Spécialité NSI en terminale
Structures linéaires 2

En Python, une structure de pile, « stack » en anglais, ou de file, « queue » en anglais, peut être construite en utilisant des listes.

En effet une pile, par exemple, peut être considérée comme une liste avec des conditions d'utilisation restreintes. Les seules opérations possibles sont l'insertion ou la suppression d'un élément uniquement à la fin de la liste. C'est le principe « dernier entré, premier sorti », ou en anglais LIFO, acronyme de « Last In First Out ». On peut penser à une pile d'assiettes. Pour obtenir une file, on n'autorise que l'insertion d'un côté et la suppression de l'autre, suivant le principe « premier entré, premier sorti », en anglais FIFO, acronyme de « First In First Out ». On peut penser à une file d'attente.

En informatique, la structure de pile est très présente. Une pile est utilisée pour mémoriser de l'information, que ce soit au niveau du processeur ou au niveau de programmes ou d'applications. La programmation récursive nécessite l'utilisation d'une pile. Un navigateur ou un traitement de texte mémorise les actions effectuées afin de pouvoir revenir en arrière pas à pas. Les pages ou les fichiers ouverts sont également mémorisés. Une pile peut être nécessaire au calcul d'expressions mathématiques suivant la notation utilisée (infixée ou postfixée). C'est le cas avec la notation polonaise et la notation polonaise inverse, utilisées par certains langages.

La structure de file est elle aussi importante par exemple dans la gestion des processus. C'est aussi le cas pour un imprimante qui gère une file de documents en attente d'impression.

1 Piles

Trois fonctions traduisent les opérations de base sur les piles :
- une fonction qui crée une pile vide ;
- une fonction qui ajoute un élément sur la pile (en anglais « push ») ;
- une fonction qui retire l'élément sur la pile, le sommet, et le renvoie ; (en anglais « pop »).

D'autres fonctions peuvent fournir des informations sur une pile. Les plus utilisées sont :
- une fonction qui permet de savoir si une pile est vide ;
- une fonction qui donne la taille de la pile ;
- une fonction qui donne le sommet de la pile.

1.1 Pile à capacité non bornée

Nous représentons une pile par une liste. À priori, la taille n'est pas limitée et nous disons que la pile a une *capacité non bornée*. Voici un code pour les fonctions `creer_pile`, `empiler` et `depiler` :

```
def creer_pile():
    """Crée une pile vide"""
    return []

def empiler(p, x):
    """Ajoute un élément x sur la pile p"""
    p.append(x)

def depiler(p):
    """Renvoie le sommet de la pile p si elle est non vide,
    le sommet est retiré de la pile"""
    if len(p) > 0:
        return p.pop()
```

Cela revient à transformer les méthodes `append` et `pop` en fonctions utilisables avec une pile.

Remarque : la fonction `empiler` ne renvoie rien.

Nous pouvons aussi coder quelques fonctions auxiliaires :

```
def pile_vide(p):
    """Renvoie un booléen
    True si la pile est vide et False sinon"""
    return p == []

def taille(p):
    """Renvoie la taille de la pile p"""
    return len(p)

def sommet(p):
    """Renvoie le sommet de la pile p,
    le sommet n'est pas retiré de la pile"""
    if len(p) > 0:
        return p[-1]
```

Utilisation des fonctions :

```
>>> p = creer_pile()
>>> empiler(p, "A")
>>> empiler(p, "B")
>>> empiler(p, "C")
>>> taille(p)
3
>>> sommet(p)
'C'
>>> depiler(p)
'C'
```

À aucun moment un utilisateur n'a besoin de savoir comment sont codées les fonctions.

L'interface nous propose des fonctions avec une spécification précisant leur rôle et la manière de les utiliser. Le code peut même être modifié sans qu'un utilisateur en ait connaissance.

1.2 Pile à capacité bornée

En pratique une pile n'a pas une capacité illimitée. Il peut y avoir un débordement lorsque la capacité maximale est dépassée.

