IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Codecamp

Semaine de rentrée – TAF DCL

1 Règles du jeu

Le but est de produire en groupe un logiciel qui contient un ensemble de fonctionnalités.

Les groupes seront tirés au sort et constitués de 6 à 7 élèves. Chaque étudiant tire le numéro de son groupe depuis un chapeau. Ce tirage se fait par groupe de tirage construit pour optimiser l'hétérogénéité des groupes. Le but étant de forcer un peu des rencontres entre étudiants qui ne se connaissent pas.

Chaque groupe construit un logiciel commun en Python. La première étape commune est décrite ci-après dans la section section 2. Pour ce faire, vous pouvez procéder de la façon que vous voulez. Quelques informations sur des éléments Python indispensables figurent en section section 3.

Chaque étape doit être validée pour passer à la suivante. La validation inclut une vérification **par le groupe** du fonctionnement du logiciel, ainsi que la validation du respect des exigences de la grille de relecture. Cette relecture du code doit être faite par **au moins deux** membres du groupe. Ensuite, la validation finale doit être **attestée par un membre d'une autre équipe** qui doit vérifier le fonctionnement et attester du respect des contraintes.

Il est alors possible de tirer auprès des enseignants l'extension suivante.

Pour éviter de trop perdre de temps, il est recommandé de bien réfléchir à la conception de chaque extension avant de passer à sa réalisation. Nous conseillons également de ne pas imaginer des solutions trop complexes. En effet, le but est de faire le plus d'extensions possible, tout en fournissant un code relativement propre.

De manière générale, il est préférable de ne pas dépendre de bibliothèques externes. On essaie d'utiliser au maximum les mécanismes des bibliothèques standards de Python. En particulier, l'usage de bibliothèques de traitement des données, comme Pandas, est proscrit.

2 Le problème initial

Le but du *codecamp* est la production d'un logiciel simple de gestion de tâches.

Le point de départ est commun à tous les groupes et consiste à réaliser les commandes suivantes :

ajout, modification et suppression de tâches. Une tâche a une description (un texte sans retour chariot) et un identifiant donné à sa création. Il conviendra également de fournir un moyen de voir la liste des tâches.

L'interaction avec l'utilisateur doit rester simple ; nous optons pour une interaction en ligne de commande dans un terminal (CLI – *Command Line Interface*).

Chaque exécution du programme recevra en paramètre un nom de fichier dans lequel sont contenues les tâches déjà créées. Le résultat de l'exécution modifiera généralement le contenu de ce fichier.

Vous être libres du choix du format du fichier, mais nous recommandons de rester simple (par exemple, un format texte avec une ligne par tâche).

Cette première version doit donc permettre d'exécuter les commandes suivantes (nous supposons que votre programme s'appelle `task` :

— `task lestaches.txt add <la description sur le reste de la ligne>` : ajoute au fichier `lestaches.txt` la ligne de la tâche, retourne son identifiant ;
— `task lestaches.txt modify id <la nouvelle description sur le reste de la ligne>` : remplace la description de la tâche d'identifiant `id` dans `lestaches.txt`, renvoie un message d'erreur si la tâche n'est pas trouvée ;
— `task lestaches.txt rm id` : retire la ligne du fichier `lestaches.txt` contenant la tâche d'identifiant `id`, renvoie un message d'erreur si la tâche n'est pas trouvée ;
— `task lestaches.txt show` : affiche la liste des tâches du fichier sous la forme ci-dessous en les triant par leurs identifiants.

```
+-----+----------------+
| id  | description    |
+-----+----------------+
| ... | ...            |
+-----+----------------+
```

3 Quelques outils Python

3.1 La lecture et l'écriture de fichier

Pour rendre le projet facile à réaliser, nous recommandons d'adopter un format texte simple dans lequel une commande est stockée par ligne. On utilisera alors les fonctions Python suivantes :

— `open(`*filename*`, `*mode*`)`, qui ouvre le fichier dans un mode donné (`r` : lecture, `w` : création du fichier si nécessaire et écriture écrasant le contenu précédent, `a` : création du fichier si nécessaire et écriture de nouvelles données à la fin du fichier).
— `read()` et `readlines()`, qui retournent respectivement le contenu du fichier et une liste de lignes de contenu. ⚠ Le fichier doit être ouvert en mode `r` ; si vous lisez deux fois le contenu du fichier, le comportement ne sera probablement pas celui que vous souhaitez.
— `write(`*string*`)`, qui écrit la chaîne de caractère en argument dans le fichier. ⚠ Le fichier doit être ouvert en mode `w` ou `a`.
— `close()`, qui ferme le fichier.

