Toutes les fonctions demandées dans ce TD doivent être écrites dans le fichier `operations.py`.
Attention, un copier coller du code dans la console ne fonctionnera peut-être pas...
Pour que l'interpréteur python trouve bien les fichier `parentheses.py` et `pile.py` dont nous allons avoir besoin (ne **PAS** les lire, c'est inutile), il faut avoir exécuté la feuille de script complète pour que le répertoire de travail (noté `wdir` dans la commande runfile) contienne bien ces fichiers.

I Préliminaire sur les parenthèses

1.1 Compter les parenthèses

Le but est ici de savoir si une expression arithmétique est bien parenthésée ou non. Par exemple

- $((a+b)-(c+d))*4$ est correcte
- $((1+3)-a$ ne l'est pas

Exercice 1
Plus précisément, nous allons travailler sur des chaînes de caractères contenant, entre autre, des parenthèses. Compléter la fonction `parentheses` pour qu'elle renvoie le booléen indiquant si la chaîne passée en argument est bien parenthésée.
On pourra compter les parenthèses ouvrante positivement et les fermante négativement. Pour trouver le bon algorithme, il est judicieux de trouver plusieurs expression mal parenthésées, pour différentes raisons.

1.2 Permettre d'autres délimiteurs

Le problème de l'approche précédente, si on veut généraliser à des expressions du type $[(a+b)-c]*d$ (voire autoriser des {, }) est qu'il faut introduire une variable par type de délimiteur permis. Une approche plus générale fait intervenir les piles.
L'algorithme est le suivant : on parcours, caractère par caractère, la chaîne à étudier et pour chaque caractère

- si c'est une "parenthèse" ouvrante, on l'empile.
- si c'est une fermante, on dépile (erreur si la pile est vide, ça tombe bien...) et on compare à la valeur dépilée. Elles doivent correspondre (c'est à dire que l'on doit avoir dépilé la parenthèse ouvrante correspondante), sinon l'expression est mal parenthésée (comme par exemple ([)).

A la fin du parcours, la pile doit être vide.

Exemple

Pour l'expression précédente $[(a+b)-c]*d$, les opérations de pile sont :

1. empiler [
2. empiler (
3. on voit), on dépile et on obtient (qui est le bon symbole
4. on voit] et on dépile [qui est bien le bon symbole

Fin du parcours de la chaîne, l'expression est bien parenthésée.

Aide technique

Le fichier `parentheses.py` contient 3 objets qui vont nous aider à implémenter cet algorithme

- `OUVRANTES` qui est la liste des parenthèses ouvrantes admissibles (on a choisi (, [et {)
- `FERMANTES` qui est la liste des parenthèses fermantes admissibles
- `correspondent` qui est une fonction testant si deux chaîne sont bien, dans l'ordre, une parenthèse ouvrante et une parenthèse fermante correspondante.

Pour utiliser ces outils, on écrira

```
import parentheses as par
# et par exemple
'(' in par.OUVRANTES  # retourne True
par.correspondent('(', '(')  # retourne False
par.correspondent(')', '(')  # retourne False
par.correspondent('(', ')')  # retourne True
```

Exercice 2
Compléter la fonction `parentheses_generale`

II Remontée dans le temps

2.1 Notations post-fixée

2.1.1 Principe de base

Partons d'une opération arithmétique simple : 3×4. La notation post-fixée consiste à placer les deux nombres **avant** l'opérateur. Dans cette nouvelle notation, on écrira 3 4 $\times$.
Si l'expression est plus complexe, on applique successivement le même principe à chaque opération, en plaçant avant tout des parenthèses.

$$1+2\times 3 = 1+(2\times 3)$$

Ainsi, en notation post-fixée on obtient successivement 1 (2×3) + (1ère application) puis finalement 1 (2 3 $\times$) + et avec la convention qu'une opération s'applique forcément aux deux nombres précédents, on peut retirer les parenthèses : 1 2 3 $\times$ +.
Pour bien comprendre cette notation, il faut avoir en tête que le résultat d'une opération (ici 2 3 $\times$) est un nombre.

