Info-F-106 — Projet d'Informatique
Projet 2 – Gestion d'une base de données et d'un langage de requêtes

Anthony Cnudde Gwenaël Joret Tom Lenaerts Robin Petit
Loan Sens Cédric Simar

version du 28 février 2025

Introduction

Contexte

Le service technique de l'ULB aimerait modifier sa manière de stocker ses informations (que ce soient les cours, le personnel académique, les étudiant.e.s, les locaux, etc.) et a chargé le Département d'Informatique de déterminer un format et une interface qui soient (i) faciles à utiliser et (ii) adaptés à nos besoins.

L'équipe du projet d'année I a pris les devants et s'est occupée de définir le format *Unique Layout for a DataBase* (que nous abrégerons par ULDB). Malheureusement à force d'enseigner, nous avons perdu la pratique et notre maîtrise de Python est trop rouillée pour en faire une implémentation. Nous vous chargeons alors de vous occuper de la partie implémentation. Attention : votre programme doit scrupuleusement respecter le format défini car ce dernier a déjà été validé par le service technique.

Formellement

Dans ce projet, vous allez devoir implémenter une base de données (ou encore DB pour *DataBase*) simplifiée. Schématiquement, nous considérons qu'une base de données est un ensemble de *tables*. Chaque table est définie par une *signature* (c'est-à-dire les *champs* qu'elle contient, leur ordre ainsi que leur type) et une séquence *non-ordonnée* d'*entrées* qui ont chacune une unique valeur pour chacun des champs.

Le nombre d'entrées d'une table est appelée sa *taille*. Cette taille peut théoriquement être arbitrairement grande, mais nous imposerons ici une restriction (explicitée plus loin) qui impose une limite implicite.

En plus d'écrire un module Python qui vous permettra d'écrire et de lire dans une telle base de données, vous écrirez également une interface permettant ces opérations sans

passer explicitement par Python. Cette interface sera une *CLI* (*Command Line Interface*, ou encore interface en lignes de commande) similaire à l'interpréteur Python que vous connaissez bien.

Afin de vous aider dans le développement de ce projet, ce dernier est découpé en 5 parties (ou phases). Nous vous invitons bien entendu à réaliser ces phases dans l'ordre en prenant en compte les potentielles aides données. Attention cependant : les classes et méthodes imposées doivent impérativement être respectées. En effet ces dernières doivent obligatoirement exister et avoir la signature demandée. Vous ne pouvez donc pas ajouter de nouveaux paramètres, même avec une valeur par défaut. Vous avez cependant la liberté d'ajouter autant de fonctions, classes, méthodes, ou autre que bon vous semble (et nous vous y incitons afin d'obtenir un code le plus clair et lisible possible).

Les phases sont les suivantes :

1. gérer un fichier binaire (donc pouvoir lire ou écrire ce que nous voulons) ;
2. gérer les opérations de base sur une table (statique à ce stade : il n'est pas encore possible de l'altérer) ;
3. insérer et récupérer des entrées dans une table ;
4. modifier et supprimer des entrées déjà insérées ;
5. implémenter la CLI.

Consignes

Ce projet est à remettre sur l'UV dans le devoir associé pour le dimanche 30 mars à 21h59. Notez bien que cette deadline est stricte et qu'*aucun* retard ne sera accepté. Pensez donc à envoyer votre projet suffisamment à l'avance (la panne internet arrive toujours juste avant de remettre son code), vous pourrez de toute façon écraser la remise actuelle pour mettre la dernière version de votre projet.

Vous devez remettre un unique fichier **au format ZIP** dont le nom correspond à votre matricule (avec le triple 0 en préfixe) contenant l'ensemble des fichiers sources demandés *et rien d'autre*. En particulier, nous ne voulons pas d'un dossier `.venv`, de `test.py`, de l'énoncé en PDF, *etc.* De plus, ce ZIP ne doit contenir *aucun sous-dossier*, mais bien uniquement vos fichiers sources *directement à la racine*. En cas de problème, vous pouvez toujours construire l'archive avec la commande suivante sur un terminal Unix (où bien sûr vous devez remplacer `000408989` par votre propre matricule) :

```
$ zip 000408989.zip binary.py database.py uldb.py
```

Ce projet 2 sera évalué comme suit : 10 points sur 40 seront donnés automatiquement sur base du nombre de tests automatiques que votre code passe et les 30 points restants seront donnés par évaluation de votre code par un correcteur. Notez bien que l'évaluation sur les tests automatiques est *entièrement automatisée* mais suppose que vous respectez les consignes de remise *à la lettre*. Tout manquement à ces consignes résultera donc en une note nulle au projet 2, soyez donc particulièrement vigilant.e.s !

Vous pouvez à tout moment vérifier que votre code passe les tests grâce à la commande suivante (vous devrez potentiellement d'abord installer `pytest` à l'aide de la commande `python3 -m pip install pytest`) :

```
$ python3 -m pytest -vv test.py
```

Afin de nous assurer que vous commencez le projet suffisamment en avance et que vous n'attendez pas la veille de la remise pour le commencer, une remise intermédiaire est ajoutée le dimanche 16 mars à 21h59 (aux deux tiers du temps imparti). Vous devez rendre votre code sur l'UV dans le devoir associé (comme pour la remise finale) et les tests automatiques qui vous ont été fournis seront automatiquement évalués sur votre code. Si vous ne passez pas les 17 premiers tests, cette remise incomplète *bloquera* votre remise finale. Plus précisément, afin que votre code soit évalué lors de la remise du 30 mars, il faut **impérativement** que vous ayez passé la remise intermédiaire, sans quoi votre note (sur 40) comptera uniquement la partie sur 10 correspondant aux tests automatiques (vous ne pourrez donc pas avoir plus de 5 sur 40).

Tout comme pour le projet 1 au premier quadrimestre, la règle d'or d'INFO-F106 est toujours la même : ne nous envoyez pas de mails [1] car nous ne les lirons pas, vous êtes beaucoup trop nombreux.ses pour cela.

Cependant, une séance de questions/réponses sera organisée tous les vendredis de 10h15 à 11h45. Cette séance aura lieu sur la page Teams du cours et *sera enregistrée*. Avant de venir poser une question, assurez-vous que la réponse n'y a pas été apportée lors d'une séance de Q&A précédente. Notez également que cette séance *n'est pas une permanence* : si vous avez des questions, venez à 10h15 car lorsqu'il n'y a plus de questions, la séance prend fin. N'attendez donc pas 11h pour la joindre car elle risque de déjà être finie, et la prochaine occasion de poser vos questions sera le vendredi suivant.

Le dernier point à traiter ici est celui de ChatGPT/GH-Copilot/autre assistance de code. Bien qu'il ne vous est pas *strictement* interdit d'utiliser de tels outils, nous vous encourageons **très fortement** à les ignorer. En effet, l'intérêt de ce projet est de vous entrainer à coder en Python, pas d'entrainer le LLM de Microsoft ! Dès lors, nous imposons que toute fonction de votre code pour laquelle vous avez utilisé un tel outil doit être *explicitement* annotée comme tel dans son docstring. Tout manquement à cette règle sera considéré comme plagiat *manifeste* et vous exposera à des sanctions administratives pouvant être très graves.

