TD – Types Abstraits

Exercice 1

1) On part d'une file vide et on effectue les opérations ci-dessous. Représenter l'état de la file à chaque fois que le code repasse sur l'instruction « répéter » (en représentant la tête de la file à droite), et indiquer quel est le contenu exact de la file à la sortie du programme.

- Créer une file vide
- Enfiler 2
- Répéter :
  - Défiler et assigner la valeur à v
  - Enfiler la valeur 2*v
  - Assigner 2**v + 1 à w
  - Enfiler w
  - Si w est strictement supérieur à 99997: stop

2) Écrire (sur feuille) le code python correspondant, en prévoyant un affichage du contenue de la file à la fin de chaque itération (avant la condition ; utiliser l'interface donnée dans le cours / on supposera que la file peut être affichée directement avec l'appel $print(file)$).

Exercice 2

On passe une pile en argument à une fonction. Les seules structures de données utilisables dans ce langage sont des piles (*so called*, $Factor$…) et leur interface est minimale, avec uniquement les opérations décrites dans le cours. Comment inverser le contenu de cette pile, en la mutant ?

Rédiger un court paragraphe expliquant les structures de données à utiliser, les structures de contrôle à utiliser, et la démarche générale de l'algorithme.

Exercice 3

Lien (à ouvrir avec ***Firefox ou Chrome***) : https://alainbusser.frama.io/NSI-IREMI-974/queuesortable.html. ***Il faut zoomer*** (beaucoup…) dans la page pour voir les valeurs cartes…

On dispose de cartes dans une pile, à gauche (top à droite). L'idée est de former à la fin une file de cartes triées en ordre croissant, en n'utilisant que les trois files intermédiaires.

Nota : L'ordre initial des cartes est toujours garanti pour qu'une solution existe en utilisant au plus 3 files. Mais dans l'absolu, il peut exister des jeux initiaux où trois files ne suffisent pas.

1) Essayer le « jeu » et trouver une façon de résoudre le problème (rafraichir la page pour recommencer).

On utilise les termes suivants pour décrire les différents composants du problème :

- $deck$ est la pile contenant au départ les cartes,
- $file1$, $file2$ et $file3$ sont les trois files centrales,
- $final$ est la file à droite.

2) Proposez une stratégie pour réussir à coup sûr à trier les cartes. Votre stratégie doit être la plus simple et systématique possible (l'idée étant de pouvoir la transformer en code facilement si besoin). Conseil : décomposez le problème en 2 grandes étapes : temps 1 = on enfile toutes les cartes dans les trois files centrales en appliquant une logique appropriée, temps 2 = on défile toutes les files centrales pour enfiler dans $finale$, avec une logique appropriée…

Exercice 4 – Reversing returns…

On dispose d'une pile et d'une file (classes définies en début de fichier), chacune pouvant contenir au départ un nombre inconnu d'éléments.

Le but est d'arriver à l'état final avec tous les éléments de la file inversés, le contenu de la pile étant exactement le même qu'au départ. Exemple, en considérant que la tête ou le top des structures est à droite :

Départ :	pile : [1,42,-1,3] (top)	file : [0,1,2,3] (tête)
Fin :	pile : [1,42,-1,3] (top)	file : [3,2,1,0] (tête)

Contraintes :

- Interdiction d'utiliser d'autres structures de données que la pile et la file passées en argument.
- Vous pouvez utiliser au plus ***deux*** variables en plus, stockant un entier.
- La fonction est ici une « procédure » : il faut muter les arguments.

1) Rappeler comment on désigne classiquement les piles et les files (acronymes utilisés + signification).
2) Quelle est à priori, des deux types de structures disponibles, celle qui porte l'idée d'une « inversion d'ordre » dans sont fonctionnement ?
3) Donner l'idée générale de ce qu'il faut faire, puis expliquer à quoi servent la ou les variables supplémentaires.
4) Implanter la fonction $reversing$ qui reçoit une pile et une file en arguments (dans cet ordre), et mute les structures de données de manière à renverse le contenu de la file.
5) Pourquoi la fonction n'a-t-elle pas besoin d'une instruction «$return$» ?

Exercice 5 – Parenthèses

On s'intéresse à la validation de chaînes de caractères comprenant des parenthèses. Ce type de problème est très important pour les compileurs/interpréteurs: c'est une des premières vérifications qui est faite sur votre fichier de code avant qu'il soit analysé. C'est aussi utilisé par votre IDE pendant que vous tapez du code dedans pour vous afficher quelle est la parenthèse en vis-à-vis de celle sur laquelle le curseur se trouve.

