Numérique et sciences informatiques
Classe de terminale

II Structures de données
A Structures de données linéaires
2. Listes chaînées

I Notion de structure de données abstraite et structure de données linéaire

I.1 La notion de structure de données abstraite : interface et implémentation

• L'année dernière, nous avons rencontré différents types de données qui existent déjà en Python :

* des types élémentaires : **`int`**, **`float`**, **`str`**, **`bool`** ;
* des types structurés : **`list`**, **`dict`**, **`tuple`**.

On dit que ce sont des **types prédéfinis** dans ce langage de programmation. Ces types de base sont des types très classiques que l'on trouve (sous des noms légèrement différents) dans tous les langages de programmation

• A partir de ces types prédéfinis, un développeur doit pouvoir construire ses propres types structurés qui pourront être ainsi adaptés au type de projet qu'il veut réaliser. Un type structuré est un moyen d'organiser les données de façon significative pour faciliter leur manipulation.

• Par exemple, si le programme à écrire doit mettre en œuvre des opérations de géométrie dans le plan, le développeur pourra créée un type **Point** qui contiendra comme information l'abscisse et l'ordonnée sous la forme deux objets de type **`float`**.

• En Python, il pourra notamment le faire en utilisant la programmation orientée objet en créant une classe **Point** qui aura deux attributs `x` et `y` qui représenteront respectivement l'abscisse et l'ordonnée du point, ainsi que différentes méthodes permettant d'appliquer certaines opérations à ce point : déplacement selon un vecteur, calcul d'une distance, etc.

•Une **structure de données abstraite** (ou un **type abstrait**) est :

* **la description d'une structure de données particulière** et **des opérations que l'on peut lui appliquer** ;
* mais qui ne précise pas le détail des types prédéfinis que l'on utilisera et le code informatique écrit pour effectuer ces différentes opérations.

Cette description peut donc être considérée comme *indépendante du langage de programmation utilisé*.

• La description de la structure de données est appelée son **interface**. La décrire, c'est **spécifier** le type abstrait à créer.

• Le détail des types prédéfinis et des codes écrits pour mettre en œuvre l'interface s'appelle **l'implémentation**. On pourra donc concevoir *différentes implémentations pour une même interface*.

• Le plus souvent, on définira dans l'interface différentes opérations qui pourront être appliquées à un objet de la structure de données. Ces opérations peuvent être classées en quatre grandes catégories :

* un **constructeur** : qui permet de *créer un objet* de cette structure de données ;
* des **accesseurs** : qui permettent *lire certaines des informations* que contient un tel objet ;

* des **mutateurs** qui permettent de modifier l'objet pour :
  - *mettre à jour* et modifier ces informations
  - *supprimer* tout ou partie de ces informations.

• des **itérateurs** qui permettent de parcourir l'ensemble des éléments qui composent cet objet.

• Dans les exemples que nous étudierons, nous verrons deux grands styles d'implémentation :
* l'implémentation **fonctionnelle** : les opérations sur la structure de données sont mises en œuvre par des fonctions ;
* l'implémentation **par programmation orientée objet** : la structure de données est représentée par une classe et les opérations sont réalisées par les méthodes de cette classe.

I.2 Les structures de données linéaires : listes chainées, piles et files

• Dans ce chapitre et le suivant, nous allons étudier plus particulièrement trois structures de données abstraites classiques :

* les listes chainées
*les piles,
* et les files.

Dans chacun de ces trois cas, on pourra se représenter la structure de données comme *une succession finie et ordonnée d'éléments de même type* reliées entre elles par une relation de la forme « ... est suivi de ... ». On peut donc illustrer cette succession par le schéma suivant :

• On se représente chaque élément de cette succession comme le « maillon » d'une « chaîne ». Ce maillon est formé de deux parties :
* dans la première partie se trouve l'information contenue dans l'élément : un objet d'un type bien défini « homogène » ;
* dans la deuxième partie on trace une flèche indiquant quel est le maillon suivant. D'une façon plus concrète, il s'agira en fait *de l'adresse mémoire* où se trouve ce maillon suivant.

