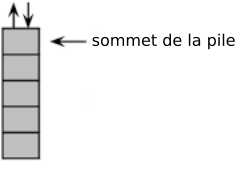
**Structure de données : les piles et files**

1. **Une structure de données :La pile :**

**Principe :**

On retrouve dans les piles une partie des propriétés vues sur les listes. Dans les piles, il est uniquement possible de manipuler le dernier élément introduit dans la pile. On prend souvent l'analogie avec une pile d'assiettes : dans une pile d'assiettes la seule assiette directement accessible et la dernière assiette qui a été déposée sur la pile.



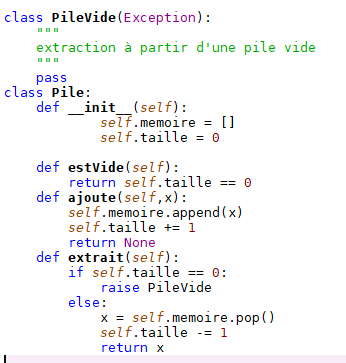
Les piles sont basées sur le principe LIFO (Last In First Out : le dernier rentré sera le premier à sortir). On retrouve souvent ce principe LIFO en informatique.

**Définition :** Une pile est une structure de données définie par :

▷son interface, c’est-à-dire la liste des méthodes qu’elle définit, à savoir : la création d’une pile vide, *estVide()* qui renvoie *True* si et seulement si la pile est vide, *ajoute(x)* qui permet d’ajouter un élément *x* à la pile et *extrait()* qui extrait un élément et renvoie sa valeur .

▷une sémantique, qui, pour la pile, est la sémantique LIFO (last in first out) .

Le plus simple, en Python, est de la programmer à l’aide d’une classe de la façon suivante

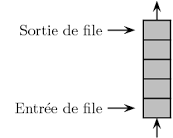


On a défini une exception *PileVide* pour rendre la programmation plus sûre. Notons qu’on aurait pu choisir une autre implémentation (c’est-à-dire une autre façon de programmer les méthodes de la classe) que celle que nous avons reprise de notre premier exemple d’introduction. L’important est de comprendre que, pour l’utilisateur qui n’a pas accès au texte de définition de la classe, mais seulement à son interface, c’est totalement indifférent. Il lui suffit de savoir qu’il créera une nouvelle pile vide p avec l’appel *p = Pile()*, qu’il pourra ajouter l’entier 3 dans la pile p en appelant *p.ajoute(3)* et qu’il pourra extraire l’élément ajouté le plus récemment avec l’instruction *x = p.extrait()*. L’usage veut aussi qu’on précise, pour chaque implémentation, le coût d’exécution de chacune des méthodes proposées par l’interface : ici, tous les coûts sont unitaires.

1. **Une autre structure de données : la file**

**Principe :**

Comme les piles, les files ont des points communs avec les listes. Différences majeures : dans une file on ajoute des éléments à une extrémité de la file et on supprime des éléments à l'autre extrémité. On prend souvent l'analogie de la file d'attente devant un magasin pour décrire une file de données.



Les files sont basées sur le principe FIFO (First In First Out : le premier qui est rentré sera le premier à sortir. Ici aussi, on retrouve souvent ce principe FIFO en informatique.

**Définition :**

La file a la même interface que la pile, mais une sémantique différente, FIFO, où la méthode *extrait()* va choisir l’élément le plus ancien résidant encore dans la file. On pourrait définir une classe comme on vient de le faire pour la pile, en remplaçant simplement dans la définition de la méthode extrait() la ligne *x = self.memoire.pop()* par *x= self.memoire.pop(0)*, ainsi qu’on l’a expliqué dansl’introduction .Mais pour bien montrer qu’il peut exister plusieurs implémentations pour une même structure de données, nous en proposons une autre



Les méthodes de création, de test de file vide, ou d’ajout, sont ici de coût unitaire .La méthode extrait a une complexité plus difficile à maîtriser. On peut démontrer que le coût d’un appel à cette méthode est unitaire.

