Python – Piles

Une pile avec une liste Python

Les listes Python ont une interface qui leur permet d'être utilisées comme des piles. La méthode `append(v)` sert à empiler et `pop()` à dépiler. Les éléments en haut de la pile se trouvent alors à la fin de la liste.

```
>>> l = []
>>> l.append(3)
>>> l.append(5)
>>> l.append(7)
>>> l.pop()
7
>>> l.append(2)
>>> l.pop()
2
>>> l.pop()
5
>>> l.pop()
3
>>> l
[]
```

Exercice 1 : Écrire les fonctions suivantes en représentant les piles avec des listes Python.

Fonction	Description
`cree_pile()`	Renvoie une nouvelle pile vide.
`empile(element, pile)`	Rajoute `element` au sommet de `pile`.
`depile(pile)`	Renvoie l'élément se trouvant au sommet de `pile`, qui ne doit pas être vide, et l'enlève.
`est_vide(pile)`	Renvoie un booléen indiquant si `pile` est vide ou non.

```
>>> pile = cree_pile()
>>> empile(4, pile)
>>> empile(5, pile)
>>> depile(pile)
5
>>> empile(3, pile)
>>> depile(pile)
3
>>> est_vide(pile)
False
>>> depile(pile)
4
>>> est_vide(pile)
True
```

Vérification de parenthésage

La vérification du parenthésage est une étape importante lors de l'analyse d'une expression mathématique ou même lors de l'interprétation/compilation d'un programme informatique. Par exemple `'(())'` et `'(()())'` sont valides mais pas `'(()'` ou `'(()))'`.
Pour l'instant nous allons supposer que les expressions sont uniquement composées de `'('` et `')'`.

Exercice 2 : 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 `depile`, votre fonction doit la récuperer et renvoyer `False`.

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

Nous allons maintenant rajouter les accolades `'{}'` et les crochets `'[]'`. Ainsi `'([{}])'` est valide mais pas `'([{]})'`.

Exercice 3 : Modifier `verif_parentheses(expression)` 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
```

La notation polonaise inversée

La **notation polonaise inversée** est une façon de représenter les expressions mathématiques en mettant les opérateurs à la fin. On parle aussi d'expressions suffixées. Par exemple "4+6" se note "4 6 +". Cette notation a l'avantage de ne pas nécessiter d'utiliser des parenthèses. En effet, les expressions "1+2×3" et "(1+2)×3" se traduisent respectivement par "1 2 3 × +" et "1 2 + 3 ×". Pour la soustraction, l'opération se fait en soutrayant le nombre de droite à celui de gauche. Ainsi "5 – 2" se traduit par "5 2 –".
Pour évaluer une expression, on a la lit de gauche à droite en faisant les calculs dès que possible avec les deux valeurs précédant l'opérateur. Voici l'exemple d'évaluation de ces deux expressions :

$$1\ \underbrace{2\ 3\ \times}\ +\qquad \underbrace{1\ 6\ +}\qquad 7$$

$$\underbrace{1\ 2\ +}\ 3\ \times\qquad \underbrace{3\ 3\ \times}\qquad 9$$

Cette notation a notamment été utilisée par les calculatrices HP. Un utilisateur habitué a cette notation peut faire les calculs plus vite qu'avec la notation classique.

Exercice 4 (sur papier) : Calculer le résultat des expressions suivantes :

1) 4 5 × 2 + 2) 1 1 + 1 1 + × 3) 1 2 7 3 − × + 4) 4 5 + 2 ×

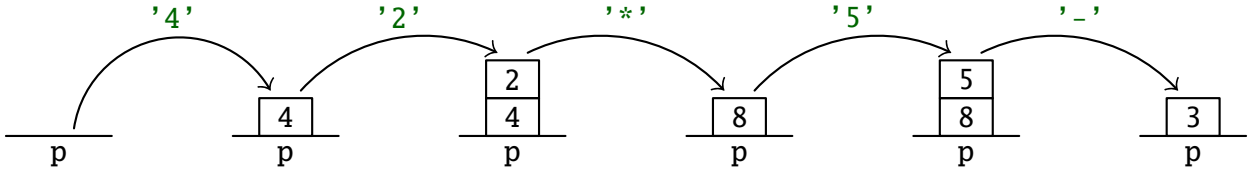
Exercice 5 (sur papier) : Traduire les expressions suivantes en notation polonaise inversée. Il faut faire attention aux priorités.

1) $5+7\times 2$
2) $1+2+3+4$
3) $5-3-1$
4) $(3+4)\times(7-2)$

Afin de représenter les expressions, nous allons utiliser des textes où chaque élément est séparé par un espace. Par exemple : `'4 6 +'` ou `'1 2 + 3 *'`. Afin d'obtenir les différents éléments, il faut utiliser la méthode `split()` des textes :