Modules autorisés et version de Python. Votre projet doit être écrit en Python 3 et sera testé et évalué en Python 3.11. Vous avez le droit d'utiliser tous les modules standards de Python (et aucun autre) du moment que ceux-ci sont bien utilisés. En particulier : utiliser des modules de la mauvaise manière (ou lorsque ce n'est pas pertinent) nuit à la qualité de votre code, or nous vous évaluons précisément là-dessus.

1. Hormis de potentiels problèmes **d'ordre administratif uniquement**, en quel cas vous devez contacter le coordinateur du cours : Gwenaël Joret.

Partie 1 – Gestion de fichiers binaires

Remarque : Dans la suite de ce document, les bytes seront toujours donnés (i) en valeur hexadécimale ; (ii) sur deux caractères ; et (iii) en police `monospace`. Un unique byte sera précédé du préfixe `0x` pour expliciter l'encodage hexadécimal, mais dans un contexte d'une séquence de bytes, les bytes successifs n'auront pas de préfixe et seront séparés d'un espace pour des raisons de clarté. En particulier nous pouvons parler de la valeur 32 qui sera décrite par `0x20` mais la séquence de valeurs « 32 46 52 » sera décrite par `20 2e 34`.

Le format ULDB ne permet que d'enregistrer des chaînes de caractères (de taille arbitraire) et des entiers (de taille fixe).

Les entiers doivent tous être encodés en complément à 2 car ils peuvent aussi bien être négatifs que positifs et en *little endian* (ou encore *petit-boutiste*, c'est-à-dire que les bytes sont écrits dans l'ordre inverse de leur poids : le byte de poids le plus faible en premier et le byte de poids le plus fort en dernier). De manière générale (et sauf mention contraire explicite), les entiers seront systématiquement encodés sur 4 bytes. Certains entiers seront encodés sur un unique byte (dans un unique cas que nous verrons dans la phase suivante) et d'autres seront encodés sur deux bytes (uniquement dans le cas des chaînes de caractères).

Puisque la longueur des chaînes de caractères n'est pas fixée à l'avance, il faut un moyen de les identifier de manière unique. Pour cela, nous obligeons les chaînes de caractères à ne contenir qu'au plus 32 767 bytes, ainsi chaque chaîne de caractères commence par un entier (disons n) positif signé sur 2 bytes (donc entre 0 et $32\,767 = 2^{15} - 1$) suivi de n bytes correspondant à l'encodage UTF-8 de la chaîne de caractères.

Attention : les caractères non-ASCII (en particulier les caractères accentués) nécessitent plus d'un byte pour être encodés en UTF-8. Par exemple le caractère `é` correspond aux bytes `0xC3` et `0xA9` alors que le caractère `e` correspond à l'unique byte `0x65`. La chaîne de caractères « `eée` » sera donc encodée sur 6 bytes comme suit : `04 00 65 c3 a9 65`. En effet, comme décrit ci-dessus, il faut d'abord encoder le nombre de bytes nécessaires sur 2 bytes (et la valeur 2 s'écrit `02 00` en *little endian*), ensuite le byte `0x65` correspond à l'entier 101 (encodage ASCII de la lettre `e` minuscule), suivi des bytes `0xC3` et `0xA9` correspondant au caractère accentué `é`, et finalement le byte `0x65` correspondant à nouveau au caractère `e`.

Code à écrire

Dans un fichier `binary.py`, écrivez une classe `BinaryFile` permettant de lire et d'écrire des entiers et des chaînes de caractères dans un fichier binaire.[2] Cette classe doit définir les méthodes :

— `__init__(self, file: BinaryIO)`, le constructeur ;

2. Lisez bien la documentation pour vous faciliter la tâche.

- `goto(self, pos: int) -> None`, qui déplace l'endroit pointé dans le fichier à `pos` bytes après le début du fichier dans le cas où `pos` est positif et à `-pos` bytes avant la fin du fichier si `pos` est négatif ;
- `get_size(self) -> int`, qui renvoie la taille (en bytes) du fichier.

Notez bien que le constructeur prend en paramètre un fichier ouvert *en mode binaire* et qu'il n'est donc pas de la responsabilité de la classe `BinaryFile` d'ouvrir ou de fermer ce fichier.

Afin de pouvoir utiliser votre classe `BinaryFile` pour pouvoir écrire des données, écrivez les méthodes :

- `write_integer(self, n: int, size: int) -> int`, qui écrit l'entier `n` sur `size` bytes à l'endroit pointé actuellement par le fichier ;
- `write_integer_to(self, n: int, size: int, pos: int) -> int`, qui fait pareil mais écrit à la position `pos` (qui peut à nouveau être négative) ;
- `write_string(self, s: str) -> int`, qui écrit la chaîne de caractère `s` à l'endroit pointé actuellement par le fichier ;
- `write_string_to(self, s: str, pos: int) -> int`, qui fait pareil mais écrit à la position `pos` (qui peut à nouveau être négative).

Ces quatre méthodes doivent toutes renvoyer le nombre de bytes écrits dans le fichier. De plus, les deux méthodes `write_*_to` ne peuvent pas changer l'endroit pointé par le fichier alors que les deux méthodes restantes doivent le faire.

Finalement, afin de pouvoir lire les valeurs écrites, écrivez les méthodes :

- `read_integer(self, size: int) -> int`, qui renvoie l'entier encodé sur `size` bytes à partir de l'endroit pointé actuellement par le fichier ;
- `read_integer_from(self, size: int, pos: int) -> int`, qui fait pareil mais en lisant à la position `pos` (qui peut à nouveau être négative) ;
- `read_string(self) -> str`, qui renvoie la chaîne de caractères encodée à l'endroit pointé actuellement par le fichier ;
- `read_string_from(self, pos: int) -> str`, qui fait pareil mais en lisant à la position `pos` (qui peut à nouveau être négative).

Notez que les deux méthodes `read_*_from` ne peuvent pas changer l'endroit pointé par le fichier alors que les deux méthodes restantes doivent le faire.

Conseil

Vous pouvez utiliser le logiciel `hexdump` pour regarder facilement le contenu d'un fichier binaire. Vous pouvez avoir sa documentation complète avec la commande `$ man hexdump`. Le paramètre `-C` vous permet d'afficher les données *byte par byte* et en plus d'avoir l'interprétation de ces bytes en ASCII à côté pour pouvoir retrouver vos chaînes de caractères de manière plus rapide. Par exemple, si `string.bin` est un fichier contenant uniquement la chaîne de caractère `eée` (détaille plus haut) telle qu'encodée par `write_string`, alors la commande `$ hexdump -C string.bin` vous affichera en effet la séquence de bytes voulue :

```
$ hexdump -C string.bin
00000000  04 00 65 c3 a9 65                                 |..e..e|
00000006
```

Les informations données par `hexdump` peuvent être séparées en trois colonnes : tout d'abord la colonne de gauche contient un nombre (représenté en hexadécimal sur 8 chiffres) qui donne l'*offset* (le décalage) entre le début de la ligne et le début du fichier ; ensuite la région de droite (délimitée par des barres verticales) est l'interprétation en ASCII des bytes lus (les bytes ne correspondant à aucun caractère affichable donnent un point) ; et finalement la région du milieu qui contient au plus 16 bytes séparés par un espace.