1. **Utiliser un module Python :**

Il existe encore une autre solution : utiliser un module prédéfini de Python, comme deque

Ce module s’importe avec la commande *from collections import deque* après laquelle on dispose des commandessuivantes :

▷d = deque([ ]) crée une collection vide

▷d.count( )renvoie le nombre d’éléments présents dans d

▷d.append(x) et d.appendleft(y) permettent d’ajouter à d ou bien x à droite ou bien y à gauche ;

▷x = d.pop() et y = d.popleft() permettent d’extraire le dernier élément x ou le premier y. Ainsi, en utilisant le couple append et pop on obtient le comportement d’une pile ; en utilisant le couple append et popleft celui d’une file. Il est important de remarquer que le module deque propose de nombreuses autres fonctions ou méthodes, que nous n’avons pas décrites ici. Mais seules celles qui sont utiles à la définition des piles et des files nous intéressent ici : c’est très souvent le cas avec des modules généralistes qui offrent beaucoup plus que ce qui est utile à la structure de données qu’on est en train d’étudier. Quand on écrit soi-même des classes, il est recommandé de se limiter à implémenter ce qu’exige la définition de la structure de données.

1. **Mini-projet :**

# Notation Polonaise Inversée, évaluation d'expressions postfixées

## Énoncé

Le but de l’exercice est de réaliser en Python une calculatrice simple, capable d’évaluer une formule en NPI, Notation Polonaise Inversée, et de retourner le résultat arithmétique. La réalisation d’une telle calculatrice se fera à l’aide d’une pile.

Un rapide coup d’oeil sur **Wikipédia** nous apprend que :

« La notation polonaise inverse (NPI) (en anglais RPN pour Reverse Polish Notation), également connue sous le nom de notation post-fixée, permet d’écrire de façon non ambiguë les formules arithmétiques sans utiliser de parenthèses. »

**Historiquement** : « Elle est dérivée de la notation polonaise utilisée pour la première fois en 1924 par le mathématicien polonais Jan Łukasiewicz, la NPI a été inventée par le philosophe et informaticien australien Charles Leonard Hamblin dans le milieu des années 1950, pour permettre les calculs sans faire référence à une quelconque adresse mémoire. À la fin des années 1960, elle a été diffusée dans le public comme interface utilisateur avec les calculatrices de bureau de Hewlett-Packard. »

## Exemple

Voici une expression algébrique en notation **infixée** (celle que nous utilisons la plupart du temps) : ((3 + 4)×2)^3

En notation **postfixée** ( NPI) cela donne : 3 4 + 2 × 3 ∧

Questions préliminaires

1. Écrire la formule 10 \* (3 + 4) – 5 en NPI
2. Déterminer le résultat de calcul décrit par la formule 12 4 + 12 4 – \*.

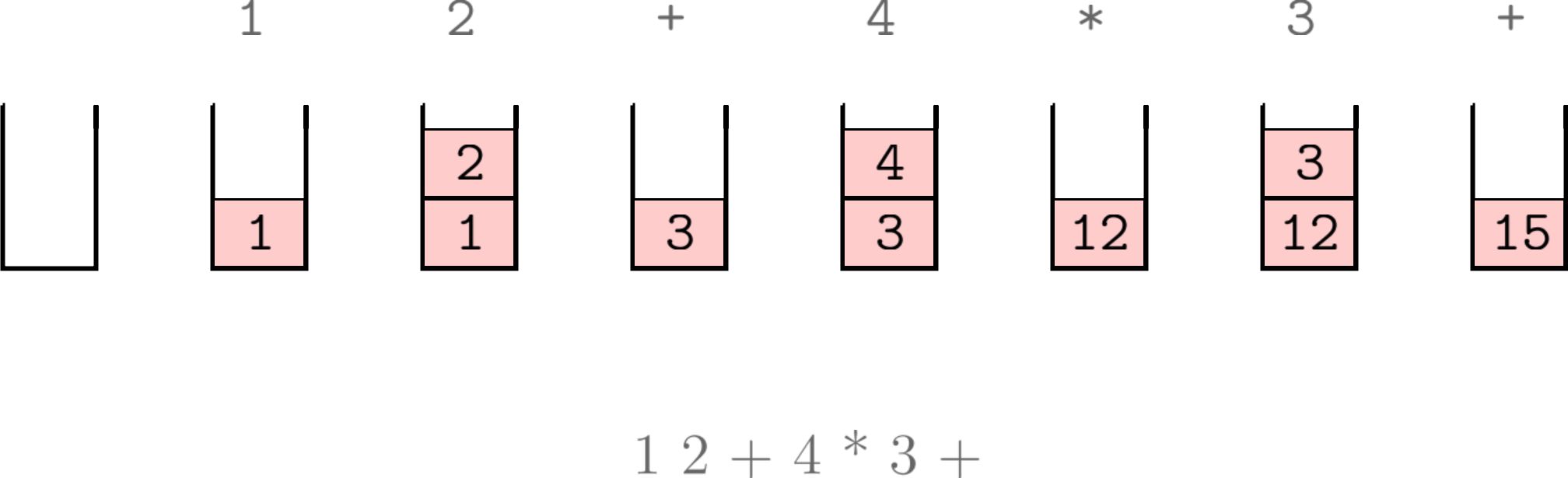
## Présentation de l'algorithme

On commence par lire un par un les caractères de l’expression.

