« Subgoal Label Worked Examples »

30 min
30 min

Olivier Goletti & Kim Mens, UCLouvain — DIDAPRO 9, Le Mans, 18 mai 2022
Utiliser des stratégies d'instruction explicites dans l'enseignement de la programmation

L'apprentissage par *subgoals*, ou **sous-étapes**, améliore les performances de résolution de problèmes des novices. Les *worked examples*, ou **exemples résolus**, permettent d'exemplifier une **stratégie de résolution** de problème au travers d'exemples. En annotant les exemples résolus avec des *labels*, ou **étiquettes**, qui correspondent aux sous-étapes, on combine ces deux outils pour mettre en évidence les sous-étapes génériques et fonctionnels que les novices ont du mal à identifier seuls.

Avantages :

L'apprentissage par *sous-étapes* :

- Facilite l'automatisation d'une *stratégie de résolution* de problèmes, au travers de *sous-étapes* bien définis ;
- Aide au transfert et à l'organisation des apprentissages grâce aux *étiquettes*, en identifiant les *sous-étapes* génériques dans les *exemples résolus* ;
- Allège la charge cognitive, en mettant en avant les *sous-étapes* génériques, au lieu des détails superficiels ;
- Égalise les performances entre les étudiants ;
- Améliore le taux de rétention des étudiants.

Quand ?

- Dès l'introduction d'un nouveau concept de programmation ;
- Avec des étudiants ayant peu de connaissances en programmation ;
- Avec des étudiants rencontrant des difficultés à identifier les étapes à suivre pour lire et utiliser ces concepts de programmation ;
- Particulièrement efficace pour les structures de données.

Comment :

- Expliquer la *stratégie de résolution* ;
- Exemplifier la *stratégie de résolution* ;
- Utilisation systématique des bonnes étiquettes en fonction du concept de programmation en jeu ;
- Simplifier si nécessaire, avec moins de *sous-étapes* et plus de *sous-sous-étapes* ;
- Alterner entre présentation d'*exemples résolus* et résolution d'exercices.

Observations intéressantes :

- « Les débutants résolvent mieux les problèmes de programmation lorsqu'ils apprennent explicitement les sous-étapes d'un concept de programmation, car ils ne reconnaissent souvent pas ces éléments fonctionnels par eux-mêmes. » [3]
- « Les automatismes des instructeurs ne sont pas toujours évident à transmettre à leurs étudiants, parce qu'ils pensent que c'est une connaissance commune ou parce qu'ils pensent que c'est intuitif. » [1]

- « Les étiquettes des sous-étapes mettent en évidence la structure du concept de programmation et la façon de les résoudre. Ce qui permet aux élèves de plus facilement reconnaître les similitudes entre les solutions. » [1]
- « L'apprentissage des sous-étapes s'est avéré efficace car il aide les élèves à reconnaître les structures abstraites des concepts de programmation, et cela avant d'avoir suffisamment de connaissances pour les reconnaître par eux-mêmes. » [1]

Références :

1. Lauren Margulieux, Briana B. Morrison, Adrienne Decker. Design and Pilot Testing of Subgoal Labeled Worked Exemples for Five Core Concepts in CS1. In *Proceedings of the 24th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE'19), 2019.
2. Olivier Goletti, Kim Mens and Felienne Hermans. Tutors' Experiences in Using Explicit Strategies in a Problem-Based Learning Introductory Programming Course. *In Proceedings of the 26th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE '21), 2021.
3. Richard Catrambone. The Subgoal learning model: Creating better examples so that students can solve novel problems. Journal of Experimental Psychology: General, 127(4), 355. 1998.
4. ITiCSE conference series (Innovation and Technology in Computer Science Education) : https://iticse.acm.org/

Comment utiliser les SLWE ?

Lecture

- Sert à automatiser l'identification de sous-étapes génériques pour différents concepts de programmation lors de la lecture d'un code :

  - Annoter au tableau un code de référence à lire avec les étiquettes des sous-étapes génériques du concept de programmation ;
  - Expliquer à voix haute comment les sous-étapes sont identifiés ;
  - Suivre ces sous-étapes en traçant le code de référence ;
  - L'exemple annoté servira de modèle par la suite.

