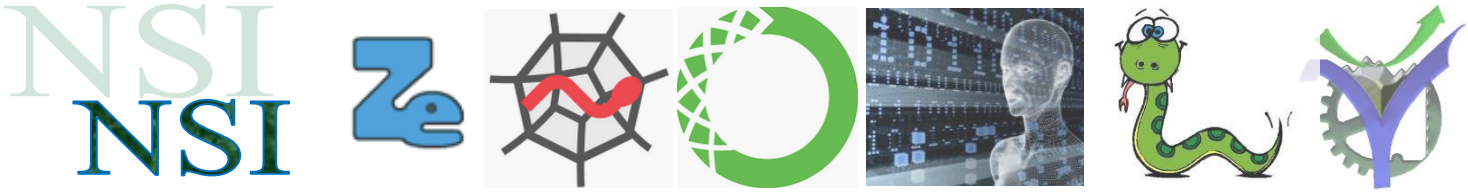
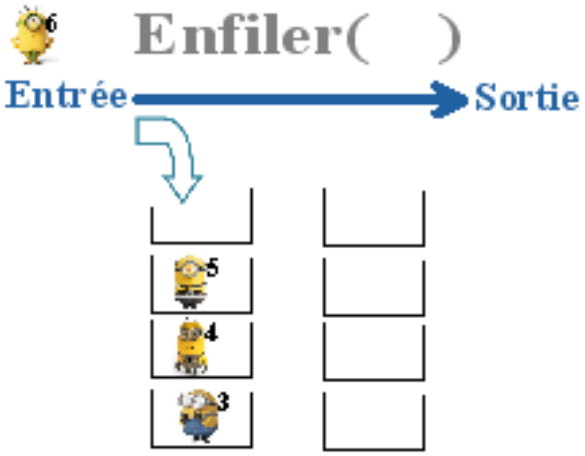


Structures de données : les piles et les files

Liste, Pile, File



Résumé :

Après la description du TAD liste, poursuite de la découverte des types abstraits de données avec trois nouveaux types très utilisés dans la suite :
Les Piles, les Files et les Files à priorité.

Sommaire

1 Le type abstrait de données : la pile

1.1 Description du type abstrait pile

Organisation

Une Pile est une structure de données dynamiques dans laquelle les éléments sont accessibles dans l'ordre inverse de l'arrivée à savoir c'est le dernier élément arrivé qui ressort en premier de la Pile.

Cette stratégie est indiquée par l'acronyme LIFO pour **L**ast **I**n **F**irst **O**ut

Vous ne mettrez plus la table de la même façon :

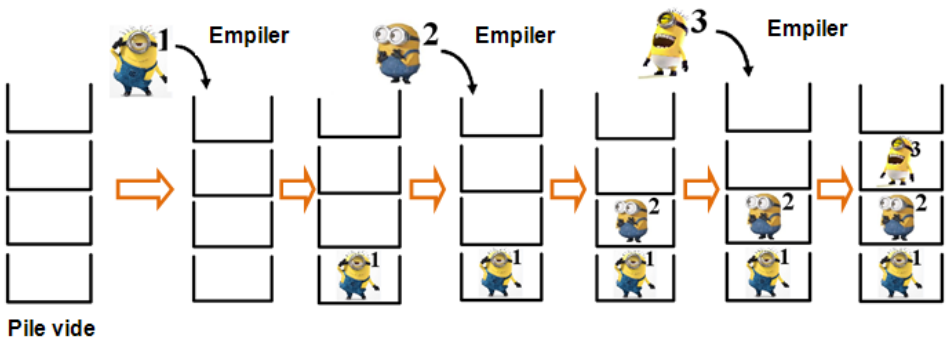
Opérations disponibles

Voilà la liste de quelques opérations disponibles avec une structure de données Pile appelée P :

estVide(P) : Renvoi un Booléen à *Vrai* si la pile ne contient aucun élément sinon renvoi la valeur *Faux*.
longueur(P) : Renvoi le nombre d'éléments de la pile P.
Empiler(x,P) : Empile l'élément x sur le sommet de la pile.
Depiler(P) : Dépile l'élément x du sommet de la pile, renvoie une exception si la tentative a lieu sur une pile vide. Renvoi x

Exemple d'utilisation

Empilement de trois éléments minions 1,2,3 :



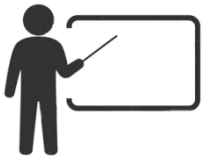
L'aspect LIFO est bien visible l'opération dépiler retournera le dernier élément entré dans la pile.

