Numérique et sciences informatiques
Classe de terminale

II Structures de données
A Structures de données linéaires
3. Piles et files

I Structure de données abstraite « pile » et « file »

• Les structures de données « pile » et « file » sont des structures de données linéaires. On peut donc les représenter avec le même schéma que nous l'avons utilisé pour les listes chaînées :

La spécificité des structures de données « pile » et « file réside dans **la nature des opérations autorisées** sur chacune de ces deux structures linéaires.

I.1 La structure de donnée « pile »

a) La notion générale

• Les opérations qui caractérisent une structure de données « pile » doivent respecter le principe fondamental suivant :

la dernière valeur ajoutée sera toujours la première à sortir.

En anglais, on dit qu'elle a un **comportement de type LIFO** pour « **Last In First Out** ». En français, on parle parfois d'un comportement DEPS (pour « dernier entré premier sorti »).

• On représente classiquement ce comportement par l'image d'une pile d'assiettes :

* chaque nouvelle assiette que l'on souhaite ranger sera placée au-dessus de la pile et sera donc *la seule accessible* ;

* c'est donc aussi cette même assiette, la dernière arrivée dans la pile, qui sera la première prélevée en cas de besoin.

• Les quatre fonctions fondamentales constituent l'**interface** de la structure de données. Ce sont donc les suivantes :

* **`creer_pile()`** : qui renvoie une pile vide ;
* **`est_vide(`**`P`**`)`** : qui renvoie un booléen, `True` si la pile `P` est vide et `False` sinon.
* **`empiler(`**`P, val`**`)`** : cette fonction place la valeur de la variable `val` « au-dessus » de la pile `P`. Cette fonction modifie la structure de donnée et ne renvoie rien ;
* **`depiler(`**`P`**`)`** : cette fonction enlève la valeur située « au-dessus » de la pile P et la renvoie *comme résultat*. Bien entendu, cette fonction ne peut être exécutée que si la pile `P` n'est pas vide.

• La propriété centrale que doit vérifier une implémentation de la structure de données « pile » est la suivante : si la fonction `empiler` doit ajouter la nouvelle donnée **à la même extrémité** de la structure linéaire où la fonction `depiler` retire une donnée. Cette extrémité peut être aussi bien le début ou la fin de la structure linéaire mais ce doit être la même.

• On représente schématiquement les éléments contenus dans une pile en les "empilant" verticalement (**représentation en colonne**) de telle façon que le *premier arrivé* soit situé tout en bas et le dernier élément arrivé (le "sommet" de la pile) soit placé tout en haut.

b) Un exemple

• Voici un exemple. On peut les fonctions décrites ci-dessus pour créer et faire évoluer une pile particulière en écrivant le code suivant :

```
p1 = créer_pile()
empiler(p1, 47)
empiler(p1, 59)
empiler(p1, 17)
nb = depiler(p1)
empiler(p1, 26)
print(p1.est_vide())
```

• On peut représenter l'évolution de la pile `p1` au cours de l'exécution du programme de la façon suivante :

• A l'issue de l'exécution,
  *la variable `p1` aura pour valeur la pile
  * la variable `nb` aura pour valeur `17` ;
  * la console aura affiché `False`.

c) Style d'écriture en programmation objet

• Si on adopte une implémentation par programmation orientée objet, on devra donc créer une classe `Pile`. Les quatre opérations fondamentales que nous venons de définir seront alors écrites *comme des méthodes* de cette classe `Pile`. En Python, on pourra donc écrire les entêtes suivantes :

```
class Pile :
    def __init__(self):
        ...
    def est_vide(self):
        ...
    def empiler(self, val) :
        ...
    def depiler(self):
        ...
```

• On pourra alors effectuer les mêmes opérations que dans l'exemple précédent en utilisant ces méthodes et en respectant la syntaxe appropriée. On devra donc écrire le code suivant :

```
p1 = Pile()
p1.empiler(47)
p1.empiler(59)
p1.empiler(17)
nb = p1.depiler()
p1.empiler(26)
print(p1.est_vide())
```

I.2 Les files

a) La notion générale

• Les opérations qui caractérisent une structure de données **file** doivent respecter le principe fondamental suivant :

le premier élément ajouté sera toujours le premier à sortir

En anglais, on dit qu'elle a un **comportement de type FIFO** pour « **First In First Out** ».

