Guillem CANTO Encadrant :

Thomas HERVIER Marc CASTELLA

Projet Cassiopée 2020 n°29 :

Développement d’un outil de gestion de QCM

Rapport

-

Guide d’utilisation et fonctionnement



# Introduction

Le projet Cassiopée « Développement d’un outil de gestion de QCM » avait pour but de créer une application Python de gestion d’une base de questions au format LaTeX dans le but de simplifier la confection de Questionnaires à Choix Multiples pour les Contrôles Finaux de signal. L’objectif a été de produire une application aussi légère qu’évolutive, permettant une utilisation simple et intuitive, tout en assurant une possible reprise par d’autres étudiants.

La base de questions est constituée d’un grand nombre de questions toutes écrites en LaTeX et en utilisant les utilitaires Auto Multiple Choice. AMC permet de générer automatiquement des QCMs tout en profitant de la puissance de la syntaxe LaTeX. Cela prend en charge la mise dans un ordre aléatoire des questions, des réponses aux questions, ainsi que la création d’un barème de notation.

Notre étude s’est donc majoritairement tournée vers la compréhension du fonctionnement d’AMC pour proposer un parser de questions efficace et fiable ainsi que la gestion a posteriori des questions parsées. Pour cela, nous avons créé une application en ligne de commande ainsi qu’un utilitaire graphique à l’aide de Python permettant de créer un QCM à partir de la base de questions.

Dans ce rapport, nous expliqueront comment utiliser l’application (graphique et CLI), comment fonctionne l’application, et quelles ont été les briques logicielles utilisées.

# Guide d’utilisation – CLI

Le module CL.py offre une interface en ligne de commande qui permet d’utiliser le coeur de l’application sans avoir à utiliser directement les fonctions internes du module Gestion.py

Pour utiliser CL il suffit de lancer le module comme un script python. Une fois lancé le module vous donnera accès à une console dans laquelle entrer des commandes.

>python ./Module/CL.py (depuis la racine du projet)

Le module CL permet d’accéder aux fonctions de du Parser de la base de donnée et de l’export mais ne permet pas de modifier directement les questions déjà dans la base (les seules modifications possibles sont l’ajout et la suppression de tags)

Le module CL permet donc :

* d’ajouter des questions dans la base depuis un fichier LaTeX
* de tagger et visualiser les questions de la base
* d’effectuer des recherches simples dans la base (par nom, tags ou mots clefs)
* d’exporter une sélection de question au format LaTeX

Pour ajouter des questions la fonction principale à connaître est « parsef » suivi du chemin vers le fichier LaTeX contenant les questions à ajouter, une foi le fichier analysé vous aurez la possibilité de visualiser et tagger les questions avant leur ajout dans la base de donné grâce au « buffer » qui sert à vérifier que l’analyse du fichier a bien donné le résultat attendu et stocke le résultat de l’analyse avant de l’ajouter formellement dans la base. Une fois que vous êtes sûr que les questions contenues dans le « buffer » sont correctes il suffit d’utiliser « saveb » afin d’ajouter les questions à la base.

Pour exporter des questions ou leur ajouter des tags la première étape est de les « sélectionner » pour cela plusieurs fonctions sont disponibles et toutes commencent par « selectby... » suivi du critère pris en compte. Une fois une ou plusieurs de ces commandes effectuées il vous sera possible de manipuler votre sélection et enfin de sauvegarder vos modifications avec « save » ou d’exporter vers un fichier avec « exportlatex » suivi d’un nom de fichier.

Il est aussi possible de visualiser directement les détails ou le code LaTeX d’une question sélectionné ou dans le « buffer » avec les commandes commençant par « print... » comme « print » ou « printlatex ».

Pour avoir une liste détaillée des commandes et de leurs effets :

* « help » affiche un message d’aide ainsi qu’une liste de commandes utile
* « help -l » affiche la liste de toutes les commandes disponibles
* « help commande » affiche un message d’aide pour la commande spécifié

IMPORTANT : Pour fermer l’application avec CL veuillez utiliser la commande « exit » dans le cas contraire les changements apportés à la base de donné (ajout de question, tags) seront potentiellement perdus.

NOTE : Le module CL.py ne permet pas d’accéder à toutes les fonctionnalités de l’application mais il permet d’éviter des erreurs qui pourraient résulter d’une mauvaise utilisation du module Gestion.py.