Voici un exemple montrant la manière « pythonique » de traiter des fichiers. Notez que Python appelle tout seul la fonction `close` à la sortie du bloc `with` lorsque vous faites ainsi.

```
# Lecture de fichier
with open('monfichier.txt', 'r') as f:
    lines = f.readlines()

# Écriture de fichier
with open('monfichier.txt', 'w') as f:
    f.write('Mon contenu\n') # \n est le caractère de retour à la ligne
```

3.2 La gestion de ligne de commande

Nous vous suggérons de rester simple et d'utiliser la bibliothèque standard `argparse`. Sa documentation est consultable en ligne, et nous proposons en annexe A une base de travail.

Le principe de cette bibliothèque est le suivant :

- On crée un parseur avec `ArgumentParser(<···>)`;
- On définit un ensemble de paramètres à donner dans la ligne de commande avec la méthode `add_argument(<···>)`;
- On utilise le parseur avec `parse_args()`.

Il y a trois types d'arguments :

1. Les options, qui peuvent être données un peu n'importe quand;
2. Les options dites positionnelles, qui doivent être données à une position particulière;
3. Les sous-commandes, qui elle-mêmes nécessitent un sous-parseur (que l'on peut ajouter avec `add_subparsers(<···>)`).

4 Les extensions

Les différentes extensions sont classées ci-dessous en trois niveaux. Chaque extension peut provoquer un changement du format de fichier, de l'affichage, du format des commandes, *etc.* Chaque extension est cumulative; une fois tirée, il faut la réaliser en fonction des autres extensions tirées.

La spécification reste un peu floue à dessein, votre première tâche est bien sûr de clarifier le besoin et spécifier l'extension que vous allez réaliser. Penser à *documenter* et à *concevoir*.

- niveau 1 :
    1. Notion de projet : chaque tâche peut être associée à un projet
    2. État pour les tâches : chaque tâche a un état, par exemple, *started*, *suspended*, *completed*, *cancelled*
    3. Niveau de priorité : une tâche à un niveau de priorité
    4. Notion de contexte de réalisation : on associe à une tâche un contexte de réalisation, par exemple *home*, *work*, ...
    5. Notion d'étiquette : il est possible d'associer à une tâche un ensemble d'étiquettes
    6. Notion d'utilisateur : une tâche peut être associée à des utilisateurs
- niveau 2 :

1. Ajout de durée : les tâches se voient associer une durée estimée et une durée réalisée
2. Ajout d'échéance : les tâches ont une date d'échéance ainsi qu'une date de réalisation
3. Attachement d'un fichier à une tâche
4. Gestion de la configuration : le logiciel devient configurable par un fichier de configuration, la configuration est fonction de votre logiciel (par exemple, les états possibles d'une tâche ou la liste des utilisateurs sont configurables)
5. Gestion de l'historique : dès qu'une tâche est modifiée ou effacée, elle est conservée dans un fichier d'historique
6. Gestion d'un journal : toute action exécutée est ajoutée avec son résultat à un fichier journal (log)

— niveau 3 :

1. Tâche récurrente : il est possible de déclarer des tâches récurrentes, par exemple, hebdomadaires ou mensuelles
2. Méta-tâche : il est possible de prédéfinir des modèles de tâches qu'il suffit ensuite d'instancier, certains paramètres de la tâche peuvent être modifiés à l'instanciation
3. Tâche future : il est possible définir une tâche qui ne pourra être réalisée que dans le futur à partir d'une date ou de la fin d'une tâche
4. Recherche / Filtrage : il est possible de rechercher un ensemble de tâches sur la base de critère, par exemple, mot contenu dans la description, valeur de caractéristique
5. Contrôle de conformité des fichiers et de leur contenu : le logiciel comporte des mécanismes de contrôle de conformité et d'intégrité des données. Les fichiers ont-ils la forme attendue ? Le contenu des fichiers est-il cohérent ? etc.
6. Notion de dépendance entre tâches : une tâche peut dépendre d'une autre tâche pour pouvoir commencer ou attendre la fin d'une autre tâche.

