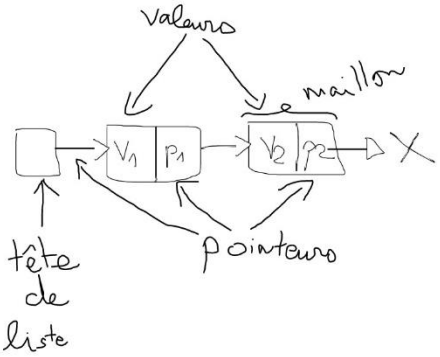
NSI : les listes chaînées

Durée :	3 séances
Classe :	Entière
Objectifs :	Comprendre comment implémenter des listes chaînées et effectuer des opérations

1. Définition

Une **liste chaînée** est une structure très simple : elle est uniquement définie par son **premier maillon** (plus précisément, une référence vers son premier maillon). Elle peut comporter d'autres **maillons**. Chaque maillon comporte une **valeur**, et une **référence au maillon suivant**.



S'il n'y a pas de maillon suivant, cette **référence** est à **None**.

Quand on dit « une référence vers », cela s'appelle aussi un **pointeur**. Un pointeur peut être vu comme une **adresse mémoire** qui localise un élément. On n'a pas besoin de « voir » cette adresse, bien que la fonction Python `id()` nous la donne.

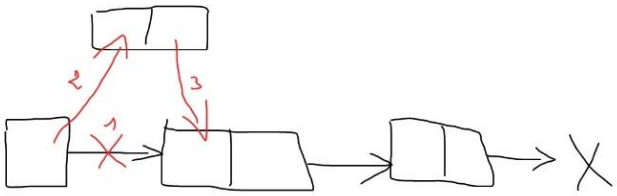
Pédagogiquement on va utiliser des valeurs littérales appartenant aux types de base (entiers, flottants, booléens, chaînes de caractères) mais on peut imaginer des valeurs beaucoup plus complexes : listes, dictionnaires, autres listes chaînées…

Ainsi, on ne peut **jamais accéder directement** à un élément d'une liste chaînée : on doit pour cela la parcourir !

Une liste chaînée a de nombreux avantages (et aussi inconvénients), le premier étant, sa flexibilité au niveau de sa taille et de la gestion de la mémoire.

Un exemple assez parlant est l'insertion d'un élément en début de liste :

- Dans une liste classique Python on doit donc parcourir toute la liste depuis la fin vers le début, et à chaque fois faire un décalage : l'élément à l'indice n-1 va à l'indice n, ainsi de suite jusqu'à l'élément à l'indice 0 que l'on va placer à l'indice 1. Enfin on mettra à l'indice 0 le nouvel élément. Donc il faut ajouter avec `append()` un élément à la liste, décaler tous les éléments 1 à 1, puis insérer à l'indice 0 le nouvel élément. Ce temps de parcours est proportionnel à la taille de la liste.
- Dans une liste chaînée c'est bien plus simple. On doit créer une méthode qui insère un élément au début de la liste. Cette méthode fait en sorte que le début de la liste pointe vers le nouvel élément, et que ce nouvel élément pointe vers l'ancienne tête de liste (voir schéma ci-contre).



2. Implémentation des classes et des premières méthodes

Pour implémenter une liste chaînée dans sa forme la plus simple, on va avoir besoin de deux classes :

- ListeChainee : contient un attribut qui est un pointeur vers son premier élément (qui est un maillon). Elle contient une méthode **ajouter(m)** qui prend en paramètre un maillon **m**, et une méthode (optionnelle car son code est très simple) permettant de dire si elle est vide ou non. On peut lui ajouter une méthode **parcours()**, qui peut être itérative ou récursive, dont le corps ressemble à **ajouter()**. Cette classe n'est instanciée qu'une fois si on n'a qu'une liste !
- Maillon : contient un attribut (disons, de type entier) et un pointeur vers l'élément suivant, pouvant valoir **None**. Elle contient aussi une méthode d'affichage

Vous retrouverez tout cela dans le fichier **`ListeChainee.py`**.

```
class Maillon:

    # un maillon a forcément une valeur et pas forcément de maillon suivant
    def __init__(self, val, suiv = None):
        self.valeur = val
        self.suivant = suiv

    #remarquez que si on print() un objet (self.suivant) on obtient son adresse mémoire
    def affiche(self):
        print("Valeur : "+str(self.valeur)+", adresse du suivant : "+str(self.suivant))
```

Et c'est tout pour la classe `Maillon` !