# Guide d’utilisation – Graphique

L’utilitaire graphique a pour but de simplifier les démarches de création de QCM.

Au démarrage, l’application informe à l’aide d’un fenêtre pop-up la création d’un fichier *db.json* s’il n’existe pas, ou son chargement s’il existe déjà. Il s’agit du même fichier que celui utilisé par la CLI. Ainsi il y a une compatibilité parfaite entre les deux applications. On peut commencer le travail sur un outil et le finir sur l’autre suivant le besoin.

Ensuite, l’application s’ouvre sur la fenêtre principale. Il s’agit d’une fenêtre scindée en 3 parties : la zone correspondant à la base de données, une zone permettant de travailler sur la question sélectionnée, et une zone d’export.

/\*Insérer screen 1\*/

Comme vous pouvez le constater, il n’y a pour l’instant pas de question dans la base de données. Pour la remplir, il faut tout d’abord importer un fichier TeX en sélectionnant « Importer un fichier TeX… » dans le menu « Importer » de la barre de menu. Un fois cela fait, la zone de base se remplit et il est possible de sélectionner une question pour afficher ses détails.

/\*Insérer screen 2\*/

Dans la zone de détail de la sélection   permet plusieurs actions.

Il est tout d’abord possible de modifier la question sélectionnée en modifiant les champs. Il est possible de modifier : le nom, l’énoncé, l’unicité de la bonne réponse, les réponses (laisser des cases vides permet de retirer des possibilités de réponse), et quel(s) est (sont) la (les) réponse(s) juste(s). Il ne faut pas oublier de valider la modification de la question en appuyant sur le bouton « Mettre à jour ». Il y a une vérification permettant de s’assurer que le nombre de réponses correctes est cohérent avec le choix du type de question.

Ensuite, il est possible de prévisualiser la question grâce au bouton éponyme. Pour avoir accès à cette fonctionnalité, il faut avoir un compilateur LaTeX, les diverses dépendances, et AMC installé sur sa machine. Cette fonction ouvre une nouvelle fenêtre qui affiche la question compilée.

/\*Insérer screen 3\*/

Enfin, le bouton sélectionner permet de mettre la question sélectionnée en zone d’export. Un fois dans cette zone, il est possible de sélectionner la question, donnant la possibilité de retirer la question de la zone d’export.

/\*Insérer screen 4\*/

Un fois la sélection des questions à exporter terminée, on peut exporter les questions en sélectionnant soit « Exporter un QCM… », soit « Exporter un fichier Moodle… » dans le menu « Exporter » de la barre de menu. Ces deux fonctionnalités permettre d’exporter les questions respectivement au format LaTeX classique et au format LaTeX Moodle.

Au moment de fermer l’application, une fenêtre demandera si vous voulez sauvegarder la base de données ou non. Le fait de sauvegarder la base de données permettra de conserver les imports et les mises à jour de question pour une prochaine utilisation de l’application. L’ensemble des modifications sera enregistré dans *db.json*.

# Fonctionnement logiciel

Notre application repose sur 7 fichiers python qui ont chacun un rôle bien défini :

* QCM.py : fichier permettant la représentation sous forme de classe d’une question ainsi que les méthodes permettant de manipuler et exporter une question.
* Parser.py : fichier permettant de créer des objets Question à partir d’une source LaTeX.
* DB.py : fichier contenant toutes les méthodes nécessaires pour gérer la base de données d’objet Question.
* Gestion.py : il s’agit du fichier contenant l’ensemble des fonctions permettant le fonctionnement algorithmique des applications.
* LatexDisplay.py : fichier permettant l’affichage de la prévisualisation d’une question.
* CL.py : exécutable python contenant le fonctionnement front-end de l’application en ligne de commande.
* GUI.py : exécutable python contenant le code front-end de l’application graphique.

Nous allons maintenant détailler le fonctionnement de chacun des modules.

## QCM.py

Le fichier QCM.py est un fichier important dans le modèle du projet car il défini les classes permettant de représenter informatiquement une question de QCM.

On y définit alors 3 classes :

* Question
* Réponse
* TypeQCM (qui est en réalité une énumération)

Tout d’abord la classe Question a pour champs :

* type : TypeQCM correspondant au type de la question.
* nom : String contenant le nom de la question.
* amc\_option : Array contenant les options AMC de la question.
* enonce : String contenant l’énoncé de la question.
* reponses : Array de Réponse qui contient toutes les possibilités de réponse.
* tags : Array de String contenant des tags qui permettent de catégoriser une question. Les tags n’existent pas pour LaTeX, c’est uniquement pour l’application.

