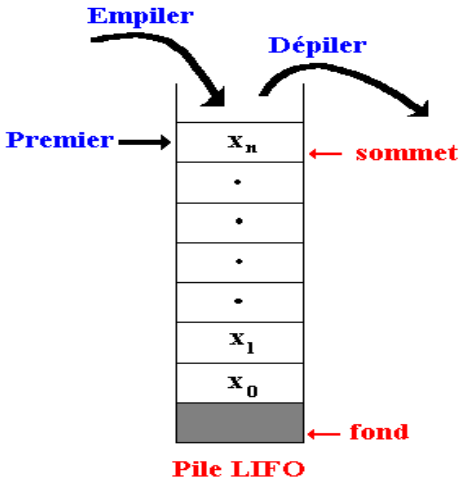
Chapitre 1.

Structures de données avancées

Les Piles

Les Piles



Une **pile** (**stack** pour les anglo-saxons) est une structure de données qui met en œuvre le principe : **dernier entré, premier sorti** ou **LIFO** (**L**ast - **I**n - **F**irst - **O**ut).
L'image qu'on peut en donner est celle d'une pile d'assiettes :
La dernière assiette placée dans la pile est au sommet de celle-ci.
Ce sera bien sûr aussi la première assiette que l'on enlèvera de cette pile. Cette structure permet de stocker provisoirement des éléments, en attendant de les utiliser plus tard.

Les seules opérations dont on a besoin avec cette structure sont :

1. **INSERER** qu'on appelle ici **EMPILER** (**push** en anglais) : Elle permet d'arranger un élément au sommet de la pile.
2. **SUPPRIMER** qu'on appelle ici **DEPILER** (**pop** en anglais) : Elle permet de récupérer les données les plus récentes.

Autres exemples :

- Pile d'appels d'une fonction récursive,
- Les deux boutons « page suivante » et « page précédente» de votre navigateur, chacun utilise une pile.

Il y a beaucoup de façons de concevoir une pile : listes, tableaux, ...

Exercices d'application : Mise en œuvre d'une pile avec une liste

ⓘ Indications pour la réalisation d'une pile en Python avec les listes :
Le type **list** intègre déjà toutes les méthodes pour réaliser une pile :

- **L.append(x)** : ajoute un élément à la fin de la liste.
- **L.insert(i, x)** : insère un élément à la position indiquée. Le premier argument est la position de l'élément courant avant lequel l'insertion doit s'effectuer, donc **L.insert(0, x)** insère l'élément en tête de la liste, et **L.insert(len(a), x)** est équivalent à **L.append(x).**
- **L.remove(x)** : supprime de la liste le premier élément dont la valeur est x. Une exception est levée s'il existe aucun élément avec cette valeur.
- **L.pop(i)**, qui enlève de la liste l'élément situé à la position indiquée, et le retourne. Si aucune position n'est indiquée, **L.pop()** enlève et retourne le dernier élément de la liste.

Exercice 1 :

1. Créer un module Python « **pile.py** » contenant les fonctions suivantes en essayant de fixer une taille maximale de la pile par l'introduction d'une constante **NMAX=40**.
2. Ecrire une fonction **creer_pile()** permettant de créer et renvoyer une pile vide;
3. Écrire une fonction **pile_vide(p)** qui retourne **True** si la pile est vide et **False** sinon.

4. Ecrire la fonction **empiler(p, x)** permettant d'ajouter au sommet de la pile p l'élément x, uniquement à l'aide de la méthode **append** de la classe **List**. Prévoir les traitements nécessaires au bon fonctionnement du module en cas de saturation de la pile.
5. Ecrire la fonction **depiler(p)** permettant de supprimer le sommet de la pile **p**, uniquement à l'aide de la méthode **pop** de la classe **List**. Que se passe-t-il quand on essaye de dépiler une pile vide ? On y mettra les traitements nécessaires au bon fonctionnement du module en cas de sortie brutale du programme.
6. Ecrire la fonction **sommet(p)** qui renvoie l'élément du sommet de la pile p.
7. Ecrire la fonction **afficher(p)** qui permet d'afficher tous les éléments de la pile.
8. Ecrire la fonction **depilerKelt(p,k)** qui dépile **k** éléments si la pile contient au moins **k** éléments, sinon elle dépile toute la pile.
9. Ecrire la fonction **appartient(p,x)** qui renvoie **True** si l'élément **x** appartient à la pile, **False** sinon. Attention : il est important que la pile ne change pas.
10. Ecrire la fonction **depilerJusqua(p,x)** qui dépile la pile jusqu'à l'élément **x**. L'élément **x** n'est pas dépilé. Si l'élément **x** n'appartient pas à la pile, alors la fonction dépile toute la pile.
11. Ecrire la fonction **inverser_pile(p)** qui inverse les éléments de la pile à laquelle elle est appliquée. On a le droit d'utiliser des piles temporaires.