Une autre manière de coder les fonctions gérant des piles peut tenir compte de cette question.

Plusieurs manières sont envisageables dans ce cas pour représenter une pile à l'aide de listes.

Par exemple, on peut stocker la capacité `c` et les éléments de la pile sous la forme `[c, liste]` où `liste` est la liste des éléments contenus dans la pile, liste dont la taille ne peut dépasser `c`. Plus simplement, on peut utiliser la forme `[c, e1, e2, e3, ...]` où `e1`, `e2`, `e3`, ... sont les éléments contenus dans la pile.

Une autre possibilité est d'utiliser une liste de taille `c` qui contient les éléments de la pile puis des valeurs `None` pour les places libres : `[e1, e2, e3, ..., None, None, ...]`. Avec cette représentation, la capacité de la pile est la longueur de la liste, mais la taille de la pile est plus difficilement accessible.

Coder la taille dans la représentation est encore une autre possibilité :
`[t, e1, e2, e3, ..., None, None, ...]`.

Les différentes représentations proposées présentent des avantages et des inconvénients. Dans chaque cas, le codage des fonctions est différent, mais l'utilisation de ces fonctions reste cependant identique.

Codage de la fonction `creer_pile` avec la dernière représentation :

```
def creer_pile(c): # crée une pile de capacité c
    p = (c+1) * [None] # une pile vide
    p[0] = 0 # la taille de la pile
    return p
```

La capacité et la taille sont deux notions différentes : la taille est le nombre d'éléments contenus dans la pile, la capacité est la taille maximale de la pile.

Codage des fonctions `empiler` et `dépiler` :

```
def empiler(p, x):
    if p[0] < len(p)-1: # si la pile n'est pas pleine
        p[0] = p[0] + 1 # la taille augmente d'une unité
        p[p[0]] = x # l'élément est placé sur la pile

def depiler(p):
    if p[0] > 0: # si la pile n'est pas vide
        s = p[p[0]] # le sommet de la pile
        p[p[0]] = None # le sommet est retiré de la pile
        p[0] = p[0] - 1 # la taille diminue d'une unité
        return s
```

Remarque : `p[0]` est à la fois la taille de la pile et l'index du sommet. C'est l'un des intérêt de cette représentation.

Il n'est pas tenu compte des possibles erreurs. Si la pile est pleine, la fonction `empiler` ne fait rien mais ne provoque pas d'erreur. Si la pile est vide, la fonction `depiler` ne renvoie rien mais ne provoque pas d'erreur.

Codage des fonctions auxiliaires :

```
def pile_vide(p):
    return p[0] == 0

def taille(p):
    return p[0]

def sommet(p):
    if p[0] > 0:
        return p[p[0]]
```

Il est important d'insister : une pile est manipulée uniquement avec les fonctions spécifiques qui ont été créées. L'utilisateur ne doit pas se préoccuper de la manière dont sont codées ces fonctions.

Il est donc interdit d'écrire par exemple `pile[i]` pour obtenir un élément de la pile, même si cette instruction ne renvoie évidemment aucune erreur.

Construction à l'aide de classes

Les représentations précédentes utilisent simplement un type existant, le type list. Dans le style de la programmation orientée objet, nous pouvons définir une structure nouvelle.

Nous considérons des piles à capacité non bornée. Il s'agit de définir une classe `Pile` avec :

des attributs :

- taille : un entier (nombre d'éléments contenus dans la pile) ;
- éléments (la pile) : une liste de longueur taille (type list) ;

des méthodes :

- procédure `init`, le constructeur (fixe la taille à 0 et crée une liste vide) ;
- procédure `empiler` ;
- fonction `depiler` ;
- fonction `pile_vide` ;
- fonction `sommet` ;
- des méthodes d'affichage.

```
class Pile:
    def __init__(self):
        self.taille = 0
        self.vals = []

    def empiler(self, x):
        self.taille += 1
        self.vals.append(x)

    def depiler(self):
        if self.taille > 0:
            self.taille -= 1
            return self.vals.pop()

    def pile_vide(self):
        return self.taille == 0


    def sommet(self):
        if self.taille > 0:
            return self.vals[-1]

    def affiche(self):
        print(self.vals)

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

Quelques tests des fonctions :

