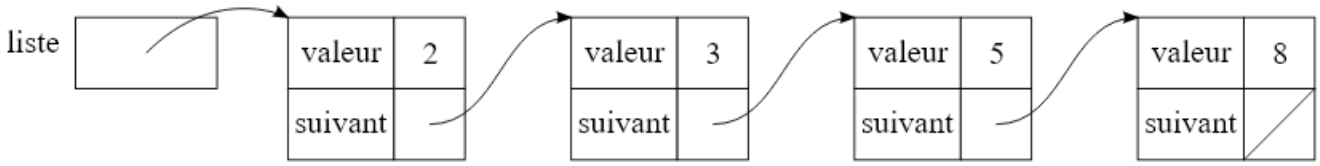
Les listes chaînées

I Description de la structure

On rappelle que la structure d'une liste chaînée ressemble à ceci :



L'illustration montre une liste chaînée composée de 4 maillons. Chaque maillon est composé de 2 champs : une valeur et un pointeur vers le maillon suivant.

Pour implémenter une liste chaînée en python nous pourrions utiliser

- un tableau à 2 éléments décrivant un maillon
  - le premier élément du tableau est la valeur du maillon
  - le second élément est l'indice du maillon suivant (ou None pour le dernier élément)
- un tableau constitué de la liste de tous les maillons

La liste chaînée donnée en illustration pourrait donc être représentée ainsi :

```
maillons = [[3, 2], [8,None], [5, 1], [2,0]]
premier = 3
```

Vous constaterez que j'ai *volontairement* inscrit dans le tableau `maillons` les éléments de la liste dans le désordre car il n'y a aucune raison que l'ordre des éléments corresponde à l'ordre dans le tableau `maillons` si par exemple on insère en cours de route des éléments au milieu de la liste.

la variable `premier` contient l'indice du premier élément de la liste, c'est donc le maillon `[2,0]` correspondant à la valeur 2. L'élément suivant sera le maillon à l'indice 0 dans le tableau donc `[3,2]` dont la valeur est 3. Ensuite on accède au maillon d'indice 2 donc `[5,1]` de valeur 5 pour terminer par le maillon `[8,None]` de valeur 8 marquant la fin de la liste. Au final la liste est donc bien

Cette approche a le mérite de se baser sur des types python standard que vous connaissez bien mais nous avons besoin de **deux variables** pour décrire un seul *objet*

- le tableau contenant les maillons
- l'indice du premier élément de la liste chaînée dans notre tableau

Cela alourdit énormément l'écriture des programmes que d'avoir un *objet* unique décrit par deux variables distinctes. Il y a moyen de faire beaucoup mieux grâce à la structure d'objets.

Dans ce TP, nous allons créer une classe **ListeChainee** permettant de créer et manipuler des objets de type liste chaînée.

```
class ListeChainee:
    """Implementation d'une liste chaînée"""
    def __init__(self, maillons=[], premier=None):
        """Initialisation de la liste chainée"""
        # tableau contenant les maillons de la liste
        self.__maillons = maillons
        # indice du premier élément.
        self.__premier = premier # None lorsque la liste est vide
```

Comme vous le voyez, la classe **ListeChainee** regroupe en un seul objet les deux informations nécessaires à la gestion de notre liste chaînée à savoir

- les maillons qui sont une liste au format que nous venons de décrire
- l'indice du premier élément

Nous utilisons ici des *paramètres optionnels* qui permettent le cas échéant d'initialiser notre liste chaînée avec du contenu.

II Propriété privée - défense d'entrer !

Parfois nous ne voulons pas qu'il soit possible de modifier les valeurs des propriétés d'une classe autrement qu'en passant par les méthodes de la classe afin de préserver la cohérence des données.

Afin de protéger les propriétés d'un objet, on ajoute __ devant leur nom. Ainsi, elles ne seront accessibles que depuis les méthodes de la classe, pas au travers des instances de l'objet. On parle alors de propriétés *privées* (sous python, elles sont plutôt *cachées*).

III A vous de jouer

Vous allez devoir implémenter les méthodes suivantes

- `premier()`
  - renvoie la valeur du premier élément de la liste ou **None** si la liste est vide
- `longueur()`
  - renvoie la longueur de la liste chainée (0 si elle est vide).
- `valeurs()`
  - renvoie un tableau avec toutes les valeurs de la liste
- `insere_debut(v)`
  - prend en paramètre la valeur à insérer `v`
  - insère la valeur `v` au début de la liste
- `insere_fin(v)`
  - prend en paramètre la valeur à insérer `v`
  - insère la valeur `v` à la fin de la liste

Avant de vous laisser implémenter ces fonctionnalités, nous allons faire ensemble la première méthode et tester le comportement de notre classe.

```
class ListeChainee:
    """Implementation d'une liste chaînée"""
    def __init__(self, maillons=[], premier=None):
        """Initialisation de la liste chainée"""
        # tableau contenant les maillons de la liste
        self.__maillons = maillons
        # indice du premier élément.
        self.__premier = premier # None lorsque la liste est vide

    def premier(self):
        """renvoye la valeur du premier élément de la liste ou None si la liste est vide"""
        return None if self.__premier is None else self.__maillons[self.__premier][0]
```

```
liste1 = ListeChainee()
# Aucun paramètre n'est passé, la liste1 est vide
print(liste1.premier())
```

```
liste1 = ListeChainee([[3, 2], [8,None], [5, 1], [2,0]], 3)
```

