Pile et File

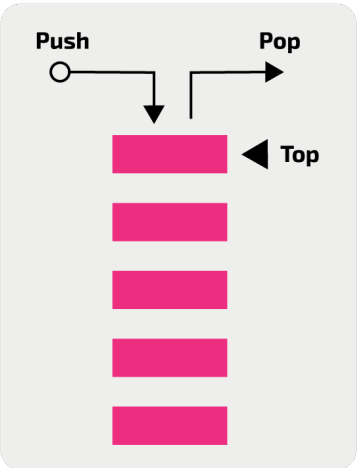
Les piles

La notion de pile est une notion fondamentale en informatique. Tout processeur utilise une pile.
Une pile informatique est tout à fait comparable à une pile d'assiettes. Quand on range une assiette dans une pile, on la pose sur le sommet de la pile et quand on veut une assiette dans une pile, on prend celle qui est au sommet. Dans une pile informatique, c'est le même principe, on a accès uniquement au sommet de la pile, les autres éléments étant invisibles.

Pour les piles, il existe deux opérations :

- La fonction empile (*push* en anglais) qui ajoute un élément au sommet de la pile.
- La fonction dépile (*pop* en anglais) qui retourne l'élément qui est au sommet de la pile en le supprimant de la pile.

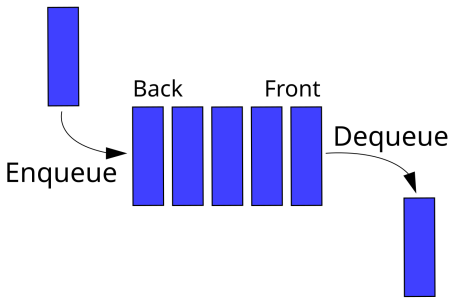
À ces deux opérateurs, on ajoute une fonction de création de pile et une fonction pour savoir si la pile est vide.



Dans la littérature anglo-saxonne, les piles se nomment *stack*. Dans une pile, le dernier élément rentré est le premier qui sort ce qui donne en anglais : (*Last In, First Out*), on parle de façon abrégée de pile *LIFO*.

Les files

Les files sont comparables au file d'attente d'une caisse d'un supermarché, la première personne qui est rentrée dans la file est la première qui doit en sortir. En informatique, nous avons aussi besoin de gérer des files.



Pour les files, il existe deux opérations de base :

- La fonction enfile (*enqueue* en anglais) qui ajoute un élément en fin de file.
- La fonction défile (*dequeue* en anglais) qui renvoie l'élément qui est à l'avant de la file en le supprimant de la file.

À ces deux opérateurs, il faut ajouter une fonction de création de file et une fonction pour savoir si la file est vide.

Dans la littérature anglo-saxonne, les files se nomment *queue*. Dans une file, le premier élément rentré est le premier qui sort ce qui donne en anglais : (*First In, First Out*), on parle de façon abrégée de file *FIFO*.

Exercice 1 : Si les fonctions étaient implémentées, qu'afficheraient les codes suivants ?

```
# Code 1 : pile
p = cree_pile()
empile(p, 1)
empile(p, 2)
empile(p, 3)
print(p)      # ???
a = depile(p)
print(p)      # ???
empile(p, 4)
empile(p, a)
print(p)      # ???
```

```
# Code 2 : file
f = cree_file()
enfile(f, 1)
enfile(f, 2)
enfile(f, 3)
print(f)      # ???
a = defile(f)
print(p)      # ???
enfile(f, 4)
enfile(f, a)
print(f)      # ???
```

Exercice 2 : Implémentation d'une pile en utilisant des listes

On utilise les listes Python pour implémenter notre pile.

```
# Initialisation d'une pile vide
def cree_pile() :
    return []
```

```
# Création d'une pile vide
p = cree_pile()
```

1. Écrire la fonction `empile` qui prend en paramètre une pile `p`, un élément `a` et ajoute l'élément à la pile.

2. Écrire la fonction `pile_vide` qui renvoie le booléen correspondant au caractère vide de la pile.

3. Écrire la fonction `depile` qui prend en entrée une pile `p` et retourne l'élément qui est au sommet de la pile tout en le supprimant de la pile.