```
>>> p = Pile()
>>> for i in range(8):
        p.empiler(i)

>>> p.sommet()
7
>>> p.taille
8
>>> p.affiche()
[0, 1, 2, 3, 4, 5, 6, 7]
>>> p.depiler()
7
>>> print(p)
[0, 1, 2, 3, 4, 5, 6]
```

Une autre implémentation à l'aide des listes chaînées est proposée en exercice.

2 Files

2.1 Implémentation d'une file

Une file peut être implémentée à l'aide d'un tableau ou d'une liste en Python. Cela nécessite deux variables dont les valeurs sont les indices de début et de fin de la file. L'utilisation d'une liste est plus compliqué que dans le cadre de l'implémentation d'une pile. En effet une liste permet l'ajout d'un élément en fin de liste avec un coût constant (méthode `append`) mais la suppression du premier élément n'a pas un coût constant (méthode `pop` avec `pop(0)`).

Nous pouvons implémenter une file à l'aide de deux piles et réciproquement une pile à l'aide de deux files. Cette manière d'implémenter une file est proposée en exercice.

2.2 Avec des classes

Comme pour les listes chaînées et les piles, nous pouvons créer une nouvelle structure dans le style de la programmation orientée objet.

Nous allons donc implémenter une classe `File` sur le modèle des listes chaînées, plus précisément comme on le ferait pour des listes doublement chaînées : à partir d'un élément, on peut accéder à l'élément précédent ou à l'élément suivant. Ceci est nécessaire afin que les actions d'ajout d'un élément et de retrait d'un élément s'exécutent avec un coût constant.

Nous commençons par définir une classe `Element`, comme nous l'avons fait pour la classe `Maillon` avec les listes chaînées, avec un attribut pour la valeur et un attribut pour l'élément suivant. Et nous ajoutons un attribut pour l'élément précédent.

```
class Element:
    def __init__(self, x):
        self.val = x
        self.precedent = None
        self.suivant = None

    def __str__(self):
        return str(self.val) + " - " + str(self.suivant)
```

Ensuite, la définition de la classe `File` comporte deux attributs pour le premier élément et le dernier et les principales fonctions de gestion d'une file : ajouter un élément dans la file, retirer un élément de la file, dire si la file est vide. On complète avec une fonction pour afficher la file.

```
class File:
    def __init__(self):
        self.premier = None
        self.dernier = None

    def ajoute(self, x):
        e = Element(x)
        if self.premier == None:
            self.premier = e
        else:
            e.precedent = self.dernier
            self.dernier.suivant = e
        self.dernier = e

    def file_vide(self):
        return self.premier is None

    def retire(self):
        if not self.file_vide():
            e = self.premier
            if e.suivant is None:
                self.premier = None
                self.dernier = None
            else:
                self.premier = e.suivant
                self.premier.precedent = None
            return e.val

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

Exemple d'utilisation :

```
>>> f = File()
>>> f.ajoute(5)
>>> f.ajoute(8)
>>> f.ajoute(13)
>>> print(f)
5 - 8 - 13 - None
>>> x = f.retire()
>>> x
5
>>> f.file_vide()
False
```

Note : dans la librairie standard en Python, une implémentation existe, utilisable pour les files et les

piles dans le module `queue`. L'utilisation est simple. Le code qui suit le montre. On peut consulter la documentation : `https://docs.python.org/fr/3/library/queue.html`.