Écriture

- Sert à automatiser l'utilisation de sous-étapes pour différents concepts de programmation lors de l'écriture de code :

  - Écrire les étiquettes des sous-étapes génériques du concept de programmation au tableau ;
  - Expliquer à voix haute comment les sous-étapes s'utilisent dans le cas de l'exemple ;
  - Construire le code sur base des sous-étapes ;
  - L'exemple annoté servira de modèle par la suite

Lecture

Affectation

1. Pour chaque sous-expression à droite du signe égal :
   a. Évaluer le **type** de la sous-expression ;
   b. Déterminer l'ordre de priorité des opérateurs ;
   c. Effectuer les opérations en fonction du type des opérandes.
2. **Évaluer** l'expression finale et son type
3. **Affecter** (assigner) la valeur à la variable

```
a = input ()
b = input ()
result = a + b
```

① type
② évaluer
③ affectation

	Name	Val
	a	"3"
	b	"4"
③ →	result	"34"

Condition

1. **Rassemblez** les expressions qui vont ensemble
2. **Evaluer** la condition
3. **Tracer** la branche sélectionnée :
   a. Si `True`, suivre la branche vraie,
   b. Si `False`, suivre les branches `elif` suivantes ou `else` s'il y en a une. Ne rien faire sinon.

```
a = 2
b = 3
if a > b:
    min = b
else :
    min = a
print (min)
```

② eval cond
2 > 3 vaut False
① if - block
else - block
③ tracer branche

	name	Val
	a	2
	b	3
→	min	2

Boucle

1. **Identifier** la boucle
   a. Instruction de **début**
   b. Instruction de mise à jour (**update**)
   c. Condition de **fin** (ou de continuation)
   d. **Corps** qui sera répété
2. **Tracer** la boucle

```
names = ["Olivier", "Kim", "Florian"]
for name in names :
    print ("Bonjour " + name)
```

① Identifier
Corps

stdout :
Bonjour Olivier
Bonjour Kim
Bonjour Florian

② Tracer

	names	Val
	names	↳ "Olivier", "Kim", "Florian"
	name	~~"Olivier"~~ ~~"Kim"~~ "Florian"

Départ
fin

```
i = 1
sum = 0
While i <= n:
    sum = sum + i
    i += 1
```

① Identifier
fin : i > n
update

② Tracer (pour n = 3)

i	~~1~~ ~~2~~ ~~3~~ 4
sum	~~0~~ ~~1~~ ~~3~~ 6

Fonction

1. **Lire** la spécification, l'en-tête et le corps de la fonction
2. Vérifier les **arguments** dans l'appel de la fonction
   a. le nombre d'arguments respecte l'en-tête
   b. les valeurs et types des arguments respectent les spécifications
3. Déterminer l'**effet** de la fonction (valeur et type de retour, effet de bord, affichage, changement d'état)
4. **Tracer** le corps de la fonction

```
def affiche(n):
    """
    pre : n est un entier positif
    post : imprime n fois le caractère
           'x' sur une ligne
    """
    caract = 'x'
    print(caract * n)   <- (3) Effet : imprime un string de longueur n

n = 9
affiche(n)   (2) args : 9 est un entier > 0
                        OK
```

(1) lire

(4) Tracer

⟹ imprime "xxxxxxxxx"

```
def pair(n):
    """
    pre : n > 0
    post : retourne True si le nombre passé
           en argument est pair,
           False sinon
    """
    reste = n % 2
    return (reste == 0)   <- (3) effet : retourne un booléen

result = pair(33/3)   (2) args : 1 argument > 0
                          OK
```

(1) lire

(4) Tracer

pair	Name	Value
	n	11
	reste	1

main		
	result	False

Ecriture

Affectation

1. Déterminer **l'expression** pour calculer la valeur à affecter ;
2. Déterminer **l'identifiant** de la variable de retour ;
3. Écrire l'**expression Python** avec les bons opérateurs ;
4. Vérifier que les types des opérandes soient bien **compatibles** avec les opérateurs ;
5. Vérifier si la valeur de l'expression est du **type** attendu.