On va dans un premier temps se limiter à vérifier si des chaînes de caractères ne contenant que des "(" et ")" sont valide ou non :

```
'()'       => oui          ')('       => non
'(()())'   => oui          '(()())('  => non
'([]{})'   => oui          '(){])'    => non
```

Partie 1 : parenthèses uniquement

La fonction ci-dessous est une version naïve et inefficace pour accomplir cette tâche :

```
def naive_valid_parentheses(expr):
    old = -1
    while old != len(expr):
        old = len(expr)
        expr = expr.replace('()','')
    return len(expr) == 0
```

1) Un appel à la méthode $str.replace(a, b)$ parcourt en une fois l'intégralité de la chaîne de caractère str et remplace toutes les occurrences de a rencontrées par b. Expliquer, en FRANÇAIS comment procède cette fonction pour dire si l'expression est valide ou non (→ des phrases/paragraphes : je ne veux pas de pseudo code ! Vous pouvez passer par le débogueur de votre IDE si besoin).
2) Combien de fois le code de la boucle `while` va-t-il être exécuté dans les cas suivants :
   a) `"()()()()()()"`
   b) `"((((()))))"`
3) Si on note N le nombre de caractères d'$expr$, quelle est à priori la complexité en temps de cette méthode ?

Partie 2 :

On veut implanter maintenant une approche linéaire pour répondre à la même question, en ne parcourant la chaîne qu'une seule fois de gauche à droite.

L'idée de la méthode de la partie 1 peut en fait être vue sous un autre angle : lorsqu'on voit une parenthèse ouvrante, on avance, et lorsqu'on voit une parenthèse fermante, on vérifie que c'est bien une parenthèse ouvrante qui est placée juste avant. Si ce n'est pas le cas, la chaîne est fausse, si c'est le cas, on peut ignorer cette paire de parenthèses, ce qui fait qu'on peut poursuivre l'exploration en appliquant la même logique.

4) Dans chacun des cas suivants, identifier le premier caractère rencontré qui fait que l'expression est identifiée comme incorrecte.

`"( ) ( ) ) ( )"` `"( ( ) ( ) ) ( ( ) ) ) )"`

`"( ( ( ( ) ( ) ( )"`

5) Quelle structure de données est appropriée pour une recherche utilisant ce type de logique ?

6) Cette nouvelle version de la fonction peut être implantée à partir du pseudo-code ci-dessous (qui n'est pas tout à fait complet). Implanter cette fonction, en complétant les expressions avec les points de suspension. Vérifier ensuite que les tests passent et/ou déboguer votre code an analysant les cas d'échecs.

```
Fonction valid_parentheses(expr:str) -> bool:
    Créer une pile vide
    Pour Chaque caractère d'expr:
        Si c'est une parenthèse ouvrante:
            Stocker le caractère dans la pile
        Ou Si c'est une parenthèse fermante:
            Si le caractère au top de la pile n'est pas une parenthèse ouvrante:
                ...
            Sinon:
                ...
            Fin Si
        Fin Si
    Fin Pour
    Renvoyer ...
Fin Fonction
```

7) *Pour ceux qui ont le temps :*
   a. *Quels caractères contient la pile lors des exécutions ?*
   b. *Du coup, dans le cas où on valide uniquement des parenthèses, on n'a en fait pas du tout besoin d'une pile et on peut se contenter d'utiliser un entier à la place. Redéfinir la fonction sans la pile.*

Partie 3:

On veut maintenant valider les parenthèses d'une "vraie" expression, contenant des littéraux, des nombres, des signes et différents types de parenthèses. Exemple :

```
"{[3x + 25*(3-y)] * (j+y)} * {x-(x} + 3)"  -> False
                                fausse ici: ^
```

On veut donc implanter la fonction `valid_more_parentheses`, qui est une version « augmentée » de la fonction précédente, et qui va parcourir une chaîne de caractères quelconques pour valider trois types différents de parenthèses : ( ), [ ] et { } (nota : on partira de la version de `valid_parentheses` avec la pile).

8) Comment gérer les nouveaux jeux de parenthèses dans le code ? (= que modifier dans la boucle ?)