Pour un exemple un peu plus grand, considérez un fichier `file.txt` contenant 32 fois le caractère `e` suivi des 26 lettres latines en minuscule, que vous pouvez par exemple créer de la manière suivante :

```
$ printf "eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee" > file.txt
$ printf "abcdefghijklmnopqrstuvwxyz" » file.txt
```

Dans ce cas, `hexdump` vous donnera l'output suivant :

```
$ hexdump -C file.txt
00000000  65 65 65 65 65 65 65 65  65 65 65 65 65 65 65 65  |eeeeeeeeeeeeeeee|
*
00000020  61 62 63 64 65 66 67 68  69 6a 6b 6c 6d 6e 6f 70  |abcdefghijklmnop|
00000030  71 72 73 74 75 76 77 78  79 7a                    |qrstuvwxyz|
0000003a
```

Vous pouvez remarquer que la deuxième ligne contient simplement une astérisque. Cela signifie que la ligne précédente est répétée jusqu'à la ligne suivante. Ici cela dit que les 16 premiers bytes (puisqu'une ligne correspond à 16 bytes) ne contiennent que des `e` (`0x65` en ASCII) mais que les 16 bytes suivants le sont aussi puisque la ligne suivante commence à l'adresse *hexadécimale* `0x00000020` (donc adresse 32 en décimal). Les bytes aux positions `0x00000020` jusque `0x00000039` contiennent alors les bytes `0x61` jusque `0x7a` qui correspondent aux lettres latines en minuscule.

Si vous ne voulez lire qu'un certain nombre de bytes dans le fichier, vous pouvez le spécifier avec le paramètre `-n <nb_bytes>`.

Partie 2 – Gestion basique des tables

Maintenant que vous pouvez interagir avec un fichier binaire pour y encoder ou lire ce que vous voulez, il est temps de s'attaquer à la DB et à son formatage. Une DB doit avoir un nom et peut avoir un nombre arbitraire de tables (potentiellement aucune). Nous représenterons cela par le fait que la DB de nom `<db>` est représentée par un dossier du même nom. Dans ce dossier, chaque table sera représentée dans son propre fichier. Dès lors la table `<table>` dans la DB `<db>` sera encodée dans le fichier `<db>/<table>.table` (remarquez l'extension `.table` permettant d'identifier les différentes tables). Tout fichier ayant une extension différente de `.table` est simplement ignoré par la base de données.

Chaque fichier de table est découpé en trois parties ordonnées : le *header* (également appelé *en-tête*) ; le *string buffer* (également appelé *mémoire de chaînes de caractères*) ; et l'*entry buffer* (également appelé *mémoire des entrées*).

Détaillons maintenant le format spécifique de chacune de ces régions.

2.1 Le header

Le *header* contient les informations générales relatives à la table en question. Plus précisément, il est structuré comme ceci :

1. Constante magique "ULDB" (4 bytes) ;
2. nombre de champs (4 bytes) ;
3. signature de la table (taille variable) ;
4. offset du *string buffer* (4 bytes) ;
5. première place disponible dans le string buffer (4 bytes) ;
6. offset de la première entrée (4 bytes) ;

La plupart des formats de fichier binaires utilisent un système de *constante magique* permettant de les reconnaître. C'est grâce à cela qu'un lecteur de PDF (tel que Evince) pourra afficher cet énoncé, même si vous changez son extension.

La signature de la table est encodée comme ceci : pour chaque champ, son type est encodé comme un entier sur 1 byte (1 pour un entier et 2 pour une chaîne de caractères) et ensuite son nom est encodé comme un string, c'est-à-dire d'abord un entier n sur 2 bytes représentant sa longueur et ensuite une suite de n bytes avec son encodage UTF-8 (*cf.* la section Gestion de fichiers binaires).

Exemple Considérons une table en version 1 avec la signature suivante :

1. MNEMONIQUE (entier) ;
2. NOM (string) ;
3. COORDINATEUR (string) ;
4. CREDITS (entier).

Offset	Bytes	Description
00	55 4c 44 42	ULDB
04	04 00 00 00	4 champs
08	01	INTEGER
09	0a 00	String sur 0x000a = 10 bytes
0b	4d 4e 45 4d 4f 4e 49 51 55 45	‘MNEMONIQUE‘
15	02	STRING
16	03 00	String sur 0x0003 = 3 bytes
18	4e 4f 4d	‘NOM‘
1b	02	STRING
1c	0c 00	String sur 0x000c = 12 bytes
1e	43 4f 4f 52 44 49 4e 41 54 45 55 52	‘COORDINATEUR‘
2a	01	INTEGER
2b	07 00	String sur 0x0007 = 7 bytes
2d	43 52 45 44 49 54 53	‘CREDITS‘
34	40 00 00 00	Pointeur vers string buffer
38	40 00 00 00	1e place disponible
3c	50 00 00 00	Pointeur vers entry buffer
40	00 00 00 00 ... 00 00 00 00	String buffer (16 bytes)

De manière équivalente, si cette table s’appelle `cours` et fait partie de la DB `programme`, alors le contenu du fichier `programme/cours.table` peut être affiché avec `hexdump` et donne :

```
$ hexdump -C -n 80 programme/cours.table
00000000  55 4c 44 42 04 00 00 00  01 0a 00 4d 4e 45 4d 4f  |ULDB.......MNEMO|
00000010  4e 49 51 55 45 02 03 00  4e 4f 4d 02 0c 00 43 4f  |NIQUE...NOM...CO|
00000020  4f 52 44 49 4e 41 54 45  55 52 01 07 00 43 52 45  |ORDINATEUR...CRE|
00000030  44 49 54 53 40 00 00 00  40 00 00 00 50 00 00 00  |DITS@...@...P...|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050
```

2.2 Le string buffer

Le string buffer est la région du fichier dans laquelle toutes les chaînes de caractères vont devoir être écrites. Leur format est le même que celui décrit dans la phase Gestion de fichiers binaires : un entier n sur 2 bytes suivi de n bytes correspondant à l’encodage UTF-8 du string.

La taille de ce buffer doit obligatoirement être une puissance de 2, et vaut 16 à l’initialisation. Dès qu’une nouvelle chaîne de caractères doit être encodée, elle doit être écrite dans ce buffer. Si le buffer ne contient pas assez de place, alors il faut l’agrandir et donc réécrire le fichier associé à la table. Lorsque ce buffer est agrandi, il faut garantir que sa taille reste bien une puissance de 2 mais qu’il soit suffisamment grand pour pouvoir contenir la nouvelle chaîne de caractères.

