Chapitre 5 - Les piles

Objectifs :

- ▷ Connaître et comprendre le type abstrait pile.
- ▷ Modéliser puis simuler le comportement d'une pile.
- ▷ Distinguer les modes FIFO et LIFO des piles et des files.

1 Introduction

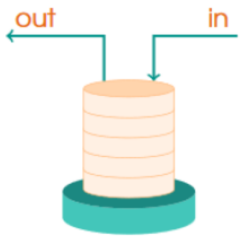
Dans ce chapitre nous allons décrire des **structures de données linéaires** appelées **piles**. Il faut bien comprendre que lorsqu'on parle de structure de données, on parle d'une représentation **abstraite** qui n'est pas en lien direct avec son implémentation qui peut être réalisée de diverses manières suivant le langage de programmation, voire au sein d'un même langage de programmation.

2 Définition

A retenir !

En informatique, une pile (*stack* en anglais) est une structure de données fondée sur le principe du **dernier arrivé, premier sorti** (ou **LIFO** pour *Last In, First Out*).

Cela signifie que les derniers éléments ajoutés à la pile seront les premiers à être récupérés.



Le fonctionnement est donc celui d'une **pile d'assiettes** : on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée.

3 Exemples d'usage

Voici quelques exemples d'usage courant d'une pile :

- ▷ Dans un navigateur web, une pile sert à **mémoriser l'historique des pages web visitées**. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton *«Afficher la page précédente»*.
- ▷ L'évaluation des **expressions mathématiques en notation post-fixée** (ou polonaise inverse) utilise une pile.
- ▷ La fonction **"Annuler"** (en anglais *Undo*) d'un logiciel (photo, traitement de texte, ..., etc.) mémorise les dernières modifications effectuées dans une pile.

4 Implémentation

Pour implémenter une structure de pile, on a besoin d'implémenter seulement un nombre réduit d'opérations de bases qui sont :

- ▷ **empiler** : ajoute un élément sur la pile. Terme anglais correspondant : *push*
- ▷ **dépiler** : enlève un élément de la pile et le renvoie. En anglais : *pop*
- ▷ **vide** : renvoie vrai si la pile est vide, faux sinon
- ▷ **remplissage** : renvoie le nombre d'éléments dans la pile

La structure de pile est un **concept abstrait**. Comment faire pour réaliser une pile dans la pratique ?

L'idée principale étant que les fonctions de bases pourront être utilisées indépendamment de l'implémentation choisie.

4.1 Implémentation n°1

Nous utiliserons une simple liste pour représenter la pile. Il se trouve que les méthodes `append` et `pop` sur les listes jouent déjà le rôle de *push* et *pop* sur les piles.

Voici les fonctions de base :

```
def pile():
    ''' Retourne une liste vide'''
    return []

def vide(p):
    '''Renvoie True si la pile est vide et False sinon'''
    return p == []

def empiler(p,x):
    '''Ajoute l'element x a la pile p'''
    return p.append(x)

def depiler(p):
    '''Depile et renvoie l'element au sommet de la pile p'''
    assert not vide(p), "Pile vide"
    return p.pop()
```

A faire 1 : Tester les instructions suivantes :

```
p = pile()

for i in range(5):
    empiler(p,2*i)

a = depiler(p)
print(a)
print(vide(p))
```

Expliquer les affichages obtenus :

A faire 2 : Réaliser les **fonctions** `taille(p)` et `sommet(p)` qui retournent respectivement la taille de la pile et le sommet de la pile (sans le supprimer).