```
>>> '1 2 + 3 *'.split()
['1', '2', '+', '3', '*']
```

On obtient alors une liste où chaque élément est un des éléments de l'expression. On supposera que toutes les expressions utilisées sont valides et on ne testera pas si elles le sont.
Afin d'évaluer l'expression, il faut construire une pile avec les valeurs déjà lues. Lorsqu'on voit un opérateur, on dépile alors les deux dernières valeurs et on rempile le résultat de l'opération. Les opérateurs que vous devez manipuler sont +, * et -. Il faudra bien penser à convertir les nombres à l'aide de la fonction `int(nb)` avant de les empiler.
Voici l'évolution de la pile lors de l'évaluation de `'4 2 * 5 -'`.



Exercice 6 (sur papier) : Décrire les différents états de la pile lors de l'évaluation des expressions suivantes :

1) `'9 1 - 3 +'` 2) `'2 3 4 + *'`

Exercice 7 (sur papier) : On appelle `p` la pile utilisée. On note `s` le symbole lu dans l'expression.

1) Quels sont les différents cas possibles pour `s` ?
2) Décrire les actions à faire pour mettre à jour la pile si `s` est un nombre.
3) Décrire les actions à faire pour mettre à jour la pile si `s == '+'`.
4) Combien de valeurs restent dans la pile à la fin de l'évaluation d'une expression valide ?

Exercice 8 : Écrire une fonction `evaluation(expression)` qui renvoie le résultat de l'évaluation de `expression` qui est un texte correspondant à une expression en notation polonaise inversée.

```
>>> evaluation('3 4 +')
7
>>> evaluation('2 3 4 + *')
14
```

```
>>> evaluation('2 3 + 5 4 + *')
45
>>> evaluation('2 3 -')
-1
```

Exercice 9 : Déterminer le résultat des expressions suivantes :

1) `'5 3 4 + * 2 2 2 1 2 + * + * -'`
2) `'1 1 + 1 1 + * 1 1 + * 1 1 + * 1 1 + *'`
3) `'1 2 + 3 4 + * 5 6 + 7 + 8 * 9 - +'`
4) `'1 2 3 4 5 6 7 8 9 + * + * + * + *'`

Faire une pile avec une liste mutable

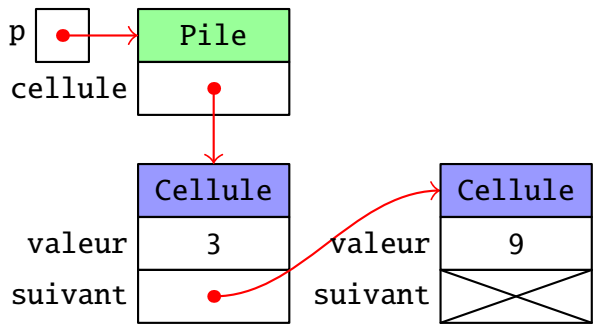
Comme pour les listes, il y a plusieurs façons de représenter une pile. Nous allons reprendre une structure de liste chaînée mutable, c'est-à-dire dont le contenu peut être modifié.

```
class Cellule:
    def __init__(self, valeur, suivant):
        self.valeur = valeur
        self.suivant = suivant