Par exemple, après avoir inséré les entrées :
- MNEMONIQUE = 101
- NOM = "Programmation"
- COORDINATEUR = "Thierry Massart"
- CREDITS = 10

et
- MNEMONIQUE = 102
- NOM = "Fonctionnement des ordinateurs"
- COORDINATEUR = "Gilles Geeraerts"
- CREDITS = 5

le string buffer aura une taille de 128 bytes et sera donné par :

```
$ hexdump -C -s 64 -n 128 programme/cours.table
00000040  0d 00 50 72 6f 67 72 61  6d 6d 61 74 69 6f 6e 0f  |..Programmation.|
00000050  00 54 68 69 65 72 72 79  20 4d 61 73 73 61 72 74  |.Thierry Massart|
00000060  1e 00 46 6f 6e 63 74 69  6f 6e 6e 65 6d 65 6e 74  |..Fonctionnement|
00000070  20 64 65 73 20 6f 72 64  69 6e 61 74 65 75 72 73  | des ordinateurs|
00000080  10 00 47 69 6c 6c 65 73  20 47 65 65 72 61 65 72  |..Gilles Geeraer|
00000090  74 73 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |ts..............|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000000c0
```

De plus, le header aura également été modifié puisque la première place disponible dans le string buffer est maintenant `0x00000092` et l'entry buffer commence à la position `0x000000c0`.

2.3 L'entry buffer

L'entry buffer est la région du fichier dans laquelle toutes les entrées doivent être encodées. Cette région commence également par un *mini-header* et contient ensuite une *liste doublement chaînée* des entrées.

Ce mini-header sur 20 bytes contient :

1. le dernier ID utilisé (0 si la aucune insertion n'a été réalisée) (4 bytes) ;
2. le nombre d'entrées présentes dans cette table (4 bytes) ;
3. un pointeur vers la première entrée (4 bytes) ;
4. un pointeur vers la dernière entrée (4 bytes) ;
5. un pointeur sur 4 bytes réservé (il servira lors de la phase Modification et suppression des entrées).

Si l'entry buffer est vide, alors ces trois pointeurs doivent valoir -1 (donc `ff ff ff ff`). En particulier, à la création de la table, ce header est :

```
00 00 00 00 00 00 00 00  ff ff ff ff ff ff ff ff
ff ff ff ff
```

Ensuite, la liste chaînée contient des éléments formatés comme suit :

1. l'identifiant (unique) de l'entrée (4 bytes) ;
2. les champs de l'entrée *dans l'ordre de la signature* donc :
   — si le *i*ème champ est un entier, cet entier est encodé sur 4 bytes ;
   — si le *i*ème champ est un string, alors il y a un pointeur sur 4 bytes vers le string dans le string buffer ;
3. un pointeur vers l'élément précédent (4 bytes) ;
4. un pointeur vers l'élément suivant (4 bytes).

Notez que s'il n'y a aucune entrée dans la table, alors le pointeur dans le header doit être à -1 (*i.e.* `0xFFFFFFFF`). De la même manière, le pointeur *précédent* du premier élément de la liste chaînée doit valoir -1 et le pointeur *suivant* du dernier élément doit également valoir -1.

Par exemple, après avoir inséré les mêmes entrées que ci-dessus :
— MNEMONIQUE = 101
— NOM = "Programmation"
— COORDINATEUR = "Thierry Massart"
— CREDITS = 10

et

— MNEMONIQUE = 102
— NOM = "Fonctionnement des ordinateurs"
— COORDINATEUR = "Gilles Geeraerts"
— CREDITS = 5

L'entry buffer est donné par

```
000000c0  02 00 00 00 02 00 00 00  d4 00 00 00 f0 00 00 00  |................|
000000d0  ff ff ff ff 01 00 00 00  65 00 00 00 40 00 00 00  |........e...@...|
000000e0  4f 00 00 00 0a 00 00 00  ff ff ff ff f0 00 00 00  |O...............|
000000f0  02 00 00 00 66 00 00 00  60 00 00 00 80 00 00 00  |....f...`.......|
00000100  05 00 00 00 d4 00 00 00  ff ff ff ff              |............|
0000010c
```

De manière plus détaillée, souvenez-vous que la signature de la table est (MATRICULE, NOM, COORDINATEUR, CREDITS), dès lors on a :

Offset	Bytes	Description
00c0	02 00 00 00	Dernier ID utilisé est 2
00c4	02 00 00 00	2 éléments dans la table
00c8	d0 00 00 00	Première entrée à l'adresse 0x000000d0
00cc	ec 00 00 00	Dernière entrée à l'adresse 0x000000ec
00d0	ff ff ff ff	Servira plus tard
00d4	01 00 00 00	ID de la première entrée
00d8	65 00 00 00	MNEMONIQUE = 101 = 0x65
00dc	40 00 00 00	Adresse du NOM (*cf.* string buffer ci-dessus)
00e0	4f 00 00 00	Adresse du COORDINATEUR (idem)
00e4	0a 00 00 00	CREDITS = 10 = 0x0a
00e8	ff ff ff ff	Pas d'entrée avant celle-ci
00ec	f0 00 00 00	Adresse de l'adresse suivante
00f0	02 00 00 00	ID de la deuxième entrée
00f4	66 00 00 00	MNEMONIQUE = 102 = 0x66
00f8	60 00 00 00	Adresse du NOM (*cf.* string buffer ci-dessus)
00fc	80 00 00 00	Adresse du COORDINATEUR (idem)
0100	05 00 00 00	CREDITS = 5 = 0x05
0104	d4 00 00 00	Adresse de l'entrée précédente
0108	ff ff ff ff	Pas d'entrée après celle-ci

Récapitulatif

Dans l'exemple donné ici (la table `cours` dans laquelle on insère les cours INFO-F101 et INFO-F102), le fichier final est donné par :

```
$ hexdump -C programme/cours.table
00000000  55 4c 44 42 04 00 00 00  01 0a 00 4d 4e 45 4d 4f  |ULDB.......MNEMO|
00000010  4e 49 51 55 45 02 03 00  4e 4f 4d 02 0c 00 43 4f  |NIQUE...NOM...CO|
00000020  4f 52 44 49 4e 41 54 45  55 52 01 07 00 43 52 45  |ORDINATEUR...CRE|
00000030  44 49 54 53 40 00 00 00  92 00 00 00 c0 00 00 00  |DITS@...........|
00000040  0d 00 50 72 6f 67 72 61  6d 6d 61 74 69 6f 6e 0f  |..Programmation.|
00000050  00 54 68 69 65 72 72 79  20 4d 61 73 73 61 72 74  |.Thierry Massart|
00000060  1e 00 46 6f 6e 63 74 69  6f 6e 6e 65 6d 65 6e 74  |..Fonctionnement|
00000070  20 64 65 73 20 6f 72 64  69 6e 61 74 65 75 72 73  | des ordinateurs|
00000080  10 00 47 69 6c 6c 65 73  20 47 65 65 72 61 65 72  |..Gilles Geeraer|
00000090  74 73 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |ts..............|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000000c0  02 00 00 00 02 00 00 00  d4 00 00 00 f0 00 00 00  |................|
000000d0  ff ff ff ff 01 00 00 00  65 00 00 00 40 00 00 00  |........e...@...|
000000e0  4f 00 00 00 0a 00 00 00  ff ff ff ff f0 00 00 00  |O...............|
000000f0  02 00 00 00 66 00 00 00  60 00 00 00 80 00 00 00  |....f...`.......|
00000100  05 00 00 00 d4 00 00 00  ff ff ff ff              |............|
```

`0000010c`

Code à écrire

Dans un fichier `database.py`, écrivez une *énumération*[3] entière `FieldType` définissant les constantes `INTEGER` et `STRING`.