Calculez la moyenne m des variables a, b, c

$m = \frac{a+b+c}{3}$ ② id ① Expression lang. nat.

m = (a + b + c) / 3 ③ Expression Python

④ Compatible

⑤ Type ? float

Condition

1. Définir tous les **cas** mutuellement exclusifs nécessaires ;
2. Les **ordonner** pour faciliter la lisibilité des conditions ;
3. Écrire if, la **première condition**, « : » et les instructions du cas `True` indentées ensuite ;
4. Si vous avez plusieurs conditions, faites de même pour chacune avec **`elif`** ;
5. Finir si nécessaire par **`else`**, « : » et les instructions correspondantes.

Calculez la valeur absolue abs de x

x ≥ 0 ou x < 0 ① cas

```
if x < 0 :
    abs = -x
```

② ordre ③ Cond. 1

```
else :
    abs = x
```

⑤ else

Boucle

1. **Choisir** la structure de boucle appropriée (`while`, `for`) ;
2. Définir et **initialiser** les variables ;
3. Déterminez la **condition d'arrêt** *(ou de continuation)*
4. Écrire le **corps** de la boucle *(mettre à jour la variable de contrôle jusqu'à la condition d'arrêt si nécessaire).*

Calculez la somme Sum des n premiers entiers positifs

① Choisir

```
i = 1
Sum = 0
While (i <= n) :
    Sum += i
    i += 1
```

② init ③ arrêt ④ Corps

```
Sum = 0
for i in range(1, n+1):
    Sum += i
```

Fonction

1. Écrire l'**en-tête** de la fonction
   a. Choisir l'identifiant de la fonction pour qu'il traduise l'intention (snake_case)
   b. Choisir le nombre de paramètres et les identifiants pour qu'ils traduisent leur rôle
2. Rédiger les **spécifications** de la fonction
   a. Donner les préconditions sur les paramètres de la fonction et leur type
   b. Donner les postconditions décrivant l'effet, la valeur et le type de retour de la fonction
3. Écrire le **corps** de la fonction
   a. Déterminer la logique du programme (affectation, condition, boucle, etc.)
   b. Déclarer les variables locales nécessaires
   c. Écrire les instructions du programme
4. Terminer par un **return** en fonction des spécifications
   a. Il faut un return pour chaque chemin d'exécution dans la fonction
   b. Pas de return est équivalent à retourner None

```
def calcule_max(a, b):                    (1) en-tête
    """
    pre : a et b sont des nombres réels
    post : retourne le maximum entre les
           deux réels a et b              (2) Spécifications
    """
    if a > b:
        max = a                           (3) Corps
    else :
        max = b
    return max                            (4) return
```

Parcours de chaîne ou de liste

1. Lire et comprendre **l'énoncé**
2. Écrire un **test** préalablement pour valider la compréhension et l'implémentation a posteriori
3. Déterminer s'il faut itérer sur les **indices** (`for index in range(len(l))`) ou sur les **éléments** (`for element in l`)
4. Exprimer la **condition de traitement** (ou de sortie de boucle si la liste ne doit pas être parcourue en entier)

```
def maximum_index(l):
    """
    @pre : une liste non-vide l d'entiers
    @post : retourne l'indice de la dernière
            occurence du plus grand entier
            dans la liste
    """
    max_val = l[0]
    max_index = 0
(5) for index in range(len(l)):           (3) Indices
        if l[index] >= max_val :          (4) Condition de traitement
            max_val = l[index]
            max_index = index
    return max_index
Boucle
maximum_index([1,2,3,2,1])                (2) Test
# 2
```

5. Écrire la **boucle** en se rappelant des sous-étapes correspondants
6. Vérifier les **cas particuliers** (liste vide, bornes de début et fin de parcours).

```
def compter (s,c):
    """
    @pre : une chaîne de caractères s,
           un caractère c
    @post : retourne le nombre d'occurences
            de c dans s
    """
    compteur = 0
    for car in s :          (3) éléments
        if car == c :       (4) Condition de traitement
            compteur += 1
    return compteur

compter ("banana", "a")
# 3
```