4.2 Implémentation n°2

Nous allons utiliser la **POO** afin de créer une classe `Pile` pour implémenter cette structure.

```
class Pile:
    '''classe Pile, creation d'une instance Pile avec une liste'''

    def __init__(self):
        '''Initialisation d'une pile vide'''
        self.L = []

    def vide(self):
        '''Teste si la pile est vide'''
        return self.L == []

    def depiler(self):
        '''Depile la dernier element de la pile'''
        assert not self.vide(),'Pile vide'
        return self.L.pop()

    def empiler(self,x):
        '''Empile l'element x en haut de la pile'''
        return self.L.append(x)
```

Tester les instructions suivantes :

```
p = Pile()
for i in range(5):
    p.empiler(2*i)

print(p.L)
a = p.depiler()
print(a)

print(p.L)
print(p.vide())
```

A faire 3 : Réaliser les **méthodes** `taille(self)` et `sommet(self)` qui retournent respectivement la taille de la liste et le sommet de la liste (sans le supprimer).

5 Exercices

Exercice 1 : On considère l'enchaînement d'opérations ci-dessous.

Écrire à chaque étape l'état de la pile `p` et la valeur éventuellement renvoyée. On prendra pour convention que la tête de la pile est à droite.

```
p = Pile()
p.empiler(3)
p.empiler(5)
p.est_vide()
p.empiler(1)
p.depiler()
p.depiler()
p.empiler(9)
p.depiler()
p.depiler()
p.est_vide()
```

Exercice 2 : *Historique de navigation*

Simulez une gestion de l'historique de navigation Internet, en créant une classe `Nav` qui utilisera une pile. Attention, il ne faut pas réinventer la classe `Pile`, mais s'en servir !

Exemple d'utilisation :

```
>>> n = Nav()
>>> n.visite('lemonde.fr')
page actuelle : lemonde.fr
>>> n.visite('google.fr')
page actuelle : google.fr
>>> n.visite('ifs.edu.sg')
page actuelle : ifs.edu.sg
>>> n.back()
page quittee : ifs.edu.sg
>>> n.back()
page quittee : google.fr
```

Exercice 3 : *Vérificateur de parenthèses*

La vérification du parenthésage est une étape importante lors de l'analyse d'une expression mathématique ou informatique lors de l'interprétation dans un éditeur de code tel que *Thonny*, *EduPython* ou *Studio Code*.

Par exemple `'(())'` et `'(()())'` sont valides mais pas `'(()'` ou `'(()))'`. Pour l'instant nous allons supposer que les expressions sont uniquement composées de `'('` et `')'`.

Écrire une fonction `verif_parentheses(expression)` qui prend un texte et renvoie un booléen qui indique si l'expression est correcte au niveau des parenthèses.

Pour vérifier la validité d'une expression, nous allons utiliser une pile. Les parenthèses ouvrantes seront empilées et elles seront dépilées lors de la lecture de parenthèses fermantes. Une expression est correcte si après avoir lu toute l'expression, la pile est vide. Si une exception est levée lors de l'appel de la méthode de `Pile`, votre fonction doit la récuperer et renvoyer `False`.

```
>>> verif_parentheses('(((())))')
True
>>> verif_parentheses('((()()))')
True
>>> verif_parentheses('((()()))(((()((()))(()))()))')
True
>>> verif_parentheses('((()()))((((((()((()))))()(()))()))(()()(()))))')
False
>>> verif_parentheses('((()()))((((((()((()))))()(()))()))(()()(()))))')
False
```

Exercice 4 : *Vérificateur de parenthèses amélioré*

Nous allons maintenant rajouter les accolades `'{}'` et les crochets `'[]'`.

Ainsi `'([{}])'` est valide mais pas `'([{]})'`. Modifier la fonction `verif_parentheses(expression)` de l'exercice précédent pour pouvoir utiliser ces nouveaux types de parenthèses. Il faut vérifier lorsqu'on dépile que la parenthèse fermante correspont à la parenthèse ouvrante dépilée.

```
>>> verif_parentheses('{{}}[[]](())')
True
>>> verif_parentheses('{{[([][{}])](([{}]))}}')
True
>>> verif_parentheses('([)]')
False
>>> verif_parentheses('{{[([][{}])][({{{{[]}}}}[]])}{{[{()}]}}}')
False
>>> verif_parentheses('{{[([][{}])]([{{([]){{}}()([])}}}[]])}{{[{()}]}}}')
False
>>> verif_parentheses('{{[([][{}])]([{{{{[]}}}}[]])}{{[{()}]}}}')
False
```

Exercice 5 : *Sujet Bac 2022 (épreuve pratique n°16)*

Cet exercice utilise des piles qui seront représentées en Python par des listes (de type `list`).

