M2 ID APPRENTISSAGE
PYTHON ET MODÈLES DÉCISIONNELS COMPRÉHENSIBLES
LANGAGES DE SCRIPT (ANCIEN INTITULÉ)
TRAVAUX DIRIGÉS
Brice Mayag

TD1 : INTRODUCTION AU LANGAGE PYTHON
PARTIE 3

Exercice 1 : Exercice sur la structure de données Pile

Les piles définissent une structure de données de stockage qui suit une politique LIFO (Last In, First Out) d'ajout et de retrait d'élément. Un moyen simple d'implanter la structure de données des piles est d'utiliser les listes python.

1. En supposant que les piles ont été implantées au moyen des listes python, écrire les fonctions usuelles d'ajout `(empiler(e,P))` et de retrait `(depiler(P))` ainsi que la fonction `sommet(P)` qui rend le sommet de la pile et la fonction `estVide(P)` qui teste si la pile est vide.
2. Evaluation des expressions arithmétiques postfixées.

   On suppose manipuler des expressions arithmétiques sans variable définies à partir des 4 opérations arithmétiques $+;\ -;\ \times;\ \div$ et des entiers. La notation postfixée consiste à mettre les opérations arithmétiques après leurs 2 arguments. Ainsi, une expression arithmétique de la forme $n \bullet m$ où $\bullet \in \{+;\ -;\ \times;\ \div\}$ a pour notation postfixée l'expression $nm\bullet$. On suppose que les expressions arithmétiques postfixées sont stockées dans une liste de chaînes de caractères où chacune des chaînes de caractères représente soit un entier soit une opération arithmétique. En utilisant une pile, écrire une fonction qui à partir d'une expression arithmétique en notation postfixée, évalue cette expression et retourne le résultat.

Test de bon parenthésage

On considère une expression arithmétique dont les éléments appartiennent à l'alphabet suivant :

$$A = \{0; \ldots; 9;\ +;\ -;\ \times;\ \div;\ (;\ );\ [;\ ]\}$$

— Écrire un algorithme qui permet de vérifier la validité des parenthèses et des crochets contenus dans une expression arithmétique.
— Justifiez le choix de la structure de données utilisée.

Exercice 2 : Exercice sur les listes.

On va dans cet exercice surtout implanter différents algorithmes classiques de tris permettant de ranger les éléments d'une liste dans l'ordre croissant. Pour cet exercice, on supposera que tous les éléments des listes sont de même type et ce type est muni d'un ordre total.

1. Donner le code de la fonction python permettant de rechercher un élément dans une liste triée.
2. Donner le code permettant de générer la liste de tous les nombres premiers inférieurs à un entier n.
3. Algorithme du tri par sélection

   (a) rechercher l'élément le plus petit de la liste ;

(b) le placer en début de liste ;

(c) puis recommencer avec la liste privée du plus petit élément.

Donner le code python implantant le tri par sélection.

4. Algorithme du tri dit à "bulle"

   (a) Parcourir la liste en intervertissant toute paire d'éléments consécutifs non correctement ordonnés,

   (b) Ainsi de proche en proche, on place le plus grand élément mis à la fin de la liste.

   (c) Réitérer le processus jusqu'à ce que tous les éléments soient bien placés.

   Donner le code python implantant le tri à bulle.

5. Algorithme du tri par insertion (tri des joueurs de cartes)

   (a) Le premier élément constitue le point de départ pour construire une liste triée.

   (b) L'élément suivant (le second pour commencer) est insérér (i.e. bien placé) dans la liste triée (partielle).

   (c) Recommencer le processus jusqu'à ce qu'il n'y ait plus d'élément de la liste initiale à insérer.

   Donner le code python implantant le tri par insertion.

6. Algorithme du tri rapide

   (a) Partager la liste selon la valeur v d'un élément arbitraire (le pivot), en deux sous-listes respectivement des éléments inferieurs ou égale à v et des éléments de valeur supérieure à v.

   (b) Trier séparément les deux sous-listes extraites en suivant la même méthode.

   (c) Réarranger les sous-listes triées avec l'élément de référence.

   Donner le code python implantant le tri rapide.

7. En utilisant le principe du tri rapide, rechercher le kème élément plus petit dans une liste.

8. Soit une liste de n chaînes de caractères dont le contenu de chacune est soit "bleu", soit "blanc" ou soit "rouge". Écrire un algorithme et l'implanter en python permettant de trier une telle liste selon les couleurs du drapeau français avec une complexité en nombre de comparaisons de exactement n.

Exercice 3 : Velib

L'objectif de ce problème est de simuler une journée d'activités du système des Velibs.

— Un vélo est représenté par un identifiant entier unique strictement positif.

— Une station est identifiée par son nom. Elle contient un certain nombre d'emplacements ; chaque emplacement peut être vide ou contenir un vélo.