(5) Boucle

(2) Test

Lecture de fichier | *Écriture de fichier*

1. **Ouvrir** le fichier
   a. Identifier le nom et le chemin du fichier (typiquement dans `filename`)
   b. Choisir le mode `"r"` | b. Choisir le mode `"w"`
   c. Ouvrir le fichier avec :
      ```
      with open(filename , mode) as f :
      ```
      ou
      ```
      f = open(filename , mode)
      ```
2. **Traitement** du fichier en fonction du format

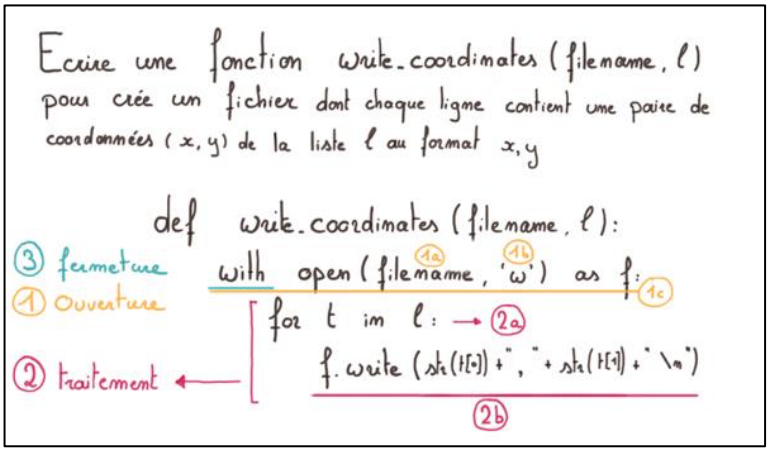
   a. Parcours des lignes
   - ligne par ligne avec `f.readline()`
   - en itérant sur les lignes avec
     `for line in f:`
     ou
     `for line in f.readlines())`

   b. Traitement des lignes
   - Retrait des blancs en début et fin de ligne (`line.strip()`)
   - Séparer les éléments en fonction du format (`line.split()`)
   - Convertir les éléments en fonction du type attendu

   c. Traiter les erreurs de formatage du fichier (ignorer ligne, raise ValueError)

   a. Trouver l'expression pour écrire une ligne en fonction du format par concaténation
   b. Parcourir la structure de données et écrire les lignes au fur et à mesure
   - Avec `f.write()`
   - Prendre en compte les retours à la ligne

3. **Fermeture** du fichier
   - Avec un `with`, il n'y a rien à faire
   - Sinon, avec un `f.close()`
4. Gérer les **exceptions** susceptibles de se produire durant le traitement du fichier (typiquement `IOError`)
   a. Mettre le code dans un `try : … except :`
   b. Traiter les exceptions par le suite avec des `except error_type: …`