Et pour méthodes de classe :

* short\_str : permet d’avoir une représentation simplifiée de la question sous la forme d’un String.
* to\_dict : permet d’avoir un dictionnaire python représentant l’objet Question.
* to\_latex : permet d’avoir une String formatée en LaTeX de l’objet Question.
* to\_moodle\_latex : permet d’avoir une String formatée en LaTeX Moodle de l’objet Question.
* get\_answers : renvoie les objets Réponses liés à la Question.
* get\_right\_answers : renvoie les objets Réponses étant marqués corrects.
* get\_wrong\_answers : renvoie les objets Réponses n’étant pas marqués corrects.
* add\_tag : permet l’ajout d’un tag à l’objet question.
* remove\_tag : permet de retirer un tag à l’objet question. Si ce tag n’est pas sur l’objet, ne fait rien.

Ensuite, la classe Réponse a pour champs :

* est\_correcte : booléen true si il s’agit d’une réponse correcte.
* enonce : String de l’énoncé de la réponse

Et pour méthodes de classe :

* to\_latex : renvoie un String formaté en LaTex de la réponse.
* to\_moodle\_latex : renvoie un String formaté en LaTeX Moodle de la réponse.

Notre troisième classe est un simple type énuméré qui permet de préciser si une question possède une ou plusieurs réponses. L’utilisation d’une énumération permet de s’assurer qu’il n’y aura pas de problème de type inconnu de question au moment de la création de l’objet.

Nous définissons également trois fonctions : type\_from\_str, str\_from\_type, et moodle\_from\_type qui servent à gérer le type de Question au moment de l’import (type\_from\_str) et de l’export (str\_from\_type et moodle\_from\_type).

## Parser.py

Cette classe contient l’ensemble des fonctions permettant de créer des objets QCM.Question à partir d’un fichier LaTeX source. Notre parser est fonctionnelle pour les implémentations françaises d’AMC.

Au sein de ce fichier, on peut tout de même différencier 2 types de fonctions : les fonctions utilitaires et les fonctions de parsing. Les fonctions de parsing vont se servir des fonctions utilitaires pour traiter un fichier TeX.

On compte tout d’abord deux fonctions utilitaires :

* get\_block(text, block\_open\_char, block\_close\_char) : cette fonction permet de renvoyer le premier block de texte contenu entre deux caractères. Il s’agit d’une fonction très importante pour le parser car en LaTeX, de nombreux paramètres importants sont contenus entre brackets {}. Il faut pouvoir facilement récupérer ces paramètres.
* pattern\_at(text, index, pattern) : cette fonction renvoie un booléen permettant de savoir si un pattern est bien présent à l’index spécifié dans un texte. Cela permet notamment de savoir si une réponse est bonne ou mauvaise.

Ensuite, on compte trois fonctions de parsing :

* parse\_reponses(r\_lines) : permet de récupérer l’énoncé et la justesse des réponses contenues dans le texte r\_lines.
* parse\_question(q\_lines) : permet de récupérer un objet QCM.Question à partir des lignes de texte contenues dans q\_lines. Cette fonction fait appelle à parse\_reponses pour constituer les objects QCM.Reponse associés à la question.
* parse\_latex(latex) : permet de récupérer une liste d’objets QCM.Question à partir d’un texte quelconque latex. Cette fonction va appeler tout simplement parse\_question sur chacune des lignes comprises entre des balises "\begin{question" et "\end{question".

## DB.py

Ce module permet la création et la gestion d’une base de données d’objets QCM.Question au format JSON. Il s’agit d’une manière assez simple de stocker les données. Ce que l’on perd en manipulations SQL, on le gagne en portabilité. En effet, JSON a été pensé pour être facilement parsé. Nous avons donc là une base de données facilement réutilisable dans une autre application. De plus, l’utilisation d’un outil comme Code Beautify (<https://codebeautify.org/jsonviewer>) permet à un humain de facilement lire et comprendre le contenu de la base de données. Il est donc possible d’intervenir directement sur les données sans avoir recours à un outil plus complexe qu’un éditeur de texte.