• Bien entendu, rien n'empêche qu'une même valeur apparaisse plusieurs fois dans la succession. En revanche, il est important qu'une liste chainée, une pile ou une file soit **homogène**, c'est-à-dire que toutes les valeurs contenues dans la partie « information » des différents maillons soient d'un même type.

• Les piles, les files et les listes chainées sont **des structures de données dynamiques** : on doit pouvoir ajouter ou enlever des maillons à la structure de données, un par un, au fur et à mesure de l'avancement du programme qui les utilise.

• La différence entre une liste chainée, une pile ou une file ne dépend pas de cette organisation des données mais *de la nature des opérations de modification et de lecture que l'on sera autorisé à effectuer sur cette structure de données*.

II Interface et implémentation de la structure de donnée « liste chaînée »

II.1 La structure de données abstraite « liste chaînée »

• On voudrait utiliser une structure de données représentant une succession ordonnées d'objets d'un même type. Cette nouvelle structure de données s'appellera une **liste chaînée**. Par exemple, on pourrait représenter la succession des températures maximum à Lyon en janvier 2023 une liste chaînée d'objets de type `float` suivante :

a) Interface minimale

• L'interface minimale d'une liste chaînée comportera les fonctions fondamentales suivantes :

(A) Un constructeur qui crée une liste :

* `creer_liste()` : crée et renvoie une liste chaînée vide.

(B) un opérateur qui modifie une liste :

* `ajouter(L, elem)` : ajoute l'élément `elem` en tête de la liste `L` et renvoie la nouvelle liste.

(C) des accesseurs qui renvoie des informations extraites d'une liste sans la modifier :

* `est_vide(L)` : renvoie `True` si la liste `L` est vide et `False` sinon.
* `tete(L)` : renvoie l'élément qui se trouve en tête de la liste `L`, si elle n'est pas vide.
* `queue(L)` : renvoie la liste privée de son premier élément, si elle n'est pas vide

(D) un itérateur

* `elements_liste(L)` renvoie un tableau contenant tous les éléments de la liste chaînée.

b) Fonctions supplémentaires

• Pour disposer de plus de souplesse dans l'utilisation de cette structure de données, on pourra éventuellement y ajouter certaines fonctions. En voici quelques exemples.

• Les opérateurs suivants ;

* `inserer(L, elem, i)` : ajoute l'élément `elem` à l'indice `i` de la liste `L`. Ceci implique un décalage « vers la droite » de tous les éléments situés à partir de l'indice `i`.
* `supprimer(L, i)` : supprime l'élément situé à l'indice `i` de la liste `L` et renvoie sa valeur. Bien entendu, cette opération conserve tous les autres éléments de la succession. Ceci implique donc un décalage « vers la gauche » de tous les éléments situés à partir de l'indice `i`.
* `remplacer(L, i, elem)` : remplace la valeur actuelle de l'élément situé à l'indice `i` de la liste `L` par l'élément `elem`.

• Les accesseurs suivants :

* `taille(L)` : renvoie le nombre d'éléments dans la liste `L`.
* `lire(L, i)` : renvoie l'élément situé à l'indice `i` dans la liste `L`.

• On supposera ici que les indices d'une liste chaînée se comportent comme ceux des tableaux : l'indice du premier élément sera 0 et le dernier élément de la liste `L` aura donc pour indice `taille(L)-1`.

II.2 Une implémentation fonctionnelle avec des tuples

• L'idée fondamentale de cette implémentation est de représenter une liste chaînée par un tuple selon le principe suivant :

* Si la liste chaînée est vide, on la représente par le tuple vide `()`.

* Si elle n'est pas vide, on la représente par un tuple à deux places `(tete, queue)`
  - où `tete` est le premier élément de la liste ;
  - et `queue` est tout le reste de la liste et donc une autre liste chainée (donc un tuple).

