Piles et Files

File avec priorités

Contexte

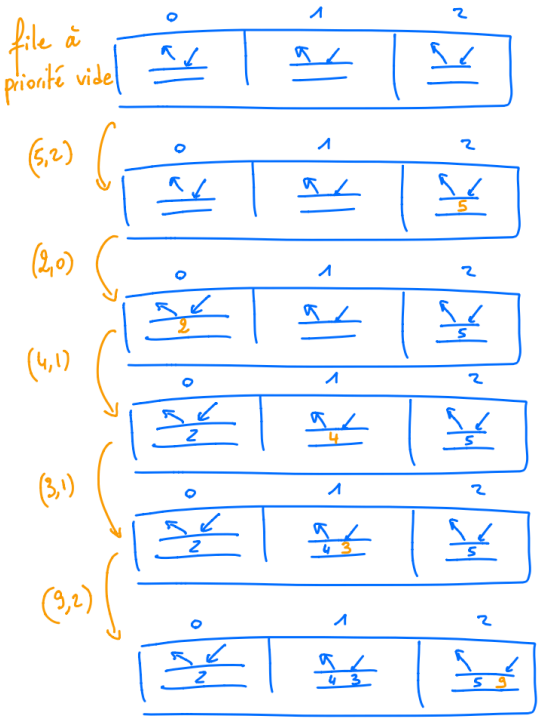
On souhaite implémenter une nouvelle structure de données pour représenter une file d'attente avec priorités. La file d'attente avec priorité permet d'enfiler des valeurs auxquelles on associe une priorité. Ainsi lorsqu'on défile :

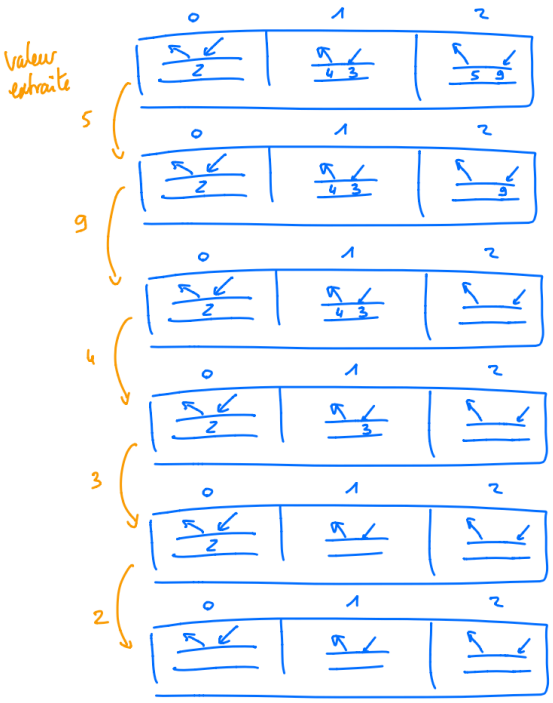
- Entre 2 valeurs de même priorité, on "servira" selon la politique classique FIFO (First In First Out = une file classique) , c'est-à-dire qu'on défile la première valeur enfilée de cette priorité.
- Entre 2 valeurs de priorités différentes, on "servira" selon la priorité.
  Exemple : Si on enfile successivement les couples (valeur,priorité) suivants dans une file de 3 priorités (0 à 2, 2 étant la priorité la plus forte) :

```
(5,2)
(2,0)
(4,1)
(3,1)
(9,2)
```

Alors, lorsqu'on "servira" (défilera/extraira), on aura dans l'ordre 5 , 9, 4, 3 puis 2. On "sert" d'abord 5 et 9 car ces valeurs sont de priorité 2 et 5 est "arrivé" avant 9. Puis on sert 4 et 3 de priorité 1 (4 arrivé avant 3), et enfin 2 de plus basse priorité. Pour que la représentation soit efficace, on propose de séparer les valeurs dès leur arrivée dans la file suivant leur priorité.