4. Tester votre code avec le code 1 de l'exercice 1.

5. Écrire la fonction `getSommet` renvoie l'élément présent au sommet de la pile sans le dépiler, et `None` si la pile est vide.

6. En utilisant les quatre fonctions précédentes, écrire une fonction `somme_pile` qui additionne les éléments d'une pile d'entiers. Tester votre code.

Exercice 3 : Implémentation d'une pile en utilisant des listes et la programmation orientée objet

```
class Pile :
    def __init__(self) :
        self.contenu = []
```

```
# Création d'une pile vide
p = Pile()
```

1. Écrire les méthodes `empile`, `pile_vide`, `depile` et `getSommet` sur le modèle de l'exercice 2.

2. Écrire la méthode `__repr__` qui retourne une chaîne de caractère représentant le contenu de la pile. La représentation sera de la forme :

```
|élément_au_sommet|
|élément|
|...|
|élément_au_fond|
```

3. Adapter le code 1 de l'exercice 1 à la programmation orientée objet pour pouvoir tester votre code.

Exercice 4 : Implémentation d'une file en utilisant des listes

On utilise les listes Python pour implémenter notre file.

```
# Initialisation d'une file vide
def cree_file() :
    return []
```

```
# Création d'une file vide
f = cree_file()
```

1. Écrire la fonction `enfile` qui prend en paramètre une file `f`, un élément `a` et ajoute l'élément à la file.

2. Écrire la fonction `file_vide` qui renvoie le booléen correspondant au caractère vide de la file.

3. Écrire la fonction `defile` qui prend en entrée une file `f` et retourne l'élément qui est à l'avant de la file tout en le supprimant de la file.

4. Tester votre code avec le code 2 de l'exercice 1.

5. Écrire la fonction `getTete` renvoie l'élément présent à l'avant de la file sans le défiler, et `None` si la file est vide.

6. En utilisant les quatre fonctions précédentes, écrire une fonction `somme_file` qui additionne les éléments d'une file d'entiers. Tester votre code.

Exercice 5 : Implémentation d'une file en utilisant des listes et la programmation orientée objet

```
class File :
    def __init__(self) :
        self.contenu = []
```

```
# Création d'une file vide
f = File()
```

1. Écrire les méthodes `enfile`, `file_vide`, `defile` et `getTete` sur le modèle de l'exercice 3.

2. Écrire la méthode `__repr__` qui retourne une chaîne de caractère représentant le contenu de la file. La représentation sera de la forme :

```
dernier_élément->élément->...->premier_élément.
```

3. Adapter le code 2 de l'exercice 1 à la programmation orientée objet pour pouvoir tester votre code.

Exercice 6 : Parenthésage

Écrire une fonction `bon_parenthesage` qui prend en argument une chaîne de caractère représentant une expression algébrique, et vérifie à l'aide d'une pile le bon parenthésage de l'expression.
L'algorithme est le suivant :
On parcours la chaîne de caractères et on empile/dépile un jeton si le caractère est respectivement une parenthèse ouvrante/fermante. Dépiler une pile vide indique une erreur. Une pile non vide à la fin de l'algorithme indique une erreur.
Utiliser la classe `Pile` de l'exercice 3.

```
assert bon_parenthesage( "(1+1)" ) == True
assert bon_parenthesage( "((1+1)" ) == False
assert bon_parenthesage( ")(1+1" ) == False
assert bon_parenthesage( "(8(94+5)))-8)*(5(75+(2-56)))" ) == False
assert bon_parenthesage( "(8(94+5)-8)*(5(75+(2-56)))" ) == True
```

Exercice 7 : Implémentation d'une file avec deux piles

1. Écrire une fonction retourne qui prend en argument deux piles et "retourne" la deuxième pile sur la première. Si la deuxième pile est vide, la fonction ne fait aucune action.

2. Dans cette partie, nous allons implémenter une file à l'aide de deux piles. L'idée est la suivante : le sommet de la première pile (la pile tête) correspond à l'avant de la file, tandis que le sommet de la seconde pile (la pile queue) correspond à l'arrière de la file.
Les éléments entrant dans la file sont empilés dans la pile queue. Pour défiler, on dépile la pile tête. Si la pile tête est vide, la seconde pile peut être "retournée" sur la première. Et on est alors en mesure de défiler. La file est vide lorsque les deux piles le sont.