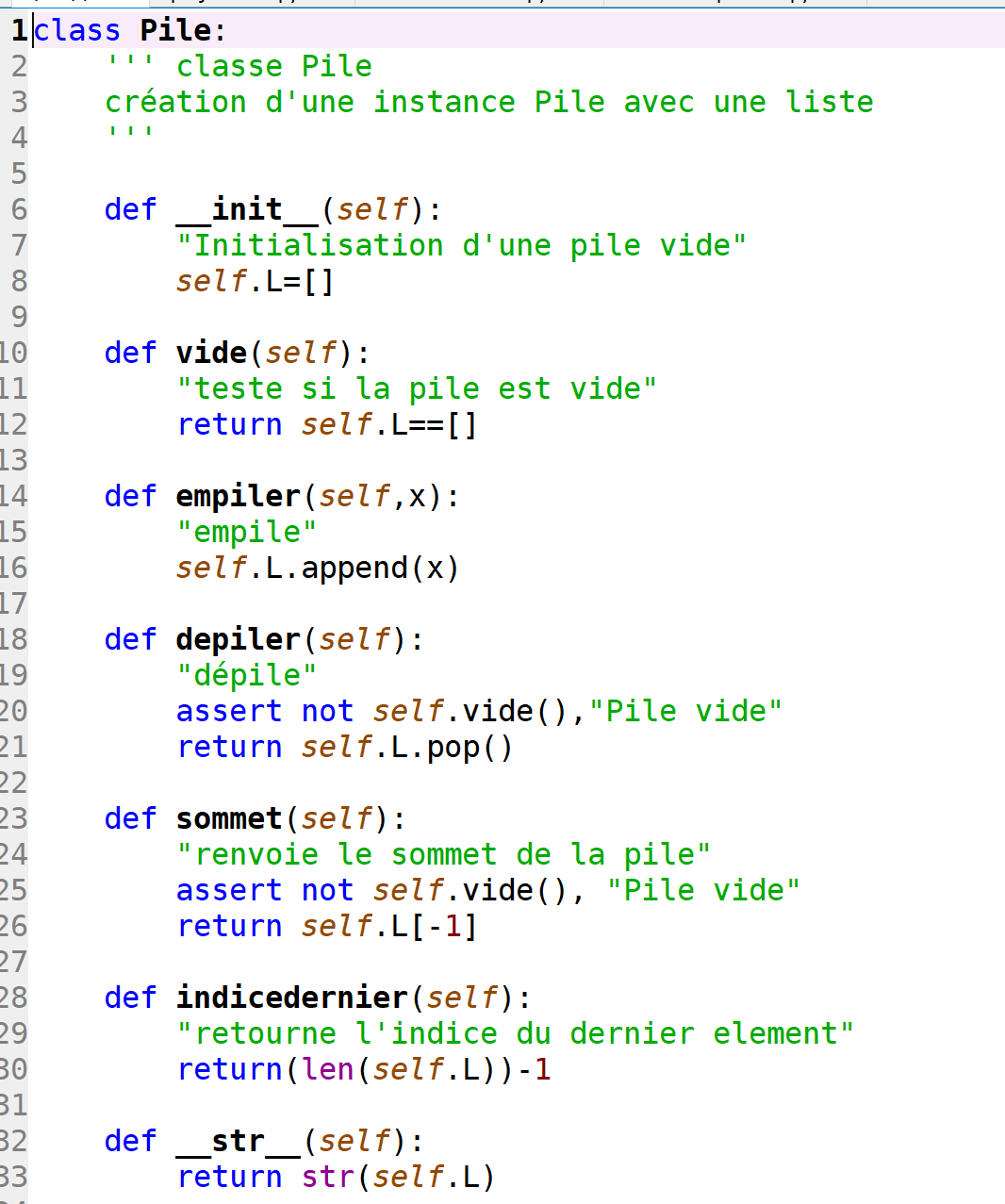
* Si le caractère lu est un opérateur, alors on dépile les deux éléments se trouvant en haut de la pile, on calcule le résultat en appliquant l’opérateur sur les deux opérandes dépilés et on empile le résultat.
* Si le caractère lu est un opérande alors on l’empile.