— Un déplacement est décrit par un vélo, une heure de départ, une station de départ, une station d'arrivée, et une heure d'arrivée qui sera par défaut l'heure de départ plus 30 minutes.

— Une heure (dans le sens moment dans la journée) est représentée en interne par le nombre de minutes écoulées depuis minuit. L'affichage par contre se fera sous la forme 9h27.

Pour simuler l'activité des vélos, deux listes sont utilisées. La première contient les stations. La deuxième contient les demandes de déplacements. Votre travail consiste alors à afficher dans l'ordre chronologique les événements de départ et d'arrivée des vélos. Pour cela on a besoin d'une file contenant les vélos en cours de déplacement. Dans une première approche quand la station de départ est vide, on ignore tout simplement l'événement, c'est à dire que la demande de déplacement n'est pas prise en compte et que le cycliste parisien prendra le métro. De même, si la station d'arrivée est pleine, l'événement est ignoré, c'est à dire qu'il ne termine pas. Dans une deuxième approche on se servira des coordonnées pour trouver la station disponible la plus proche de celle demandée.

Gestion des heures

1. Écrire la fonctions `AfficherHeure` qui renvoie l'heure formatée de la manière suivante : 12h05.
2. Écrire la fonction `EstAvant(h1,h2)` qui teste la chronologie entre deux heures données.

Gestion des stations

1. Écrire des fonctions permettant de tester si une station est pleine ou vide.
2. Écrire une méthode `garer(velo)` qui gare un vélo à la station dans le premier emplacement encore libre et génère une exception erreur si aucun n'est libre.
3. Écrire une fonction `retirer()` qui retire un vélo du premier emplacement occupé et génère une exception erreur si aucun n'est occupé.
4. Écrire une fonction afficher qui permet d'afficher l'état courant d'une station.

Gestion des déplacements

1. Écrire une fonction qui étant donné un déplacement produit une chaîne sous la forme [Velo 7 de Opera a 10h02 pour Nation a 10h32]
2. On va utiliser une file pour la gestion des déplacements. Écrire une fonction `ajouter(d)` qui ajoute un déplacement en fin de file. Par un mécanisme d'exceptions, on vérifiera que pour aucun déplacement `e` de la file `d.heurearrivee` est strictement avant `e.heurearrivee`.
3. Écrire une fonction qui enlève un déplacement de la file et une fonction prochain qui donne le prochain déplacement.

Gestion des velibs

1. On utilisera une table de hachage qui associe un nom de stations à une station.
2. Une liste de déplacements `plan` permet de représenter les déplacements planifiés.
3. Une liste de déplacements `Encours` permet de représenter les déplacements en cours.
4. La fonction `arriveeVelo(deplacement)` permet de gérer l'arrivée du vélo du déplacement à la station définie dans le déplacement. Si la station n'est pas pleine, le vélo est garé et un message est affiché. Sinon, seul un message est affiché.
5. La fonction `departVelo(deplacement)` fait partir un vélo de la station de départ donnée par le déplacement. Si la station n'est pas vide, alors on utilise un vélo de la station pour le déplacement et on ajoute le déplacement à la liste `Encours`. Dans tous les cas, un message est affiché.
6. Une fonction `afficher()` permet l'affichage de l'état des stations.
7. Enfin, on va simuler une activité de Velibs par l'algorithme suivant :

   - Tant qu'il y a des déplacements dans `plan` ou `Encours`
     * Si le prochain événement est un départ alors
       Enlever le déplacement de `plan` et appeler `departVelo(deplacement)`
     * Sinon (le prochain événement est une arrivée)
       Enlever ce déplacement de `Encours` et appeler `arriveeVelo(deplacement)`

Gestion des stations vides ou pleines

Dans la méthode `ajouter(d)`, on relâche la contrainte que l'heure d'arrivée de `d` est plus tard que toutes les dates d'arrivées dans la file. Par contre, la tête de file est le déplacement qui a une heure d'arrivée minimale. On a ce qu'on appelle une **file de priorité**.

1. Écrire une méthode `inserer(deplacement,liste)` qui insère le déplacement à sa place dans la liste et qui renvoie la liste.
2. Modifier la fonction `ajouter(d)` pour que `d` soit ajouté correctement dans la liste.
3. On ajoute à la représentation d'une station, ses coordonnées (x,y) dans la ville.
4. Écrire une fonction `ProchaineNonVide` et `ProchaineNonPleine` qui retourne la station la plus proche respectivement non vide ou non pleine. On pourra s'inspirer pour cela du précédent TD et des exercices précédents.
5. Dans le cas où une station est pleine, on dispose alors de 15 min supplémentaires pour finir son déplaement. Modifier la fonction `arriveeVelo(deplacement)` en conséquence.