(a) Nous disposons de la file `f1` dons la tête est la pile `tete = [1, 2]` et la pile `queue = [3, 4, 5]`. Établir les valeurs de deux piles composant la file f1 à chacune des actions suivantes :

```
f1 = File()
f1.tete.contenu = [1, 2]
f1.queue.contenu = [3, 4, 5]
f1.defile()
f1.enfile(2)
f1.defile()
f1.enfile(3)
f1.defile()
f1.defile()
f1.defile()
```

(b) Les méthodes ne feront appel qu'aux méthodes définies pour les piles. On définie la classe `File` :
Écrire les méthodes `file_vide` , `enfile` , `defile`. Tester votre code avec le code précédent.

```
class File :
    def __init__(self):
        self.tete = Pile()
        self.queue = Pile()
```

Exercice 8 : Hors programme – Pile et file avec des listes chaînées

Une liste chaînée est une structure de donnée permettant, comme les tableaux, de stocker plusieurs valeurs.
Une liste chaînée est composé de maillons, contenant les informations suivantes :

- La valeur à stocker ;
- Une référence vers le maillon suivant ;
- Dans le cas d'une liste doublement chaînée, une référence vers le maillon précédent.

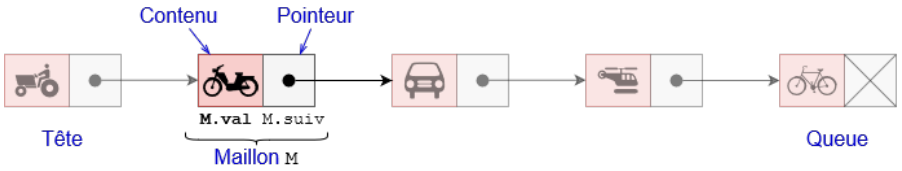


Illustration d'une liste simplement chaînée

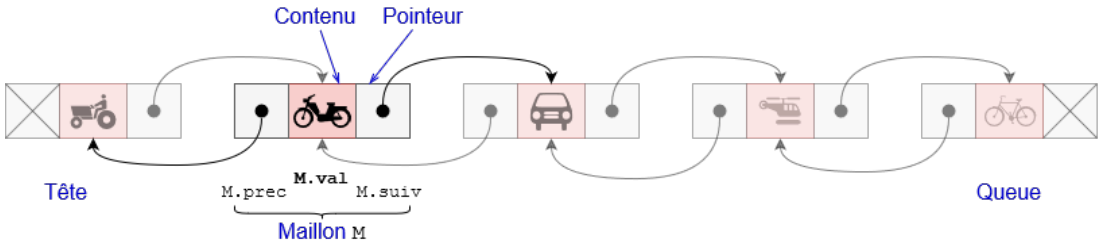


Illustration d'une liste doublement chaînée

Les classes `Pile` et `File` doivent gérer les créations et les suppressions de maillons. Les signatures des méthodes restent inchangées peu importe la façon avec laquelle on implémente le stockage des éléments.

```
class Maillon :
    def __init__(self) :
        self.valeur = None
        self.suivant = None
```

```
class PileChainee :
    def __init__(self) :
        self.tete = None
p =  PileChainee()
```

1. Écrire les méthodes `empile`, `pile_vide`, `depile` et `getSommet` de la classe `PileChainee`.

2. Écrire la méthode `__repr__` qui retourne une chaîne de caractère représentant le contenu de la pile. La représentation sera de la forme :

```
|élément_au_sommet|
|élément|
|...|
|élément_au_fond|
```

3. Adapter le code 1 de l'exercice 1 à la programmation orientée objet pour pouvoir tester votre code.

```
class MaillonDouble :
    def __init__(self) :
        self.valeur = None
        self.suivant = None
        self.precedent = None
```

```
class FileChainee :
    def __init__(self) :
        self.tete = None
        self.queue = None
f =  FileChainee()
```

4. Écrire les méthodes `enfile`, `file_vide`, `defile` et `getTete` de la classe `FileChainee`.

5. Écrire la méthode `__repr__` qui retourne une chaîne de caractère représentant le contenu de la file. La représentation sera de la forme :

```
dernier_élément->élément->...->premier_élément.
```

6. Adapter le code 2 de l'exercice 1 à la programmation orientée objet pour pouvoir tester votre code.