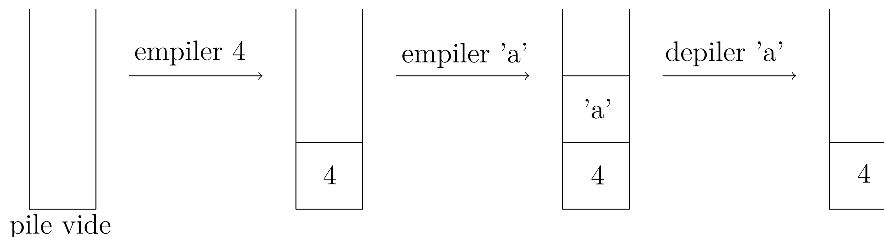
I Définition d'une pile

1. Généralités

Une **pile** (stack en anglais) est une structure de données qui s'apparente aux listes ou aux tableaux à une dimension. C'est une structure linéaire dont les données sont stockées de sorte que seule la dernière donnée ajoutée à la liste est directement accessible. Les données sont dites empilées et leur traitement au sein de la pile se fait par une succession d'opérations appelées **empilement** et **dépilement**. Ainsi, les données ne sont pas toutes accessibles de la même façon. Le **sommet** de la pile contient la dernière donnée empilée qui peut être récupérée par dépilement. La **base** de la pile contient la première donnée empilée qui ne peut être récupérée qu'après avoir dépilé toute la pile.
Un des intérêts d'une pile est que le temps d'accès aux données est constant ($O(1)$) puisqu'on n'agit que sur le dernier élément.



Une pile met en œuvre le principe du « dernier entré - premier sorti », noté LIFO (last-in first-out) en informatique. Les microprocesseurs gèrent nativement des piles. Ainsi, la pile d'exécution (call-stack) garde en mémoire les fonctions actives dans la machine et permet leur exécution séquentielle. D'autres usages de la pile sont fréquents en informatique : gestion historique des pages visitées dans un navigateur web, actions d'annulation dans un traitement de texte, parenthésage d'expression arithmétiques, exécution de fonction récursive.

2. Empiler et dépiler

Commençons par un petit exercice pour mieux comprendre la structure de pile. On considère une pile, initialement vide, et on adopte, pour simplifier, la notation suivante pour une succession d'opérations lues de gauche vers la droite : on note `*` une opération de dépilement et `A` une opération d'empilement d'un élément `A`. Par exemple `AB*A` correspond à la séquence d'opérations : empiler `A`, empiler `B`, dépiler puis empiler `A`.

- On considère la séquence d'opérations `PSI**P**SIP*S**IPS***I`. Quel est le contenu de la pile à la suite de ces opérations ? Quel est la succession des éléments dépilés écrite de gauche vers la droite ?
- Inversement, à partir de la séquence d'empilements `FACILE` (on doit empiler les éléments de `FACILE` une seule fois chacun), insérer des `*` pour que la succession des éléments dépilés donne, quand c'est possible :
  - `FACILE`
  - `ELICAF`
  - `CIFALE`
  - `AICLFE`

3. ADT d'une `pile`

Un **type de données abstrait (ADT)** – Abstract Data Type en anglais – est la description d'un ensemble organisé d'objets et des opérations de manipulation sur cet ensemble.
Nous pouvons définir de manière formelle l'ADT d'une pile dont le type sera noté `pile`. Avant cela, donnons une définition d'une pile.

Définition : Une **pile** est une structure de données linéaire qui suit le protocole LIFO.

Un jeu de fonctions minimal sur une ADT de type `pile` est le suivant.

Désignation	Fonction	Type objet retourné
`CreerPile()`	crée une pile vide	`pile`
`EstVide(p)`	retourne l'état vide ou non vide de la pile `p`	booléen
`TaillePile(p)`	retourne la taille de la pile `p`	entier
`Empiler(p,elt)`	empile `elt` au sommet de la pile `p`	rien : pile modifiée sur place
`Depiler(p)`	dépile le sommet de la pile	rien : pile modifiée sur place
`Sommet(p)`	retourne le sommet de la pile	type des éléments de la pile

II Implémentation python

Python ne gérant pas nativement les piles, les implémentations se font en utilisant des listes ou des tableaux. On oubliera par la suite la type de structure utilisé et on n'appliquera aux piles "simulées" sous Python que le jeu de fonctions décrit précédemment, comme si les piles étaient un type d'objet différent de celui réellement utilisé.

1. Implémentation à l'aide d'une liste

L'avantage d'utiliser une liste est la possibilité de manipuler des piles de tailles non bornées (à priori) et d'empiler des objets de natures différentes. Le défaut est par contre que certaines opérations ne se font pas avec un coût constant.

```
def CreerPile():
    return []

def EstVide(pile):
    return len(pile) == 0

def TaillePile(pile):
    return len(pile)

def Empiler(pile,elt):
    pile.append(elt)

def Depiler(pile):
    pile.pop()

def Sommet(pile):
    return pile[-1]
```

Avec une telle implémentation, il faut faire attention de ne pas dépiler une pile qui est déjà vide.