1.2 Une première implémentation : avec le type list de Python

Le type list est très proche de ce fonctionnement de pile avec les méthodes pop et append.
L'implémentation d'une classe Pile fondée sur ce type de donnée Python est très simple :

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

    def estVide(self):
        return self.lst  ==  []

    def empiler(self,  x):
        self.lst.append(x)

    def depiler(self):
        if self.estVide():
            raise ValueError("pile  vide")
        return self.lst.pop()

    def longueur(self):
        return len(self.lst)
```

Script_TAD_7. Coder le script ci-dessus et ajouter une méthode __str__ qui renvoie la valeur de l'élément au dessus de la pile si il existe ou une chaine 'Pile vide' dans le cas contraire. Tester votre classe avec l'exemple des trois minions ci-dessus.



Q1. Utiliser la classe Pile pour empiler les trois minions comme sur l'exemple ci-dessus. Indiquez les instructions utilisées.

1.3 Implémentation avec les listes chainées

Les listes chainées permettent également l'implémentation des Piles. Voilà un exemple de code :

Nous retrouvons le 'maillon' de la chaine

```
class Cell:
    def __init__(self,  x):
        self.val  =  x
        self.next  =  None
```

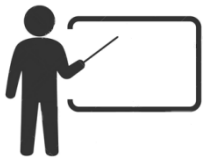
Exemple d'utilisation

```
Création de la pile p=Pile()
Empilement 'minion_1'
Sommet de la pile :  minion_1
Empilement 'minion_2'
Sommet de la pile :  minion_2
Empilement 'minion_3'
Sommet de la pile :  minion_3

Contenu de la pile à ce stade
minion_3
minion_2
minion_1
```

La classe Pile

```
class Pile:
    def __init__(self):
        self.lst  =  None

    def estVide(self):
        return self.lst is None

    def empiler(self,  x):
        c  =  Cell(x)
        c.next  =  self.lst
        self.lst  =  c

    def depiler(self):
        if self.estVide():
            raise ValueError("pile  vide")
        c  =  self.lst
        self.lst  =  c.next
        return c.val

    def __str__(self):
        if self.estVide() is True:
            return "Pile vide"
        else:
            c = self.lst
            return str(c.val)

    def longueur(self):
        """
        Renvoie la longueur de la pile
        A compléter
        """
```

Script_TAD_8. Coder le script ci-dessus et compléter la méthode longueur qui retourne le nombre de maillons de la chaine, donc la taille courante de la pile.



On constate sur l'exemple l'utilisation de la méthode *__str__(self)* via la commande *print(p)* qui renvoie le sommet de la pile. *p.lst* pointe sur le sommet de la pile, à chaque fois qu'un élément est empilé alors l'élément précédent est poussé à l'intérieur de la pile et le nouvel élément prend sa place. C'est le principe LIFO.

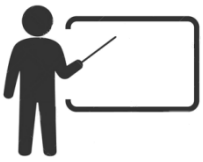
Après avoir empilé nos trois minions nous pouvons explorer la structure de données voilà le résultat :

Commande	Résultat

`print("1: ",p.lst)`
`print("2: ",p.lst.val)`
`print("3: ",p.lst.next)`
`print("4: ",p.lst.next.val)`
`print("5: ",p.lst.next.next)`
`print("6: ",p.lst.next.next.val)`
`print("7: ",p.lst.next.next.next)`
	`1:  <__main__.Cell object`
	`2:  minion_3`
	`3:  <__main__.Cell object`
	`4:  minion_2`
	`5:  <__main__.Cell object`
	`6:  minion_1`
	`7:  None`

Il serait plus pratique de pouvoir afficher le contenu entier de la pile à vous de jouer !

Script_TAD_9. Ajouter la possibilité d'afficher le contenu de la Pile en complétant le script : Script_TAD_9.py



2 Utilisation des piles quelques exemples

2.1 Inverser une chaine de caractères

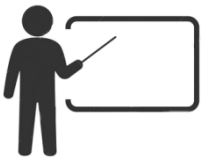
Il est très simple d'inverser les caractères d'une chaine en utilisant une pile. A vous de jouer !

Voilà un exemple de résultat :