Il existe aussi dans le module `collections` la structure `deque`.

```
from queue import *

q = Queue(5) # une file
q.put(3)
q.put(6)
q.put(12)

a = q.get()
print(a)
print(q.empty())
print(q.qsize())

p = LifoQueue(5) # une pile
p.put(3)
p.put(6)
p.put(12)

a = p.get()
print(a)
print(p.empty())
print(p.qsize())

def sommet_pile(pile):
    s = pile.get()
    pile.put(s)
    return s
```

3 Exercices

3.1 Exercice 1

Piles

1. Reprendre les fonctions `creer_pile`, `empiler`, `depiler` et `taille` permettant de créer à l'aide de listes une pile à capacité bornée et les écrire dans un fichier `mes_piles.py`. Dans la suite, ce sont les seules fonctions utilisables avec une pile. Elles constituent l'interface.
2. Créer une pile de capacité 5. Empiler les caractères 'A', 'B', 'C', 'D'. Vérifier que la liste représentant la pile est `[4, 'A', 'B', 'C', 'D', None]`.
3. Écrire une fonction `vider_pile` qui prend en argument une pile p et renvoie la pile vidée.
4. Écrire une fonction `inverse_pile` qui prend en argument une pile `p` et renvoie une autre pile avec les éléments empilés dans l'ordre inverse. La pile `p` peut être vidée.
5. Reprendre la question précédente avec la condition que la pile `p` ne soit pas modifiée.

Pour la question 5, utiliser une pile temporaire.

3.2 Exercice 2

Notation polonaise inverse

On considère une expression écrite en notation polonaise inverse.

Cette expression est représentée par une liste. Par exemple, l'expression « 8 3 + 5 × » est représentée par la liste `[8, 3, '+', 5, '*']`. En mathématiques, la forme d'écriture habituelle est $(8 + 3) \times 5$ et la valeur est 55.

Pour l'algorithme, on utilise une pile :

```
Pour e dans expression
    si e est un nombre
        alors on l'empile
    sinon
        on dépile deux opérandes
        on effectue l'opération
        on empile le résultat
On renvoie le résultat
```

1. Écrire une fonction `calcule` qui prend en paramètre une liste représentant une expression comme ci-dessus, notée `exp` et renvoie la valeur de l'expression. Les opérateurs peuvent être "+", "*", "-", ou "/".
2. Tester la fonction en vérifiant les résultats qui suivent :
   `calcule([7, 2, '+', 3, '*'])` (réponse : 27);
   `calcule([2, 5, '*', 4, '+'])` (réponse : 14);
   `calcule([8, 2, '/', 3, '-'])` (réponse : 1).

3.3 Exercice 3

Tours de Hanoï

On dispose de trois piles p1, p2 et p3. Dans la pile p1 sont empilés dans l'ordre les nombres $n, n-1, n-2, \ldots, 1$. Les piles p2 et p3 sont vides.

L'objectif est d'obtenir une pile p3 identique à la pile p1 initiale en un minimum de « coups ». Les piles p1 et p2 sont alors vides.

Un « coup » est le déplacement d'un nombre, sommet d'une pile, sur une autre pile. Pour ce faire, on utilise les fonctions `depiler` et `empiler`.

Règle : on empile un nombre sur une pile uniquement si la pile est vide ou si le nombre est plus petit que le sommet de la pile.

Ce jeux se joue en pratique avec trois tours A, B, C. Sur la tour A sont placés n disques dont le diamètre est de plus en plus petit. On ne peut poser un disque que sur un disque plus grand.

Si A, B et C sont les trois tours et x_n le nombre de déplacements de disques nécessaires au déplacement d'une tour complète de A vers C, alors pour déplacer une tour de n disques de A vers C, on effectue ces trois étapes :

- déplacer la tour des $n-1$ premiers disques de A vers B (soit x_{n-1} déplacements) ;
- déplacer le plus grand disque de A vers C (un déplacement) ;
- déplacer la tour des $n-1$ premiers disques de B vers C (x_{n-1} déplacements).

Au total cela fait $2x_{n-1}+1$ déplacements.

La relation de récurrence est : $x_0 = 0$ et $x_n = 2x_{n-1}+1$ si $n \geq 1$.

Après calculs, on obtient $x_n = 2^n - 1$.

