**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)
```





## 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)

**ACTIVITÉ 1**

**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.*

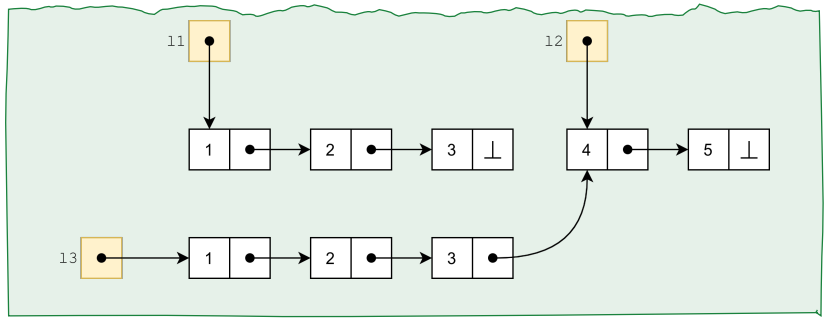
```
>>> print( longueur(None) )
0
```

*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
```








### 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**

[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 :





**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
```

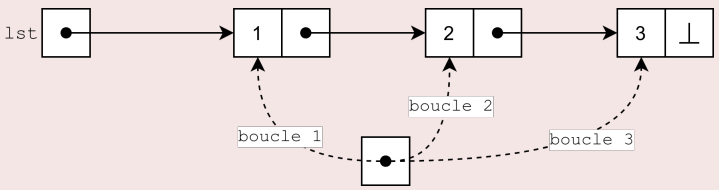
**REMARQUE**

**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*





`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`.

## ACTIVITÉ 5

**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
```

*`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





*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,`