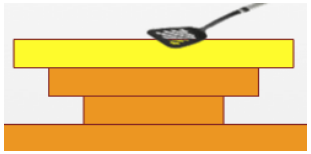
I Pile de crêpes

Le cadre de notre premier problème est assez simple. Un cuisinier a préparé des crêpes (un certain nombre) et les a empilées dans une assiette.
Avant de pouvoir présenter son travail, il vaut s'assurer que ses crêpes qui sont toutes de diamètres distincts sont rangées de la plus grande en bas de la pile à la plus petite en haut. Pour ce faire, il ne dispose que de sa fidèle spatule qui lui permet de retourner toute une partie de la pile (ou la pile en entier)



1.1 Modélisation de la situation

On représente la pile de crêpes par une instance de **Pile**, les diamètres étant donnés par des entiers dans cette pile.
Pour créer une pile dans le désordre on pourra utiliser

```
import random
from pile import Pile
p = Pile()
L = list(range(1, 11))  # on évite le diamètre nul...
random.shuffle(L)
for i in range(len(L)):
    p.empile(L[i])
```

Remarque : le deuxième import ne fonctionnera que si vous exécutez ces commandes avec F5 qui permet de changer le répertoire de travail de la console. Il faut en plus que le module **pile.py** soit dans le même dossier que votre script.

1.2 Premiers outils

Il nous faut avant tout pouvoir retourner une partie de notre pile.

Les piles

Comme en cours, nous allons utiliser une boite noire : le module *pile.py* qui est fourni. Un objet **Pile** possède les *méthodes* : **empile**, **depile**, **estvide**, **sommet** et une méthode supplémentaire **affiche** qui nous permettra de vérifier les opérations.
Le but de ce TD est de n'utiliser que des piles, même pour les opérations intermédiaires.

Exercice

Remplir les fonctions **vide_dans** (qui est une des opérations classiques vu en cours) **taille**, et **retourne** (qui représente un retournement par spatule).

A la recherche de l'algorithme

Créer une pile aléatoire de taille 5, puis essayer de la trier grâce à l'opération de retournement. On pourra utiliser la commande

```
retourne(p, k); p.affiche()  # en changeant k !
```

ainsi que la flèche du haut du clavier pour répéter les opérations.

1.3 Trions !

Normalement, vous avez trouvé une opération intermédiaire que nous devrons utiliser un certain nombre de fois. Créer une fonction qui effectue cette opération, puis implémenter votre algorithme dans la fonction **tri**

II Opérations arithmétiques

Cette fois on se propose d'implémenter une calculatrice de base, capable au départ d'effectuer les 4 opérations algébriques de base.

2.1 Notations post fixée

Une manière simple de calculer avec des piles le résultat d'une opération est d'utiliser la notation post-fixées (ou polonaise inversée).

Exemple

L'écriture de $3 + 5 * 7 - 12$ devient

$$357 * +12-$$

En python, nous représentons ceci par la liste

```
[3, 5, 7, '*', '+', 12, '-']
```

où les nombres sont entrés directement et les opérations représentées par des chaînes de caractère.

Evaluation

Implémenter la fonction **evalue** dans la feuille de script *operations.py*
Pour mémoire l'algorithme est le suivant :

- 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 /)
- le résultat est le dernier nombre restant dans la pile à la fin du parcours de la liste

Pour aller plus loin

Si vous voulez plus tard pouvoir utiliser des fonctions usuelles, on pourrait utiliser un dictionnaire.

2.2 Transformation d'une expression

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.

Préparer les données

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.
Si s est une chaîne, on pourra utiliser la fonction

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

On prendra garde à bien séparer les entités par des espaces, et à ne pas mettre d'espace au début et en fin de chaîne.

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 de notre liste convertie

- si c'est un nombre, concaténer à la sortie.
- si c'est une parenthèse ouvrante , l'empiler
- si c'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 du sommet de la pile est supérieure (ou égale) à celle de l'élément lu, dépiler dans la variable x et concaténer x à la sortie.
  Enfin, empiler l'élément lu (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

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.