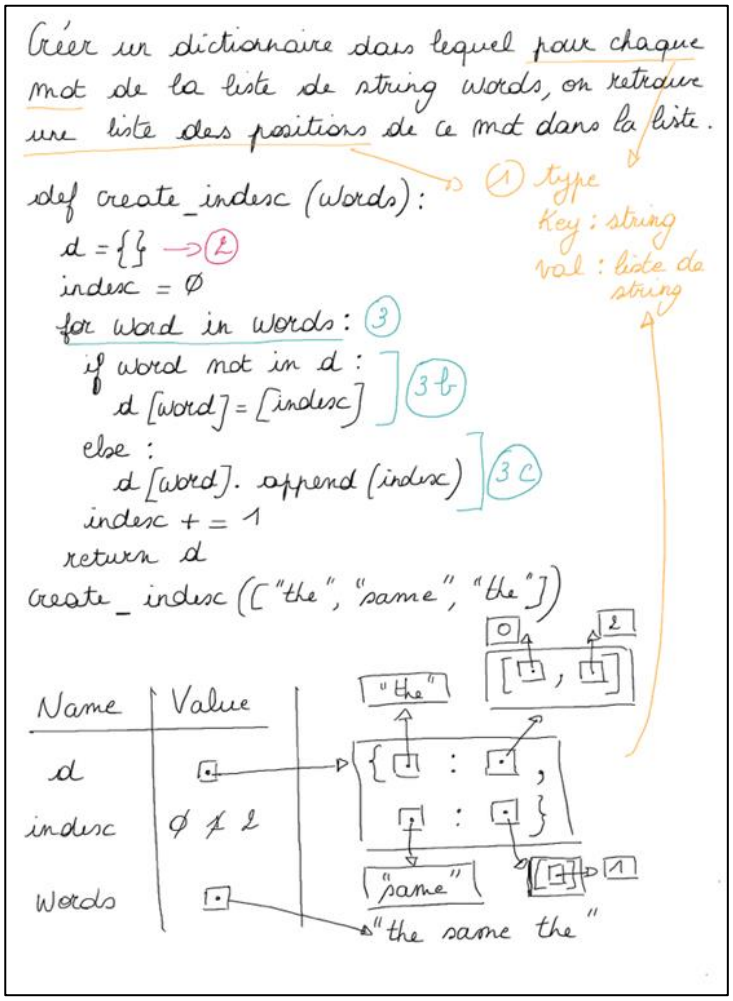
Ecrire une fonction read_coordinates(filename) qui lit les coordonnées du fichier nommé filename dont chaque ligne est au format x,y et retourne une liste de tuples (x, y)

```
def read_coordinates(filename):
    l = []
    try:                                  (1a) (1b)
        with open(filename, 'r') as f:
            for line in f:                (2a)
                tokens = line.strip().split(',')   (2b-c)
                if len(tokens) != 2
                    raise ValueError(
                        "Il faut deux valeurs par ligne
                        séparées par une virgule")
                l.append(float(tokens[0]), float(token[1]))   (2b)
    except IOError:
        return []
    return l
```

(3) fermeture
(1) ouverture
(2) traitement
(2c)
(4) Exceptions

Création/mise à jour de dictionnaire

1. Déterminer le **type** des **clés** et des **valeurs** du dictionnaire, éventuellement à l'aide d'un dessin ;
2. **Créer** un dictionnaire vide (`d={}`) s'il n'existe pas encore ;
3. Pour **chaque** clé à modifier dans le dictionnaire, vérifier si la clé existe dans le dictionnaire
   a. Établir **l'expression** qui donnera la valeur à mettre dans le dictionnaire
   b. Si non, **ajouter** la clé au dictionnaire avec une valeur par défaut en fonction du type des valeurs (`d[key] = default_val`) ou directement la bonne valeur
   c. Si oui, **mettre à jour** la valeur correspondante à la clé (nouvelle valeur ou valeur modifiée en fonction, par exemple `d[key] = new_val` ou `d[key].append(new_elem)`)



Création de Classes

1. Bien **nommer** la classe
   a. Utiliser un substantif qui décrit le type d'objet
   b. Utiliser du `CamelCase` si le nom est composé de plusieurs mots

2. Déterminer les **attributs** et **méthodes** en fonction de ce que représente cette classe :
   a. Déterminer le nombre et les types des attributs
   b. Déterminer le nombre et les objectifs des méthodes

3. Créer la méthode d'initialisation **`__init__`**
   a. **Entête** :
      i. Ajouter `self` comme premier paramètre
      ii. Déterminer les types et les identifiants des paramètres pour qu'ils traduisent leur rôle
   b. Pour chaque attribut nécessaire :
      i. Choisir un identifiant qui traduise son rôle (le même que le paramètre associé si possible)
      ii. Trouver l'expression pour l'initialiser en utilisant (ou non) les paramètres (`self.attr = expression`)

4. Pour chaque autre **méthode** d'instance
   a. Ecrire l'**entête** :
      i. Donner un identifiant en `snake_case` à la méthode qui traduise son rôle
      ii. Ajouter `self` comme premier paramètre
      iii. Déterminer les types et les identifiants des paramètres pour qu'ils traduisent leur rôle
   b. Ajouter au moins un **test** pour cette méthode
   c. Dans le **corps** de la méthode :
      i. Utiliser `self.attribut` pour accéder à la valeur d'un attribut
      ii. Pour appeler une autre méthode de la même classe écrivez `self.autre_méthode()`
      iii. Si la méthode doit retourner un résultat ne pas oublier return et bien réfléchir à l'expression et au type de la valeur retournée

5. Créer la méthode **__str__**
   a. **Entête** :
      i. Écrire l'entête `def __str__(self)`
   b. Déterminer l'expression pour représenter l'objet au format désiré en utilisant les bons attributs
      i. Concaténer les attributs via `+ self.nom_de_variable +`
      ii. Attention à bien convertir des valeurs qui ne seraient pas des strings avec l'instruction str(otherType)

      Alternativement, utiliser `.format()` pour bien formatter le string à retourner.

