07 - Piles et files

Avant de commencer ce TD, créer un dossier TD07, et y sauvegarder le fichier `classes.py`*. Sauvegarder dans TD07 votre fichier de travail (par exemple* `TD07.py`*), et taper dans ce dernier en première ligne :*

```
from classes import *
```

Ceci importera tout ce dont au aura besoin par la suite.

Sauf mention contraire, ***tout sera codé en éditeur****, et* ***compilé immédiatement*** *après écriture.*

Exercice 1 Approche des classes fournies

Taper en console `help(pile)`, et lire la documentation.

Exercice 2 Autour des piles

L'idée de cette partie est de se familiariser avec la classe `pile`*. On notera lors de l'import la présence de la fonction* `affiche_pile`*, qui permet de visualiser le contenu de la pile sans la modifier (l'élément en haut est le premier à sortir).*

2.1 Générer une pile vide, appelée `P`, puis la remplir avec `'A'`, `'B'`, `'C'` et `'D'`

2.2 Prévoir avant de tester en console ce qu'il va se passer si on tape `P.depile()`. Ne pas oublier d'afficher `P` pour voir (on notera que `'D'` est perdu pour toujours...).

2.3 A partir de cet état, que doit on taper pour avoir la pile :

```
C
E
B
A
------
```

On pourra tester les fonctions qui suivent grâce à la fonction `genere_pile_aleatoire(n)`*, qui génère une pile de taille* `n`*.*

2.4 Coder une fonction `ttop(P)`, qui retourne le deuxième élément à sortir de la pile, sans modifier *in fine* la pile.

2.5 *(Une application fort utile des piles)* On cherche à savoir si une expression mathématiques est correctement parenthésée, autrement dit si le nombre de parenthèses ouvrantes est le même que le nombre de parenthèses fermantes, et qu'on ne ferme pas une parenthèse avant de l'ouvrir. Par exemple, les expressions suivantes ne sont pas correctement parenthésées :

$$2+3)+4 \quad ; \quad 4*((4+5)-2 \quad ; \quad x*(3x-2)+(5-x))$$

Le principe est le suivant : on parcourt la chaine de caractère, et on empile un élément quelconque (1 par exemple) dès qu'on croise une parenthèse ouvrante, et on dépile quand on croise une parenthèse fermante. A la fin, on doit avoir une pile vide. On retourne faux :

- si on doit dépiler alors que la pile est vide
- si la pile n'est pas vide à la fin

Coder la fonction en commençant comme suit :

```
def test_parenthese(S):
    test=pile_vide()
    for s in S:
        ----
```

Compléter les `---`

2.6 On souhaite coder une fonction donnant la taille de la pile sans modifier *in fine* cette dernière. Le principe est le suivant : on empile dans `Paux` ce qu'on dépile de `P`, jusqu'à ce que `P` soit vide, et on refait le processus inverse pour remplir à nouveau `P`. On commence comme suit :

```
def taille_pile(P):
    taille=0
    Paux=pile_vide()
    while not....:
        ....
        taille+=1
    while not ....:
        ....
    return taille
```

Compléter les `....`

2.7 Coder sur le même principe une fonction qui retourne une pile en ordre inverse de `P`, sans toucher *in fine* à `P`. Compléter les `----` :

```
def renverse_pile(P):
    Paux=pile_vide()
    Pretour=pile_vide()
    ----
    return Pretour
```

Les questions suivantes, ***plus difficiles et optionnelles****, vont vous permettre de réviser la récursivité.*

2.8 On propose la fonction

```
def taille_pile_rec(P):
    if P.est_vide():
        return 0
    top=P.depile()
    a=1+taille_pile_rec(P)
    P.empile(top)
    return a
```

Expliquer ligne à ligne pourquoi cette fonction retourne bien la taille de la pile, sans en modifier le contenu *in fine*[1].

Exercice 3 Autour des files

L'idée de cette partie est de se familiariser avec les classes ***file*** *et* ***filep*** *(files de priorité). On notera lors de l'import la présence de la fonction* ***affiche_file****, qui permet de visualiser le contenu de la pile sans la modifier (l'élément à droite est le premier à sortir).*

1. Dans cet exemple, on utilise la pile de récursivité comme pile auxiliaire pour empiler les éléments en attente.

3.1 Générer une file vide, appelée `F`, puis la remplir avec `'A'`, `'B'`, `'C'` et `'D'`

3.2 Prévoir avant de tester en console ce qu'il va se passer si on tape `F.retire()`. Ne pas oublier d'afficher `F` pour vérifier.

3.3 Prévoir avant de tester en console ce qu'il va se passer si on tape `F.top()`. Ne pas oublier d'afficher `F` pour vérifier.

3.4 On souhaite coder une fonction donnant la taille de la file sans modifier *in fine* cette dernière. Le principe est le suivant : on insere dans `Faux` ce qu'on enlève de `F`, jusqu'à ce que `F` soit vide, et on refait le processus inverse pour remplir à nouveau `F`. S'inspirer de la question 2.6 pour coder `taille_file(F)`

3.5 *(Un peu plus subtil)* **Si on sait que tous les éléments de la file sont distincts**, coder une fonction `taille_file_u(F)`, qui renvoie la taille de la file sans utiliser de file auxiliaire.

3.6 *(Un peu difficile)* S'inspirer de la question 2.7 en utilisant **une pile auxiliaire** pour générer une file en ordre inverse de `F`, tout en conservant *in fine* `F`. On aura donc :

```
def renverse_file(F):
    Paux=pile_vide()
    Fretour=file_vide()
    ----
    return Fretour
```