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

Codecamp

Semaine de rentrée – TAF ILSD

1 Règles du *codecamp*

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

Les groupes seront constitués par les enseignants et comporteront entre 4 et 5 élèves.

Chaque groupe construit un logiciel similaire en Python. La première étape commune est décrite ci-après dans la section 2. Pour ce faire, vous pouvez procéder de la façon que vous voulez. Quelques informations sur des éléments Python indispensables figurent en 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 du sujet. Cette relecture du code doit être faite par au moins un membre du groupe. Ensuite, la validation finale doit être attestée par une relecture externe (d'un enseignant ou d'une autre équipe). 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

L'objectif de ce *codecamp* est la production d'un logiciel simple facilitant la gestion par un traiteur des commandes passées par ses clients. Le point de départ est commun à tous les groupes et consiste à réaliser les fonctionnalités suivantes : ajout, modification, suppression et affichage de commandes de clients. Une commande a un *identifiant* fourni par le système lors de son ajout, un *nom de client*, une *date de prestation*, un *montant* et une zone de *détails* (le contenu de la commande).

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 commandes 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 commande).

Cette première version doit donc permettre d'exécuter les fonctions suivantes :

— `orders commandes.txt add <···>` : ajoute au fichier `commandes.txt` la nouvelle commande, puis retourne son identifiant ;

— `orders commandes.txt modify id <···>` : remplace les informations de la commande portant l'identifiant `id` dans `commandes.txt`, ou renvoie un message si la commande n'est pas trouvée ;

— `orders commandes.txt rm id` : retire la ligne du fichier `commandes.txt` contenant la commande portant l'identifiant `id`, ou renvoie un message si la commande n'est pas trouvée ;

— `orders commandes.txt show` : affiche la liste des commandes enregistrées dans le fichier `commandes.txt` sous la forme ci-dessous, en les triant par identifiant.

```
+-----+--------+------+---------+---------+
| id  | client | date | montant | détails |
+-----+--------+------+---------+---------+
| ... | ...    | ...  | ...     | ...     |
+-----+--------+------+---------+---------+
```

La zone de détails est libre. Vous devez fournir des options que l'utilisateur pourra spécifier afin d'utiliser votre programme (par exemple, `-c "Famille Martin" -d 2025-12-24 -m 60.00 "2 × Menu Prestige"`). Les valeurs de certaines informations sont contraintes :

date une date valide

montant un montant valide

3 Quelques outils Python

3.1 La lecture et l'écriture de fichiers

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 quatre 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
a. Notion d'état : chaque commande peut avoir un état (par exemple, *à payer*, *payée*, *etc.*).
b. Ajout d'étiquettes : il est possible d'associer à une commande un ensemble d'étiquettes (par exemple, *végétarien*, *végan*, *etc.*).
c. Différenciation entre les différentes dates : chaque commande peut avoir une date de création, de facturation, de paiement et de prestation, chacune différente.
d. Ajout d'une ristourne : les bons clients peuvent bénéficier d'une réduction sur le prix total de leur commande.

Niveau 2
a. Composition des commandes : le champ de détails suit désormais un format standard qui facilite son traitement automatique.
b. Gestion de la configuration : le logiciel devient configurable par un fichier de configuration, adapté aux fonctions de votre logiciel.

c. Gestion d'un journal : toute action exécutée est ajoutée avec son résultat à un fichier de journal (*log*).

d. Formats d'export : il est possible d'exporter les informations du logiciel dans un ou plusieurs formats de votre choix.

Niveau 3

a. Calcul automatique du montant [*req.* extension 2.a] : le champ de montant est désormais automatiquement calculé à partir de la composition de la commande.

b. Composition des plats [*req.* extension 2.a] : chaque plat vendu dispose d'une liste d'ingrédients avec leurs quantités.

c. Création de menus : il est possible de définir des menus composés de plusieurs articles, et de les utiliser lors de l'ajout de nouvelles commandes (avec possibilité d'ajustements, par exemple, rajouter « sans poivrons » à l'entrée).

d. Recherche et filtrage : il est possible de rechercher et trier un ensemble de commandes sur la base de critères (par exemple, mot contenu dans les détails, état, valeur, date de la prestation, *etc.*).

e. 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.*).