Dans certaines implémentations des listes chaînées, seule la classe `Maillon` est utilisée. L'intérêt selon moi d'avoir une classe `ListeChainee` est qu'on peut lui ajouter plusieurs attributs : par exemple, sa taille, qui sera mise à jour en cas d'ajout/suppression.

Dans sa plus simple expression, `ListeChainee` se déclare ainsi :

```
class ListeChainee:

    # le premier élément est de type maillon, par défaut None (liste vide)
    def __init__(self, premierElement = None):
        self.pe = premierElement

    # teste si la liste est vide
    def estVide(self):
        return self.pe == None
```

Avec ces deux classes on peut déjà créer une liste chaînée et la remplir. Pour cela, on crée un nouveau fichier nommé par exemple `testListesChainees.py` et on y importe nos classes (attention il doit se trouver dans le même répertoire).

```
from ListeChainee2024 import *

lc = ListeChainee()
m1 = Maillon(12)
m2 = Maillon(24)
m3 = Maillon(48)
lc.pe = m1
m1.suivant = m2
m2.suivant = m3 #équivalent à : m1.suivant.suivant=m3
```

Quand on se retrouve avec plus d'une dizaine de maillons, ce travail peut devenir vite fastidieux si on le fait ainsi, car il faut créer manuellement à chaque fois une instance de `Maillon` et la placer dans une variable. **Gardons ça à l'esprit et continuons.** Nous y reviendrons plus tard dans une discussion.

3. Parcours de la liste

a) Itératif

On veut maintenant **parcourir** notre liste. La méthode `parcours()` se crée naturellement dans la classe `ListeChainee`. On va tout d'abord en faire une **version itérative**. L'algorithme de parcours est très simple :

1. On regarde si la liste possède un premier élément, sinon on affiche « Liste Vide »
2. Si oui, on dit que c'est l'élément courant.
3. On appelle `affiche()` sur ce maillon que son suivant n'est pas égal à `None`
4. On dit que l'élément courant est le suivant de ce maillon
5. Si en revanche son suivant est `None` (sortie de boucle `while`), on affiche ce maillon (qui donc est le dernier)

```
#parcours itératif
def parcours(self):
    if not(self.estVide()):
        maillonCourant = self.pe
        while maillonCourant.suivant != None:
            maillonCourant.affiche()
            maillonCourant = maillonCourant.suivant
        maillonCourant.affiche()#affiche le dernier maillon qui pointe vers None
    else:
        print("Liste vide")
```

On peut maintenant tester le parcours toujours dans notre fichier de test :

```
lc.parcours()
```

Donne comme exécution :

```
Valeur : 12, adresse du suivant : <ListeChainee2024.Maillon object at 0x000001C0F8829FD0>
Valeur : 24, adresse du suivant : <ListeChainee2024.Maillon object at 0x000001C0F882A180>
Valeur : 48, adresse du suivant : None
```

b) Récursif

On peut maintenant penser à créer une **méthode récursive de parcours**. Comme vous l'avez compris (c'est assez intuitif), cette méthode va se rappeler de maillon en maillon, il faudra donc deux choses pour ce parcours récursif :

- une méthode dans ListeChainee qui ne sera appelée qu'une fois
- une méthode dans Maillon qui va s'appeler récursivement

La première (**dans `ListeChainee`**) est simplissime, on teste si la liste est vide et on affiche dans ce cas « Liste Vide », sinon on appelle une méthode (on aurait pu lui donner le même nom) de parcours sur Maillon :

```
#parcours récursif
def parcoursRec(self):
    if self.estVide():
        print("Liste vide")
    else:
        self.pe.parcoursR()
```