```
INVERSION D'UNE CHAINE AVEC UNE PILE

Entrez votre chaine de caractères :
Choix par défaut :  anticonstitutionnellement
tnemellennoitutitsnocitna
```

Script_TAD_10. Réalisez l'inversion d'une chaine avec une pile

Script_TAD_10_Inversion_Chaine.py



Que veut dire le mot anticonstitutionnellement ?

Avec 25 lettres au compteur, l'adverbe ***anticonstitutionnellement*** est réputé pour être le mot le plus long de la langue française, à l'écrit. Disons-le tout de suite : ce n'est pas exact. D'autres mots à rallonges ont plus de lettres, par exemple des mots techniques, des verbes conjugués ou d'autres curiosités.

Cette idée s'est répandue car, à une époque donnée, ***anticonstitutionnellement*** était le mot le plus long de tous les mots du dictionnaire.

Alors, qu'est-ce que ça veut dire ?

Ce mot signifie « de manière contraire à la constitution ».

Rappelons-le, la constitution d'un État, c'est l'ensemble des textes juridiques qui régissent le fonctionnement du gouvernement de cet État. Par exemple, la France est actuellement régie par la Constitution du 4 octobre 1958, qui fixe les règles de fonctionnement de la V^{e} République.

Anticonstitutionnellement est donc presque un synonyme de ***illégalement***.

2.2 Vérification du bon parenthésage d'une expression ( )

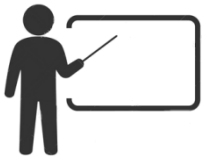
Dans les expressions arithmétiques il est nécessaire d'avoir un bon parenthésage c'est-à-dire un équilibre dans les parenthèses ouvrantes et fermantes comme dans l'expression ci-dessous :

```
(2+4*(5+22))
```

Nous pouvons implanter cette vérification avec une pile.

Script_TAD_11. Vérifiez automatiquement le bon parenthésage d'une expression, compléter le script :

Script_TAD_11_Parenthesage.py



```
Entrez votre chaine de caractères :
Choix par défaut :  (2+4*(5+22)
False
```

Q2. Est-il absolument nécessaire d'utiliser une pile pour faire cette vérification ?

2.3 Vérification du bon parenthésage d'une expression ( ) [ ] { }

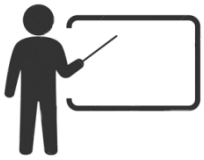
Nous allons maintenant étendre la vérification du bon 'parenthésage' dans une situation plus complexe où il y a non seulement des parenthèses ( ) mais également des accolades { } ou bien des crochets [ ].