Exercice 3
Ecrire $3+5\times 7-12$ sous forme post-fixée.

2.2 Encodage

En python, nous représentons une expression post-fixée par une liste où les nombres sont entrés directement et les opérations représentées par des chaînes de caractère (on utilisera +, -, *, / comme dans les calculs python).

2.3 Evaluation

Implémenter la fonction **evalue** dans la feuille de script *operations.py*
L'algorithme est le suivant : on parcourt la liste, élément par élément

- si on lit un chiffre on l'empile
- si on lit une opération, on dépile deux nombres, on calcule le résultat que l'on empile (attention à la non-commutativité de - et / : on convient que le sommet de la pile va à droite et l'élément d'avant à gauche de l'opérateur, comme dans notre exemple).
  On doit empiler le résultat obtenu d'après le principe que le résultat d'une opération est un nombre, utilisable par d'autres opérations.
- le résultat est le dernier nombre restant dans la pile à la fin du parcours de la liste. Si la pile est vide ou contient plus d'un élément à la fin du parcours de la liste, on utilisera

```
raise ValueError(''La liste donnée est mal formée.'')
```

  à la place d'un `return`. Ceci déclenche une erreur (le fameux texte rouge dans la console...).

Pour aller plus loin

Si vous voulez, plus tard, pouvoir utiliser des fonctions usuelles, on pourrait utiliser un dictionnaire comme la variable `_par` du fichier `parentheses.py`

III Calcul classique

Le but ici est de partir d'une expression écrite de manière usuelle, puis de la transformer en expression post-fixée et enfin de l'évaluer. Pour simplifier la transformation depuis les chaînes de caractères, on ne permet l'utilisation que de nombres entiers.

3.1 Préparer les données

3.1.1 Séparer

Si s est une chaîne, on pourra utiliser les méthodes

```
s.strip().split(' ')
```

et pour tester l'appartenance ou non on pensera aux mots clés `in` et `not in`

Exercice 4
Remplir la fonction **converti** qui prend une chaîne de caractère où les nombres et les opérations sont tous séparés par des espaces et retourne une liste contenant des entiers et des chaînes de caractères (qui représentent les opérations, comme précédemment). On doit cette fois permettre l'écriture de parenthèses, qui seront représentées par les chaînes correspondantes.
On prendra garde à bien séparer les entités par des espaces, pour tester la fonction.

3.1.2 Transformation en post-fixée

Les données sont prêtes. Construisons la **liste** post-fixée qui correspond. Dans la suite nous l'appelons "sortie". Nous allons utiliser l'algorithme suivant, qui utilise une pile pour se souvenir des opérations et des parenthèses lues :
Pour chaque élément `item` de notre liste convertie

- si `item` est un nombre, concaténer à la sortie (ie l'ajouter à la fin de la liste sortie).
- si `item` est une parenthèse ouvrante , l'empiler
- si `item` est une parenthèse fermante :
  - dépiler dans une variable x
  - tant que x n'est pas une parenthèse ouvrante, concaténer x à la sortie et dépiler dans la variable x
- dans les autres cas : tant que la pile n'est pas vide et la précédence (calculée par la fonction fournie) du sommet de la pile est supérieure (ou égale) à celle de `item`, dépiler dans la variable x et concaténer x à la sortie.
  Enfin, empiler `item` (après la boucle...).

A la fin de la lecture, il suffit de dépiler et concaténer tous les éléments restant dans la pile.
Pour l'implémentation, vous disposez de deux fonctions auxiliaires **est_entier** qui vérifie si son argument est un entier et **precedence** qui calcule la précédence d'un symbole qui avait été empilé. Complétez la fonction transforme

3.2 Finalement

Pour conclure, on peut créer une fonction **calcule** qui prend une chaîne de caractères en entrée et calcule si possible le résultat de l'opération. Pour rendre ceci interactif, on peut utiliser la fonction **input** de python.