On utilisera donc un **tableau** de **files**.
Ainsi, dans l'exemple précédent on aura un tableau de taille 3. Dans chaque case du tableau on aura une file.
Dans la case 2, on aura la file [5,9] de priorité 2 (à dire de gauche à droite selon l'arrivée).
Dans la case 1, on aura la file [4,3] de priorité 1.
Et dans la case 0, on aura la file [2] de priorité 0.
Ce qui fait donne le tableau [[2],[4,3],[5,9]].
Voici le shéma de l'insertion puis de l'extraction de l'exemple





Enoncé

On souhaite afficher dans l'ordre les valeurs selon leur priorité. Pour cela, on vous demande de compléter le programme donné qui définit les opérations suivantes :

- `taille_prio` : pour une file de priorités donnée, renvoie sa taille (son nombre de priorités, la taille du tableau).
- `file_vide_prio` : pour une file de priorités donnée, renvoie si elle est vide, c'est-à-dire que chacune de ses files est vide.
- `extraire_prio` : pour une file de priorités donnée, il faut **extraire** (servir/défiler) la file (non vide) de plus haute priorité. Si la file de priorités est vide, on renverra `None` (plutôt qu'une erreur pour les besoins de PIXAL).
- et enfin `servir` : pour une file de priorités donnée, affiche les élements dans l'ordre de leur priorité. C'est-à-dire extraire par ordre de priorité tant qu'il y a des valeurs dans la file de priorités.

Le programme doit utiliser l'implémentation proposée de la structure de donnée file et restreindre l'utilisation des listes python aux opérations des tableaux (pas de suppression, pas d'ajout, taille fixe...).

Implémentation

Les fonctions définies dans le cours et nécessaires à la manipulation des files sont disponibles grâce à l'`import` suivant :

```
from file import *
```

La liste des opérations est rappelée ici (et dans l'avant-propos) :

```
ma_file = creer_file_vide() # création file vide
valeur = extraire(ma_file) # récupère le premier arrivé et le supprime (si vide erreur)
inserer(ma_file,valeur) # ajoute valeur à la file
est_vide = file_vide(ma_file) # vérifie si la file est vide
taille = nombre_elements(ma_file) # le nombre d'éléments
```

Entrée du programme

Un tableau de files (lisible à l'aide d'un eval classique)

Sortie du programme

Les sorties de l'appel au programme donné :

Voici le programme à compléter :

```
from file import * #POUR PIXAL

### FONCTIONS A DEFINIR PAR LES ETUDIANTS

#nb de priorites
def taille_prio(file_prio) :
    # A COMPLETER. Rappel, file_prio est un tableau de files

# renvoie vrai si les files de toutes les priorités sont vides
def file_vide_prio(file_prio):
    # A COMPLETER. Rappel, file_prio est un tableau de files

#extraire le plus prioritaire
def extraire_prio(file_prio):
    # A COMPLETER. Rappel, file_prio est un tableau de files

#afficher les éléments dépilés selon leur priorité
def servir(file_prio) :
   # A COMPLETER. Rappel, file_prio est un tableau de files

### FONCTION DE VERIFICATION FOURNIE
def verification(file_prio) :
    #affichage brut
    print(file_prio)

    #taille = nombre de priorités
    print("taille :",taille_prio(file_prio))

    #test de file priorités vide
    print("est-vide ? ", file_vide_prio(file_prio))

    #extraire une valeur prioritaire
    print("avant extraire :", file_prio) #affichage brut
    valeur = extraire_prio(file_prio)
    print("valeur extraite :",valeur,"après extraire :",file_prio)
    #extraire une autre valeur prioritaire
    print("avant extraire :", file_prio) #affichage brut
    valeur = extraire_prio(file_prio)
    print("valeur extraite :",valeur,"après extraire :",file_prio)

    #servir les valeurs
    servir(file_prio)

### Programme principal

file_prio = eval(input())
verification(file_prio)
```

Exemples

Entrée :

```
[[4,2,1],[5,3],[7,8]]
```

Sortie :

```
[[4, 2, 1], [5, 3], [7, 8]]
taille : 3
est-vide ?  False
avant extraire : [[4, 2, 1], [5, 3], [7, 8]]
valeur extraite : 7 après extraire : [[4, 2, 1], [5, 3], [8]]
avant extraire : [[4, 2, 1], [5, 3], [8]]
valeur extraite : 8 après extraire : [[4, 2, 1], [5, 3], []]
5
3
4
2
1
```

Entrée :

```
[[]]
```

Sortie :

```
[[]]
taille : 1
est-vide ?  True
avant extraire : [[]]
valeur extraite : None après extraire : [[]]
avant extraire : [[]]
valeur extraite : None après extraire : [[]]
```

Conseils

N'hésitez pas à mettre en commentaire les appels des fonctions demandées dans la fonction `verification`.