Exercice 2 :

Écrire une fonction **conversion(nb10)** qui retourne la représentation en base 2 du nombre nb10 représenté en base 10 à l'aide de piles.

Exercice 3 : Évaluation d'une expression arithmétique

Une des tâches fondamentales d'un interpréteur comme celui de Python (ou d'un compilateur dans un autre langage de programmation) est d'attribuer une valeur à une expression arithmétique. Une expression arithmétique est formée des 10 chiffres $0,1,2,\ldots,9$ des signes $+, -, \times$ et $/$ et des parenthèses ouvrantes et fermantes.

Par exemple : $(3 + (4 \times 5))/2$ est une expression arithmétique dont la valeur est 11,5.

En général, l'utilisateur tape cette expression à l'aide de son clavier, ce qui génère une chaîne de caractères $\text{ch} = "(3 + (4 \times 5))/2"$.

Le problème est de traduire cette chaîne en un nombre, tout en gérant les priorités des opérations et les parenthèses : l'interpréteur doit se livrer pour cela à une **analyse syntaxique**.

1. Créer un script Python qui joue le rôle d'un analyseur syntaxique minimaliste vérifiant seulement si une expression arithmétique est bien parenthésée.
   Exemples d'expressions arithmétiques bien parenthésées : (1+ 2) , ( ) , (1+6) / 2
   Exemples d'expressions arithmétiques mal parenthésées : (1+ 2 ,)(,)(1+6) / 2

Une même expression arithmétique a au moins trois représentations naturelles :

La représentation **infixée**. C'est celle qu'on utilise tout le temps : **(3 + 2) × 5**

La représentation **préfixée** où on place tous les opérateurs avant les opérandes : **2 + 5** s'écrit **+ 2 5**
(3 + 2) × 5 s'écrit **× + 3 2 5** et **3 + (2 × 5)** s'écrit **+ 3 × 2 5**

On peut également placer les opérateurs après les opérandes : c'est la représentation **postfixée**.

Dans ce cas **2 + 5** s'écrirait **: 2 5 +** et si on partait de **(3 + 2) × 5** on aurait : **3 2 + 5 ×.**

2. Donner les deux autres formes des expressions suivantes et indiquer à chaque fois leurs valeurs :
   a) **2 + (3 - (5 + 1) × 2)** b) **2 3 4 5 + - 6 × /**

L'évaluation d'une expression **postfixée** peut se faire de façon très simple, en une seule lecture de la gauche vers la droite à l'aide d'une pile qui stocke les résultats intermédiaires.
Sur l'exemple suivant, le caractère ▶ a été rajouté pour indiquer la limite de la partie déjà traitée de l'expression.
▶ "2 3 4 + - 5 ×" pile P : []
"2 ▶ 3 4 + - 5 ×" pile P : [2]
"2 3 ▶ 4 + - 5 ×" pile P : [2,3]
"2 3 4 ▶+ - 5 ×" pile P : [2, 3,4]
"2 3 4 + ▶ - 5 ×" pile P : [2,7]
"2 3 4 + - ▶ 5 ×" pile P : [-5]
"2 3 4 + - 5 ▶ ×" pile P : [-5,5]
"2 3 4 + - 5 ×" ▶ pile P : [-25] **Résultat :** -25

3. Écrire une fonction **calc_exp_arith(ch)** qui prend en paramètre une chaîne **ch** représentant une expression arithmétique et renvoie sa valeur numérique.
   Exemple : Partons de la chaîne postfixée : **ch** = "**27 × 5 +** ", le résultat renvoyé sera **19**.

Exercice 4 :

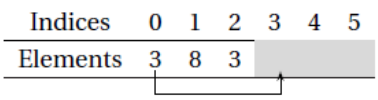
Une pile est une structure de données qui met en œuvre le principe : dernier entré, premier sorti. Il y a beaucoup de façons de concevoir une pile en Python. La plus simple consiste à utiliser un conteneur de type **list**. Les deux opérations possibles sur les piles sont :

- L'opération **Empiler** : insère un élément au sommet d'une pile.
- L'opération **Dépiler** : retire le dernier élément entré dans la pile et renvoie cet élément.