5 Critères de validation

1. Qualité du code :
   a. Aucune variable globale ne doit être utilisée, les fonctions prennent toutes leurs données nécessaires en paramètres et retournent tous leurs résultats.
   b. Les fonctions gardent une taille raisonnable.
   c. Il n'y a pas d'élément inutile (fonction ou variable non utilisée par exemple).
   d. Les noms des variables, des fonctions et des fichiers sont pertinents (en adéquation avec leurs contenus et leurs usages). Le code est écrit de manière homogène.
   e. Les fonctions sont testées.
2. Fonctionnalités :
   a. Le code s'exécute (facilement) sur une machine autre que celle des membres du groupe.
   b. Le code supporte de manière correcte les fonctionnalités annoncées.
3. Documentation :
   a. Chaque fonction du code doit être décrite, ainsi que ses paramètres, résultats, effets, contraintes, son (ou ses) auteur(s)...
   b. Chaque commande utilisable doit être décrite dans l'aide (`-h` ou `--help`).
   c. un *readme* à jour est fourni, il contient une description rapide des fonctionnalités et de leur usage.

A Utilisation du parseur

A.1 Exemple de fichier exécutable

Nous supposons que le parseur se trouve dans un fichier `options.py`.

```
#!/usr/bin/env python3
import commands
from options import create_parser

# Création du parseur de ligne de commande
options = create_parser().parse_args()
try:
    # Lecture du fichier de commandes, s'il existe
    with open(options.file, 'r') as f:
        tasks = f.readlines()
    # Exécution de la commande
    if options.command == 'add':
        commands.add(' '.join(options.details), options.file, tasks)
    elif options.command == 'modify':
        commands.modify(options.id, ' '.join(options.details), options.file,
                        tasks)
    elif options.command == 'rm':
        commands.rm(options.id, options.file, tasks)
    elif options.command == 'show':
        commands.show(tasks)

except FileNotFoundError:
    print(f"The file {options.file} was not found")
```

A.2 Exemple de fichier de commandes

Nous supposons que le fichier de commandes est nommé `commands.py` et que les fonctions sont implémentées dans `core.py`.

```
import core

# Implémentation des différentes commandes, utilisant les fonctions contenues dans le module core
def add(details, filename, tasks):
    with open(filename, 'a') as f:
        task = core.add(tasks, details)
        f.write(task)
    print(f"Succesfully added order {task[0]} ({task[1]})")

...
```

A.3 Exemple de parseur

Le code ci-dessous donne une base de travail pour l'implémentation du parseur (`options.py`).

```
import argparse

def create_parser():
    # Création du parseur de ligne de commande
    parser = argparse.ArgumentParser(description='Simple task manager')
    # Ajout d'un argument positionnel (le fichier contenant les tâches)
    parser.add_argument('file', help='The tasks file')
    # Ajout d'un sous-parseur pour les sous-commandes
    subparsers = parser.add_subparsers(help='The commands to manage orders',
                                       dest='command', required=True)
    # Création du parseur pour la commande add
    parser_add = subparsers.add_parser('add', help='Add a new task.
      The rest of the command line is used for the task details, the default
      being "no details".')
    parser_add.add_argument('details', nargs='*', default="no details",
                            help="task details")
    # Création du parseur pour la commande modify
    parser_modify = subparsers.add_parser('modify',help='Modify a task given
      its id. The rest of the command line is used for the task details, the
      default being "no details"')
    parser_modify.add_argument('id', help="the task id")
    parser_modify.add_argument('details', nargs='*', default="no details",
                               help="the new details")
    # Création du parseur pour la commande rm
    parser_rm = subparsers.add_parser('rm', help='Remove an task given its id')
    parser_rm.add_argument('id', help="the task id")
    # Création du parseur pour la commande show
    parser_show = subparsers.add_parser('show', help='Show the tasks')
    return parser
```