Négatifs et positifs

Enoncé

Écrire un programme qui accepte en entrée une série de nombres entiers relatifs. Dès que l'entrée est `0`, le programme affiche dans l'ordre :

- Le nombre (éventuellement nul) de nombres négatifs entrés.
- Les nombres négatifs entrés (dans leur ordre d'entrée).
- Le nombre (éventuellement nul) de nombres positifs entrés.
- Les nombres positifs entrés (dans leur ordre d'entrée).

Le `0` final n'est pas affiché.

Le programme ne doit utiliser aucune autre structure de donnée qu'une (ou plusieurs) file(s).

Implémentation

Les fonctions définies dans le cours et nécessaires à la manipulation des files sont disponibles grâce à l'`import` suivant :

```
from file import *
```

La liste des opérations est rappelée ici (et dans l'avant-propos) :

```
ma_file = creer_file_vide() # création file vide
valeur = extraire(ma_file) # récupère le premier arrivé et le supprime (si vide erreur)
inserer(ma_file,valeur) # ajoute valeur à la file
est_vide = file_vide(ma_file) # vérifie si la file est vide
taille = nombre_elements(ma_file) # le nombre d'éléments
```

Entrée du programme

Une séquence d'entiers relatifs

Sortie du programme

Des entiers

Exemples

Entrée :

```
61
-53
-42
75
-92
0
```

Sortie :

```
3
-53
-42
-92
2
61
75
```

Entrée :

```
65
86
96
0
```

Sortie :

```
0
3
65
86
96
```

Entrée :

```
-65
-86
-96
0
```

Sortie :

```
3
-65
-86
-96
0
```

Afficher chiffres

Enoncé

Écrire la fonction **`afficher_chiffres`** qui accepte en paramètre un entier positif ou nul et affiche dans l'ordre gauche-droite les chiffres de sa représentation en base 10. Écrire cette fonction en n'utilisant aucune autre structure de donnée qu'une (ou plusieurs) pile(s).

Écrire le **programme** qui lit un entier donné en entrée et appelle la fonction **`afficher_chiffres`** pour réaliser l'affichage.

Implémentation

Les sources de la classe `pile.py` sont données dans le document d'implémentation, elles sont nécessaires si vous voulez utiliser le pas à pas. Voici la liste des fonctions :

Entrée du programme

Un entier.

Sortie du programme

Les chiffres de cet entier.

Exemples

Entrée :

```
678215
```

Sortie :

```
6
7
8
2
1
5
```

Entrée :

```
0
```

Sortie :

```
0
```

Vérification de parenthésage

Énoncé

On considère une chaîne de caractères permettant de stocker une expression mathématique telle que :

```
({p*(q/(r-s))}/t) - {(x + y) - {(a/b)*c}} + d
```

On souhaite vérifier que les **deux** types de parenthésages, c'est-à-dire les parenthèses `()` et les accolades `{}`, sont équilibrés.

Lors du parcours de gauche à droite de la chaîne de caractères contenant l'expression mathématique, seules 4 situations peuvent être rencontrées :

- Pendant le parcours, on rencontre un fermant...
  - ...qui ne correspond pas au dernier ouvrant rencontré (**mauvais fermant**) : `"(a - {b*c))"`
  - ...qui n'est précédé d'aucun ouvrant (**trop de fermants**) : `"(a - {b*c}))))"`
- Une fois le parcours terminé :
  - Il reste des ouvrants non fermés (**trop d'ouvrants**) : `"(a - (b*c)"`
  - Sinon, le parenthésage est **correct** : `"({a - b} - {b*c})"`

L'**objectif** de l'exercice est d'écrire la **fonction `verifie_parenthesage(expression)`** qui accepte en paramètre une chaîne de caractères `expression` et retourne un code d'erreur qui est `0`, `1`, `2` ou `3` selon si `expression` a un mauvais fermant, trop de fermants, trop d'ouvrants ou est correcte.

Indications :

- On s'arrête à la première erreur détectée (s'il y en a).
- Pour simplifier, on ne vérifie pas le placement correct des arguments des opérateurs : on considère par exemple que l'expression `(a*)-{bc}` a un parenthésage correct.
- Le parcours cacactère par caractère d'une chaîne de cacactères `chaine` peut s'effectuer en utilisant la boucle `for car in chaine:`.
- Attention à ne pas utiliser la méthode `split()` dans cet exercice.
- Il est vivement conseillé d'utiliser une pile (de l'implémentation `pile`) pour stocker les parenthèses/accolades ouvrantes rencontrées.
- Recopier et compléter le programme fourni ci-dessous, qui prend en entrée une **liste d'expressions**, effectue les appels de la fonction **`verifie_parenthesage(expression)`**, et retourne la **liste des codes d'erreur** correspondants à chacune de ces expressions.

