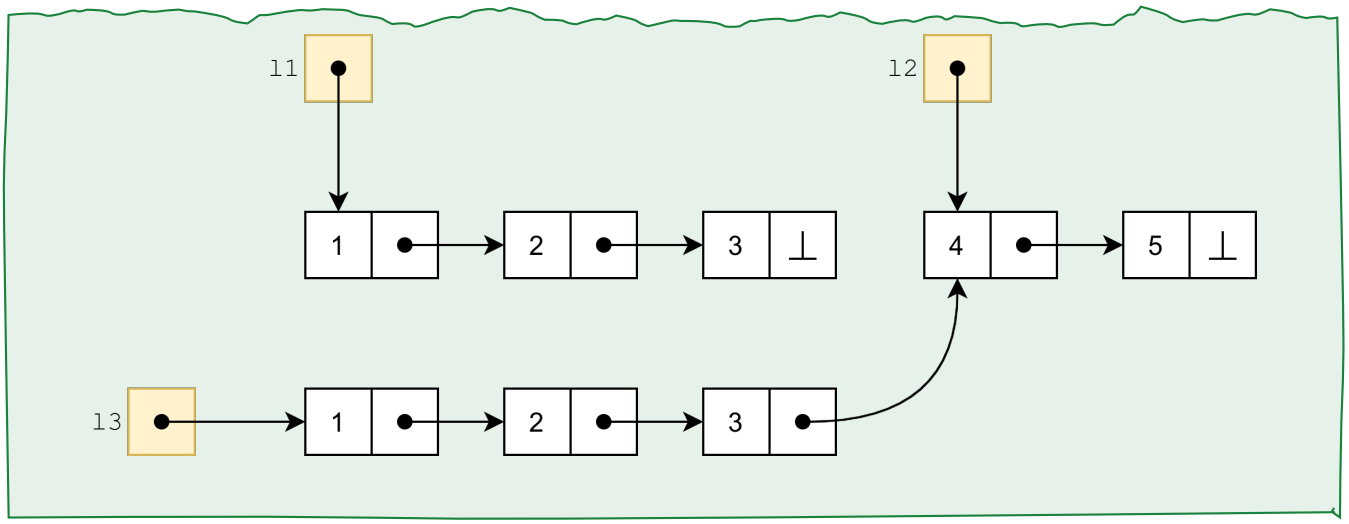
[7]:
```
TestResults(failed=0, attempted=32)
```

REMARQUE

Il est important de comprendre ici que les listes passées en argument à la fonction concatener ne sont pas modifiées. Plus précisément, les éléments de la liste `l1` sont copiés et ceux de `l2` sont partagés. Illustrons-le avec la concaténation des listes 1, 2, 3 et 4, 5. Après les trois instructions

```
l1 = Maillon(1, Maillon(2, Maillon(3, None)))
l2 = Maillon(4, Maillon(5, None))
l3 = concatener(l1, l2)
```

on a la situation suivante avec 8 maillons au total :



3.4 — Renverser une liste

ACTIVITÉ 6

Implémenter une fonction `renverser(lst)` qui reçoit en argument une liste comme 1, 2, 3 et renvoie la liste renversée 3, 2, 1.

Exemples et tests :

```
>>> l1 = Maillon(1, Maillon(2, Maillon(3, None)))
>>> l2 = renverser(l1)
>>> assert nieme_element(0, l2) == 3
>>> assert nieme_element(1, l2) == 2
>>> assert nieme_element(2, l2) == 1

>>> nieme_element(3, l2)
Traceback (most recent call last):
IndexError: index out of range
```