Dans la cellule ci-dessus, j'ai fourni une liste de maillons et un premier élément à la création de la liste. Cela va permettre de l'initialiser avec du contenu. Nous allons pouvoir vérifier que le premier élément de la liste est bien **2** :

```
liste1.premier()
```

Essayons de récupérer l'indice du premier élément stocké dans la propriété **__premier** : Vous constaterez une erreur. Cette propriété est cachée, on ne peut y accéder ni pour la consulter ni pour la modifier. C'est une sécurité nous garantissant l'intégrité des données.

```
liste1.__premier
```

IV Implémentez à présent les méthodes demandées

L'ossature de la classe est déjà présente, il vous suffira de remplacer **pass** par la méthode que vous fabriquerez. Vous vous rappelez qu'une méthode se code comme une fonction, la seule différence est le paramètre **self** qui fait référence à l'objet qui contient la méthode. Ce paramètre est *toujours* le premier paramètre passé à la méthode.

```
class ListeChainee:
    """Implementation d'une liste chaînée"""
    def __init__(self, maillons=[], premier=None):
        """Initialisation de la liste chainée"""
        # tableau contenant les maillons de la liste
        self.__maillons = maillons
        # indice du premier élément.
        self.__premier = premier # None lorsque la liste est vide

    def premier(self):
        """renvoye la valeur du premier élément de la liste ou None si la liste est vide"""
        return None if self.__premier is None else self.__maillons[self.__premier][0]

    def longueur(self):
        """renvoie la longueur de la liste chainée (0 si elle est vide)."""
        return 0 if self.__premier is None else len(self.__maillons)

    def valeurs(self):
        """renvoye un tableau avec toutes les valeurs de la liste"""
        pass

    def insere_debut(self, v):
        """prend en paramètre la valeur à insérer v
        insère la valeur v au début de la liste"""
        pass

    def insere_fin(self, v):
        """prend en paramètre la valeur à insérer v
        insère la valeur v à la fin de la liste"""
        pass
```

```
# YOUR CODE HERE
raise NotImplementedError()
```

```
# Test de la méthode valeur

liste1 = ListeChainee([[3, 2], [8,None], [5, 1], [2,0]], 3)
assert liste1.valeurs() == [2, 3, 5, 8]
```

```
# Test de la méthode longueur

liste1 = ListeChainee([[3, 2], [8,None], [5, 1], [2,0]], 3)
assert liste1.longueur() == 4
liste2 = ListeChainee()
assert liste2.longueur() == 0
```

```
# Test de la méthode insere_debut

liste1 = ListeChainee([[3, 2], [8,None], [5, 1], [2,0]], 3)
liste1.insere_debut(-5)
assert liste1.valeurs() == [-5, 2, 3, 5, 8]
```

```
# Test de la méthode insere_fin

liste1 = ListeChainee([[3, 2], [8,None], [5, 1], [2,0]], 3)
liste1.insere_fin(11)
assert liste1.valeurs() == [2, 3, 5, 8, 11]
```

Nous allons poursuivre l'ajout de méthodes à notre classe. Nous voudrions implémenter la méthode suivante :

- `insere_milieu(valeur, place)`
  - prend en paramètre une `valeur` et un emplacement `place` dans la liste
  - insère la `valeur` à l'emplacement `place`

Cette méthode va nécessiter d'accéder aux propriétés privées **__maillons** et **__premier**. Puisque ce n'est pas possible directement, nous allons écrite des méthodes qui permettent de renvoyer ces valeurs sans toutefois permettre leur modification. Afin de sécuriser davantage les données, on renverra une **copie** du tableau maillon et non l'objet original. On est ainsi assuré qu'aucune modification ne sera réalisée sur ce dernier.

En *programmation orientée objet*, ces méthodes permettant d'accéder de manière sécurisée à des propriétés cachées s'appellent des **accesseurs**. Voici comment nous pouvons les implémenter :

```
class ListeChainee:
    """Implementation d'une liste chaînée"""
    def __init__(self, maillons=[], premier=None):
        """Initialisation de la liste chainée"""
        # tableau contenant les maillons de la liste
        self.__maillons = maillons
        # indice du premier élément.
        self.__premier = premier # None lorsque la liste est vide

    def lire_premier(self):
        return self.__premier
    def lire_maillons(self):
        return [[m[0], m[1]] for m in self.__maillons]
    def remplace_maillons(self, maillons):
        self.__maillons = maillons
```

```
liste1 = ListeChainee([[3, 2], [8,None], [5, 1], [2,0]], 3)
print(liste1.lire_premier())
print(liste1.lire_maillons())
```