• Ainsi la liste chaînée suivante :

sera implémentée par le tuple suivant :

```
('Alice', ('Bob', ('Carole', ())))
```

• On en déduit l'implémentation suivante de l'interface minimale :

```
def creer_liste() :
    """renvoie une liste vide"""
    return ...

def ajouter(L, elem) :
    "ajoute l'élément elem en tête de la liste L et renvoie
cette nouvelle liste"""
    return (..., ...)

def est_vide(L) :
    """renvoie True si la liste L est vide, False sinon"""
    return L == ...

def tete(L) :
    """ renvoie l'élement en tête de la liste L"""
#    assert ...
    elem = ...
    return elem

def queue(L) :
    """ renvoie la liste L privée de son premier élement"""
#    assert ...
    reste = ...
    return reste

def elements_liste(L) :
    """renvoie le tableau des éléments de la liste L"""
    res = []
    liste = ...
    while not ... :
        tete = ...
        liste = ...
        res = res.append(...)
    return ...
```

II.3 Une implémentation en programmation orientée objet

• Pour implémenter les listes chaînées par programmation orientée objet, nous allons créer une classe `Maillon` comme première brique de cette construction. Voici le code qui crée cette classe :

```
class Maillon :
    def __init__(self, val=None, suiv=None) :
        """Un maillon vide possède un attribut 'valeur'
        et un attribut 'suivant' qui contiennent None.
        """
        self.valeur = val
        self.suivant = suiv

    def __str__(self) :
        """Affiche toutes les valeurs
        de la succession à partir du   maillon 'self'.
        """
        if self.valeur is None :
            return "None"
        else :
            return str(self.valeur) + \
             " ; " + str(self.suivant)
```

• Nous pouvons maintenant créer la classe `ListeChainee` qui comportera deux attributs :
* l'attribut `premier` de type `Maillon` qui correspond au premier élément de la succession de valeur ;
* l'attribut `taille` qui contiendra le nombre d'éléments dans la liste et qui sera mis à jour à chaque ajout ou suppression d'un élément dans la liste.

• La définition de la classe, avec son constructeur `__init__`, sa méthode d'affichage `__str__` et sa méthode pour déterminer le nombre d'éléments sera donc :

```
class Liste_chainee :
    """Les objets de la classe Liste_chainee
     sont mutables."""

    def __init__(self, val=None) :
        """Une chaine vide a pour attribut 'premier'
          un maillon vide."""
        self.premier = Maillon(val)
        if val is None :
            self.taille = 0
        else :
            self.taille = 1

    def __str__(self) :
        return str(self.premier)
```

a) Implémentation de l'interface minimale

• Dans cette implémentation en programmation orientée objet, le constructeur `__init__()` joue le rôle du constructeur `creer_liste()`. Pour appeler ce constructeur il suffit d'utiliser la syntaxe Python pour créer un objet de la classe `Liste_chainee`. Par exemple, pour créer une liste vide et l'affecter à la variable `maListe` on écrira l'instruction suivante :

```
maListe = Liste_chainee()
```

• Nous allons maintenant implémenter *sous la forme de méthodes* de la classe `ListeChainee` les autres opérations de l'interface minimale d'une liste chaînée :

* la méthode `est_vide(self)` qui renvoie `True` si la liste chainée est vide et `False` sinon.

* la méthode `ajouter(self, elem)` qui ajoute un élément en tête de la liste chainée concernée *et ne renvoie rien*.

* la méthode `tete(self)` qui renvoie la valeur de l'élément de *tête sans modifier la liste chaînée* .

* la méthode `elements_liste(self)` renvoie un tableau contenant tous les éléments de la liste chaînée.

```
    def est_vide(self):
        return self.taille == 0

    def ajouter(self, val) :
        nouv_maillon = Maillon(val)
        nouv_maillon.suivant = self.premier
        self.premier = nouv_maillon
        self.taille += 1

    def tete(self):
        assert not self.est_vide(),\
                "La liste chaînée est vide."
        return self.premier.valeur

    def elements_liste(self):
        res = []
        pointeur = self.premier
        while pointeur != None and pointeur.valeur != None
:
            res.append(pointeur.valeur)
            pointeur = pointeur.suivant
        return res
```

b) Implémentation de fonctions supplémentaires

• Nous allons maintenant utiliser la souplesse permise par l'implémentation choisie pour implémenter des méthodes supplémentaires :

* `taille(self)` qui renvoie le nombre d'éléments dans la liste chaînée.