La seconde (**dans `Maillon`**) est assez intuitive elle aussi :

```
#parcours récursif
def parcoursR(self):
    self.affiche() #on affiche le maillon

    #on rappelle sur son suivant s'il existe
    if self.suivant : #raccourci pour dire self.suivant != None
        self.suivant.parcoursR()
```

On est dans le cas d'une condition d'arrêt implicite. Le test est aussi simple à faire que le premier :

```
lc.parcoursRec()
```

Et le résultat identique :

```
Valeur : 12, adresse du suivant : <ListeChainee2024.Maillon object at 0x0000025A1B2ED610>
Valeur : 24, adresse du suivant : <ListeChainee2024.Maillon object at 0x0000025A1B3112B0>
Valeur : 48, adresse du suivant : None
```

Note : j'ai rebooté entre temps mon IDE (VSCode) c'est la raison pour laquelle les adresses en mémoire ont changé.

4. Ajout d'un élément à la liste

Par ajout d'un élément, on entend « en dernière position ». Ainsi, ajouter un élément à une liste chaînée revient à ajouter un maillon en bout de chaîne. On devra donc, ici aussi, parcourir la liste jusqu'à son dernier élément avant d'ajouter un nouveau maillon. Si vous avez compris comment parcourir une liste, l'ajout d'un nouveau maillon ne devrait poser de grandes difficultés.

À la fin du 2 je vous avais dit de garder à l'esprit qu'il pouvait être fastidieux de créer à chaque fois manuellement des maillons, nous en parlerons à la fin de cette partie.

a) Itératif

Ici l'algorithme est simple. En paramètre, notre méthode `ajout()` prend un objet de type Maillon :

1. Si la liste ne possède pas de premier élément, on y place notre maillon.
2. Si elle en a un, on dit que c'est l'élément courant.
3. Tant que son suivant n'est pas égal à `None`, on « avance »
4. Si on tombe sur un suivant qui est `None` (sortie de boucle `while`), on y place le nouveau maillon (qui devient par conséquent le dernier)

Notez les énormes similitudes avec le code de la méthode `parcours()` et les redondances…

Question : comment les supprimer / les réduire ?

```
# ajoute un maillon en itératif
def ajouteMaillon(self, m):
    if self.estVide():
        self.pe = m
    else:
        # initialisation : le premier maillon est le maillon courant
        maillonCourant = self.pe

        # on avance tant que le maillon courant contient une référence vers un autre
        while maillonCourant.suivant != None:
            maillonCourant = maillonCourant.suivant

        # ici on arrive à un maillon qui n'a pas de suivant : c'est la queue...
        # on lui accroche le nouveau maillon !
        maillonCourant.suivant = m
```

On peut maintenant éviter les syntaxes avec d'innombrables points représentant de fastidieuses navigations entre les objets. Il suffit de créer nos maillons, et de les passer en paramètre de la méthode `ajout()` que l'on appelle toujours sur l'instance de `ListeChainee`.

```
lc = ListeChainee()
m1 = Maillon(12)
m2 = Maillon(24)
m3 = Maillon(48)
lc.ajouteMaillon(m1)
lc.ajouteMaillon(m2)
lc.ajouteMaillon(m3)
```

Faites vous-même le test du parcours (itératif ou récursif) que je n'ai pas fait figurer ici pour alléger le document.

b) Récursif

Ici encore on peut facilement créer une méthode d'ajout récursif. Le raisonnement est encore une fois très similaire à celui du parcours. On distingue deux cas :

- La liste est vide : on procède exactement comme en itératif et on lui accroche le maillon
- La liste n'est pas vide : on va à la fin en la parcourant de maillon en maillon (donc on doit créer une méthode d'ajout dans la classe `Maillon`)

Dans la classe **`ListeChainee`** on a :

```
# ajoute un maillon en récursif
def ajouteMaillonRec(self,m):
    if self.estVide():
        self.pe = m
    else:
        self.pe.ajouteMaillonR(m)
```

Et dans la classe **`Maillon`** on a :

```
#ajout récursif
def ajouteMaillonR(self,m):
    if self.suivant : #raccourci pour dire self.suivant != None
        self.suivant.ajouteMaillonR(m)
    else:
        self.suivant=m #condition d'arrêt
```

On teste tout ça :

```
lc = ListeChainee()
m1 = Maillon(12)
m2 = Maillon(24)
m3 = Maillon(48)
lc.ajouteMaillonRec(m1)
lc.ajouteMaillonRec(m2)
lc.ajouteMaillonRec(m3)
```

Faites vous-même encore une fois le test du parcours (itératif ou récursif).

c) Discussion

On arrive à la discussion que je vous avais promise : comment procéder pour créer un grand nombre d'éléments ?