Considérons alors l'alias de type `TableSignature = list[tuple[str, FieldType]]`. Ce type sert à représenter la signature d'une table qui n'est donc autre qu'une liste de paires (`nom`, `type`) où `nom` est un `str` et `type` est un `FieldType`.

Dans ce même fichier, écrivez une classe `Database` avec les méthodes :

- `__init__(self, name: str)`, le constructeur ;
- `list_tables(self) -> list[str]`, qui renvoie une liste avec le nom de toutes les tables existant dans cette DB ;
- `create_table(self, table_name: str, *fields: TableSignature) -> None`, qui crée une nouvelle table de nom `table_name` et de signature `fields` ;
- `delete_table(self, table_name: str) -> None`, qui supprime la table de nom `table_name` ;
- `get_table_signature(self, table_name: str) -> TableSignature`, qui renvoie la signature de la table de nom `table_name`.

Ces trois dernières méthodes doivent lancer une exception de type `ValueError` si l'opération est impossible (soit parce que la table existe déjà dans le cas de la création, soit parce qu'elle n'existe pas dans le cas de la suppression ou de la récupération de la signature).

Voici un exemple d'utilisation de ces méthodes :

```
db = Database('programme')
db.create_table(
    'cours',
    ('MNEMONIQUE', FieldType.INTEGER),
    ('NOM', FieldType.STRING),
    ('COORDINATEUR', FieldType.STRING),
    ('CREDITS', FieldType.INTEGER)
)
print(db.list_tables())  # doit afficher ['cours']
db.delete_table('cours')
print(db.list_tables())  # doit afficher []
db.delete_table('cours')  # doit lancer une exception
```

3. Lisez bien la documentation du module `enum`.

Partie 3 – Insertion et récupération d'entrées dans une table

Il est à présent temps de pouvoir faire des requêtes à la DB : nous voulons pouvoir insérer de nouvelles entrées, récupérer des entrées existante et même récupérer des champs spécifiques d'entrées existantes.

Il est assez évident que de nouvelles entrées doivent pouvoir être créées, sinon la DB restera vide à jamais, ce qui ne servirait à rien. Les échanges d'entrées (tant à ajouter qu'à récupérer) avec la DB se feront par l'intermédiaire de dictionnaires dont les paires (clef, valeur) correspondent aux paires (nom du champ, valeur du champ).

Il est à noter que l'identifiant (unique) de chaque entrée ne peut pas être décidé lors de l'insertion. En effet, permettre de spécifier un ID lors de l'insertion d'une nouvelle entrée risquerait d'aller à l'encontre de la règle d'unicité (ou alors il faudrait vérifier la table en entier à chaque insertion pour garantir que cet identifiant n'a jamais été utilisé, ce qui serait beaucoup trop coûteux en temps). Heureusement, le fait de sauver explicitement dans le fichier l'ID de la dernière entrée à avoir été ajoutée, permet de s'assurer d'une distribution unique des IDs en temps constant.

Bien que l'ID ne puisse être spécifié lors d'une insertion, il doit être possible de récupérer l'ID d'une entrée existante.

Cependant, l'entièreté des champs d'une entrée ne sont pas toujours nécessaires. Parfois, nous ne voulons récupérer qu'un sous-ensemble des champs et dans de telles situations, devoir lire l'entrée en entier pour en ignorer une majorité peut être très coûteux. En effet, imaginons la situation suivante : une des tables de la DB sert à stocker tous les fichiers HTML de la page web de chaque département de l'ULB. Chaque entrée correspond donc à un département, et les différents champs (disons une centaine, et tous des chaînes de caractères) sont le contenu de petits fichiers (disons $\sim$ 10 Ko chacun) HTML utilisés dans le layout de la page web du département en question. Nous pouvons potentiellement vouloir le contenu d'un unique fichier (par exemple `index.html`) pour chaque département mais sans pour autant devoir lire $\sim$ 1 Mo de donnée à chaque fois. Dans une telle situation, il est clair que pouvoir demander un sous-ensemble de champs est fondamental pour une bonne utilisation.

Nous utiliserons le vocabulaire suivant :

- un *ajout* d'entrée (*add* en anglais) correspond à l'insertion d'une entrée dans une table ;
- une *demande* d'entrée (*get* en anglais) correspond à la récupération d'une entrée dans une table ;
- une *sélection* de champs d'entrées (*select* en anglais) correspond à la récupération de certains champs d'une entrée dans une table.

Lors d'une demande ou d'une sélection, nous voulons rarement récupérer l'entièreté de la table, mais habituellement seules les entrées remplissant certains critères sont demandées.

Par exemple si nous considérons la même DB que proposée ci-dessus (qui encode les cours donnés en BA-INFO), nous pouvons vouloir récupérer les cours à seulement 10 crédits (et donc ignorer les cours à 5 crédits). Nous pouvons également chercher à récupérer *le nom* (donc lors d'une sélection) de tous les cours donnés par un coordinateur donné.

Notez que le nom du champ associé à l'identifiant est `id` (en minuscules).

Code à écrire

Puisque tous les champs sont soit des chaînes de caractères, soit des entiers, considérons l'alias de type `Field = str | int`. Comme mentionné plus haut, les communications avec la classe `Database` se feront via des `dict`, donc considérons également l'alias de type `Entry = dict[str, Field]`.

Complétez votre classe `Database` en lui ajoutant les méthodes suivantes :

1. ```
   def add_entry(self, table_name: str, entry: Entry) -> None
   ```

   qui ajoute l'entrée `entry` à la table de nom `table_name`.

2. ```
   def get_complete_table(self, table_name: str) -> list[Entry]
   ```

   qui renvoie *toutes* les entrées de la table de nom `table_name` dans une liste.

3. ```
   def get_entry(self, table_name: str, field_name: str,
                 field_value: Field) -> Entry | None
   ```

   qui renvoie une entrée (quelconque) de la table de nom `table_name` dont le champ `field_name` contient la valeur `field_value` si une telle entrée existe, et qui renvoie `None` sinon.

4. ```
   def get_entries(self, table_name: str, field_name: str,
                   field_value: Field) -> list[Entry]
   ```

   qui renvoie *toutes* les entrées de la table de nom `table_name` dont le champ `field_name` contient la valeur `field_name`.