9) Comment gérer les caractères qui ne nous intéressent pas ?

10) Écrire la fonction et déboguer votre code.

11) Un code propre et bien conçu pour `valid_more_parentheses` devrait contenir au plus trois conditions. Si ce n'est pas votre cas, trouver une solution pour limiter le nombre de conditions (rappel : cherchez le code dupliqué et demandez-vous comment n'écrire qu'une fois les parties redondantes, et comment abstraire les parties qui changent.

Exercice 6 – Calculatrice polonaise inverse (voir fichier « TD - Stack-Queue - 6 - calculatrice polonaise.py »)

L'écriture polonaise inverse des expressions arithmétiques place l'opérateur après ses opérandes. Cette notation a l'avantage de ne nécessiter aucune parenthèses ni aucune règle de priorité des opérateurs (=« précédence »), ce qui rend l'évaluation des expressions polonaises extrêmement facile à faire via ordinateur.

Exemples :	*Notation classique*		*Notation polonaise inverse*
	$2 * 3$	$\Leftrightarrow$	$2\ 3\ *$
	$2 * 3 - 4$	$\Leftrightarrow$	$2\ 3\ *\ 4\ -$
	$2 * (3 - 4)$	$\Leftrightarrow$	$2\ 3\ 4\ -\ *$
	$4 * (11 + 2 * 3)$	$\Leftrightarrow$	$4\ 11\ 2\ 3\ *\ +\ *$

L'évaluation d'une telle expression se fait en parcourant les éléments de l'expression de gauche à droite, à l'aide d'une pile. Pour un élément de l'expression :

a) Si on c'est un nombre, on l'empile

b) Si c'est un opérateur, on dépile les deux derniers éléments et on leur applique le calcul correspondant à l'opérateur. Le résultat est ensuite empilé.

Si l'expression évaluée est valide, il y a toujours au moins deux éléments sur la pile lorsqu'un opérateur doit être appliqué, et il ne reste qu'un seul élément sur la pile à la fin de l'évaluation : c'est le résultat de l'évaluation de l'expression complète.

1) Reprendre le second et le troisième exemple ci-dessus et vérifier que l'application de la méthode d'évaluation des expressions polonaises inverses donne bien les résultats attendus. Pour cela, utiliser le modèle suivant (donné sur le premier exemple) :

Élément de l'expression traité	*(début)*	2	3	*	*(fin)*
Contenu de la pile après traitement de l'élément (top en haut)	Vide	2	3 2	…	6

On veut écrire une fonction $clacRPN(s)$ prenant une chaîne de caractères s en entrée et appliquant le raisonnement ci-dessus. Les spécifications des chaînes de caractères utilisées sont les suivantes :

- Toutes les entrées sont des calculs valides, en notation polonaise inverse
- Les seuls opérateurs utilisés sont l'addition et la multiplication
- Les valeurs numériques sont des entiers quelconques
- Chaque élément (valeur ou opérateur) est séparé du suivant par un espace

Exemple :

La fonction reçoit `"4 11 2 3 * + *"` et doit renvoyer 28.

2) Vu le format de l'argument, que faudra-t-il faire en premier sur la chaîne de caractères ? (= comment faites-vous pour extraire les valeurs et les opérateurs ? ⇒ chercher dans les méthodes de la classe str).

3) Écrire la fonction $clacRPN(s)$ qui évalue la chaîne s ne contenant pas d'autres opérateurs que $+$ et $*$, et renvoie le résultat sous forme d'entier.

On veut maintenant complexifier un peu le problème, en ajoutant des nombres négatifs (le signe sera toujours collé à la valeur dans la chaîne de caractères) et les opérations de soustraction et de division :

```
"11 2 -3 * + 4 * 1 -"    <=>    (11 + 2 * (-3)) * 4 - 1
```

Notez bien l'ordre d'utilisation des opérandes pour la soustraction : `"a b -" <=> a - b`. Il en sera de même pour la division.

Nota : on utilise la division en virgule flottante, et non la division entière.

4) Modifier le code en conséquence (deuxième version) jusqu'à passer les tests.

5) Si votre code ressemble à du « spaghetti-code » (plein de conditions et de répétition de code partout), essayer d'améliorer son écriture en utilisant :

- Quatre fonctions annexes (*aucune ne doit recevoir la pile en argument !*)
- Un dictionnaire

Nota : il est possible d'écrire cette fonction avec un seul if/else.