TP n° 2 : corrigé

MP Lycée Camille Jullian

15 octobre 2021

Tous les programmes de ce corrigé seront mis à disposition sous forme de fichier Python, seuls quelques commentaires sont présents dans ce document.

1 Exercice 1

Toutes les fonctionnalités supplémentaires demandées sont volontairement programmées en utilisant uniquement les méthodes de base **vide**, **depiler** et **empiler**. Il est bien sûr possible d'utiliser n'importe laquelle des méthodes définies à l'intérieur de la définition de classe pour programmer d'autres méthodes de cette même définition.

- la méthode **hauteur** est rendue particulièrement pénible par la restriction aux trois opérations fondamentales. De plus, il faut penser à tout rempiler avant de quitter le programme pour ne pas modifier l'état de la Pile. Bien sûr, un simple return len(self.valeurs) irait beaucoup plus vite.
- la méthode **affichage** n'est pas censée modifier la Pile qu'elle affiche, il faut donc bien faire attention, après avoir tout dépilé et affiché, à rempiler tous les éléments, et dans le bon ordre (donc après un petit reverse de la liste obtenue).
- la méthode **videpile** est simple, utiliser un raccourci du genre self.valeurs=[] serait vraiment vilain, on n'est pas censé accéder à self.valeurs dans les programmations de méthodes (sauf pour le créer dans __init__ et tester que la Pile est vide, bien entendu).
- la méthode **echange** ne justifie aucun commentaire.
- j'ai choisi pour le dépilage en bloc de renvoyer la liste dans l'ordre de dépilage (le premier élément de la liste est donc celui qui était tout en haut de la Pile), un choix qu'on peut tout à fait discuter, l'ordre inverse étant aussi pertinent.
- les deux version de **inverse** ne font que réutiliser des techniques déjà vues dans les programmes précédents.
- enfin, le principe de la méthode **decalage** est simple : on dépile en deux temps pour mettre d'un côté un premier tas constitué des k derniers éléments de la Pile, puis un deuxième contenant tous les autres, et on rempile le tout, mais en commençant par les derniers éléments, ce qui aura exactement l'effet souhaité.

2 Exercice 2

La fonction qui sert à enlever les accents est très brutale mais facilement compréhensible (je me suis limité aux accents théoriquement existants en français, le programme ne marchera donc pas sur un texte suédois, tant pis). Ensuite, j'ai décidé de réutiliser la fonctionnalité supplémentaire d'inversion de Pile programmée à l'exercice 1 pour créer deux Piles (une dans chaque sens) ne contenant que les minuscules non accentuées des caractères de la chaîne. Notez que je teste ensuite toutes les paires de caractères alors qu'on pourrait se contenter de s'arrêter à la moitié. En fait, le programme ne marche pas, car pour une raison que je n'arrive pas à comprendre, les caractères accentués ne sont pas intégrés à la Pile (alors qu'ils sont correctement convertis en caractère non

accentués. J'ai l'impression qu'il y a deux formats différents de stockage des caractères dont l'un n'est pas compatible avec la fonction **ord** mais j'avoue avoir le flemme de trouver ce qui ne va pas...

3 Exercice 3

Pas de commentaire à faire ici, la fonction **randint(0,1)** renverra aléatoirement 0 ou 1. Selon le cas, on dépile l'une ou l'autre des deux Piles. Une fois qu'une des deux Piles est vide, on regarde laquelle c'est et on finit de vider l'autre.

4 Exercice 4

On se contente ici de faire ce qui est demandé dans l'énoncé, avec une taille de File limitée à 100 objets. La méhode __init__ est un peu plus complète puisqu'on a trois objets à attacher à notre File au lieu d'un seul pour une Pile. Ensuite, il faut bien faire attention quand on empile à ne pas avoir déjà une File plein, ce qui se produit dans deux cas : la queue est juste avant la tête, ou bien la queue est à 99 et la tête à 0. Le reste est simple à comprendre.

5 Exercice 5

La fonction calculant les cases disponibles est d'une brutalité exemplaire, on ajoute toutes les cases qui ne font pas sortir de l'échiquier (ici, n représente le nombre de lignes et de colonnes, supposé égal, de l'échiquier). Le programme **cavalier** applique l'algorithme décrit dans l'énoncé, en utilisant des remove pour supprimer les cases « sans issue » des listes de cases disponibles, et en testant brutalement pour chaque nouvelle case atteignable si on ne l'a pas déjà visitée (la ligne avec le « not in »). Ce sont ces tests qui rendent le programme très peu efficace (on doit parcourir une liste à chaque fois, répété un paquet de fois, ça prend du temps). En pratique, le programme tourne très bien pour $n = 5$ ou même $n = 6$, mais pas pour des valeurs plus grandes de n (on ne peut donc pas résoudre le problème du cavalier « classique » pour $n = 8$ avec).