[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
        """
        self.valeur = valeur
        self.suivant = suivant
```

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.

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

Exemple :

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

[3]:
```
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()
```

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

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`.

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
```

[4]:
```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()
```

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

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 à gauche, on obtient la situation représentée à droite :

(schéma)

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

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
```

[5]:
```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()
```

5]: TestResults(failed=0, attempted=6)

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`.

[6]:
```
# 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()
```

6]: TestResults(failed=0, attempted=8)

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__()`.

[8]:
```
# 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()
```

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

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.

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
```

[10]:
```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()
```

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

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
```

[12]:
```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()
```

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

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.

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
```

[16]:
```
# 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()
```

6]: `TestResults(failed=0, attempted=9)`

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.