```
Entrez votre chaine de caractères :
Choix par défaut :  ((e([[{{}kju}]]+)45)28.9)
Chaine bien formée :  True
```

A vous de jouer :

Script_TAD_12. Vérifiez automatiquement le bon parenthésage amélioré, compléter le script :

Script_TAD_12_Parenthesage_Multiples.py

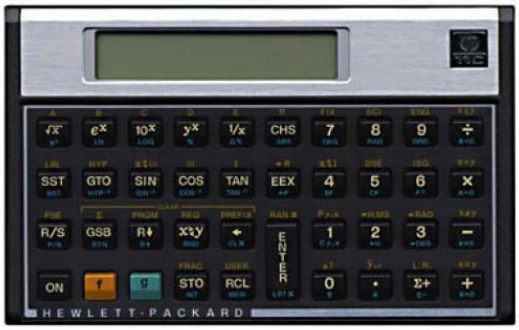


2.4 La notation polonaise inverse les expressions post-fixées

Présentation

La notation polonaise inverse[1] (NPI, en anglais RPN pour Reverse Polish Notation), également connue sous le nom de notation postfixée, est utilisée par certaines calculatrices HP. Dérivée de la notation polonaise présentée en 1924 par le mathématicien polonais Jan Łukasiewicz, se différencie de l'écriture 'habituelle' des expressions par l'ordre des termes, en effet, les opérandes y sont présentés avant les opérateurs et non l'inverse.

[1] https://fr.wikipedia.org/wiki/Notation_polonaise_inverse



HP-11C (1981-1989)

Expression en notation infixée

(3 + 6) * 2 'habituelle' 7 caractères

Expression en notation postfixée

3 6 + 2 * 5 caractères

Postfixée : l'opérateur est après les deux opérandes.

Exemples

L'évaluation de ces expressions post-fixées est réalisée avec une pile. (Image de la calculatrice[2])

Q3. Écrire en notation postfixée l'expression suivante : (4 – 9) * 3

Q4. Écrire en notation postfixée l'expression suivante : (3 + 4) * (3 – 2)

Q5. Écrire en notation postfixée l'expression suivante : 2 * (6 * 2 + 1)

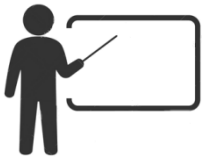
Q6. Écrire en notation postfixée l'expression suivante : (2 + 3) + (4 * (7 – 9))

Evaluation d'expression postfixée avec une pile

A vous de jouer :

Script_TAD_13.Complétez le script pour réaliser l'évaluation d'expression écrite au format NPI avec des nombres à un seul chiffre et les 4 opérations + - * /

Script_TAD_13_Notation_Polonaise.py



Amélioration : travailler avec des entiers positifs quelconques

Pour pouvoir travailler avec des nombres entiers positifs avec plusieurs chiffres nous allons procéder différemment de l'exercice précédent pour séparer notre chaine d'entrée.

Au lieu de lire la chaine expression caractère par caractère en séparant sur les espaces nous allons découper la chaine avec la méthode *.split()* qui réalise le découpage automatiquement. De cette manière nos nombres entiers peuvent être de longueur quelconque.

[2] http://www.ac-grenoble.fr/ugine/m/docs/calculatrice_rpn.html

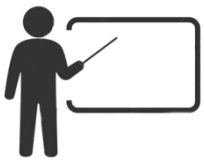
Illustration de l'utilisation de .split()

```
>>> expression = "23 6 12 * 14 + *"

>>> expression.split()
['23', '6', '12', '*', '14', '+', '*']
```

Amélioration de notre évaluation d'expression postfixée

Script_TAD_14.Complétez le script pour réaliser l'évaluation d'expression écrite au format NPI avec l'utilisation de split()

Script_TAD_14_Notation_Polonaise_Ameliore.py



Exemple de résultats :