Une fois tous les caractères lus, la pile ne contient qu’un seul élément qui correspond au résultat final.

Voyons avec un exemple l’état de la pile après la lecture de chaque caractère de l’expression ((1+2)∗4)+3, ou 1 2 + 4 \* 3 + en NPI.



Pour réaliser le programme qui permet d'évaluer différentes expressions en NPI, suivez les étapes.



Étape 1 : la classe Pile

Utilisez la classe pile ci-contre (elle est un peu plus complète

Que celle du cours) . Il s’agit d’un type de données abstrait qui n’est pas présent nativement dans Python.

Le constructeur est \_\_init\_\_ (ne pas oublier le double underscore avant et après init) !

Les primitives sont les méthodes vide, empiler, depiler, sommet,indicedernier. La méthode \_\_str\_\_ permet de simplifier l’écriture pour les élèves.

Étape 2  : entrée de l'utilisateur

L'utilisateur entre une chaîne de caractères sous le bon format : c'est-à-dire des nombres (opérandes) et des opérateurs (+,-,\*,/) séparés par des espaces. Une fois cette chaîne obtenue, on a besoin de traiter les opérandes et opérateurs séparément. Le plus simple est de découper la chaîne selon les espaces qu'elle contient.

Pour cela on utilise la méthode **split**. Essayez de comprendre comment elle fonctionne avec [cet exemple](https://www.w3schools.com/python/trypython.asp?filename=demo_ref_string_split).

L’expression précédente de l'exemple graphique sera représentée par la liste : expr=[1,2,’+’,4,’\*’,3,’+’]

Étape 3 : définir la fonction

Écrivez une fonction **eval\_postfix** qui prend en paramètres une chaîne de caractères représentant l'expression postfixée et renvoie son évaluation.

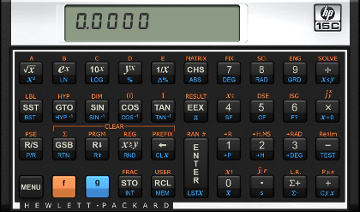
Pensez à la documentation de la fonction.

Étape 4 : jeu de test

Votre fichier Python devra comporter un moyen pour tester la fonction eval\_postfix. Le but est d'avoir toujours un moyen de tester rapidement ce qui marche ou pas.   
Il existe deux procédes :

* l’utilisation d’une fonction test qui utilisera et vérifiera des résultats avec **assert()**.
* L’utilisation de la bibliothèque doctest qui permet de lancer les tests dans la documentation de la fonction.

Pour information

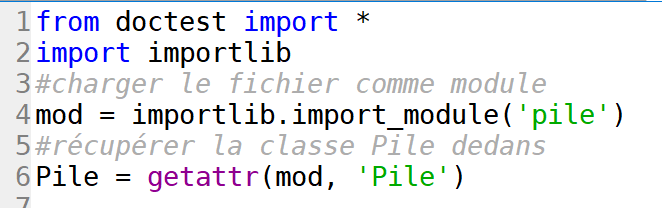
Certaines calculatrices utilisent toujours cette notation (comme la hp 15c de 1982 dans la figure ci-dessous), la valeur affichée à l'écran est celle qui est au sommet de la pile et on dispose d'une touche enter qui permet d'empiler un nombre une fois que l'on a fini de le saisir, alors que l'appui sur un opérateur lance le calcul immédiatement. Notez que cette calculatrice ne contient pas de touche =.

Outre le mécanisme d'évaluation beaucoup plus simple, la notation NPI présente un certain nombre d'avantages, même pour l'utilisateur :