Au sein de ce fichier, on commence d’abord par définir quatre fonctions utilitaires :

* tag\_check(tags, required) : renvoie un bouléen permettant de savoir si la liste tags contient bien tous les tags contenus dans la liste required.
* keyword\_check(text, keywords) : renvoie un booléen permettant de savoir si le texte text contient bien l’ensemble des mots clés de la liste keywords.
* Reponse\_from\_dict(rdict) : renvoie un objet QCM.Reponse représenté par le dictionnaire rdict.
* Question\_from\_dict(qdict) : renvoie un objet QCM.Question représenté par le dictionnaire qdict. Cette fonction fait elle-même appel à reponse\_from\_dict pour obtenir les objets QCM.Reponse associés à cette question.

Ensuite, on va définir une classe Base. Cette classe est la représentation logicielle de la base de données *db.json*. Elle permet de définir un grand nombre de méthodes de classe pour manipuler les données contenues au sein de la base. Cette classe est facilement réutilisable pour une reprise du projet car elle fournit tous les outils nécessaires à la gestion d’une base de données.

La classe Base est constituée de 3 champs :

* filename : le nom du fichier JSON dans lequel sera stockée de manière persistante la base de questions.
* data : une liste de dictionnaires représentant chacun une question. Le champ data contient une image logicielle de ce qui est contenu dans le fichier de base de données. Cela signifie que tant que l’on ne l’a pas explicitement demandé, les modifications apportées aux questions contenues dans data ne seront pas sauvegardées dans le fichier. Cela permet de travailler librement sur les questions et dans le cas d’une erreur, les anciennes données ne sont pas perdues. Cependant, cela signifie aussi qu’il faut se souvenir de sauvegarder la base de données à chaque modification désirée.
* nextindex : entier permettant de connaître l’index de la prochaine question à ajouter dans la base. Cela permet de donner un identifiant unique à chaque question (son index dans la liste data).

Ensuite, l’importance de cette classe réside principalement dans ses méthodes permettant de manipuler les données :

* persist() : permet de sauvegarder l’image de la base *data* dans le fichier représenté par *filename*. Cela assure la persistance des données.
* add\_question(question) : permet d’ajouter un objet QCM.Question à *data* tout en incrémentant *nextindex*.
* add\_multiple(questions) : permet d’ajouter une liste d’objets QCM.Question à *data*. Il s’agit en réalité d’un appel itératif à add\_question.
* del\_question(index) : retire la question à l’index index dans *data*. Cette fonction n’a pas d’action sur nextindex.
* get\_question(index) : renvoie l’objet QCM.Question présent à l’index index. Cette méthode ne gère pas les *IndexError*.
* update\_question(index, question) : remplace la question *data*[index] par question. S’il n’y a initialement pas de question à cet index, crée ce nouvel index et y insère la question.
* question\_by\_name(name) : renvoie une liste d’objets QCM.Question dont le champ *name* est name. Nous sommes partis du principe que l’unicité des noms des questions n’était pas forcément à respecter. C’est pourquoi nous renvoyons une liste. Pour obtenir une question bien précise, il vaut mieux connaître son index et utiliser get\_question.
* question\_by\_tag(tags) : renvoie une liste d’objets QCM.Question qui contiennent tous les *tags* indiqués dans tags.
* question\_by\_keyword(keywords) : renvoie une liste d’objets QCM.Question qui contiennent tous les keywords de la liste keywords dans leur champ *enonce*.
* all\_questions() : renvoie une liste d’objets QCM.Question qui est composée de toutes les questions dans la base.

## Gestion.py

Il s’agit du module le plus important de notre solution logicielle. En effet, Gestion.py est le module qui implémente les fonctions permettant d’effectuer toutes les opérations sur les questions et l’image logicielle de la base de questions. Cela gère aussi les ensembles de questions à afficher et à sélectionner.

Pour utiliser les fonctions de ce module, il suffit de l’importer et d’appeler la fonction init() une fois. Cela fait qu’il est possible de se servir des fonctions de Gestion.py dans la console python. Cependant il est conseillé d’utiliser les applications car les fonctions de Gestion.py sont un peu trop brutes. Par exemple, la plupart ne prennent pas en compte les exceptions, il faut donc penser à wrapper les fonctions au niveau applicatif. Cependant, il peut s’avérer pratique d’utiliser la console notamment pour comprendre le fonctionnement des fonctions définies dans ce fichier.