```
EXPRESSION 3 :  23 6 12 * 14 + *
1978
```

3 Les files

3.1 Description du type abstrait file

Organisation

Après avoir étudié les Piles qui ont un comportement LIFO nous allons maintenant introduire les Files. Nous les rencontrons tous les jours dans la vie courante ce sont les files d'attentes où la gestion est de type FIFO à savoir First In First Out.

First In First Out.

Opérations disponibles

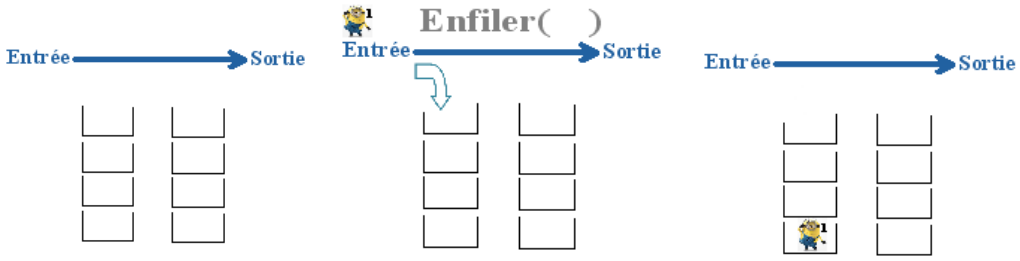
Voilà la liste de quelques opérations disponibles avec une structure de données File appelée F :

estVide(F) : Renvoi un Booléen à *Vrai* si la file ne contient aucun élément sinon renvoi la valeur *Faux*.
Enfiler(x,F) : Enfile l'élément x dans la file
Defiler(F) : Sort l'élément x de la file, renvoi un message si la tentative a lieu sur une file vide.

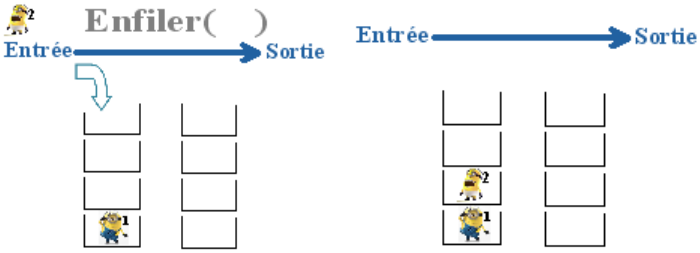
3.2 Implémentation du TAD file avec deux piles

Une solution pour l'implémentation d'une file consiste à utiliser deux files in et out selon le scénario ci-dessous où les minions arrivent à notre rescousse :

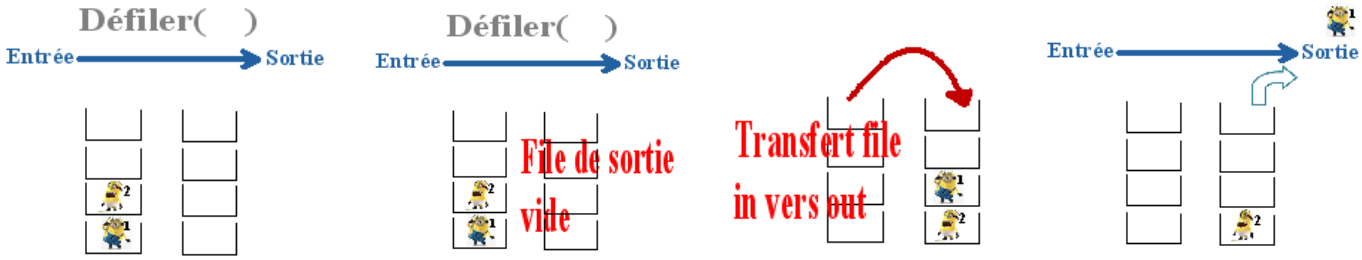
Un premier minion entre dans la file



Un deuxième minion se présente



On souhaite extraire le premier élément de la file



Q7. A partir de la présentation ci-dessus décrire l'algorithme par un texte expliquant les étapes permettant de réaliser une file avec deux piles in et out.

3.3 Implémentation en Python

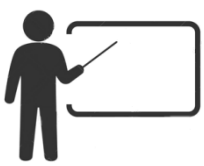
La classe file peut être réalisée à partir de la classe pile décrite au début de ce document basée sur les listes de Python voir Script_TAD_13.py . Le code partiel est présenté ci-dessous :

```
class File:
    def __init__(self):
        self.inbox  =  Pile()
        self.outbox =  Pile()

    def enfiler(self, x):
```

```
    def defiler(self):


