DS Pile et Files en Python

Exercice 1 : Piles

La classe *Pile* utilisée dans cet exercice est implémantée en utilisant des **listes Python**. et propose quatre méthodes :

- Le constructeur qui permet de créer une pile vide.
- Une méthode *est_vide* qui renvoie *True* si la pile est vide, *False* sinon.
- La méthode *empiler* qui prend un entier en paramètre et l'ajoute au sommet de la pile.
- La méthode *dépiler* qui retire l'entier au sommet de la pile et le renvoie.

On considère que la représentation de la pile affiche le sommet comme l'élément le plus à droite.

Question 1 :
La méthode suivante `est_triee` renvoie *True* si la pile est triée dans l'ordre croissant, *False* sinon.

1. **Recopier sur la copie le numéro de la ligne associée et la ligne complétée.**

```
def est_triee(self):
    if not self.est_vide():
        e1 = self.depiler()
        while not ... :
            if e1 ... e2 :
                return False
            e1 = ...
        return True
```

Question 2:
Écrire la méthode `element_maximum` qui renvoie la valeur maximale de la pile. Les autres éléments dépilés doivent rester dans la pile..

Question 3:
On se munit d'une méthode `traiter` et de la pile B précédente représentée par [9, -7, 8, 12, 4].

```
def traiter(self):
      q = Pile()
      while not self.est_vide():
         q.empiler(self.depileMax())
      while not q.est_vide():
         self.empiler(q.depiler())
```

1. **Donner l'état des deux piles B et q lors des 4 premières itérations de la première boucle**.
2. **Que réalise ce traitement ?**

Exercice 2 : Files

On propose deux structures de données :

Pile	File
`creer_pile_vide()` : Qui renvoie une pile vide.	`creer_file_vide()` : Qui renvoie une file vide.
`est_vide()` : Renvoie *True* si la pile est vide, *False* sinon.	`est_vide()` : Renvoie *True* si la file est vide, *False* sinon.
`empiler()` : Prend une chaine de caractère en paramètre et l'empile au sommet de la pile.	`enfiler()` : Prend une chaine de caractère en paramètre et l'enfile à la queue de la file.
`depiler()` : Renvoie l'élément au sommet de la pile et le retire.	`defiler()` : Renvoie l'élément à la tête de la file et le retire.

Question 1 :

On considère la file F suivante :

$\underline{"rouge","vert","jaune","rouge","jaune".}$

La flèche correspond au sens de défilement.

1. Quel sera le contenu de la pile P et de la file F après l'exécution du programme Python suivant?

```
P = creer_pile_vide()
while not est_vide(F):
   P.empiler(defiler(F))
```

2. Corriger la méthode `taille_file` suivante qui renvoie le nombre d'élément de la file.

```
def taille_file(self):
    """Retourne la taille de la file sans utiliser len
    """
    taille = 1
    while self.est_vide() == True:
        file.defiler()
        taille = 1
    return taille
```

Question 2

1. **Compléter la méthode `former_pile` suivante qui prend en paramètre une file F et qui ajoute les éléments de la pile dans le même sens que la file.**
   Recopier sur la copie le numéro de la ligne et la ligne complétée

```
1  def former_pile(file):
2      """
3      Prend une file en paramètre et renvoie une pile.
4      Le sommet de la pile correspond à la tête de file.
5      """
6      pile = Pile()
7      temp = Pile()
8      while not file.est_vide():
9          val = ...
10         temp.empiler(val)
11     while not temp. ... :
12         ... (temp.depiler())
13     return pile
```

Exemple :

$$\hline \text{"}rouge\text{"}, \text{"}vert\text{"}, \text{"}jaune\text{"}, \text{"}jaune\text{"}, \text{"}vert\text{"} \hline$$

Après l'appel, la pile correspondante sera :

$$\begin{array}{|c|}\hline \text{"vert"} \\ \hline \text{"jaune"} \\ \hline \text{"jaune"} \\ \hline \text{"vert"} \\ \hline \text{"rouge"} \\ \hline \end{array}$$

Question 3

Écrire une méthode de file `occurence_element` qui prend en paramètre un élément et qui renvoie le nombre de fois où l'élément apparait dans la file. La file doit conserver tous ses éléments dans le même ordre après appel de cette méthode.

Exercice 3 : Problèmes sur les piles

Les questions suivantes seront indépendantes et traiteront d'algorithmes sur les piles

Question 1 : Écriture polonaise inversée

La pile suivante est définie par une liste Python, on dispose des méthodes `pop` et `append`.
On dispose de la pile P suivante (le sommet se situe à gauche sur la représentation) :
P = [6 3 7 + *].
Cette pile représente l'expression mathématique suivante : (6 + 3) * 7.

Rappel : L'algorithme se déroule de la manière suivante :
Tant que l'élément dépilé est un entier, on le place dans une pile temporaire.

Si l'élément suivant est un caractère représentant un opérateur (+,*,-,/), on dépile deux entiers a et b de la pile temporaire et on empile le résultat de l'opération a `operateur` b sur la pile temporaire.

Compléter la fonction `polonaise_inversee` suivante qui prend en paramètre une pile que l'on considère correcte pour cet algorithme et renvoie l'évaluation de l'expression mathématique associée.
Recopier sur la copie le numéro de la ligne associée et la ligne complétée.

```
1  def polonaise_inversee(liste : list) -> int:
2      """Fonction qui calcule la valeur d'une expression en notation polonaise inversée"""
3      pile = ...
4      for element in liste:
5          if element in "+-*/": #Si l'élément est un operateur"
6              b = pile.pop(0)
7              a = pile.pop(...)
8              if element == "+":
9                  pile = pile + [a+b]
10             elif element == ... :
11                 pile = pile + [a-b]
12             elif element == ... :
13                 pile = pile + [a*b]
14             elif element == ... :
15                 pile = pile + [a/b]
16         else: #Si l'élément n'est pas un opérateur
17             pile = pile + [int(...)]
18     return ...
```

Bonus: Vérification de parenthésage d'expression

La pile suivante est définie par une pile modélisée en Programmation Orientée Objet.
On dispose des opérations suivantes : `depiler()`, `empiler()`, `est_vide()`, `creer_pile_vide()`.

On souhaite vérifier qu'une expression mathématique est correctement parenthésée, c'est-à-dire que chaque parenthèse ouvrante est couplée à une parenthèse fermante.
L'algorithme se déroule de cette manière :
On regarde pour chaque caractère de la chaine:

- Si le caractère est une parenthèse ouvrante, on l'empile dans une pile.
- Si le caractère est une parenthèse fermante, on dépile un élément de la pile.
  Si la pile est vide à la fin de l'exécution de la fonction, l'expression est bien parenthésée, sinon elle ne l'est pas.

Écrire une fonction `verification_parenthesage` qui prend en paramètre une chaine de caractère représentant une expression et renvoie *True* si l'expression est bien parenthésée, *False* sinon.

```
>>> verification_parenthesage('(4+2)*6)')
False
>>> verification_parenthesage('((4+3)*(5+6))')
True
```