Au début du fichier on commence par déclarer 3 variables :

* sel : une liste de tuples(string, QCM.Question) dans laquelle on va mettre toutes les questions sélectionnées en base pour l’exportation.
* view : une liste de tuples(string, QCM.Question) qui sert à afficher les questions de la partie gauche du GUI.
* buffer : une liste de QCM.Question formée après un parsing
* db : une DB.Base qui correspond à l’image logicielle de la base de données de question.

Nous commençons d’abord par définir une fonction init() qui permet d’initialiser la base de données à l’aide du fichier *db.json*. L’appel à cette fonction est obligatoire avant d’utiliser les fonctions de Gestion.py sinon tous les accès à la base de données produiront des exceptions.

Ensuite, nous définissons trois getters pour le buffer, sel, et view.

Par la suite, nous définissons un ensemble de fonctions pour parser et exporter. Comme plusieurs d’entre elles sont dupliquées suivant le format (LaTeX ou Moodle), nous remplacerons les mots latex et moodle par xxx ici :

* export\_sel\_xxx(filename) : permet d’exporter les questions de *sel* au format sélectionné dans le fichier filename.
* export\_buffer\_xxx(filename) : permet d’exporter les questions de *buffer* au format sélectionné dans le fichier filename.
* get\_xxx\_str(index) : renvoie une String formatée de la question à l’index index dans *sel*.
* get\_buffer\_xxx\_str(index) : renvoie une String formatée de la question à l’index index dans *buffer*.
* get\_all\_xxx\_str() : renvoie une liste de Strings formatées de toutes les questions dans *sel*.
* get\_all\_buffer\_xxx\_str() : renvoie une liste de Strings formatées de toutes les questions dans *buffer*.
* parse\_file(filename) : remplit *buffer* avec les questions trouvée dans le fichier filename.

Il convient maintenant d’ajouter les fonctions qui permettent les manipulations de base de *sel*, *buffer*, et *db*. Comme certaines fonctions sont en double pour *buffer* et *sel* nous les remplaceront par xxx :

* save\_xxx() : sauvegarde le contenu de xxx dans *db*.
* clear\_xxx() : vide le tableau xxx.
* remove\_xxx(index) : retire la QCM.Question présente à l’index index dans xxx.
* get\_index(index) : renvoie la QCM.Question présente à l’index index dans *db*.
* update\_index(index, update) : remplace la question à l’index index par update dans *db*.
* persist\_db() : sauvegarde *db* sur le disque dur, dans le fichier *db.json*.

Nous avons ensuite implémenté des fonctions permettant de manipuler les fonctions au sein de *sel* et de *buffer*. Comme ci-dessus, xxx peut soit faire référence à *sel*, soit *buffer* :

* apply\_tag(index, tag) : applique le tag tag à la question à l’index index au sein de *sel*.
* apply\_tag\_all(tag) : applique le tag tag à toutes les questions de *sel*.
* apply\_tag\_buffer(index, tag) : applique le tag tag à la question à l’index index au sein de *buffer*.
* apply\_tag\_all\_buffer(tag) : applique le tag tag à toutes les questions de *buffer*.
* get\_xxx\_str(index) : renvoie la String non formatée de la question à l’index index dans xxx.
* get\_short\_xxx\_str(index) : renvoie la String raccourcie non formatée de la question à l’index index dans xxx.
* get\_all\_xxx\_str() : renvoie une liste de Strings non formatées des questions de xxx.
* get\_all\_short\_xxx\_str() : renvoie une liste de Strings raccourcies non formatées des questions de xxx.

Il y a également de présent des fonctions permettant de manipuler des questions au sein de *db*:

* remove\_duplicates() : retire les questions dupliquées au sein de *db*.
* select\_name(name) : ajoute à *sel* les questions de *db* dont le nom est name.
* select\_tags(tags) : ajoute à *sel* les questions de *db* qui ont l’ensemble des tags.
* select\_keywords(keywords) : ajoute à *sel* les questions de *db* qui possèdent l’ensemble des keywords dans leur énoncée.
* select\_all() : ajoute à *sel* l’ensemble des questions de *db*.
* select\_id(db\_id) : ajoute à *sel* la question ayant pour id db\_id dans *db*. Renvoie un booléen permettant de savoir si cette question est effectivement présent en base.
* refresh\_sel() : met à jour *sel* avec les nouvelles données de *db*. Permet de récupérer les mises à jour apportées aux questions dans *db*.