    def estVide(self):
        return self.inbox.estVide() and self.outbox.estVide()
```

Script_TAD_15. Compléter dans le script de la classe File décrit ci-dessus les méthodes enfiler et defiler.

Script_TAD_15.py



4 En guise de conclusion

4.1 Utilisation des piles et des files dans les parcours d'arbre.

Nous retrouverons ces types abstraits Pile et File dans nos études sur les parcours d'arbres.

4.2 Le module collections de Python

La bibliothèque standard de Python comprend un module nommé collections qui contient des structures de données supplémentaires dont le type *deque* qui implémente la File. Se référer à la documentation.

Ce module implémente des types de données de conteneurs spécialisés qui apportent des alternatives aux conteneurs natifs de Python plus généraux `dict`, `list`, `set` et `tuple`.

`deque`	conteneur se comportant comme une liste avec des ajouts et retraits rapides à chaque extrémité

Il existe également le module queue qui implémente des files FIFO, LIFO dans un environnement plus complexe de multi threading.

Bilan de la complexité de quelques opérations[3]

Opérations sur les listes/piles

Type Python	Type abstrait	Opération	Exemple	Complexité
`list lst` avec `n=len(lst)`	Tableau dynamique	Ajout à la fin	`lst.append(x)`	$O(1)$
		Accès à un élément	`lst[i]`	$O(1)$
		Modification d'un élément	`lst[i] = x`	$O(1)$
		Effacement d'un élément	`del lst[i]`	$O(n)$
		Insertion d'un élément	`lst.insert(i,x)`	$O(n)$
		Recherche d'un élément	`x in lst`	$O(n)$
	Pile	*push*	`lst.append(x)`	$O(1)$
		pop	`lst.pop()`	$O(1)$

Opérations sur les dictionnaires

Type Python	Type abstrait	Opération	Exemple	Complexité
`dict d` avec `n=len(d)`	Tableau associatif	Ajout d'un élément	`d[key] = val`	$O(1)$
		Modification d'un élément	`d[key] = val`	$O(1)$
		Effacement d'un élément	`del d[key]`	$O(1)$
		Accès à un élément	`d[key]`	$O(1)$
		Recherche d'une clé	`key in d`	$O(1)$
		Recherche d'une valeur	`val in d.values()`	$O(n)$

Opérations sur les ensembles

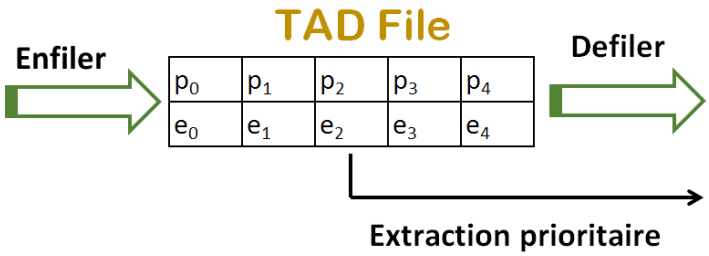
Type Python	Type abstrait	Opération	Exemple	Complexité
`set s,t` avec `n=len(s)` `m=len(t)`	Ensemble	Ajout d'un élément	`s.add(elt)`	$O(1)$
		Retrait d'un élément	`s.remove(elt)`	$O(1)$
		Test d'appartenance	`elt in s`	$O(1)$
		Union	`s | t`	$O(n+m)$
		Intersection	`s & t`	$O(\min(n,m))$
		Différence	`s - t`	$O(n)$
		Différence symétrique	`s ^ t`	$O(n)$

[3] D'après PrépaBac TNSI Hatier page 100

5 Projet : Implémentation d'une File à priorité

5.1 Présentation du Type Abstrait de Données File à priorité

La file à priorité est une file qui possède une possibilité de sortir des éléments en fonction d'une valeur appelée priorité.



5.2 La cellule de base pour l'implémentation

Il faut ajouter un attribut priorite à nos éléments de Classe Cell :

```
def __init__(self,  x, p):
    self.val  =  x
    self.priorite = p
    self.next  =  None
```

5.3 Programme à réaliser

Réaliser à partir des différents exemples traités dans ce cours l'implémentation d'une file à priorité. Le fonctionnement de la file est 'standard' avec les opérations enfiler et defiler. Par contre l'extraction, (et donc la suppression), de l'élément le plus prioritaire, ou d'un des éléments les plus prioritaires, est possible directement.

5.4 Principe de l'implémentation d'une solution possible

L'implémentation de ce type abstrait est réalisée avec deux piles comme vu précédemment. Il faut par contre ajouter la gestion de la priorité aux piles. Voilà quelques éléments du corrigé pour votre inspiration.

Opérations de la classe Pile

```
## CLASSE PILE SUR LA BASE D'UNE LISTE CHAINEE
class Pile:
    '''
    La classe Pile est implémentée sur la base de cellule de liste chainée
    avec un attribut priorité.

    Le type construit Pile possède les opérations suivantes :
    - .estVide()              : test si la pile est vide (True) ou non (False)
    - .empilerParCellule(elt) : empile un élément elt de classe (Cell) dans la Pile, utilisé avec les Files
    - .empilerParValeur(x,p)  : empile un élément créé par Cell(x,p) dans la Pile, utilisé en direct
    - .depiler()              : retourne l'élément en haut de la pile (Cell) ou bien (None) si la pile est vide
    - .longueur()             : retourne la longueur de la Pile (Entier)
    - .afficher()             : imprime le contenu de la pile, le rang de valeur 0 est attribué au sommet
    - .prioritaireMax()       : retourne la priorité et le rang dans la pile de l'élément le plus prioritaire [pmax,rang]
                                retourne [-1,-1] si la pile est vide
    - .retirerElement(n)      : retire l'élément de rang n de la pile et le retourne (Cell)  retourne None si la pile est vide
    - __str__ : Impression par print(Pile) du dernier élément de la pile
    - __len__ : Retour par len(Pile) de la longueur
    '''
```

Opérations de la classe File