CORRECTION

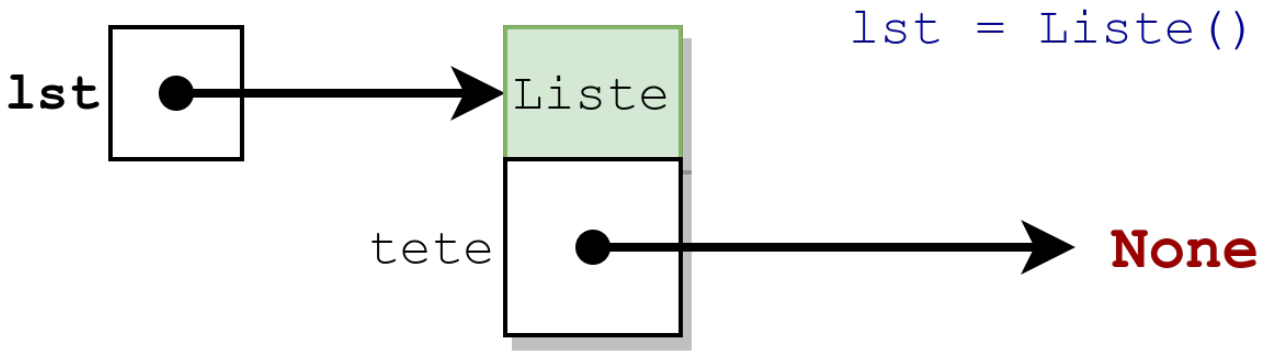
[8]:
```
def renverser(lst):
    """ renvoyer une nouvelle liste chainée renversée

    Exemples et tests:
    >>> l1 = Maillon(1, Maillon(2, Maillon(3, None)))
    >>> l2 = renverser(l1)
    >>> assert nieme_element(0, l2) == 3
    >>> assert nieme_element(1, l2) == 2
    >>> assert nieme_element(2, l2) == 1

    >>> nieme_element(3, l2)
    Traceback (most recent call last):
    IndexError: index out of range
    """
    n = longueur(lst)
    new_lst = None
    for i in range(n):
        valeur = nieme_element(i, lst)
        new_lst = Maillon(valeur, new_lst)
    return new_lst


testmod()
```

[8]:
```
TestResults(failed=0, attempted=38)
```

4 — Encapsulation dans un objet

Méthodes de bases

Pour finir nous allons maintenant encapsuler une liste chaînée **dans un objet**.
L'idée consiste à définir une nouvelle classe, `Liste`, qui possède un unique attribut, `tete`, qui contient une liste chaînée. On l'appelle tete car il désigne la tête de la liste, lorsque celle-ci n'est pas vide (et `None` sinon).
Le constructeur initialise l'attribut `tete` avec la valeur `None`.



Il y a de multiples intérêts à cette encapsulation :

- D'une part, il cache la représentation de la structure à l'utilisateur. Ainsi, celui qui utilise notre classe `Liste` n'a plus à manipuler explicitement la classe `Maillon`. Mieux encore, il peut complètement ignorer son existence. De même, il ignore que la liste vide est représentée par la valeur `None`. En particulier, la réalisation de la classe `Liste` pourrait être modifiée sans pour autant que le code qui l'utilise n'ait besoin d'être modifié à son tour.
- D'autre part, l'utilisation de classes et de méthodes nous permet de donner le même nom à toutes les méthodes qui sont de même nature. Ainsi, on peut avoir plusieurs classes avec des méthodes `est_vide`, `ajoute`, etc. Si nous avions utilisé de simples fonctions, il faudrait distinguer `liste_est_vide`, `pile_est_vide`, `ensemble_est_vide`, etc.

ACTIVITÉ 1

Implémenter la classe `Liste` avec un constructeur qui initialise l'attribut `tete` à None.

Exemple :

```
>>> lst = Liste()
>>> print(lst.tete)
None
```

[ ]:
```python
class Liste:
    def __init__(self):
        """
        Constructeur d'une liste vide.

        Exemples :
        >>> lst = Liste()
        >>> print(lst.tete)
        None
        """

        self.tete = None


# test avec l'exemple
testmod()
```

Ainsi, un objet construit avec `Liste()` représente une liste vide.

On peut également introduire une méthode `est_vide` qui renvoie un booléen indiquant si la liste est vide. En effet, notre intention est d'encapsuler, c'est-à-dire de cacher, la représentation de la liste derrière cet objet. Pour cette raison, on n e souhaite pas que l'utilisateur de la classe `Liste` teste explicitement si l'attribut `tete` vaut `None`, mais qu'il utilise cette méthode `est_vide`.

ACTIVITÉ 2

Ajouter à la classe `Liste` la méthode `est_vide()` qui renvoie `True` si la liste est vide et `False` sinon.