Enfin, on dispose ici d’une dernière catégorie de fonctions, celles permettant de manipuler les questions dans *view*:

* clear\_view() : permet de vider *view*.
* view\_all() : met toutes les questions de *db* dans *view*.
* view\_name(name) : mets les questions de *db* dont le nom est name dans *view*.
* view\_tags(tags) : met l’ensemble des questions de *db* qui contiennent tous les tags tags dans *view*.
* view\_keywords(keywords) : met l’ensemble des questions de db qui contiennent les keywords dans leur énoncé dans view.
* refresh\_view() : met à jour *view* avec les nouvelles données de *db*. Permet de récupérer les mises à jour apportées aux questions dans *db*.

## LaTeXDisplay.py

Ce fichier est un module tout simple ne contenant qu’une seule méthode. La méthode on\_latex(master, latex\_str) permet d’afficher la fenêtre de prévisualisation d’une question dans l’application GUI. Les arguments sont simplement master, la fenêtre à laquelle cette sous fenêtre est rattachée et latex\_str, une String formatée en LaTeX. Grâce à la bibliothèque Sympy et plus particulièrement grâce à la fonction preview, on peut commander la compilation d’un texte LaTeX avec le compilateur de la machine et l’enregistrer sous la forme d’une image. On va ensuite charger cette image avec PIL et l’afficher en plein écran de la nouvelle fenêtre.

## CL.py

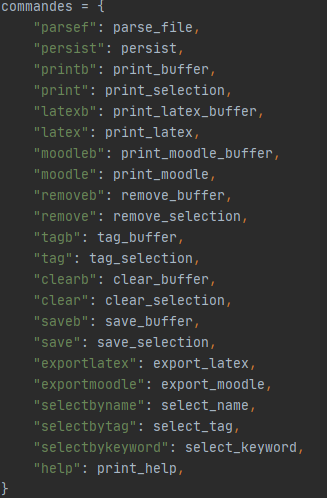
Ce module a pour rôle de créer l’outil en ligne de commande dont l’utilisation a été expliquée précédemment.

Il s’agit d’une fonction main() dans laquelle on va avoir une boucle while qui ne s’arrêtera que quand l’utilisateur aura entré la commande ‘exit’. Cela va aller chercher dans le dictionnaire *commandes* quelle est la fonction associée à la commande entrée avant de l’exécuter.

Les commandes disponibles sont les suivantes :

* help <commande> : affiche un message d'aide
* parsef <fichier>... : ouvre le(s) fichier(s) spécifié(s) et recherche les QCM écrites en LaTeX
* print(printb) <?index>... : affiche les questions de la sélection (du buffer) ou seulement les indexs spécifiés
* latex(latexb) <?index>... : affiche le code LaTeX des questions sélectionnées (du buffer) spécifiées
* moodle(moodleb) <?index>... : affiche le code LaTeX moodle des questions sélectionnées (du buffer) spécifiées
* save(saveb) : enregistre les modifications sur la sélection (enregistre le buffer) dans la base
* persist : répercute les chagements dans la base de donnée sur le disque dur
* clear(clearb) : Remet à zéro la sélection (le buffer) sans sauvegarder
* remove(removeb) <index>... : enlève les questions aux indexs voulues de la sélection (du buffer)
* tag(tagb) <tag>... : applique un tag à la sélection (au buffer)
* selectbytag, selectbyname, selectbykeyword <arg>... : sélectionne dans la base suivant un critère
* exportlatex (exportmoodle) <fichier> : exporte la sélection au format LaTeX (Moodle LaTeX) dans le fichier
* exit : enregistre la base et ferme l'application help <commande> : affiche un message d'aide
* parsef <fichier>... : ouvre le(s) fichier(s) spécifié(s) et recherche les QCM écrites en LaTeX
* print(printb) <?index>... : affiche les questions de la sélection (du buffer) ou seulement les indexs spécifiés
* latex(latexb) <?index>... : affiche le code LaTeX des questions sélectionnées (du buffer) spécifiées
* moodle(moodleb) <?index>... : affiche le code LaTeX moodle des questions sélectionnées (du buffer) spécifiées
* save(saveb) : enregistre les modifications sur la sélection (enregistre le buffer) dans la base
* persist : répercute les chagements dans la base de donnée sur le disque dur
* clear(clearb) : Remet à zéro la sélection (le buffer) sans sauvegarder
* remove(removeb) <index>... : enlève les questions aux indexs voulues de la sélection (du buffer)
* tag(tagb) <tag>... : applique un tag à la sélection (au buffer)
* selectbytag, selectbyname, selectbykeyword <arg>... : sélectionne dans la base suivant un critère
* exportlatex (exportmoodle) <fichier> : exporte la sélection au format LaTeX (Moodle LaTeX) dans le fichier
* exit : enregistre la base et ferme l'application

Le dictionnaire des commandes associées à leur fonction est le suivant :



Chacune de ces fonctions est en général un wrapper simple des fonctions de Gestion.py. On y traite généralement les arguments passés à la commande pour les donner aux fonctions de Gestion.py.