Niveau 4

a. Gestion de la TVA [*req.* extension 2.a] : trois taux de TVA sont applicables aux produits alimentaires et aux boissons ; le logiciel permet de les spécifier.

b. Préparation du poste de travail [*req.* extension 2.a] : pour simplifier la confection des articles mis en vente, le logiciel produit, pour chaque plat, un document permettant le suivi des étapes et des préparations restantes (par exemple, n cuissons des chapons, m sauces aux morilles, k mises en barquette pour 4 personnes, *etc.*).

c. Traçabilité des produits [*req.* extension 3.b] : pour traiter correctement les rappels de produits, chaque ingrédient utilisé dans la préparation des plats est tracé avec son numéro de lot fournisseur.

d. Liste de courses [*req.* extension 3.b] : le logiciel produit automatiquement une liste de courses synthétisant l'intégralité des ingrédients à acheter ; les ingrédients déjà achetés sont indiqués comme tels.

e. Formats d'import [*req.* extension 2.d] : il est possible d'importer les informations depuis un ou plusieurs formats de votre choix ; les modifications doivent être répercutées dans le fichier initial.

Les dépendances entre les extensions sont résumées en figure 1. Notez que les extensions peuvent avoir des interactions non triviales les unes avec les autres, indépendamment de ces dépendances.

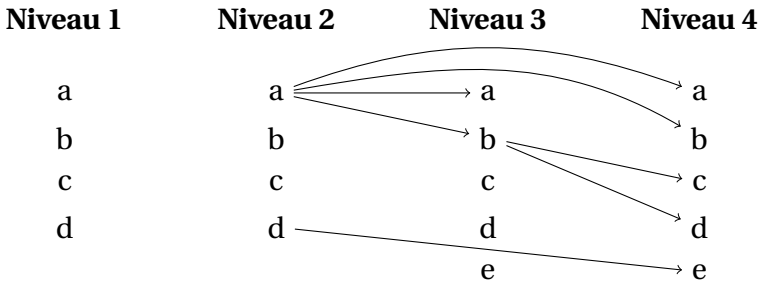

Figure 1 – Dépendances des extensions

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. On adopte un style fonctionnel[1] (exercice de style), c'est-à-dire :
      — Les variables utilisées sont non mutables ; une fois créées elle ne sont pas modifiées.
      — On évite les boucles. On préfère la récursivité. On utilise des itérateurs de type `map()`.
      — Les fonctions n'ont pas d'effet de bord (sauf I/O).
      — Pensez à introduire des contrats/assertions.
      — Pourquoi pas des lambdas pour généraliser certaines fonctions…
   f. Les fonctions sont testées par des tests unitaires fournis.
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, contraintes, effets, son (ou ses) auteur(s)…
   b. Chaque commande utilisable doit être décrite dans une page d'aide (`-h` ou `--help`).
   c. Un *readme* à jour est fourni, il contient une description rapide des fonctionnalités et de leur usage.
4. Process et outils :
   a. Vous utiliserez Git pour travailler en groupe. Un dépôt par groupe.
   b. Vous utiliserez un environnement de développement adapté à la programmation.

1. `https://docs.python.org/3/howto/functional.html`

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:
        orders = f.readlines()
    # Exécution de la commande
    if options.command == 'add':
        commands.add(' '.join(options.details), options.file, orders)
    elif options.command == 'modify':
        commands.modify(options.id, ' '.join(options.details), options.file,
                        orders)
    elif options.command == 'rm':
        commands.rm(options.id, options.file, orders)
    elif options.command == 'show':
        commands.show(orders)

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, orders):
    with open(filename, 'a') as f:
        order = core.add(orders, details)
        f.write(order)
    print(f"Succesfully added order {order[0]} ({order[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 order manager')
    # Ajout d'un argument positionnel (le fichier contenant les commandes)
    parser.add_argument('file', help='The orders 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 order.
      The rest of the command line is used for the order details, the default
      being "no details".')
    parser_add.add_argument('details', nargs='*', default="no details",
                            help="order details")
    # Création du parseur pour la commande modify
    parser_modify = subparsers.add_parser('modify',help='Modify an order given
      its id. The rest of the command line is used for the order details, the
      default being "no details"')
    parser_modify.add_argument('id', help="the order 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 order given its id')
    parser_rm.add_argument('id', help="the order id")
    # Création du parseur pour la commande show
    parser_show = subparsers.add_parser('show', help='Show the orders')
    return parser
```