2. Implémentation à l'aide d'un tableau

On peut aussi créer des piles à l'aide de tableaux (objets de types array du module numpy) de la façon suivante : on définit au départ la taille maximale `n` des piles à manipuler, les piles sont alors des tableaux `P` de taille `n+1` pour lesquels le premier élément `P[0]` contient l'indice correspondant au sommet de la pile.
L'avantage d'un tableau est d'avoir en fait une longueur fixe donc le coût des opérations est constant ; en revanche, la taille des piles est bornée et les éléments d'une pile doivent être de la même nature (des entiers par exemple).

```
from numpy import array

n = 15
def CreerPile():
    return array([0]*(n+1))

def EstVide(pile):
    return pile[0] == 0

def TaillePile(pile):
    return pile[0]

def Empiler(pile,elt):
    pile[0] += 1
    pile[pile[0]] = elt

def Depiler(pile):
    pile[0] -= 1

def Sommet(pile):
    return pile[pile[0]]
```

Il faut cette fois penser à vérifier qu'une pile n'est pas vide avant de la dépiler mais aussi faire attention à ce qu'elle ne soit pas déjà pleine avant d'empiler un nouvel élément.

III Utilisations d'une pile

1. Navigateur Internet

Les pages web visitées par un internaute sont placées dans une structure composée de deux piles `p` et `q`. Lorsque le navigateur lit une page s (que l'on appellera « page actuelle »), cette page est un objet placé au sommet de la pile `p`. On supposera que pour lire une page donnée visitée avant la page actuelle, il suffit qu'elle se trouve au sommet de `p`.

— On donne la fonction suivante :

```
def Retour_arriere(p,q): # permet de revenir à la page précédente et de l'afficher
    if TaillePile(p) >=2 :
        a = Sommet(p)
        Empiler(q,a)
        Depiler(p)
        print(Sommet(p))
    else:
        print(Sommet(p))
```

Que se passe-t-il quand la pile `p` contient plus de 2 éléments ? Si elle contient 1 seul élément ? Quel est l'intérêt de la pile `q` ?

— Coder la fonction `Aller_avant(p,q)` qui permet de revenir à la page que l'on lisait avant d'avoir effectué un retour à la page précédente, et de l'afficher. Si une telle page n'existe pas (par exemple si on n'a jamais fait de retour en arrière), la fonction réaffichera la page actuelle.

— Coder la fonction `Nouvelle_page(p,q,s)` qui permet d'ouvrir une nouvelle page `s` en la plaçant au sommet de la pile `p` et qui l'affiche. Attention ! On ne peut plus alors que visiter des pages se trouvant encore dans la pile `p`.

Exemple de navigation à partir d'un navigateur vide

```
>>> p,q = CreerPile(), CreerPile()
>>> Nouvelle_page(p,q,'google')
google
>>> Nouvelle_page(p,q,'psimontaigne')
psimontaigne
>>> Aller_avant(p,q)
psimontaigne # on ne peut plus aller plus loin en avant
>>> Retour_arriere(p,q)
google
```

```
>>> Retour_arriere(p,q)
google # on ne peut pas aller plus loin en arrière
>>> Nouvelle_page(p,q,'scei')
scei
>>> Retour_arriere(p,q)
google
>>> Retour_arriere(p,q)
google
>>> Aller_avant(p,q)
scei # on n'a plus accès à 'psimontaigne'
```

2. vérification du parenthésage

Une expression est dite « bien parenthésée » si à chaque parenthèse ouvrante est associée une parenthèse fermante. Bien que l'on parle de parenthèse, cette définition vaut pour toute expression encadrée par des balises ouvrantes et fermantes. Dans la suite, nous nous limitons, sans perte de généralité toutefois, à des expressions parenthésées. Par exemple, l'expression :

`(a + b) * c - b * ( 2 * a - c)`

est bien parenthésée. Ce n'est pas le cas de l'expression :

`(a + b) * c - b * ( 2 * a - c`

où il manque une parenthèse.