• Pour représenter le principe d'une structure de données « file », on utilise classiquement l'image d'une file d'attente devant une boutique :

* chaque nouvelle personne qui arrive dans la file d'attente doit se placer à la fin de celle-ci ;

* c'est la première personne arrivée dans la file (parmi celles qui s'y trouvent encore) qui est la première autorisée à en sortir. A ce moment, cette personne se trouve donc au début de la file.

• Les quatre méthodes fondamentales qui constituent l'**interface** de la structure de données « File » sont donc les suivantes :

* **`creer_file()`** : qui crée une file vide ;
* **`est_vide(`**`F`**`)`** : qui renvoie un booléen indiquant si la file `F` est vide.
* **`enfiler(`**`F, val`**`)`** : on place la valeur de la variable `val` « à la fin » de la file ;
* **`defiler(`**`F`**`)`** : on enlève la valeur située « au début » de la file et on la renvoie comme résultat de la fonction. Là encore, cette méthode ne peut être exécutée que si la file n'est pas vide.

• Une implémentation de la structure de données « file » doit vérifier la propriété centrale suivante : la fonction `enfiler` ajoute une nouvelle donnée **à l'extrémité opposée** à celle où la fonction `defiler` retire une donnée. L'extrémité choisie pour l'enfilement peut être le début ou la fin de la structure linéaire mais elle doit être opposée à celle choisie pour le défilement.

• *O*n représente traditionnellement les éléments contenus dans une file en les plaçant côte à côté horizontalement (**représentation en ligne**).

Attention : Il faut lire attentivement l'énoncé de l'exercice pour savoir si la tête de la file (le dernier arrivé dans la file) est représentée par l'élément le plus à droite ou le plus à gauche. Dans ce qui suit, nous avons choisi la première de ces deux conventions possibles.

b) Style d'écriture en programmation objet

• Comme pour les piles, on pourra aussi choisir une programmation orientée objet pour implémenter les files et donc créer une classe `File`. Les quatre opérations fondamentales que nous venons de définir seront alors créées *comme des méthodes* de cette classe `File`. En python, on pourra donc écrire les en-têtes suivantes :

```
class File :
    def __init__(self):
        ...
    def est_vide(self):
        ...
    def enfiler(self, val) :
        ...
    def defiler(self):
        ...
```

c) Un exemple

• Prenons un exemple de manipulation de ces opérations

implémentation fonctionnelle

```
f1 = creer_file()
print(est_vide(f1))
enfiler(f1, 33)
enfiler(f1, 15)
enfiler (f1, 91)
nb1 = defiler(f1)
enfiler(f1, 8)
nb2 = defiler(f1)
```

implémentation par POO

```
f1 = File()
print(f1.est_vide())
f1.enfiler(33)
f1.enfiler (15)
f1.enfiler (91)
nb1 = f1.defiler()
f1.enfiler(8)
nb2 = = f1.defiler()
```

• Dans les deux cas, on pourra décrire l'évolution des variables par le schéma suivant :

II Une implémentation des structure de données Pile et File par programmation orientée objet

II.1 La classe `Maillon`

• Nous noterons ici **`t`** le type des valeurs de notre succession. Ce type homogène pour les éléments de la succession pourra être le type **`int`**, **`float`**, **`str`**, **`tuple`**, etc.

• Comme pour les listes chainées, nous allons d'abord créer la classe **`Maillon`** qui nous sert pour toutes les structures de données linéaires. Les objets de cette classe représenteront donc les éléments de la succession de valeurs. Ils comportent deux attributs :

* un attribut `valeur` qui contiendra la valeur de type **`t`** associée à cet élément ;
* un attribut `suivant` qui contiendra une valeur de type **`Maillon`** et qui renverra à l'élément suivant de la succession. Si un élément est le dernier de la succession de valeurs, on l'indiquera en plaçant la valeur `None` dans son attribut `suivant`.