## GUI.py

Le fichier GUI.py définit toutes les variables et les méthodes permettant d’afficher l’application graphique. On y utilise la bibliothèque Tkinter car c’est la bibliothèque python standard pour la création d’interfaces graphiques. Cela a de nombreux avantages car cela signifie que l’application est facilement maintenable sans avoir à installer de bibliothèque graphique. Cela rend l’utilitaire encore plus léger. Cependant, Tkinter est une bibliothèque vieillissante et assez peu puissante. Ainsi, on se retrouve à devoir faire beaucoup de compromis pour se plier aux exigences de la bibliothèque.

La plupart des fonctions de ce fichier font appel à des fonctions du module Gestion.py. Ainsi, on ne s’occupe ici que du front-end de l’application. Tout le back-end est fait à part.

Pour ce qui est de la structure de ce fichier, il est structuré comme suit :

Des lignes 8 à 51, on va déclarer la plupart des variables correspondants à des éléments graphiques.

On commence tout d’abord par ajouter les frames. Cela nous permet de partitionner la fenêtre en 3 parties : la base, le détail de la question courante et la sélection. Pour la stratégie d’affichage de ces frames, on optera pour un simple pack. Cela permet de les placer horizontalement.

Ensuite, on va déclarer les listes permettant d’afficher l’ensemble des questions dans la base et la sélection.

On va consacrer beaucoup plus de lignes de code à la déclaration des éléments graphiques de la frame détail. C’est dans ces lignes que l’on voit le manque de praticité de Tkinter. En effet, pour chaque Entry (zone de texte modifiable), déclarer une StringVar. Cela permettra de get et de set le texte dans les entries. Il n’est pas possible d’avoir une action directement sur ces dernières, tout doit se faire sur les StringVars. De même, il faudra déclarer des IntVar pour chaque checkbox, les petites cases que l’on peut cocher. Ces éléments seront affichés selon une stratégie grid car cela va nous permettre de placer plus précisément les éléments qu’avec une simple méthode pack. On peut ainsi donner plus d’ergonomie à l’application.

On définit la fonction display\_detail\_question(question) qui prend en argument un objet QCM.Question et qui s’occupe de set les StringVar et IntVar pour afficher la question sélectionnée au centre de l’écran. On se servira de cette fonction à deux reprises : pour l’affichage des questions en base ou en sélection.

Par la suite, on va définir les fonctions que l’on passera au paramètre ‘command’ des boutons de la zone de détail. Il est bon de noter que l’on définit deux fonctions différentes pour le bouton export : button\_export\_command() et button\_export\_retirer(). En effet, en fonction de si l’on sélectionne une question en base ou en selection. Ainsi, on va devoir dynamiquement modifier la commande associée au bouton pour pouvoir ajouter ou retirer la question de la sélection.

Il va maintenant falloir définir le comportement des listes de base et de sélection, pour cela on déclare deux fonctions, base\_onselect(event) et selection\_onselect(event), que l’on va ensuite binder à l’événement ListboxSelect comme suit : xxx\_view.bind(‘<<ListboxSelect>>’, xxx\_onselect). Ces méthodes appellent display\_detail\_question pour afficher la question sélectionnée et modifient le comportement des boutons pour qu’ils correspondent bien à de quel côté la question sélectionnée vient.

Vient ensuite la fonction load\_db(). On va appeler cette fonction au démarrage de l’application. C’est dans cette méthode que l’on va initialiser le module Gestion et qu’on va procéder à l’affichage initial des questions dans la base de données – si elle existe.

Il faut ensuite déclarer les fonctions import\_tex(), export\_tex(), et export\_moodle(). Ces fonctions effectueront les opérations d’import et d’export proposées dans le menu en haut de la fenêtre.