Exemples :

```
>>> lst = Liste()
>>> print(lst.est_vide())
True

>>> lst = Liste()
>>> lst.tete = Maillon(1, None)
>>> print(lst.est_vide())
False
```

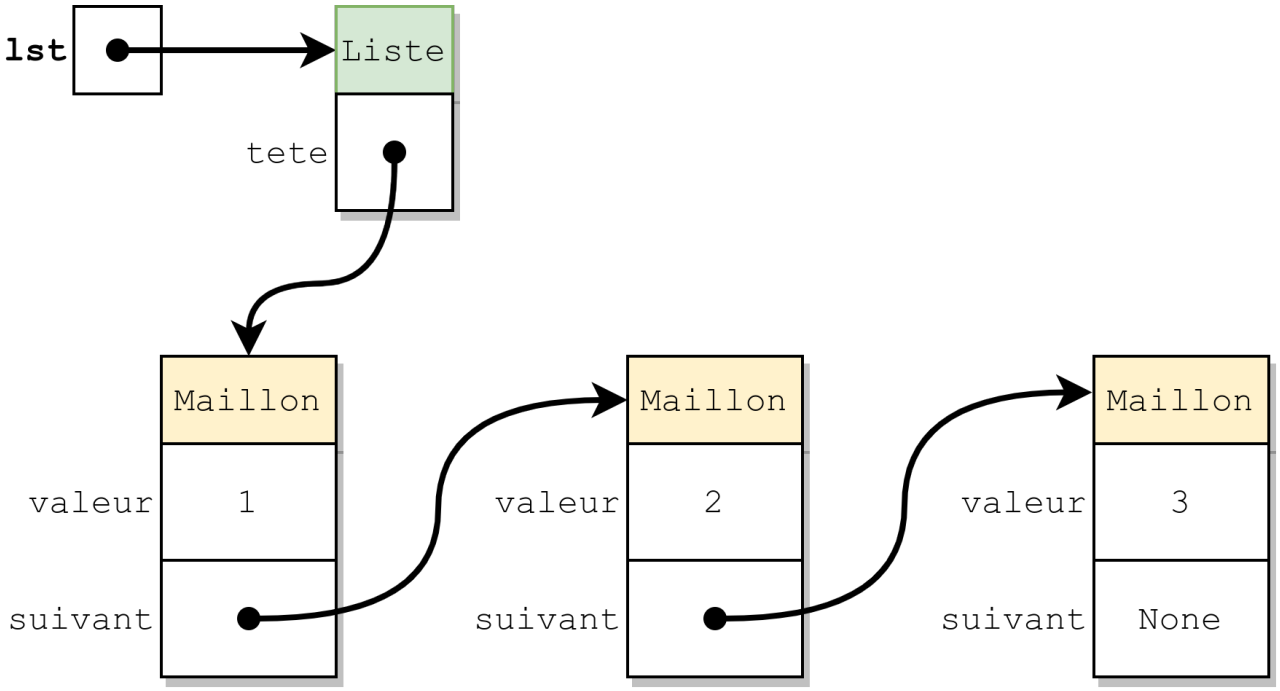
[ ]:
```python
# modification de la classe Liste existante
#
# Pour ne pas supprimer les implémentations précédentes,
# il faut "étendre" la classe Liste pour l'enrichir.
#
# Pour cela, il faut écrire
# `class Liste(Liste):` à la place de `class Liste:`

class Liste(Liste):
    def est_vide(self) -> bool:
        """ Est ce que la liste est vide ?

        Returns:
            bool: True si et seulement si la liste est vide

        Exemples :
        >>> lst = Liste()
        >>> print(lst.est_vide())
        True

        >>> lst = Liste()
        >>> lst.tete = Maillon(1, None)
        >>> print(lst.est_vide())
        False
        """

        return self.tete is None


# tests avec les exemples
testmod()
```

On poursuit la construction de la classe `Liste` avec une méthode pour `ajouter` un élément en tête de la liste.

Cette méthode modifie l'attribut `tete` et ne renvoie rien. Si par exemple on exécute les quatre instructions ci-dessous, on obtient la situation représentée par le

schéma :

```
lst = Liste()
lst.ajoute(3)
lst.ajoute(2)
lst.ajoute(1)
```



On a donc construit ainsi la liste 1, 2, 3, dans cet ordre.

ACTIVITÉ 3

Implémenter dans la classe `Liste` la méthode `ajoute` ayant un paramètre `valeur`. Cette méthode ajoute un nouveau maillon en tête de la liste ayant pour valeur : `valeur` et pour attribut `suivant` : l'ancien attribut `tete` de la liste.