Pour le comprendre vous devez vous convaincre qu'une fois qu'une instance de maillon est créée, elle doit être placée dans la liste chaînée. Et donc on ne doit même pas pouvoir y accéder individuellement, car une liste chaînée n'est pas une liste classique ! On doit donc, pour atteindre un élément, traverser la liste de maillon en maillon depuis son premier élément.

Ainsi, dans les exemples ci-dessus, on a explicitement nommé nos maillons `m1`, `m2`…mais on ne devrait même pas avoir à faire ça car la définition des listes chaînées nous interdit d'accéder à un élément directement !

Ainsi à terme on doit pouvoir faire quelque chose de ce genre (j'ai choisi des ajouts et un parcours récursifs mais on aurait tout aussi bien pu les faire en itératif ou bien une combinaison des deux) :

```
import random

lc = ListeChainee()

for i in range(100):
    m = Maillon(random.randint(1,10000))
    lc.ajouteMaillonRec(m)

lc.parcoursRec()
```

De la sorte, il m'est impossible d'accéder directement au nième élément de la liste.

5. Accéder au nième élément d'une liste chaînée

Je peux par contre créer une méthode qui le fait. À moi de m'assurer que le maillon existe (par exemple renvoyer None s'il n'existe pas). Encore une fois, j'ai le choix du paradigme, qu'il soit itératif ou récursif ☺

a) Itératif

Je vais donc créer une méthode dans ListeChainee qui prendra en paramètre un entier correspondant au maillon que je veux trouver (s'il existe). Par analogie avec les listes classiques, je considère que la liste vide ne possède pas d'élément, et que le premier élément d'une liste non vide se trouve à l'indice 0 (c'est une image, une liste chaînée de par sa définition n'a pas réellement d'indices !)

Je dois donc avancer de maillon en maillon tant que je n'ai pas compté n éléments **ET** tant que je peux avancer (tant que le maillon courant possède un suivant). J'aurais pu créer un compteur i mais je préfère me servir de n pour économiser les variables, et donc je dois compter à rebours.

J'ai 2 manières de sortir de ma boucle : j'ai fini de compter **OU** l'élément courant n'a pas de suivant (les deux sont également possibles : si par exemple je veux le nième élément d'une liste à n éléments !)

Si je finis mon parcours avant d'avoir trouvé l'élément voulu, je renvoie `None`

```
#accéder au nième élément d'une liste
def accesNiemeElement(self, n):
    if self.pe :
        courant = self.pe
        while n>0 and courant.suivant:
            courant = courant.suivant
            n = n - 1

        #1 : je suis arrivé à compter à rebours jusqu'à 0
        if n==0:
            return courant

        #2 : je n'ai pas fini de compter mais je suis sorti du while
        else:
            return None

    #cas de la liste vide
    else :
        return None
```

Dans mon fichier test, avant d'afficher le maillon que j'ai récupéré (i.e. appeler dessus la méthode `affiche()`) je dois m'assurer qu'il existe ! Voici un exemple complet illustrant un test que l'on pourrait faire.

```
from ListeChainee2024 import *
import random

lc = ListeChainee()
for i in range(10):
    m = Maillon(random.randint(1,10000))
    lc.ajouteMaillonRec(m)

print("Parcours complet :")
lc.parcours()
print("Recherche d'éléments :")
l = [5,0,9,10]
for i in l:
    print('élément',i,end='-->')
    m = lc.accesNiemeElement(i)
    if m : m.affiche()
    else : print("l'élément",i,"n'existe pas")
```