```
## CLASSE FILE SUR LA BASE DE DEUX PILES
class File:
    '''
    La classe File est implémentée sur la base de deux Piles une pour les données
    d'entrée et une pour les données de sortie.
    L'attribut priorité est géré. Quand on souhaite extraire l'élément le plus prioritaire
    alors la recherche de cet élément ce fait sur les deux Piles in et out et le plus
    prioritaire des deux est extrait directement. Avec priorité à la pile in en cas d'égalité.
    Cela n'empêche pas par ailleurs de retirer les éléments de la File normalement par .defiler()

    Le type construit File possède les opérations suivantes :
    - .estVide()                  : test si la file est vide (True) ou non (False)
    - .enfiler(x,p)               : entrée d'un élément dans la file défini à partir de sa valeur x et
                                    de sa priorité p
    - .defiler()                  : retourne le premier élément entré dans la File au format Cell
    - .afficher()                 : Affiche le contenu complet de la File les rangs 0 sont les sommets des Piles in et out
    - .enleverElementPrioritaire() : Recherche l'élément le plus prioritaire de la File
                                    l'élément est supprimé de la File est retourné au format Cell
                                    renvoi (None) si la File est vide
    '''
```

Exemple de fonctionnement

```
## Utilisation de la classe File à priorité
print("ESSAIS FILE A PRIORITE")
print("Création de la File")
ma_file = File()
print("Entrée élément (val:1,priorite:2)")
ma_file.enfiler(1,2)
print("Entrée élément (val:2,priorite:10)")
ma_file.enfiler(2,10)
print("Entrée élément (val:3,priorite:3)")
ma_file.enfiler(3,3)
ma_file.afficher()
print("Sortie du premier élément ")
element = ma_file.defiler()
if element is not None:
    print("Sortie élément  val:",element.val,
    " priorité:",element.priorite)
ma_file.afficher()
print("Entrée élément (val:4,priorite:0)")
ma_file.enfiler(4,0)
print("Entrée élément (val:5,priorite:1)")
ma_file.enfiler(5,1)
ma_file.afficher()
print("Extraction de l'élément le plus prioritaire")
element = ma_file.enleverElementPrioritaire()
if element != None:
    print("Sortie élément  val:",element.val,
    " priorité:",element.priorite)
ma_file.afficher()
print()
```

```
ESSAIS FILE A PRIORITE
Création de la File
Entrée élément (val:1,priorite:2)
Entrée élément (val:2,priorite:10)
Entrée élément (val:3,priorite:3)
-------------------------------------------------
File.afficher : Etat des piles Pile_in et Pile_out
Pile d'entrée
0  Valeur :  3   Priorité :  3
1  Valeur :  2   Priorité :  10
2  Valeur :  1   Priorité :  2
Pile de sortie
Pile vide
-------------------------------------------------
Sortie du premier élément
Sortie élément  val: 1  priorité: 2
-------------------------------------------------
File.afficher : Etat des piles Pile_in et Pile_out
Pile d'entrée
Pile vide
Pile de sortie
0  Valeur :  2   Priorité :  10
1  Valeur :  3   Priorité :  3
-------------------------------------------------
Entrée élément (val:4,priorite:0)
Entrée élément (val:5,priorite:1)
-------------------------------------------------
File.afficher : Etat des piles Pile_in et Pile_out
Pile d'entrée
0  Valeur :  5   Priorité :  1
1  Valeur :  4   Priorité :  0
Pile de sortie
0  Valeur :  2   Priorité :  10
1  Valeur :  3   Priorité :  3
-------------------------------------------------
Extraction de l'élément le plus prioritaire
Sortie élément  val: 2  priorité: 10
-------------------------------------------------
File.afficher : Etat des piles Pile_in et Pile_out
Pile d'entrée
0  Valeur :  5   Priorité :  1
1  Valeur :  4   Priorité :  0
Pile de sortie
0  Valeur :  3   Priorité :  3
-------------------------------------------------
```