Exemples :

```
>>> lst = Liste()
>>> lst.ajoute(1)
```

```
>>> print(lst.tete.valeur)
1
>>> lst.ajoute(2)
>>> print(lst.tete.valeur)
2
>>> print(lst.tete.suivant.valeur)
1
```

[ ]:
```python
class Liste(Liste):
    def ajoute(self, valeur):
        """
        Ajouter un nouveau maillon en tête de liste

        Args:
            valeur (type): valeur du nouveau maillon

        Exemples :
        >>> lst = Liste()
        >>> lst.ajoute(1)
        >>> print(lst.tete.valeur)
        1
        >>> lst.ajoute(2)
        >>> print(lst.tete.valeur)
        2
        >>> print(lst.tete.suivant.valeur)
        1
        """
        self.tete = Maillon(valeur, self.tete)


testmod()
```

Autres méthodes

On peut maintenant reformuler nos opérations, à savoir `longueur`, `nieme_element`, `concatener` ou encore `renverser`, comme autant de méthodes de la classe `Liste`. Ainsi, on peut écrire par exemple la méthode `longueur` qui nous permet d'écrire `lst.longueur()` pour obtenir la longueur de la liste `lst`.

[ ]:
```python
# il faut importer la fonction `longueur`
# d'une liste chaînée (cf. partie 3 du cours)
from operations_base import longueur


class Liste(Liste):
```

```
    def longueur(self) -> int:
        """
        Longueur d'une liste chaînée

        Returns:
            int: longueur de la liste

        Exemples:
        >>> lst = Liste()
        >>> assert lst.longueur() == 0
        >>> lst.ajoute(4)
        >>> assert lst.longueur() == 1
        >>> lst.ajoute(2)
        >>> assert lst.longueur() == 2
        >>> lst.ajoute(1)
        >>> assert lst.longueur() == 3
        """
        return longueur(self.tete)


testmod()
```

REMARQUE

Il est important de noter qu'il n'y a pas confusion ici entre la fonction `longueur` définie précédemment et la méthode `longueur`. En particulier, la seconde est définie en appelant la première. Le langage Python est ainsi fait que, lorsqu'on écrit `longueur(self.tete)`, il ne s'agit pas d'un appel récursif à la méthode `longueur` (un appel récursif s'écrirait `self.longueur()`).

`longueur()` et `self.longueur()` sont deux fonctions différentes !

On peut donner à cette méthode le nom `__len__` et Python nous permet alors d'écrire `len(lst)` comme pour un tableau. En effet, lorsque l'on écrit `len(e)` en Python, ce n'est qu'un synonyme pour l'appel de méthode `e.__len__()`.

[ ]:
```
# il faut importer la fonction `longueur`
# d'une liste chaînée (cf. partie 3 du cours)
from operations_base import longueur


# étendre la classe Liste pour ajouter des méthodes
class Liste(Liste):

    def __len__(self):
        """
        Permet d'utiliser :
            - la fonction len avec les Liste
```

```
            - une instance de Liste comme une expression booléenne :
                True  si et seulement si l'instance est de longueur > 0
                False si et seulement si l'instance est de longueur nulle

        Exemples:
        >>> lst = Liste()
        >>> assert len(lst) == 0
        >>> if lst: print("expression booléenne évaluée à False")
        >>> lst.ajoute(4)
        >>> assert len(lst) == 1
        >>> if lst: print("expression booléenne évaluée à True")
        expression booléenne évaluée à True
        """
        return longueur(self.tete)


testmod()
```

De même, on peut ajouter à la classe `Liste` une méthode pour accéder au n-ième élément de la liste, c'est-à-dire une méthode qui va appeler notre fonction `nieme_element` sur `self.tete`. Le nom de la méthode est arbitraire et nous pourrions choisir de conserver le nom `nieme_element`. Mais là encore nous pouvons faire le choix d'un nom idiomatique en Python, à savoir `__getitem__`.

Ceci nous permet alors d'écrire `lst[i]` pour accéder au i-ième élément de notre liste, exactement comme pour les tableaux.

ACTIVITÉ 4

Implémenter dans la classe `Liste` la méthode `__getitem__` de paramètre `index` permettant de renvoyer la valeur du maillon de rang `index` de la liste. Utiliser pour cela la fonction `nieme_element()`.