Il est impossible de faire mieux car, pour déplacer la tour de n disques de A vers C, on doit forcément déplacer le plus grand disque de A vers C, et pour ce faire, on doit avoir empilé les $n-1$ premiers disques en B.

Avec 32 disques, le minimum de déplacements est $2^{32}-1 = 4294967295$. S'il faut une seconde pour déplacer un disque à la main, il faudrait environ 136 années pour finir le jeu. Avec 60 disques, pour 500 000 déplacements à la seconde sur un ordinateur, il faudrait plus de 73000 années.

Dans les questions qui suivent, les piles sont des instances d'une classe Pile.

1. Écrire une procédure récursive `hanoi` qui utilise la méthode décrite ci-dessus.

   La procédure `hanoi` prend quatre paramètres : `n` le nombre de disques utilisés, `d` la pile de départ, `a` la pile d'arrivée, `i` la pile intermédiaire.
2. Écrire une procédure `joue` qui prend en paramètre un entier `n`, représentant le nombre de disques, crée trois piles, `p1` la pile initiale contenant les `n` disques, `p2` et `p3`, et affiche les trois piles finales en utilisant la procédure `hanoi`.
3. Compléter le programme avec un compteur afin d'afficher le nombre de déplacements effectués.

Pour répondre à la troisième question, on peut exceptionnellement utiliser une variable globale `cpt` dans la fonction `hanoi` et dans la fonction `joue`.

3.4 Exercice 4

Le tri par insertion d'une liste `[x1, x2, ..., xn]` est connu. À chaque étape, pour i allant de 1 à n, on insère `xi` dans la liste `[x1, x2, ..., xi-1]` qui est triée. Une liste vide est triée. On suppose que le tri est effectué suivant l'ordre croissant.

1. Écrire une fonction `insertion` qui insère un élément `x` à la bonne place dans une pile où les éléments sont empilés du plus grand au plus petit. La pile représente la partie triée de la liste dans l'ordre décroissant. L'élément `x` et la pile sont les paramètres de la fonction.
2. Écrire une fonction `tri` qui insère les éléments de la liste à trier dans une pile à l'aide de la fonction `insertion`. Lorsque tous les éléments ont été insérés, ils sont dépilés et placés un à un dans la liste qui est alors triée dans l'ordre croissant.

   Pour effectuer des tests, on utilise les fonctions `creer_pile`, `empiler`, `depiler`, `pile_vide` et `sommet` du cours.

3.5 Exercice 5

Il s'agit d'implémenter les piles à l'aide de listes chaînées en s'inspirant de ce qui est fait dans le cours pour les files. Compléter le code qui suit.

```
class Element:
    def __init__(self, x):
        self.val = ...
        self.suivant = ...

    def __str__(self):
        return ...

class Pile:
    def __init__(self):
        self.dernier = ...

    def ajoute(self, x):
        ...

    def pile_vide(self):
        ...

    def supprime(self): # déjà complété
        if not self.pile_vide():
            e = self.dernier
            self.dernier = e.suivant
            return e.val

    def __str__(self):
        ...
```

3.6 Exercice 6

Implémenter une structure de file à l'aide de deux piles.

Pour cela, écrire une classe `File` dont les deux attributs sont des instances d'une classe `Pile`.

Définir une méthode `ajoute` qui ajoute un élément dans la file. Cela se traduit par le placement de l'élément sur la première pile.

Définir une méthode `supprime` qui retire un élément de la file. Cela se traduit par le retrait d'un élément de la deuxième pile si elle n'est pas vide. Sinon on dépile préalablement tous les éléments de la première pile, que l'on empile sur la deuxième.

Faire des dessins pour comprendre l'utilité des deux piles dans la simulation d'une file. Ou bien, prendre un paquet de cartes à jouer, le présenter face visible et retirer les cartes une à une pour les faire défiler. C'est une file. Ensuite, poser une partie des cartes une à une, face non visible, sur une table pour former une première pile. Prendre le paquet déposé et le retourner à côté. C'est une deuxième pile. Retirer alors les cartes une à une, les faces sont visibles.