On rappelle que :

- l'expression `T1 = list(T)` fait une copie de `T` indépendante de `T`
- l'expression `x = T.pop()` enlève le sommet de la pile `T` et le place dans la variable `x`
- l'expression `T.append(v)` place la valeur `v` au sommet de la pile `T`.

Compléter le code Python de la fonction positif ci-dessous qui prend une pile `T` de nombres entiers en paramètre et qui renvoie la pile des entiers positifs dans le même ordre, sans modifier la variable `T`.

```
1  def positif(T):
2      T2 = ...(T)
3      T3 = ...
4      while T2 != []:
5          x = ...
6          if ... >= 0:
7              T3.append(...)
8      T2 = []
9      while T3 != ...:
10         x = T3.pop()
11         ...
12     print('T = ',T)
13     return T2
```

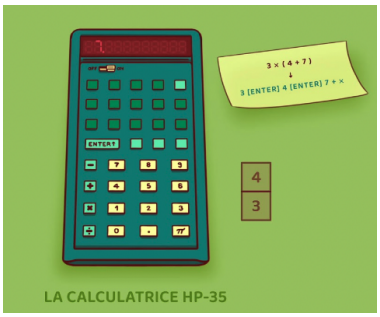
Exemple :

```
>>> positif([-1,0,5,-3,4,-6,10,9,-8])
[0, 5, 4, 10, 9]
```

Exercice 6 : *Notation polonaise inverse*

Historique

La **notation polonaise inverse** notée **NPI** (en anglais **RPN** pour *Reverse Polish Notation*), également connue sous le nom de **notation post-fixée**, permet d'écrire de façon non ambiguë les formules arithmétiques sans utiliser de parenthèses. Elle est dérivée de la notation polonaise utilisée pour la première fois en 1924 par le mathématicien polonais *Jan Łukasiewicz*, la NPI a été inventée par le philosophe et informaticien australien *Charles Leonard Hamblin* dans le milieu des années 1950, pour permettre les calculs sans faire référence à une quelconque adresse mémoire. À la fin des années 1960, elle a été diffusée dans le public comme interface utilisateur avec les calculatrices de bureau de *Hewlett-Packard*.



Exemple :

Voici une expression algébrique en notation infixée (celle que nous utilisons la plupart du temps) : $((3+4)*2)^3$

En notation post-fixée (NPI) cela donne : `3 4 + 2 3 ^`

L'objectif est d'écrire un programme pour évaluer des expressions écrites en notation polonaise inverse.

Les expressions en NPI seront représentées par des tableaux contenant des entiers et des caractères.

Ainsi, l'expression précédente sera représentée par le tableau : `expr = [3,4,'+',2,'*',3,'^']`

Le principe

Une pile vide est créée ; le tableau est parcouru de gauche à droite.

- ▷ Chaque nombre rencontré est empilé.
- ▷ Si l'élément rencontré est un opérateur, on dépile le sommet et le sous-sommet puis on empile le résultat du calcul (sous-sommet "opération" sommet).
- ▷ Si l'élément rencontré est une fonction (comme : p), on dépile le sommet et on calcule la valeur de la fonction pour le sommet, puis on empile le résultat

1. Donner les tableaux qui vont représenter les calculs suivants :

   (a) `2 + 20 * 2`
   (b) `2 * (10 * 2 + 1)`
   (c) `10 + 20 + 12`
   (d) `(10 + 4 / 2) * 2 + (3 * 3 * (4 / 2))`

2. Écrire un programme Python qui permette d'implémenter le calcul d'expressions en NPI et le tester avec les expressions précédentes.