Exemple :

```
>>> lst = Liste()
>>> lst.ajoute(4)
>>> lst.ajoute(2)
>>> lst.ajoute(1)
>>> assert lst[0] == 1
>>> assert lst[1] == 2
```

```
>>> assert lst[2] == 4
```

[ ]:
```python
# il faut importer la fonction `longueur`
# d'une liste chaînée (cf. partie 3 du cours)
from operations_base import nieme_element


# étendre la classe Liste pour ajouter des méthodes
class Liste(Liste):

    def __getitem__(self, index):
        """ Permet d'utiliser la syntaxe des listes

        Exemples:
        >>> lst = Liste()
        >>> lst.ajoute(4)
        >>> lst.ajoute(2)
        >>> lst.ajoute(1)
        >>> assert lst[0] == 1
        >>> assert lst[1] == 2
        >>> assert lst[2] == 4
        """
        return nieme_element(index, self.tete)


testmod()
```

Pour la fonction `renverser`, on fait le choix de nommer la méthode `reverse` car là encore c'est un nom qui existe déjà pour les tableaux de Python.

Implémenter dans la classe `Liste` une méthode `reverse` ne renvoie rien mais inverse l'ordre des maillons de la liste.

Exemple :

```
>>> lst = Liste()
>>> lst.ajoute(3)
>>> lst.ajoute(2)
>>> lst.ajoute(1)
>>> lst.reverse()
>>> assert lst[0] == 3
>>> assert lst[1] == 2
```

```
>>> assert lst[2] == 1
```

[ ]:
```python
# il faut importer la fonction `renverser`
# d'une liste chaînée (cf. partie 3 du cours)
from operations_base import renverser


class Liste(Liste):
    def reverse(self):
        """
        Renverse la liste en place

        Exemples:
        >>> lst = Liste()
        >>> lst.ajoute(3)
        >>> lst.ajoute(2)
        >>> lst.ajoute(1)
        >>> lst.reverse()
        >>> assert lst[0] == 3
        >>> assert lst[1] == 2
        >>> assert lst[2] == 1
        """
        self.tete = renverser(self.tete)


testmod()
```

Enfin, le cas de la concaténation est plus subtil, car il s'agit de renvoyer une nouvelle liste, c'est-à-dire un nouvel objet. On choisit d'appeler la méthode `__add__`, qui correspond à la syntaxe + de Python.

ACTIVITÉ 6

Implémenter la méthode `__add__` de paramètre `autre_liste` qui renvoie une nouvelle liste, résultat de la concaténation de la liste actuelle et de `autre_liste`.

Exemple :

```
>>> lst_1 = Liste()
>>> lst_1.ajoute(1)
>>> lst_2 = Liste()
>>> lst_2.ajoute(3)
>>> lst_2.ajoute(2)
```

```
>>> lst_3 = lst_1 + lst_2
>>> assert lst_3[0] == 1      # lst_3 concaténée !
>>> assert lst_3[1] == 2
>>> assert lst_3[2] == 3
```

[ ]:
```python
# il faut importer la fonction `concatener`
# d'une liste chaînée (cf. partie 3 du cours)
from operations_base import concatener


# étendre la classe Liste pour ajouter des méthodes
class Liste(Liste):
    def __add__(self, liste):
        """
        Permet d'utiliser l'opérateur + entre instances de Liste

        Exemples:
        >>> lst_1 = Liste()
        >>> lst_1.ajoute(1)
        >>> lst_2 = Liste()
        >>> lst_2.ajoute(3)
        >>> lst_2.ajoute(2)
        >>> lst_3 = lst_1 + lst_2
        >>> assert lst_3[0] == 1      # lst_3 concaténée !
        >>> assert lst_3[1] == 2
        >>> assert lst_3[2] == 3
        """

        concat = Liste()
        concat.tete = concatener(self.tete, liste.tete)
        return concat


testmod()
```

Une liste chaînée est une structure de données pour représenter une séquence finie d'éléments. Chaque élément est contenu dans un Maillon, qui fournit par ailleurs un moyen d'accéder au maillon suivant. Les opérations sur les listes chaînées se programment sous la forme de parcours qui suivent ces liaisons, en utilisant une fonction récursive ou une boucle.