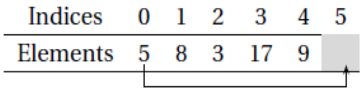
Dans cet exercice, on vous demande de réaliser une pile pouvant contenir **n+1** entiers :

- La première case (de rang 0) contient l'indice du prochain élément à insérer dans la pile.
- Les n "cases" suivantes, d'indices 1 à n, contiennent les éléments insérés dans la pile.
- Le dernier élément inséré est appelé sommet de la pile.
- Si P[0] égal à 1 la pile est considérée comme vide.
- À chaque fois qu'on insère un élément, on augmente P[0] d'une unité.
- Lorsque P[0] égal à n+1, la pile est considérée comme pleine.

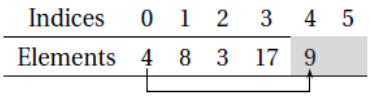
Exemple :
Réalisation d'une pile P d'au plus 5 éléments. Les éléments de la pile apparaissent uniquement aux positions blanches. P[0] contient l'indice du prochain élément à insérer.



- La pile contient seulement les entiers 8 et 3.
- L'élément au sommet de la pile est 3 situé à la position 2.



- État de la pile P après les appels **empiler(p, 17)** et **empiler(p, 9).**



- État de la pile P après que l'appel **depiler(p)** ait retourné 9.
- Bien que 9 apparaisse encore dans la liste, il ne fait plus partie de la pile.

Questions :

1. Écrire une fonction **init_pile(p, n)** permettant de créer et d'initialiser une pile **p** de **n+1** zéros.
2. Écrire une fonction **pile_vide(p)** qui retourne **True** si la pile est vide et **False** sinon.
3. Écrire une fonction **pile_pleine(p)** qui retourne **True** si la pile est pleine et **False** sinon.
4. Créer une fonction **empiler (p, x)** qui insère l'élément **x** au sommet de la pile **P**. Il faut gérer le fait qu'on ne peut rien ajouter à une pile pleine.
5. Créer une fonction **depiler (p)** qui retire le dernier élément entré dans la pile et renvoie cet élément. Il faut gérer le fait qu'on ne peut rien retirer d'une pile vide.
6. Écrire une fonction **copier(p)** réalisant la copie d'une pile. La pile donnée en argument devra être laissée intacte. Il est interdit d'utiliser les fonctions prédéfinies copy et deepcopy. Seules autorisées sont les fonctions précédemment définies.
7. Écrire une fonction **echanger(p)** réalisant l'échange des deux éléments (supposés existants) en haut de pile. Rien ne doit être retourné.
8. Écrire une fonction **enroler (p, n)** réalisant la « rotation » des **n** éléments en haut de pile selon le modèle suivant :

```
>>> print(p) → [ 6 , 7 , 8 , 12 , 10 , 42 ]
>>> enroler(p, 4)
>>> print(p) → [ 6 , 7 , 42 , 8 , 12 , 10 ]
```

9. Écrire une fonction **superposition (p1, p2)** permettant de renvoyer une pile résultat de superposition de deux piles suivant le modèle suivant :

```
>>> print(p1) → [ 4 , 7 , 8 , 12 , 42 , 10 ]
>>> print(p2) → [ 6 , 7 , 42 , 8 , 12 , 10 ]
>>> superposition(p1, p2)
[ 9 , 7 , 42 , 8 , 12 , 10 , 7 , 8 , 12 , 0 , 0 ]
```

Tri d'une pile par insertions successives :

10. On rappelle qu'on peut trier une pile en empilant les éléments successifs dans une pile qui reste à tout moment triée.
Écrire une fonction **inserer (x, p)** réalisant l'insertion d'un élément **x** dans une pile **p** en maintenant l'ordre (croissant) des éléments de ladite pile. Rien n'est retourné, mais la pile donnée en argument est modifiée. Exemple :

```
>>> print(p) → [ 6 , 7 , 8 , 10 , 12 , 42, 0 , 0 , 0 , 0 ]
>>> inserer(9, p)
>>> print(p) → [ 7 , 7 , 8 , 9 , 10 , 12 , 42, 0 , 0 , 0 ]
>>> inserer(50,p)
>>> inserer(2, p)
>>> print(p) → [ 9 , 2 , 7 , 8 , 9 , 10 , 12 , 42 , 50 , 0 ]
```

11. On peut maintenant trier les piles. Écrire une fonction **tri_insertion_en_place(p)** réalisant le tri d'une pile EN PLACE (rien n'est retourné, mais la pile donnée en argument a été modifiée).

Exemple :

```
>>> print(p) → [ 11 , 4 , 41 , 3 , 4 , 5 , 40 , 21 , 33 , 50 , 30 ]
>>> tri_insertion_en_place(p)
>>> print(p) → [ 11 , 3 , 4 , 4 , 5 , 21 , 30 , 33 , 40 , 41 , 50 ]
```