6. Créer la méthode **__eq__**
   a. **Entête** :
      i. Écrire l'entête `def __eq__(self,autre)`

b. Vérifier que le paramètre autre est bien une instance de cette classe avec `type()` ou `isinstance(autre, ClassName)`
c. Comparer individuellement l'égalité des valeurs des attributs désirés

7. Tester la classe en exécutant les tests que vous avez ajouté pour ses différentes méthodes.

```
class Student:                                   (1) nommer
    def __init__(self, n):                       (4)
        # nom de l'étudiant
        self.name = n
        # score reçu par l'étudiant sur 3 tests
        # (initialement None car l'étudiant n'a
        # pas encore passé les tests)
        self.test1 = None
        self.test2 = None
        self.test3 = None                        (4b)

    def average_score(self):                     (3a) entête
        return (self.test1 + self.test2 + self.test3) / 3
                                                 (3c) acces

    def __str__(self):                           (5)
        s = "Bonjour, " + self.name + \
            "Vos scores sont: " + "\n" + \
            str(self.test1) + "\n" + \
            str(self.test2) + "\n" + \
            str(self.test3) + "\n" + \
            "Votre score moyenne est " + str(self.average_score())
        return s                                 (5b) i, ii

    def __eq__(self, other):                     (6)
        if not isinstance(other, Student):       (6b) instance
            return False
        return self.name == other.name and \
               self.test1 == other.test1 and \
               self.test2 == other.test2 and \
               self.test3 == other.test3         (6c) Comparaison
```

Listes Chainées

Le code ci-dessous fait référence au code de la classe LinkedList du cours INFO1 en Python, et se base sur des exemples tirés de l'implémentation de la Mission 11, phase de réalisation. Il s'agit d'une implémentation d'une liste chainée ordonnée, qui hérite de la classe LinkedList.

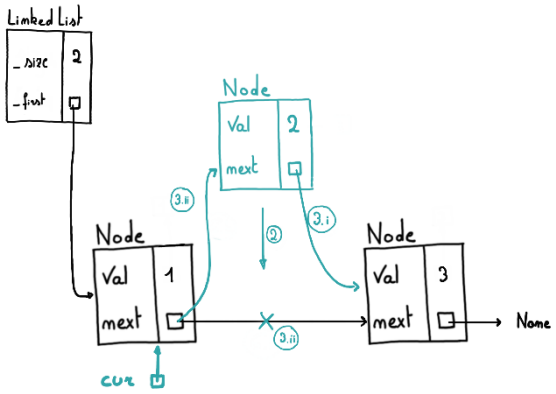
ajout d'un noeud

1. Crée un **nouveau nœud** (`new_node`) composé de la valeur à ajouter dans la liste et un paramètre next (voisin) instancié a None.
2. Déterminer l'**emplacement** où insérer `new_node`
   a. Si **tête de liste** (liste vide ou premier élément) => 3.a
   b. Sinon **parcours** nécessaire
      i. Créer une référence vers **l'élément courant** (`cur`, instancié à self.`first`)
      ii. Parcours tant qu'il y a un nœud suivant
         1. Si **parcours terminé** (fin de liste) => 3.c
      iii. Vérifier la **condition d'insertion** sur la valeur du nœud après `cur`
         1. Si la condition est vérifiée => 3.b
      iv. **Mettre à jour** la variable d'itération
3. **Insérer** `new_node`,

/!\ l'ordre des changements à tout une importance

   a. Si en **tête** de liste :
      i. `new_node.next` pointe vers l'ancienne tête de liste
      ii. a ref self.`first` pointe vers `new_node`
   b. Si en **milieu** de liste :
      i. `new_node.next` pointe vers le nœud qui suit le nœud courant
      ii. le nœud courant pointe vers `new_node`
   c. si en **fin** de liste :
      i. le nœud courant est le dernier et pointe vers `new_node`
4. **Incrémenter** la taille de la liste chainée