5. ```
   def select_entry(self, table_name: str, fields: tuple[str],
                    field_name: str, field_value: Field) \
           -> Field | tuple[Field]
   ```

   qui effectue une *sélection* des champs demandés sur une entrée de la table de nom `table_name` dont le champ `field_name` contient la valeur `field_value` et renvoie ces champs uniquement. Si un unique champ est demandé, la fonction ne doit pas renvoyer un tuple de taille 1, mais bien uniquement la valeur du champ demandé.
   ```

Sinon, le tuple renvoyé doit contenir les valeurs des champs dans le même ordre que celui demandé par le paramètre `fields`.

6.
```
def select_entries(self, table: str, fields: tuple[str],
                   field_name: str, field_value: Field) \
        -> list[Field | tuple[Field]]
```

qui se comporte comme `select_entry` mais qui renvoie les champs demandés de *toutes* les entrées de la table de nom `table_name` satisfaisant la condition.

7.
```
def get_table_size(self, table_name: str) -> int
```

qui renvoie le nombre d'entrées dans la table de nom `table_name`.

Exemple

À nouveau, considérons une DB avec une table `cours` (comme dans l'exemple de la phase précédente). Voici un exemple d'utilisation des méthodes ajoutées :

```
# db est initialisée et sa table est créée
# ...
db.add_entry('cours', {
    'MNEMONIQUE': 101, 'NOM': 'Progra',
    'CCORDINATEUR': 'T. Massart', 'CREDITS': 10
})  # ajout de Progra
db.add_entry('cours', {
    'MNEMONIQUE': 102, 'NOM': 'FDO',
    'COORDINATEUR': 'G. Geeraerts', 'CREDITS': 5
})  # ajout de FDO
db.add_entry('cours', {
    'MNEMONIQUE': 103, 'NOM': 'Algo 1',
    'COORDINATEUR': 'O. Markowitch', 'CREDITS': 10
})  # ajout d'Algo 1
# doit afficher les 3 cours
print(db.get_complete_table('cours'))
# doit afficher Progra ou Algo 1
print(db.get_entry('cours', 'CREDITS', 10))
# doit afficher Progra et Ago 1
print(db.get_entries('cours', 'CREDITS', 10))
# doit afficher [101, 103]
print(db.select_entries('cours', ('MNEMONIQUE',), 'CREDITS', 10))
# doit afficher [(101, 'Progra'), (103, 'Algo 1')]
print(db.select_entries('cours', ('MNEMONIQUE', 'NOM'), 'CREDITS', 10))
# doit afficher [('Progra', 101), ('Algo 1', 103)]
print(db.select_entries('cours', ('NOM', 'MNEMONIQUE'), 'CREDITS', 10))
# doit afficher 2
print(db.select_entry('cours', ('id',), 'NOM', 'FDO'))
# doit afficher 3
print(db.get_table_size('cours'))
```

Partie 4 – Modification et suppression des entrées

Il arrive que des cours à l'ULB soient modifiés, soient découpés en plusieurs plus petits cours, voire soient regroupés en un seul. Pour pouvoir faire cela sans perdre le reste des données, il faut pouvoir faire des modifications et des suppressions localisées dans une table.

De plus, ces deux opérations doivent être disponibles. En effet, on pourrait penser qu'une fois la possibilité de supprimer une entrée, la modification n'est pas nécessaire car il est toujours possible d'effectuer un `get_entry` suivi d'une suppression de l'entrée, de modifier le dictionnaire reçu, et finalement d'insérer l'entrée modifiée. Cependant en procédant de la sorte, l'identifiant n'est pas préservé ! En effet, lors de l'insertion de la nouvelle entrée modifiée, celle-ci recevra un nouvel identifiant, et par unicité de l'identifiant, ce dernier ne pourra pas déjà avoir été utilisé pour une autre entrée de cette table.

Dès lors, en plus d'une méthode de suppression, il va falloir une méthode de modification.

La modification d'entrée ne peut se faire qu'un seul champ à la fois, mais doit également se faire de manière conditionnelle : nous ne voulons pas mettre la même valeur à toutes les entrées de la table mais seulement à celles qui satisfont une certaine condition. Conceptuellement, une requête de mise à jour ressemble à *dans la table T, mettre le champ F à la valeur V pour toutes les entrées qui satisfont la condition C*. Comme pour les demandes et les sélections, une condition ne peut ici concerner qu'un unique champ. Notez bien que lors de la modification d'une chaîne de caractères, si possible, il faut toujours chercher à réutiliser l'espace disponible dans le string buffer avant de prendre un nouvel espace. En effet replacer une chaîne de caractères de 32 bytes par une chaîne de 5 bytes de devra *jamais* causer de réallocation du string buffer, indépendamment de la place disponible.

La suppression peut être implémentée de manière triviale mais particulièrement inefficace : pour supprimer une entrée de la table, il suffit de trouver sa position, et de recréer la table sans cette entrée. Cependant on voit ici que la suppression d'un unique élément nécessitera de toute façon de réécrire l'intégralité de la table. Il faut donc faire plus intelligent que cela.

Puisque les entrées de la table sont stockées sous la forme d'une liste chaînée dans le fichier, il est aisé de retirer une entrée de cette liste, même sans la retirer explicitement du fichier : il suffit pour cela de s'assurer qu'en suivant les pointeurs allant d'entrée en entrée, on ne passe jamais par l'entrée que l'on souhaite supprimer. On appelle *entrée effective* une entrée qui a été insérée dans la table mais qui n'a pas été supprimée ; et *entrée reliquat* une entrée qui a été insérée mais qui a déjà été supprimée.

Cependant on peut voir un certain désavantage à cette approche : si le nombre d'entrées

dans la table à tout moment reste petit mais que plein d'insertions puis de suppressions sont effectuées, bien que le nombre d'entrées *effectives* de la table (donc *réellement* présentes, pas des reliquats d'entrées supprimées) reste faible, la taille du fichier va continuer de croitre encore et encore.

Afin d'éviter ce problème, nous voulons pouvoir réutiliser les zones du fichier occupées par des entrées reliquats. Pour cela, il faut qu'en plus de la liste chaînée des entrées effectives, votre table maintienne une liste chaînée des entrées reliquats (et bien sûr, ces deux listes chaînées doivent être disjointes puisqu'une entrée est soit effective, soit reliquat, mais jamais les deux à la fois). C'est à ça que va servir le dernier pointeur du mini-header de l'entry buffer (*cf.* la phase Gestion basique des tables). Ce pointeur doit contenir l'adresse d'une extrémité de la chaîne des entrées reliquats. Ainsi, lors de l'ajout d'une entrée dans la table, il faut choisir où l'insérer : avant de choisir de placer cette entrée à la fin du fichier, il faut vérifier s'il existe une entrée reliquat, et si oui placer la nouvelle entrée à la place de l'entrée reliquat trouvée. Notez bien que la recherche d'un tel emplacement doit se faire en temps constant.

En procédant de la sorte, une suite d'ajout, suppression, ajout, suppression, *etc.* ne devrait pas faire augmenter la taille du fichier de la table (sauf dans le cas où les insertions demandent de plus en plus de place dans le string buffer).

