

```
tnsvius = tnsvius.lles
                           self.valeur = valeur
  snow no tnavius nolliam : (nolliam) tnavius
nollinm ol sand observate relicion : (oqui) ruolin
                      esselo eb metountanol"""
                def __init__(self, valeur, suivant):
       mollism sh saq is snow no travius nollism
                                 nollipm : tabuius
                 nollism el sast enstros ruelsu
                                     ədhı: unəlpu
  Jassil sau'b nolliom al ratnasàrqor ruoq assolo sal
                                          [2]: class Maillon:
                             [1]: from doctest import testmod
```

4 – Encapsulation dans un objet

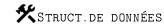
Sesed eb sebodtéM

Pour finir nous allons maintenant encapsuler une liste chaînée dans un objet.

la liste, lorsque celle-ci n'est pas vide (et None sinon). Le constructeur initialise but, tete, qui contient une liste chaînée. On l'appelle tete car il désigne la tête de L'idée consiste à définir une nouvelle classe, Liste, qui possède un unique attri-

Il y a de multiples intérêts à cette encapsulation : l'attribut tete avec la valeur None.

son tour. fiée sans pour autant que le code qui l'utilise n'ait besoin d'être modifié à None. En particulier, la réalisation de la classe Liste pourrait être moditence. De même, il ignore que la liste vide est représentée par la valeur classe Maillon. Mieux encore, il peut complètement ignorer son exiscelui qui utilise notre classe Liste n'a plus à manipuler explicitement la D'une part, il cache la représentation de la structure à l'utilisateur. Ainsi,



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]: 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

en utilisant une fonction récursive ou une boucle.

vide et False sinon.

```
Exemples:
>>> lst = Liste()
```

(()abiv_taa.tal)tnirq <<<

Łgjae

testmod()

```
səldməxə səl səvə sisəi #
                                enoN si etet. tete mruter
                                                       อธาช∄
                               (() sbiv_tss.tsl) ininq <<<
                        (snoN ,1)nollipM = stst.tel <<<
                                        () 5 1 2 i 1 = 1 2 l <<<
                               (() sbiv_tes.tel) ininq <<<
                                        () 5 1 2 i 1 = 1 21 <<<
                                                : səүdшəхд
obiv iso oisil al is imomoluos to is ourl :lood
                    def est_vide(self) -> bool:
"" Est ce que la liste est vide ?
                                                class Liste(Liste):
         ":stsi1 ssn15" sb son1q n1 \dot{\rm a} ":(stsi1)stsi1 ssn15" \rm \#
                                      # Pour cela, il faut écrire
    # Pour ne pas supprimer les implémentations précédentes, # il faut "étendre" la classe Liste pour l'enrichir.
                   etantsixe etsil essals al eb noitasitibom # :[4]
```

3 - E0.0

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

On poursuit la construction de la classe Liste avec une méthode pour a j'outer un élément en tête de la liste.

01

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)
   >>> lst.ajoute(2)
   >>> print(lst.tete.valeur)
   >>> print(lst.tete.suivant.valeur)
[5]: class Liste(Liste):
       def ajoute(self, valeur):
           Ajouter un nouveau maillon en tête de liste
              valeur (type): valeur du nouveau maillon
           >>> lst = Liste()
           >>> lst.ajoute(1)
           >>> print(lst.tete.valeur)
           >>> lst.ajoute(2)
           >>> print(lst.tete.valeur)
           >>> print(lst.tete.suivant.valeur)
           self.tete = Maillon(valeur, self.tete)
```



```
>>> 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,

3 - 60.0

(0=betqmetts (0=belist)stluseAtaeT : [d

Autres méthodes

()bomtaet

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

```
testmod()
           return longueur(self.tete)
       E == () unaugnol. isl inassa <<<
                    (1) stuojo.ts1 <<<
       S == () rusugnol. isl insess <<<
                    (S) strofp. tsl <<<
       I == () nusugnol. isl inseas <<<
                    (t) stuoin. tel <<<
       0 == () nusugnol. isl inseas <<<
                    () > 1 & i 1 = 1 & i < < <
                            :səldwəxg
        stril bl sh rusugnol :tni
        sənîndə sizil ənu'b rusugnol
               def longueur(self) -> int:
                           class Liste(Liste):
        from operations_base import longueur
(sruos ub & sitrag .ls) sininds steil sau'b #
    "Tusugnol" noitonol al retroqui tual li # :[3]
```

6=bətqmətts (0=bəlisi)atlusəAtaəT : [6

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

```
I == [8]181 179880 <<<
            S == [1]121 175220 <<<
            E == [0]1s1 1msssp <<<
                 () sersusar. tel <<<
                 (1) stuoja. tsl <<<
                 (S) stuojoute(S)
                 (E) 27no[n.1s1 <<<
                 () stsi1 = tsi <<<
         genneuse in liste en place
                    def reverse(self):
                      class Liste(Liste):
       from operations_base import renverser
# d'une liste chaînée (cf. partie 3 du cours)
  restaunar notion of al ratrogni tunt li # :[21]
        ! == [\(\int\)] == [\(\int\)]
        >>> assert lst[1] == 2
         >>   assert lst[0] == 3
               ()erieverse()
               (1) etuois.tal <<<
               (S) etuojs.tal <<<
               (5) etuois.tal <<<
               () etail = tal <<<
                            Exemble:
```

[2] TestResults(failed=0, attempted=8)

self.tete = renverser(self.tete)

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

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

testmod()

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 lonqueur 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 nième 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]: # il faut importer la fonction `lonqueur
      # 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
              >>> 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.