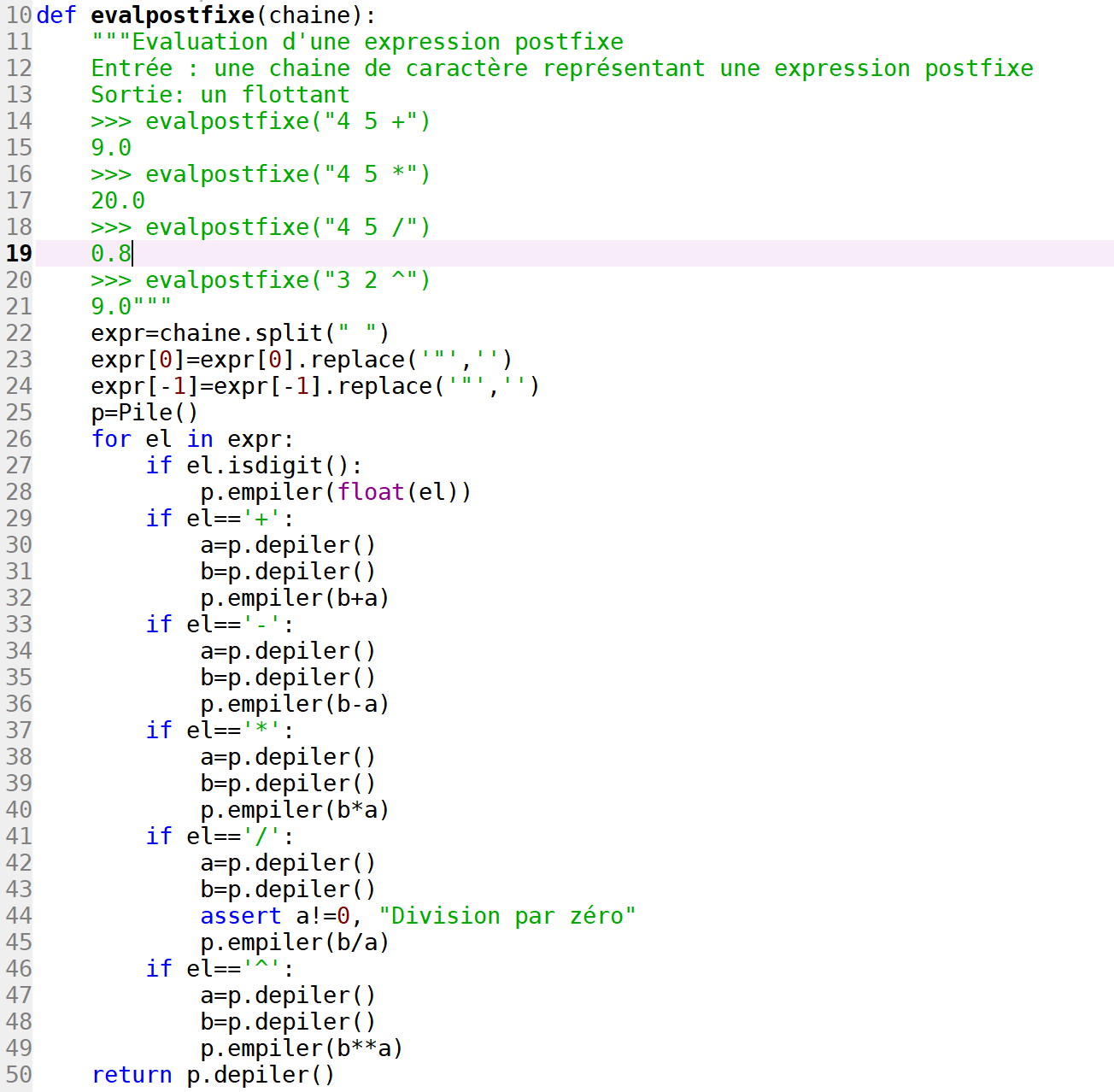
1. Les parenthèses deviennent inutiles ;
2. Les résultats intermédiaires sont tous affichés ;
3. Les erreurs de calcul se décèlent plus facilement ;
4. Les résultats intermédiaires peuvent être réutilisés facilement.

## Élément de correction

Importation des bibliothèques

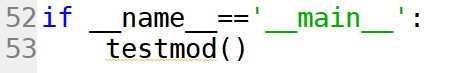
La bibliothèque doctest pour effectuer les tests dans la documentation de la fonction.

On importe ensuite importlib qui permet d’utiliser la classe Pile définit dans le fichier pile.py

On parcourt ainsi les éléments de la liste, si celui-ci est un nombre, on l’empile dans p. Si ce n’est pas le cas, on dépile des deux éléments se trouvant en haut de la pile. On fait attention au cas particulier pouvant causer une exception (le cas de la division par zéro).

Jeu de tests avec les principales opérations

Transformation de la chaîne de caractère en liste

Une fois que l’on a parcouru la liste, la pile ne contient qu’un seul élément qu’il suffit de dépiler !

Pour lancer le jeu de test, uniquement si on se trouve dans le fichier evaluationpostfixe.py, on doit rajouter ces deux lignes :