Code à recopier :

Implémentation

Les sources de la classe `pile.py` sont données dans le document d'implémentation, elles sont nécessaires si vous voulez utiliser le pas à pas. Voici la liste des fonctions :

Entrée du programme

Une liste de chaînes de caractères contenant la liste d'expressions pour lesquelles on vérifie le parenthésage, cette liste sera saisie à l'aide de `eval(input())`. Pour utiliser eval dans le pas à pas, vous pouvez insérer en début de programme le code suivant :

Sortie du programme

Une liste d'entiers.

Exemples

Entrée :

```
[ "({p*(q/(r-s))}/t) - {(x + y) - {(a/b)*c}} + d" , "{a - (b*c}", "{a - (b*c)", "a}+b"]
```

Sortie :

```
[3, 0, 2, 1]
```

La première chaîne `"({p*(q/(r-s))}/t) - {(x + y) - {(a/b)*c}} + d"` est correcte, la deuxième `"{a - (b*c}"`a un mauvais fermant, la troisième `"{a - (b*c)"` a trop d'ouvrants, et la dernière `""a}+b"` a trop de fermants.

Notation Polonaise Inversée (RPN)

Enoncé

Tiré de Wikipédia:
Reverse Polish notation (RPN) is a mathematical notation in which every operator follows all of its operands, in contrast to Polish notation (PN), which puts the operator before its operands. It is also known as postfix notation. It does not need any parentheses as long as each operator has a fixed number of operands. The description "Polish" refers to the nationality of logician Jan Łukasiewicz, who invented Polish notation in the 1920s.
The reverse Polish scheme was proposed in 1954 by Burks, Warren, and Wright and was independently reinvented by Friedrich L. Bauer and Edsger W. Dijkstra in the early 1960s to reduce computer memory access and utilize the stack to evaluate expressions. The algorithms and notation for this scheme were extended by Australian philosopher and computer scientist Charles Hamblin in the mid-1950s.
During the 1970s and 1980s, RPN was well-known to many calculator users, as Hewlett-Packard used it in their pioneering 9100Aand HP-35 scientific calculators, the succeeding Voyager series - and also the "cult" HP-12C financial calculator.

En utilisant la structure de données de pile, implémentée par la classe Lifo (vue en CTD), on souhaite écrire une fonction qui permet d'évaluer une expression (chaîne de caractères) écrite en notation polonaise inversée (RNP).
Pour donner un exemple simple, l'expression classique : "5 + ((1 + 2) * 4) – 3", est traduite en RNP de la manière suivante : "5 1 2 + 4 * + 3 –"
Dans cet exercice, il vous est demandé, à partir d'une expression comme celle-ci : "5 1 2 + 4 * + 3 –" de procéder à son évaluation. Pour l'exemple, voici le traitement à appliquer :

Entree	Action	Pile	Commentaire
5	Opérande	5	Empiler
1	Opérande	1 5	Empiler
2	Opérande	2 1 5	Empiler
+	Opérateur	3 5	Dépiler les 2 opérandes (2 et 1), calculer (2 + 1= 3) et empiler (3)
4	Opérande	4 3	Empiler

		5	
*	Opérateur	12 5	Dépiler les 2 opérandes (4 et 3), calculer (3 * 4 = 12) et empiler (12)
+	Opérateur	17	Dépiler les 2 opérandes (12 et 5), calculer (5 + 12 = 17) et empiler (17)
3	Opérande	3 17	Empiler
-	Opérateur	14	Dépiler les 2 opérandes (3 et 17), calculer (17 - 3 = 14) et empiler (14)
	Résultat	14	

Les opérateurs acceptés sont +,-,*,// et %.

Implémentation

Les sources de la classe `pile.py` sont données dans le document d'implémentation, elles sont nécessaires si vous voulez utiliser le pas à pas. Voici la liste des fonctions :

Entrée du programme

Une chaîne de caractères avec des espaces entre les opérandes et opérateurs. On peut utiliser **chaine.split()** pour découper la chaîne : le split supprime les espaces et transforme la chaîne en tuple.

Sortie du programme

Un entier, la valeur de l'expression. Si l'expression est mal écrite, la chaîne suivante : "erreur d'expression".

Exemples

Entrée :

```
5 1 2 + 4 * + 3 - 5 % 2 *
```

Sortie :

```
8
```

Entrée :

```
5 1 2 + 4 * + 3
```

Sortie :

```
erreur d'expression
```