class Pile:
    def __init__(self):
        self.cellule = None

    def est_vide(self):
        return self.cellule is None
```

La pile vide ne contient qu'un pointeur vers **None**. Chaque nouvel élément est rajouté au début de la pile. Pour dépiler, c'est l'élément au début de la pile qui est enlevé.

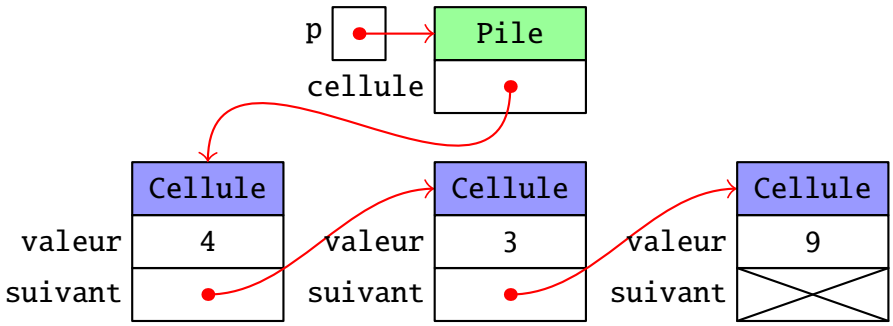


Exercice 10 (sur papier) : Répondre aux questions suivantes en utilisant le schéma-ci-dessus.

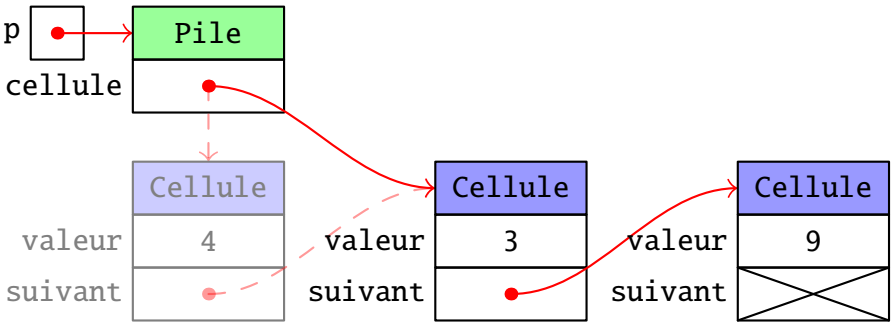
1) Quelle est la valeur de l'expression de `p.cellule.valeur` ?
2) Quelle expression permet d'obtenir la valeur 9 se trouvant dans la dernière cellule ?
3) Compléter l'expression afin d'obtenir la figure de l'exercice suivant :

```
p.cellule = Cellule(..., ...)
```

Exercice 11 : Écrire la méthode `empile(self, v)` qui rajoute une nouvelle cellule contenant `v` au début de la pile. Voici à quoi ressemblera la liste ci-dessus après l'ajout de `4` en tête de pile.



Exercice 12 : Écrire la méthode `depile(self)` qui renvoie la valeur du premier élément de la pile et l'enlève. Si la pile est vide, il faut lever l'exception `IndexError` avec le message `'pile vide'`.



Exercice 13 : Écrire une méthode `affiche(self)` qui affiche chaque élément de la pile, ligne par ligne. L'élément au sommet est affiché en premier. Si la pile est vide, rien n'est affiché. Pour faire cela, on peut créer une pile temporaire, dépiler la pile dans cette nouvelle pile, en affichant les éléments au fur et à mesure, puis à réempiler les éléments dans la pile d'origine.

```
>>> p = Pile()
>>> p.empile(5)
>>> p.empile(99)
>>> p.affiche()
99
5
```

Pour aller plus loin

Exercice 14 : Rajouter et compléter la méthode `__str__(self)` qui renvoie un texte correspondant au contenu de la pile, sans la dépiler. Si la pile est vide, le texte renvoyé est `'vide'`, sinon les éléments sont affichés de gauche à droite en les séparant par `'->'`.

```
class Pile:
    ...

    def __str__(self):
        if ...:
            return "vide"
        res = str(...) # contenu de la première cellule
        suivant = ... # cellule d'après
        while suivant is not None:
            res = ... + " -> " + ...
            suivant = ... # on passe à la cellule d'après
        return res
```

```
>>> p = Pile()
>>> print(p)
vide
>>> p.empile(5)
>>> p.empile(99)
>>> print(p)
99 -> 5
```

Exercice 15 : Rajouter une méthode `copie(self)` qui renvoie une nouvelle pile qui contient les mêmes éléments, dans le même ordre que `self`. Vous ne pouvez utiliser que les méthodes de la classe et pas les attributs, comme `self.cellule` ou `self.cellule.valeur`. Par contre, vous pouvez créer une nouvelle pile, qui servira à dépiler la pile principale. Puis les éléments seront à nouveau ré-empilés dans la pile principale et également dans la copie.

Exercice 16 : Rajouter une méthode `renverse(self)` qui renverse l'ordre des éléments de la pile. Celui qui était en premier passe en premier et ainsi de suite. Vous ne pouvez qu'utiliser les méthodes usuelles des piles. Vous pouvez créer plusieurs piles temporaires qui serviront à faire le retournement.