• L'implémentation de la classe **`Maillon`** pourra être fait avec le constructeur suivant :

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

Remarque : Puisque l'attribut `suivant` d'un objet de la classe **`Maillon`** est lui-même de la classe **`Maillon`**, il s'agit d'une définition de classe **récursive.**

• On écrira aussi une méthode `__str__` pour pouvoir imprimer la succession des valeurs situées à partir de ce maillon :

```
    def __str__(self) :
        """Pour que la récursivité fonctionne, il faut que l'on
        puisse appliquer la fonction "str(...)" à un maillon vide.
        """
        if self.valeur is None :
            return "None"
        else :
            return str(self.valeur) + "->" + str(self.suivant)
```

II.2 Une implémentation de la classe Pile

• Pour créer la classe `Pile`, on va prendre pour convention que le « *haut* de la pile » est constituée par le *début* de la succession de valeurs. Un objet de la classe `Pile` n'aura donc qu'un seul attribut `haut` qui sera de type `Maillon`.

• On crée d'abord les méthodes cachées suivantes :

* Le constructeur `__init__(self, val)` pour créer une pile vide ou bien constituée d'une seule valeur.

```
class Pile :
    def __init__(self, val) :
        """Une pile vide a un attribut 'haut' dont la valeur
        est un maillon vide.
        """
        self.haut = Maillon(val)
```

Cette méthode sera appelée par une instruction de la forme :

```
P = Pile(val)
```

* La méthode `__str(self)__` pour afficher les valeurs successives de la pile.

```
    def __str__(self) :
        """renvoie le résultat de la méthode __str__
         appliquée maillon du haut.
        """
        return str(self.haut)
```

Cette méthode sera appelée lorsque l'on réalisera une conversion en `str` d'un objet de type `Pile`, notamment lors d'une instruction d'affichage de la forme `print(P)`.

• La méthode `est_vide(self)` :

```
    def est_vide(self) :
        """renvoie un booléen"""
        return self.haut.valeur is None
```

• On écrit enfin les méthodes `empiler(self, val)` et `depiler(self)` de la façon suivante :

```
    def empiler(self, val) :
        if self.est_vide() :
            self.haut.valeur = val
        else :
            nouv_maillon = Maillon(val)
            nouv_maillon.suivant = self.haut
            self.haut = nouv_maillon

    def depiler(self) :
        assert not self.est_vide(), "depiler : la pile est vide !"
        res = self.haut.valeur
        if self.haut.suivant != None : # la pile n'a qu'un élément
            self.haut = Maillon() # la pile est maintenant vide
        else :
            self.haut = self.haut.suivant
        return res
```

II.3 Implémentation de la classe File

• Pour créer une classe `File`, nous allons prendre pour convention que la « *tête* de la file » est constituée par *le début* de la succession de valeurs. On va donc créer une classe `File` pour laquelle il n'y a qu'un seul attribut `tete` de type `Maillon`.

• On crée d'abord les méthodes cachées suivantes :

* le constructeur `__init__(self, val)` qui permet de créer une file vide ou bien constituée d'une seule valeur. Si la file est vide, son attribut `tete` pointe sur un maillon vide.

```
class File :
    def __init__(self, val) :
        "renvoie une file vide"""
        self.tete = Maillon(val)
```

Cette méthode sera appelée par une instruction de la forme :

`F = File(val)`

* la méthode `__str(self)__` pour afficher les valeurs successives de la pile :

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

• On crée enfin les méthodes `enfiler(self, val)` et `defiler(self)` de la façon suivante :

```
    def enfiler(self, val) :
        if self.est_vide() :
            self.tete.valeur = val
        else :
            nouv_maillon = Maillon(val)
            pointeur = self.tete
            while pointeur.suivant != None : # on déplace le pointeur
                pointeur = pointeur.suivant  # sur le dernier maillon
            pointeur.suivant = nouv_maillon

    def defiler(self) :
        assert not self.est_vide(), "defiler : la file est vide !"
        res = self.tete.valeur
        if self.tete.suivant == None : # la file n'a qu'un élément
            self.tete = Maillon()      # la file est maintenant vide
        else :
            self.tete = self.tete.suivant
        return res
```

• La méthode `defiler` de la classe `File` ressemble beaucoup à la méthode `depiler` de la classe `Pile` puisque, dans les deux cas nous voulons prendre la valeur à supprimer *au début* de la succession de valeurs.

• En revanche, la méthode `enfiler` est *très différente* de la méthode `empiler` car, dans le cas d'une file, nous voulons placer la nouvelle valeur *à la fin* de la succession, et non pas au début, comme nous l'avons fait pour les piles.