Ainsi, nous accédons aux données interne de notre classe sans pour autant risquer de compromettre ces dernières par une écriture malheureuse. Dans l'exemple suivant, nous allons voir comment utiliser ces accesseurs pour implémenter l'algorithme d'insertion dans une liste chaînée. Il vous restera ensuite à terminer notre classe avec la méthode **concat()**

```python
class ListeChainee:
    """Implementation d'une liste chaînée"""
    def __init__(self, maillons=[], premier=None):
        """Initialisation de la liste chainée"""
        # tableau contenant les maillons de la liste
        self.__maillons = maillons
        # indice du premier élément.
        self.__premier = premier # None lorsque la liste est vide

    def lire_premier(self):
        return self.__premier

    def lire_maillons(self):
        return [[m[0], m[1]] for m in self.__maillons]

    def remplace_maillons(self, maillons):
        self.__maillons = maillons

    def premier(self):
        """renvoye la valeur du premier élément de la liste ou None si la liste est vide"""
        return None if self.__premier is None else self.__maillons[self.__premier][0]

    def longueur(self):
        """renvoie la longueur de la liste chainée (0 si elle est vide)."""
        return 0 if self.__premier is None else len(self.__maillons)

    def insere_milieu(self, valeur, pos):
        """prend en paramètre la valeur et la position
        insère la valeur dans la liste à la position demandée
        Si la position n'existe pas dans la liste, on insère à la fin"""
      # On récupère les données internes
      maillons = self.lire_maillons()
      premier = self.lire_premier()

      # On utilise les méthodes existantes pour les cas limites
      if pos >= self.longueur():
          self.insere_fin(valeur)
      elif pos == 0:
          self.insere_debut(valeur)
      else:
          # parcour de la liste à la recherche de la position
          i = premier
          for _ in range(pos-1):
              if maillons[i][1] is not None:
                  i = maillons[i][1]
          # On crée un nouveau maillon à la fin de la liste
          maillons.append([valeur, maillons[i][1]])
          # On change les liens de manière à insérer le maillon
          maillons[i][1] = len(maillons)-1
          self.remplace_maillons(maillons)

  # Recopiez ici vos méthodes valeurs() et insere*()

  # YOUR CODE HERE
  raise NotImplementedError()
```

```
# Test de la méthode insere_milieu

liste1 = ListeChainee([[3, 2], [8,None], [5, 1], [2,0]], 3)
liste1.insere_milieu(10,2)
assert liste1.valeurs() == [2, 3, 10, 5, 8]
```

V A vous de jouer

Il ne vous reste plus à présent qu'à terminer notre classe **ListeChainee** en y ajoutant la méthode **concat()** :

- `concat(liste_c)`
  - prend en paramètre une liste `liste_c`
  - concatène (met bout à bout) `liste_c` à la fin de la liste courante

```
class ListeChainee:
    """Implementation d'une liste chaînée"""
    def __init__(self, maillons=[], premier=None):
        """Initialisation de la liste chainée"""
        # tableau contenant les maillons de la liste
        self.__maillons = maillons
        # indice du premier élément.
        self.__premier = premier # None lorsque la liste est vide

    def lire_premier(self):
        return self.__premier

    def lire_maillons(self):
        return [[m[0], m[1]] for m in self.__maillons]

    def remplace_maillons(self, maillons):
        self.__maillons = maillons

    def premier(self):
        """renvoye la valeur du premier élément de la liste ou None si la liste est vide"""
        return None if self.__premier is None else self.__maillons[self.__premier][0]

    def longueur(self):
        """renvoie la longueur de la liste chainée (0 si elle est vide)."""
        return 0 if self.__premier is None else len(self.__maillons)

    def insere_milieu(self, valeur, pos):
        """prend en paramètre la valeur et la position
        insère la valeur dans la liste à la position demandée
        Si la position n'existe pas dans la liste, on insère à la fin"""
```

```
        # On récupère les données internes
        maillons = self.lire_maillons()
        premier = self.lire_premier()

        # On utilise les méthodes existantes pour les cas limites
        if pos >= self.longueur():
            self.insere_fin(valeur)
        elif pos == 0:
            self.insere_debut(valeur)
        else:
            # parcour de la liste à la recherche de la position
            i = premier
            for _ in range(pos-1):
                if maillons[i][1] is not None:
                    i = maillons[i][1]
            # On crée un nouveau maillon à la fin de la liste
            maillons.append([valeur, maillons[i][1]])
            # On change les liens de manière à insérer le maillon
            maillons[i][1] = len(maillons)-1
            self.remplace_maillons(maillons)

    # Recopiez ici vos méthodes valeurs() et insere*()

    # YOUR CODE HERE
    raise NotImplementedError()
```

```
# Test de la méthode concat

liste1 = ListeChainee([[3, 2], [8,None], [5, 1], [2,0]], 3)
liste2 = ListeChainee([[6, 2], [8,None], [7, 1]], 0)
liste1.concat(liste2)
assert liste1.valeurs() == [2, 3, 5, 8, 6, 7, 8]
```