Pour contrôler le bon parenthésage d'une expression, les piles sont une structure de données adaptée. En parcourant un à un les caractères d'une expression, chaque parenthèse ouverte détectée donne lieu à un empilement et chaque parenthèse fermante à un dépilement. La pile étant initialement vide, une expression est bien parenthésée si la pile est encore vide après parcours de l'expression (et si on ne rencontre pas d'erreur en chemin – par exemple en tentant de dépiler une pile déjà vide –). Deux situations particulières d'expression mal parenthésée requièrent cependant une attention particulière : trop de parenthèses ouvrantes ou trop de parenthèses fermantes. Dans le premier cas, la pile n'est pas vide après l'analyse de l'expression. Dans le second cas, la pile est vide alors qu'on cherche à dépiler. Le code suivant présente une fonction qui analyse le bon parenthésage d'une chaîne de caractères.

```
def parenthesage(exp):
    p = CreerPile()
    for elt in exp :
        if elt == "(":
            Empiler(p,1)
```

```python
        if elt == ")" :
            if EstVide(p):
                return False
            else :
                Depiler(p)
    return EstVide(p)
```

Exercice : Adapter ce code pour qu'il précise, dans le cas où le parenthésage est incorrect, la nature de l'erreur ('trop de parenthèses ouvrantes' ou 'trop de parenthèses fermantes').
Exercice : Adapter le code précédent pour contrôler à l'aide d'un dictionnaire le bon parenthésage d'expressions comportant trois types de parenthèses : ( ), [ ] et { } Par exemple : `{ [(a + b) * c - b] -d } * e` est bien parenthésée.

3. Evaluations d'expressions infixes et postfixes

Comment évaluer une expression algébrique ? Pour quiconque a appris les règles élémentaires du calcul, la réponse est simple : connaître les quatre opérations +, -, *, / et les règles de préséance. Ainsi, le calcul de (1 + 2) * 3 se compose d'une addition suivie d'une multiplication. La simple lecture de gauche à droite de l'expression permet son évaluation. Ce n'est pas le cas de l'expression 1 + 2 * 3 qui nécessite une lecture de toute l'expression avant de pouvoir l'évaluer, la multiplication étant prioritaire sur l'addition. Cette observation pose la question de l'évaluation automatique d'une expression arithmétique par des machines. L'idéal serait une évaluation au fur et à mesure de la lecture, de gauche à droite. A cette première constatation suit celle de l'analyse syntaxique, par des machines encore, des expressions algébriques. Par exemple, la lecture de gauche à droite de l'expression 1 + 2 ne permet de faire l'opération qu'une fois les deux opérandes 1 et 2 et l'opération + lus. Un algorithme susceptible d'analyser cette expression ne pourrait donc faire l'opération qu'après avoir identifié le deuxième opérande. Et cela ne suffirait pas encore puisque la lecture de l'expression 1 + 2 * 3 révèle qu'il faut avoir fait la multiplication pour, ensuite seulement, faire l'addition. Ces difficultés sont liées à l'adoption d'une notation, dite notation infixe, comme pour 1 + 2. Pratique pour l'humain, cette notation présente des inconvénients majeurs pour la machine. D'autres notations existent qui placent le signe avant ou après les deux opérandes. On les qualifie respectivement de notation **préfixe** (+ 1 2) et de notation **postfixe** (1 2 +). La notation postfixe est aussi appelée notation polonaise inversée. Elle est d'un usage courant pour les machines à calculer Hewlett-Packard. Les deux notations préfixe et postfixe présentent un avantage indéniable sur la notation infixe : elles rendent les parenthèses inutiles. L'expression (1 + 2) * 3 en notation infixe devient 1 2 + 3 * en notation postfixe. Sa lecture de gauche à droite suffit pour évaluer correctement l'expression. On commence par lire les deux opérandes 1 et 2. En arrivant sur le signe +, on fait l'addition des deux derniers opérandes connus 1 et 2. Le résultat obtenu, à savoir 3, constitue un opérande pour la suite du calcul qui se poursuit en lisant alors l'opérande 3 puis le signe *. On fait le produit des deux derniers opérandes connus 3 et 3. Le résultat final est 9. S'il est possible de convertir une expression infixe en une expression postfixe, il reste à trouver un algorithme susceptible d'évaluer cette dernière. Une pile est une structure de données adaptée pour y parvenir. Au fur et à mesure de la lecture d'une expression postfixe, les opérandes sont empilés. Dès qu'une opération est rencontrée, les deux derniers opérandes sont dépilés et placés en mémoire. L'opération est évaluée puis le résultat est empilé. Quand le calcul se termine, la pile est vide.
Exercice : Compléter le code suivant afin qu'il évalue une expression postfixe initialement définie par une chaîne de caractères.

```python
# Calcul de l'opération mathématique
# op1 op op2 (notation infixe)
# op1 , op2 : opérandes
# op : opération
def calc (op,op1,op2):
    if op == '*':
        return op1*op2
    elif op == '/':
        return op1/op2
    elif op == '+':
        return op1+op2
    else:
        return op1-op2
```

```python
# Fonction d'évaluation d'une expression postfixe
def postfixe (exp):
    # conversion de la chaîne de caractères en une liste de mots, les séparateurs étant par défaut les espaces:
    liste_op = exp.split()
    # à compléter...

>>> postfixe('1 2 + 3 *')
9.0
```

4. Encore des palindromes !

Exercice : Écrire une fonction `palindrome(chaine)` qui détecte à l'aide d'une pile si la chaîne de caractères `chaine` de longueur n est un palindrome.