Remarquons tout de même qu'il peut rester un dernier problème concernant la taille du fichier. En effet, si une opération de suppression retire 75% des entrées de la table, la proportion d'entrées effectives dans la table deviendra très basse. Afin de toujours garder un fichier de taille raisonnable par rapport au nombre d'entrées contenues, il faut qu'après chaque opération de suppression vous vérifiez que cette proportion est (strictement) supérieure à 50%, sans quoi il vous faudra réencoder la table en ignorant les entrées reliquats.

Code à écrire

Ajoutez les méthodes suivantes à votre classe `Database` :

1. ```
   def update_entries(self, table_str: str,
                      cond_name: str, cond_value: Field,
                      update_name: str, update_value: Field) -> bool
   ```

   qui va remplacer le champ `update_name` par la valeur `update_value` pour toutes les entrées de la table de nom `table_name` dont le champ `cond_name` contient la valeur `cond_value`. La fonction doit renvoyer `True` si au moins une entrée a été modifiée et `False` sinon.
2. ```
   def delete_entries(self, table_name: str,
                      field_name: str, field_value: Field) -> bool
   ```

qui va supprimer de la table de nom `table_name` toutes les entrées dont le champ `field_name` contient la valeur `field_value`. La fonction doit renvoyer `True` si au moins une entrée a été supprimée et `False` sinon.

Vous devez également adapter votre méthode `add_entry` afin de correspondre à la description ci-dessus.

Remarque : avant de modifier une entrée, assurez-vous bien d'avoir la place dans le string buffer pour pouvoir y écrire la potentielle nouvelle chaîne de caractères.

Exemple

Toujours avec la même table que dans les exemples précédents, voici un exemple de code utilisant ces méthodes :

```
# db est initialisée et sa table est créée
# ...
db.add_entry('cours', {
    'MNEMONIQUE': 101, 'NOM': 'Progra',
    'CCORDINATEUR': 'T. Massart', 'CREDITS': 10
})  # ajout de Progra
db.add_entry('cours', {
    'MNEMONIQUE': 102, 'NOM': 'FDO',
    'COORDINATEUR': 'G. Geeraerts', 'CREDITS': 5
})  # ajout de FDO
# doit afficher True
print(db.update_entries('cours', 'CREDITS', 5, 'NOM', 'Programmation'))
# doit afficher False
print(db.update_entries('cours', 'MNEMONIQUE', 205, 'NOM', 'CFN'))
# doit afficher False
print(db.delete_entries('cours', 'MNEMONIQUE', 205, 'NOM', 'CFN'))
# doit afficher Programmation
print(db.select_entry('cours', ('NOM',), 'MNEMONIQUE', 101))
# doit afficher True
print(db.delete_entries('cours', 'MNEMONIQUE', 101))
# doit afficher []
print(db.select_entries('cours', ('NOM',), 'MNEMONIQUE', 101))
```

Partie 5 – Langage de requêtes

Maintenant que votre gestionnaire de DB est fini, il faut que tout le monde puisse l'utiliser, et comme mentionné plus haut, nous avons perdu l'habitude de programmer en Python depuis le temps. Dans le standard ULDB, nous avons également défini un langage simplifié pour pouvoir effectuer des requêtes basiques.

Vous allez devoir implémenter un programme (nommé `uldb`) interprétant ce langage. Tel Python, votre programme doit pouvoir fonctionner dans deux modes différents : en mode *script* et en mode *interactif*. Pour utiliser le programme en mode script, il faudra lui donner en paramètre le chemin vers un fichier contenant un script compatible ULDB. Pour utiliser le mode interactif, il faudra simplement lancer votre programme avec Python sans aucun paramètre.

5.1 Description du langage ULDB

Notez que le langage ULDB est très simple et a donc une syntaxe très rigide. En particulier, les paramètres des fonctions doivent être séparés par des virgules, mais ajouter des espaces après les virgules rend l'instruction invalide. Votre programme n'a pas besoin de pouvoir comprendre de telles instructions, mais s'il vous en prend l'envie, n'hésitez pas à écrire votre interpréteur de manière aussi robuste que possible.

On dira qu'une entrée satisfait la condition `field_name=field_value` lorsque son champ `field_name` contient la valeur `field_value`.

Voici la liste des instructions possibles dans ce langage, ainsi que leur documentation :

1. `open(db_name)`
   Ouvre la DB de nom `db_name`. Attention seule une DB peut être ouverte par session : une fois qu'une DB est ouverte, aucune autre ne peut être ouverte. Si vous tentez d'ouvrir une deuxième DB, le programme `uldb` doit afficher un message d'erreur *mais continuer à s'exécuter*.
   De plus, il faut qu'une DB soit ouverte pour chacune des instructions suivantes, sans quoi le programme `uldb` doit afficher un message d'erreur mais continuer à s'exécuter.
2. `create_table(table_name,name1=type1,name2=type2,...)`
   Crée une table de nom `table_name` dans la DB ouverte et dont la signature est donnée par les paramètres suivants. Les valeurs possibles pour `type1`, `type2`, *etc.* sont `INTEGER` et `STRING`. Le nombre de paramètres à la fonction `create_table` peut être arbitrairement grand, tant que ces derniers encodent une signature valide.
3. `delete_table(table_name)`
   Supprime la table de nom `table_name` dans la DB ouverte. Si aucune table de ce nom existe, le programme `uldb` doit afficher un message d'erreur *mais continuer à s'exécuter*.

4. `list_tables()`
   Affiche le nom des tables existantes dans la DB ouverte. Il doit y avoir un unique nom de table par ligne.
5. `insert_to(table_name,name1=value1,name2=value2,...)`
   Ajoute l'entrée définie par les paramètres suivants dans la table de nom `table_name` de la DB ouverte. La valeur d'un entier doit être écrite avec les caractères `0` jusque `9`. La valeur d'une chaîne de caractères doit être écrite entre guillemets doubles (et pas entre guillemets simples) et tous les caractères UTF-8 sont acceptés.
   Les champs peuvent être donnés dans n'importe quel ordre (pas uniquement celui donné lors de la création de la table).
   Si le nombre de champs, le nom des champs, ou le type des champs ne correspond pas à ce qui est attendu par la signature de la table, le programme `uldb` doit afficher un message d'erreur *mais doit continuer à s'exécuter*.
6. `from_if_get(table_name,cond_name=cond_value,name1,name2,...)`
   Affiche le résultat de la sélection des champs `name1`, `name2`, *etc.* des entrées de la table de nom `table_name` satisfaisant la condition `cond_name=cond_value`, *i.e.* dont le champ `cond_name` contient la valeur `cond_value`. L'affichage doit se faire une entrée par ligne.
   Le même champ peut être donné plusieurs fois dans `name1`, `name2`, *etc.*. L'instruction `from_if_get(T,F=V,F,F,F)` est donc valide.
   Si un unique champ est donné après la condition et que ce champ est `*` (une astérisque), alors il faut récupérer (et donc afficher) *tous les champs* des entrées (hormis l'identifiant), dans l'ordre donné lors de la création de la table (donc dans l'ordre de la signature de la table).
   S'il y a une erreur de type ou de nom de champ, le programme `uldb` doit afficher un message d'erreur *mais continuer à s'exécuter*.
7. `from_delete_where(table_name,cond_name=cond_value)`
   Supprime les entrées de la table de nom `table_name` satisfaisant la condition `cond_name=cond_value`.
   Si le champ `cond_name` n'existe pas ou si `cond_value` a un type qui ne correspond pas à celui attendu, le programme `uldb` doit afficher un message d'erreur *mais continuer à s'exécuter*.
8. `from_update_where(table_name,cond_name=cond_value,name=new_value)`
   Modifie les entrées de la table de nom `table_name` qui satisfont la condition `cond_name=cond_value` en mettant le champ `name` à la valeur `new_value`.
   Si le champ `cond_name` ou `name` n'existe pas ou si `cond_value` ou `new_value` a un type qui ne correspond pas à celui attendu, le programme `uldb` doit afficher un message d'erreur *mais continuer à s'exécuter*.

