TD 20 : Piles et files

Vous disposez d'un fichier `piles.py` qui contient l'implémentation d'une pile de capacité infinie. Dans le même répertoire, `from piles import *`. Faire les exercices dans l'ordre (l'ordre a été modifié pour tout faire tenir sur une même page recto-verso).

Exercice 1 (Parenthésage simple)

On désigne ici par `mot` une chaîne de caractère. Savoir déterminer si un mot est bien parenthésé est une fonction utile pour toute analyse syntaxique. Ici, on ne s'intéresse qu'au parenthésage simple `()`. Une condition nécessaire pour qu'un mot soit bien parenthésé est qu'il contienne autant de parenthèse ouvrante `'('` que fermante `')'`. Cependant, cela ne suffit pas : le mot `')('` est mal parenthésé. Un mot est dit bien parenthésé si

- il ne contient ni parenthèse ouvrante `'('` ni parenthèse fermante `')'`, ou
- il est de la forme `'(mot)'` avec `mot` un mot bien parenthésé, ou
- il est la concaténation de 2 mots bien parenthésés

1. Écrire une fonction `parenthesage_simple(mot)` qui prend en argument la chaîne de caractère `mot` et renvoie un booléen déterminant si `mot` est bien parenthésée à l'aide d'une pile.
2. Écrire une fonction `parenthesage_simple2(mot)` qui renvoie `False` si la chaîne `mot` est mal parenthésée, ou la liste des indices des parenthèses ouvrantes et fermantes sous forme de tuples.

   `parenthesage_simple2(')(')` → False

   `parenthesage_simple2('((mot)(mot))')` → `[(1,5), (6,10), (0,11)]`
3. Quelle serait l'idée de base pour faire du parenthésage multiple avec `()`, `[]` et `{}` pour poursuivre dans le cadre de la question précédente ?
4. À la maison, vous pouvez reprendre la première question pour la traiter sans pile avec un compteur : `parenthesage_simple3(mot)`.

Exercice 3 (Implémentation d'une file – Partie 1)

Une file est une structure de donnée abstraite de type FIFO :« First In, First Out ». Les opérations autorisées sur cette structure sont `creer_file(n)`, `enfiler(element, file)`, `defiler(file)`, `taille(file)` et `est_vide(file)`.
La liste doublement chaînée `deque` du module `collections` permet d'implémenter toutes les opérations autorisées sur une file `F` en temps constant à l'aide de `F.append()`, `F.appendleft()`, `F.pop()` et `F.popleft()`.
Implémenter une file à l'aide d'une `deque`. *Implémenter* signifie qu'il faut écrire des fonctions réalisant les opérations autorisées.

Exercice 2 (Écritures d'une expression algébrique)

Les notations **infixe**, **préfixe** et **suffixe** (ou postfixe) sont des formes d'écritures d'expressions algébriques qui se distinguent par la position relative des opérateurs et de leurs opérandes. Un opérateur est écrit avant ses opérandes en notation préfixe, entre ses opérandes en notation infixe et après ses opérandes en notation suffixe. Proposée en 1924 par le polonais Łukasiewicz, la notation préfixe est aussi appelée **notation polonaise** ; et la notation suffixe, la **notation polonaise inversée**. Par souci de simplification, on se restreint à des opérateurs binaires (qui ont deux opérandes). Seule la notation infixe utilise des parenthèses. Ainsi, l'expression infixe `(1+2)*(3-4*5)` donne en notation préfixe `* + 1 2 - 3 * 4 5` et en notation suffixe `1 2 + 3 4 5 * - *`.

1. Écrire l'expression algébrique `(5-7)*3` en notation préfixe puis suffixe.
2. Écrire une fonction `eval_prefixe(expr)` qui évalue une expression en notation préfixe (aides ci-dessous). L'expression sera contenue dans une liste.
3. Écrire une fonction `eval_suffixe(expr)` qui évalue une expression en notation suffixe, l'expression étant toujours contenue dans une liste.

Évaluation d'une expression en notation préfixe : nécessité d'une pile
- lire la liste de droite à gauche (`T[::-1]` renvoie le tableau `T` inversé)
- si c'est un nombre, on l'empile
- si c'est un opérateur, on dépile ses deux opérandes de la pile, on réalise l'opération et on empile le résultat
- À la fin, la pile ne contient plus qu'une donnée : la valeur de l'expression.

C'est le même principe pour la notation suffixe mais on parcourt la chaîne de gauche à droite.
Pour tester si x est un nombre, on utilisera `isinstance(x, (int,float,complex))`.

Exercice 4 (Implémentation d'une file – Partie 2)

On dispose des opérations suivantes sur les piles : `creer_pile(n)`, `empiler(element, pile)`, `depiler(pile)`, `pile_vide(pile)`, `taille_pile(pile)` et `sommet_pile(pile)`. On se propose d'implémenter une file à partir de 2 piles. On représente `file` comme la liste d'une pile `entree` et d'une pile `sortie` : `file = [entree, sortie]`. On considère des piles et files de capacité infinie.

1. Écrire la fonction `creer_file(n)`. L'entier `n` est laissé ici par souci de compatibilité avec d'autres implémentations possibles de capacité finie.
2. Écrire la fonction `est_vide(file)`.
3. Écrire la fonction `taille_file(file)`.
4. Écrire la fonction `enfiler(element, file)`.
5. Écrire la fonction `defiler(file)`.

Comme vous pouvez le constater, la représentation en mémoire de cette file n'a rien à voir avec celle de l'exercice précédent. Pourtant, la manipulation de cette structure de donnée est identique à la précédente, on a créé une couche d'abstraction entre les données et leur manipulation : la même *structure de données abstraite* (la file) peut être réalisée par différentes *implémentations concrètes*.