Structures abstraites 1
TD : Implémentation d'une pile

Rappel : Une structure de données est une organisation logique des données permettant de simplifier ou d'accélérer leur traitement. Une pile est liste particulière dans laquelle le dernier élément arrivée est le premier sorti (principe **LIFO**)

La structure de pile est un concept abstrait défini par son interface.

Je vous propose donc ici deux implémentations possibles. L'idée principale étant que l'interface (les fonctions de base) pourront être utilisées indépendamment de l'implémentation choisie. Seule l'efficacité des opérations sera modifiée.

1) Préliminaires :

Sans utiliser le cours, rappelez les trois primitives de l'interface d'une Pile. Relisez le cours si vous ne trouvez pas.

2) Implémentation n°1

Dans cette implémentation, 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.

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

def est_vide(p):
    """renvoie True si la pile est vide
     et False sinon"""
    return p == []

def empiler(p, x):
    """Ajoute l'élément x à la pile p"""
    p.append(x)

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

Voici les fonctions de base :

❐ — Exercice **1** —

- Télécharger le programme TD_Pile.py depuis bouillotvincent.github.io et tester les instructions suivantes:
- Dans le programme principal, à l'aide d'une boucle while, écrivez un algorithme permettant de dépiler totalement la pile.

```
p = pile()
for i in range(5):
    empiler(p, 2*i)
a = depiler(p)
print(a)
print(est_vide(p))
```

❐ — Exercice **2** —

- Réaliser les fonctions **taille(p)** et **sommet(p)** qui retourne respectivement la taille de la liste et le sommet de la pile, sans le supprimer.

3) Implémentation n°2

Nous allons créer une classe Pile pour implémenter cette structure.
Voici la classe que nous allons utiliser :

```
class Pile:
    """ classe Pile :
    création d'une instance Pile avec à partir d'une liste """
    def __init__(self):
        """Initialisation d'une pile vide"""
        self.L = []

    def est_vide(self):
        """teste si la pile est vide"""
        return

    def depiler(self):
        """dépile"""

        return

    def empiler(self, x):
        """empile"""
```

❐ — Exercice **3** —

- En vous aidant de la première implémentation, complétez la classe Pile dont le prototype a été fourni ci-dessus.

❐ — Exercice **4** —

- Télécharger le programme TD_Pile.py depuis bouillotvincent.github.io et tester les instructions suivantes:
- Dans le programme principal, à l'aide d'une boucle while, écrivez un algorithme permettant de dépiler totalement la pile

```
p = Pile()
for i in range(5):
    p.empiler(2*i)
print(p.L)
a = p.depiler()
print(a)
print(p.L)
print(p.est_vide())
```

❐ — Exercice **5** —

- Réaliser une méthode récursive **taille(self)** qui renvoie la taille de la pile
- Réalisez une méthode **sommet(self)** qui renvoie le sommet de la liste sans le supprimer.

4) Un exemple d'utilisation : Contrôle du parenthésage d'une expression

Dans le cadre le logiciel de calcul formel, on est amené à évaluer des expressions mathématiques.
Il s'agit ici d'écrire une fonction qui contrôle si une expression mathématique, donnée sous forme d'une chaîne de caractères, est bien parenthésée, c'est- à-dire s'il y a autant de parenthèses ouvrantes que de fermantes, et qu'elles sont bien placées.

Exemple :
(..(..)..) est bien parenthésée
(...(..(..)...) ne l'est pas

L'algorithme :

- On crée une pile
- On parcourt l'expression de gauche à droite.
- À chaque fois que l'on rencontre une parenthèse ouvrante "(" on l'empile
- Si on rencontre une parenthèse fermante ") " et que la pile n'est pas vide on dépile (sinon on retourne faux)
- À la fin la pile doit être vide...

❐ — Exercice **1** —

- Écrire cet algorithme en langage naturel (aussi appelé pseudo-code).

❒ — Exercice **2** —

- En utilisant l'une des structures Pile réalisées plus haut, écrire une fonction `verification(expr)` qui vérifie si une expression mathématique passée en paramètre est correctement parenthésée.
- Modifiez votre programme afin que celui-ci prenne en compte les crochets ouvrants et fermants.

❒ — Prolongement **1** —

- Si vous avez le temps, vous pouvez modifier votre programme afin que celui-ci indique où est l'erreur de parenthèsage — voire qu'il propose une (ou des) corrections possibles.