J'obtiens la sortie suivante :

```
Parcours complet :
Valeur : 6945, adresse du suivant : <ListeChainee2024.Maillon object at 0x00000238AB6BBE60>
Valeur : 2892, adresse du suivant : <ListeChainee2024.Maillon object at 0x00000238AB6EBC50>
Valeur : 6822, adresse du suivant : <ListeChainee2024.Maillon object at 0x00000238AB6EBC20>
Valeur : 314, adresse du suivant : <ListeChainee2024.Maillon object at 0x00000238AB6EBBF0>
Valeur : 3126, adresse du suivant : <ListeChainee2024.Maillon object at 0x00000238AB6EBB00>
Valeur : 5848, adresse du suivant : <ListeChainee2024.Maillon object at 0x00000238AB6EBAD0>
Valeur : 2585, adresse du suivant : <ListeChainee2024.Maillon object at 0x00000238AB6EBA40>
Valeur : 1909, adresse du suivant : <ListeChainee2024.Maillon object at 0x00000238AB6EB9E0>
Valeur : 5888, adresse du suivant : <ListeChainee2024.Maillon object at 0x00000238AB6EB890>
Valeur : 970, adresse du suivant : None
Recherche d'éléments :
élément 5-->Valeur : 5848, adresse du suivant : <ListeChainee2024.Maillon object at 0x00000238AB6EBAD0>
élément 0-->Valeur : 6945, adresse du suivant : <ListeChainee2024.Maillon object at 0x00000238AB6BBE60>
élément 9-->Valeur : 970, adresse du suivant : None
élément 10-->l'élément 10 n'existe pas
```

b) Récursif

Question : Sauriez-vous faire la même chose mais en récursif ?

6. Insérer un élément ailleurs qu'en bout de liste

On a vu précédemment comment insérer un élément en dernière position. On a vu aussi en cours comment insérer un élément en première position d'une liste chaînée, comme le rappelle le schéma en début de document (dans la partie Définition). C'est ce raisonnement qui nous importe maintenant : insérer un élément en début de liste est très similaire à insérer un élément n'importe où dans la liste.

On sait (cf. précédemment) retrouver, s'il existe, n'importe quel élément d'une liste. À nous de créer une méthode itérative et deux méthodes récursives pour généraliser cela. On pourrait même, à terme, de se passer de la méthode d'ajout. On suppose que si on ne lui passe pas de paramètre, on ajoute l'élément à la fin de la liste, et que sinon on l'ajoute à la position n passée en paramètre.

Question : Implémenter en itératif l'insertion d'un élément en nième position de la liste

Question : Implémenter en récursif l'insertion d'un élément en nième position de la liste

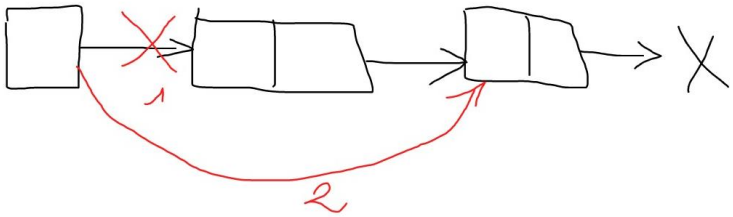
Question : Comparer sur une liste chaînée de 10000 éléments et une liste Python de 10000 éléments le temps nécessaire pour ajouter un élément en :

- Début de liste
- Fin de liste
- Milieu de liste

7. Améliorations

On peut améliorer les listes chaînées de plusieurs manières.

- Puisqu'on sait insérer un élément en fin de liste, et maintenant en nième position, on peut créer la méthode qui supprime un élément de la liste comme le montre le très beau schéma qui illustre comment supprimer le premier élément de la liste.



- Enfin, on aime bien aussi créer des listes **doublement chaînées** (on ajouter un pointeur vers le maillon précédent, ce qui permet un parcours dans les deux sens), **circulaires** (le dernier élément pointe sur le dernier), voir une combinaison des deux…dans le cas d'une liste chaînée circulaire, on doit éviter de tomber dans une boucle (ou une récursion) infinie, car un maillon a toujours un suivant. On doit par conséquent utiliser un marqueur pour s'arrêter lorsque c'est nécessaire.