Il reste une dernière instruction qui n'existe qu'en mode interactif (et pas en mode script) : les instructions `quit` et `q` permettent d'arrêter le programme `uldb`.

Voici un exemple d'exécution du programme `uldb` :

```
uldb:: open(programme)
uldb:: delete_table(cours)
uldb:: create_table(cours,MNEM=INTEGER,NOM=STRING,COORD=STRING,CRED=INTEGER)
uldb:: insert_to(cours,MNEM=101,NOM="Progra",CRED=10,COORD="T. Massart")
uldb:: insert_to(cours,MNEM=102,NOM="FDO",CRED=5,COORD="G. Geeraerts")
uldb:: insert_to(cours,MNEM=103,NOM="Algo I",CRED=10,COORD="O. Markowitch")
uldb:: insert_to(cours,MNEM=105,NOM="LDP I",CRED=5,COORD="C. Petit")
uldb:: insert_to(cours,MNEM=106,CRED=5,NOM="Projet I",COORD="G. Joret")
uldb:: list_tables()
cours
uldb:: from_if_get(cours,CRED=5,MNEM)
102
105
106
uldb:: from_if_get(cours,CRED=5,id,MNEM)
(2, 102)
(4, 105)
(5, 106)
uldb:: from_if_get(cours,CRED=5,*)
(102, 'FDO', 'G. Geeraerts', 5)
(105, 'LDP I', 'C. Petit', 5)
(106, 'Projet I', 'G. Joret', 5)
uldb:: from_if_get(cours,CRED=10,MNEM)
101
103
uldb:: from_delete_where(cours,MNEM=103)
uldb:: from_if_get(cours,CRED=10,MNEM)
101
uldb:: from_update_where(cours,id=1,CRED=0)
uldb:: from_if_get(cours,CRED=0,MNEM)
101
uldb:: from_update_where(cours,id=1,CRED=10)
uldb:: from_if_get(cours,CRED=0,MNEM)
uldb:: from_if_get(cours,CRED=10,MNEM)
101
uldb:: quit
```

Code à écrire

Écrivez votre interpréteur dans un fichier `uldb.py`. Les détails d'implémentation sont laissés libres. Faites tout de même attention à avoir un code clair, propre, lisible et documenté.

Bonus – Joindre les tables d'une DB

Attention. Cette dernière phase de *bonus* suppose que vous avez déjà réalisé l'entièreté du projet. Commencez-la uniquement lorsque vous avez terminé le projet et si vous vous désirez le poursuivre. Ne pas réaliser ce bonus ne peut pas pénaliser votre note, car comme son nom l'indique, elle ne peut vous donner que des points *bonus*.

Lors du *design* d'une DB relationnelle (comme dans ce projet), il est important de découper proprement les données en différentes tables, mais il est également nécessaire de pouvoir faire des opérations jointes sur les différentes tables. Cela permet de représenter des types de relation dites *many-to-one*, *one-to-many* ou encore *many-to-many*, mais permet également de diminuer la quantité d'information à sauver à chaque fois.

En effet, prenons l'exemple suivant : nous avons entièrement rempli la table `cours` de la DB `programme` et tous les cours de toutes les facultés y sont insérés. Nous pouvons garder un unique champ `COORDINATEUR` par cours car il ne peut y avoir qu'une seule personne à la charge d'un cours donné. Cependant, il peut y avoir un nombre arbitraire d'assistant.e.s (*e.g.* ce présent projet avec 4 assistants). Si on voulait pouvoir encoder, dans la table `cours` l'ensemble des assistant.e.s de chaque cours, il faudrait regarder à travers toute l'Université le nombre maximum d'assistant.e.s pour un unique cours afin de savoir combien de champs prévoir dans la table. Malgré tout, avec l'augmentation constante du nombre d'étudiant.e.s dans l'enseignement supérieur, nous n'avons aucune garantie que ce nombre maximum ne sera pas dépassé d'ici quelques années. De plus, si nous devons systématiquement encoder 10 champs pour les assistant.e.s des cours, alors des cours tels que INFO-F205 (Calcul Formel et Numérique) avec un unique assistant représenteraient un gaspillage de mémoire dans la table.

Pour cela, il est beaucoup plus intéressant de découper la DB comme suit :

1. une table `cours` comme jusqu'à présent (sans référence aux assistant.e.s) ;
2. une table `TA` contenant les informations relatives aux assistant.e.s (*e.g.* nom, prénom, matricule, faculté, *etc.*) ;
3. une table `TACours` contenant uniquement deux champs : l'identifiant d'un.e assistant.e (*i.e.* la table `TA`) et l'identifiant d'un cours (*i.e.* la table `cours`).

Ainsi, en supposant que INFO-F101 est toujours le cours d'ID 1 dans la table `cours`, récupérer la liste des assistant.e.s pour un cours reviendrait à récupérer, dans la table `TACours`, le premier champ de toutes les entrées dont le deuxième champ vaut 1.

Code à écrire

Pour cette phase, vous devez adapter votre code de manière à pouvoir effectuer de telles requêtes. Vous devez également adapter le programme `uldb.py` (et donc le langage imposé présenté dans la phase précédente) afin de pouvoir effectuer de telles requêtes dans l'interface interactive.

Puisque vous risquez de modifier la signature des méthodes imposées dans les phases précédentes, afin de vous assurer que les tests automatiques continuent de passer, nous vous conseillons d'écrire votre code dans un nouveau fichier `database_bonus.py`.

Aide Afin d'éviter de devoir copier/coller le code de `database.py` dans ce nouveau fichier, vous pouvez utiliser l'*héritage* (dont nous ne parlerons pas ici) en *augmentant* votre classe `Database` comme ceci :

```
1 from database import Database as _Database
2
3 class Database(_Database):
4     # ...
```

Vous pourrez alors utiliser les méthodes déjà existantes de la classe `Database` originale, en créer de nouvelles, voire même remplacer des méthodes existantes.