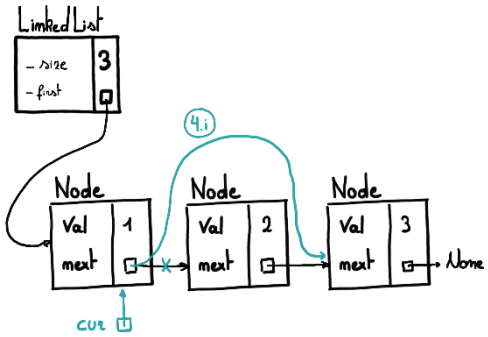
```
def add (self, value):
    ① Nouveau noeud   new_node = Node(value)
    ② Tête de liste   if self.size() == 0 or value < self.first().value():
                      # cas particulier, liste vide ou tête de liste
    ③a Insertion        i| new_node.set_next(self.first())
                        ii| self.set_first(new_node)
                      else:
                      # cas général
    ②b Parcours         i| cur = self.first()
                        ii| while cur.next() is not None:
                        iii|     if value < cur.next().value():
                                 # milieu de liste
    ③b Insertion                 i| new_node.set_next(cur.next())
                                 ii| cur.set_next(new_node)
    ④ Incrémentation             self.inc_size()
                                 return
                        iv|      cur = cur.next()
                                 # fin de liste
    ③c Insertion        i| cur.set_next(new_node)
                        self.inc_size()
    ④ Incrémentation
```



retrait d'un noeud

1. Vérifier la **taille** de la liste chainée
2. Vérifier la **valeur** de l'élément à retirer
3. Déterminer l'**emplacement** du nœud à supprimer
   a. Si **tête de liste** (liste avec un seul élément ou première élément) => 4.a
   b. Sinon **parcours** nécessaire
      i. Créer une référence vers **l'élément courant** (`cur`, instancié à self.`first`)
      ii. Parcours tant qu'il y a un noeud suivant
         1. Si **parcours terminé** (fin de liste) => pas dans la liste
      iii. Vérifier la **condition de suppression** sur la valeur du nœud après `cur`
         1. Si la condition est vérifiée => 4.b
      iv. **Mettre à jour** la variable d'itération
4. **Retirer** le nœud correspondant
   a. Si en **tête** de liste :
      i. `Self.first` pointe vers `cur.next`
   b. Si en **milieu** de liste :
      i. `cur.next` point vers `cur.next.next`
5. **Décrémenter** la taille de la liste chainée

```
def remove(self, value):
    # cas particulier : liste vide
① Taille    if self.size() == 0 :
                return False
② Valeur    if value == None :
                return False
    # cas général
    cur = self.first()
    # cas particulier : tête de liste
3.a Tête de liste  if self.first().value() == value :
④.a Retrait            self.set_first(self.first().next())
⑤ Decrementation       self.dec_size()
                       return True
    # milieu de liste
3.b Parcours  while cur.next() is not None :
                  if cur.next().value() == value :
④.b Retrait           cur.set_next(cur.next().next())
⑤ Decrementation      self.dec_size()
                      return True
    # la valeur n'est pas dans la liste
    return False
```



10 min

Mise en situation

En illustrant et utilisant les labels des boucles : crée un programme python qui retourne la somme des *n premier* entiers pairs strictement positifs en utilisant les étiquettes des concepts utilisés. (Exercice tiré de Mission 2 – Démarrage)

15 min

Mise en situation

En illustrant et utilisant les labels des classes : écrivez une classe complète Employe dont les instances représentent un employé d'entreprise. Un employé est caractérisé par son nom (un string) et son salaire (un entier positif). Il doit être possible de créer un nouvel employé avec un nom et un salaire et d'effectuer les opérations suivantes sur un objet Employe : obtenir le nom de l'employé ; obtenir son salaire ; augmenter son salaire d'un certain pourcentage ; et obtenir un texte descriptif représentant l'employé sous la forme "nom : salaire".
(Exercice tiré de Mission 8 – Bilan Final)