* `lire(self, i)` qui renvoie la valeur du maillon d'indice n sans modifier la liste chaînée.

* `supprimer(self, n)` qui supprime le maillon d'indice `n` et renvoie la valeur de ce maillon.

* `inserer(self, n, val)` qui insère un maillon de valeur `val` entre le maillon d'indice `n-1` et le maillon d'indice `n`.

* `remplacer(self, n, val)` qui remplace par `val` la valeur dans le maillon d'indice `n`.

• Pour **renvoyer la taille de la liste** :

```
    def taille(self) :
        return self....
```

• Pour **lire une valeur dans la liste** :

```
    def lire(self, n) :
        assert not self.est_vide(), "La liste est vide"
        assert 0 <= n < self.taille, "Erreur d'indice"
        pointeur = ...
        for k in range(n) :
            pointeur = pointeur....
        return ...
```

• Pour **supprimer un élément de la liste** :

```
    def supprimer(self, n) :
        assert not self.est_vide(), "La liste est vide"
        assert 0<= n < self.taille, "Erreur d'indice"
        # on place le pointeur sur le 1er maillon
        pointeur = ...
        if n == 0 :
            res = ...
            self.premier = ...
        else :
            # on place le pointeur sur le maillon d'indice n-1
            for k in range(n-1) :
                pointeur = pointeur....
            # on supprime le maillon d'indice n
            # et on récupère sa valeur
            res = pointeur.suivant....
            pointeur.suivant = ...
        self.taille = ...
        return ...
```

• Pour **insérer un élément dans la liste** :

```
   def inserer(self, n, val) :
        assert 0 <= n < self.taille), "Erreur d'indice"
        # on crée un nouveau maillon de valeur 'val'
        nouv_maillon = Maillon(val)
        # on place le pointeur sur le 1er maillon
        pointeur = ...
        if n == 0 :
            nouv_maillon.suivant = ...
            self.premier = ...
        else :
        # on place le pointeur sur le maillon d'indice n-1
            for k in range(n-1) :
                pointeur = pointeur....
            # on insère le nouveau maillon dans la liste
```

```
            nouv_maillon.suivant = ...
            pointeur.suivant = ...
        self.taille = ...
```

• Pour **remplacer une valeur dans la liste** :

```
    def remplacer(self, n, val) :
        assert not self.est_vide(), "La liste est vide"
        assert 0 <= n < self.taille, "Erreur d'indice"
        pointeur = ...
        for k in range(n) :
            pointeur = pointeur....
        pointeur.... = ...
```

c) Utilisation des méthodes « cachées » pour trois opérations

• On peut aussi le corps de la méthode cachée `__len__(self)` qui sera appelé lors d'un appel de la forme `len(L)` si `L` est un objet de la classe `ListeChainee` :

```
    def __len__(self) :
        return self.taille
```

• Python offre aussi la possibilité de *lire la valeur* d'un élément de la liste situé à l'indice `n` *avec une syntaxe identique à celle d'un tableau*. Si `L` est un objet de la classe `ListeChainee`, l'expression `L[n]` renverra la valeur de l'élément situé à l'indice `n`. Pour implémenter ceci, il suffit d'écrire le corps de la méthode cachée `__getitem__(self, n)` en utilisant la méthode `lire` :

```
    def __getitem__(self, n) :
        return self.lire(n)
```

• De la même façon, Python permet de *modifier la valeur* d'un élément de la liste situé à l'indice `n` *avec une syntaxe identique à celle d'un tableau*. Si `L` est un objet de la classe `ListeChainee`, l'instruction `L[n]=val` associera la valeur `val` à l'élément situé à l'indice `n`. Pour implémenter ceci, il suffit d'écrire le corps de la méthode cachée `__setitem__(self, n)` en utilisant la méthode `remplacer` :

```
    def __setitem__(self, n, val) :
        self.remplacer(n, val)
```