Ensuite, on définit deux fonctions create\_menu() et create\_frames(). Ces deux fonctions commandent l’affichage respectivement du menu en haut de la fenêtre et des 3 zones graphiques. On y fera appel dans set\_gui(), la fonction suivante, qui permet de mettre en place l’interface graphique après avoir chargé la base de données et clos le message qui indique à l’allumage si la base *db.json* est présente ou non.

La fonction exit\_protocol() permet d’afficher une fenêtre au moment de quitter qui demande si l’utilisateur souhaite sauvegarder la base de données ou non. Pour faire appel à cette fonction, il suffit d’ajouter un protocole à la fenêtre *master* en invoquant master.wm\_protocol("WM\_DELETE\_WINDOW", exit\_protocol).

Enfin, on va créer trois fonctions : update\_view(), update\_selection(), et refresh(). La fonction refresh va faire appel aux deux précédentes pour mettre à jour l’affichage des questions. Ce sont des fonctions qu’il faut appeler à chaque modification de la base (ajout de question, …) et de la sélection (ajout et retrait de question, …).

L’ensemble de ces fonctions permettent de définir le comportement de l’interface graphique. Il n’y a plus qu’à paramétrer la fenêtre *master* et à l’afficher à l’aide de master.mainloop().

# Propositions d’amélioration

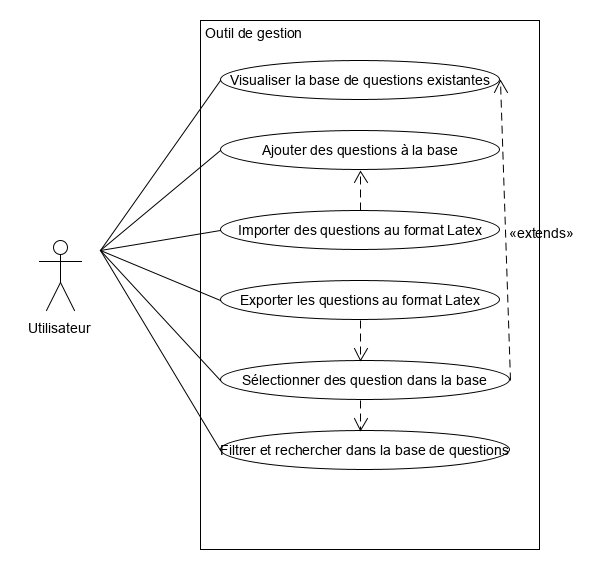
Une proposition d’amélioration majeure serait le côté front-end de notre utilitaire. En effet, nous avons pu mettre en place une structure logique simple mais puissante et évolutive. Cependant, nous n’avons pas pu mettre autant de fonctionnalités que souhaité dans notre application graphique.

Le choix de Tkinter comme framework graphique a joué en notre défaveur car il est trop simple et ne permet pas de réaliser une interface plus complexe. Ainsi, le plus clair de notre travail pour l’interface fut de trouver des manières de détourner les limitations de Tkinter plutôt que de se servir de ses fonctionnalités.

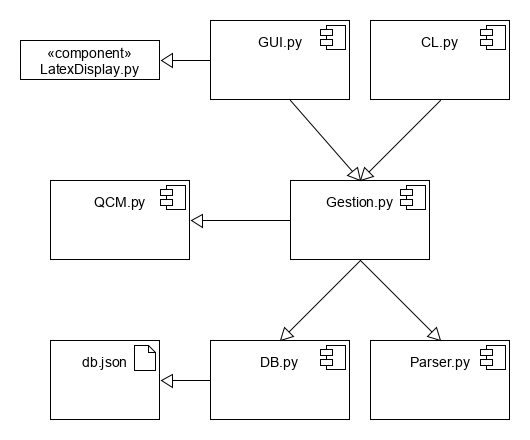
Ainsi, nous avons trouvé d’autres bibliothèques qui semblent plus puissantes au niveau de la création d’interface. On peut citer PyGame. Malgré son nom, cette bibliothèque permet de faire toute sorte d’interface et non pas que des jeux-vidéos. Son avantage est qu’elle permet une bien plus grande diversité de composants graphiques que Tkinter et leur agencement à l’écran est plus paramétrable. Ce qui permet bien plus simplement de réaliser une application sur mesure.

# Annexes

## Diagramme de cas d’utilisation



## Structure du programme



## Diagramme de classes

