CSC 370 - SPRING 2018
DATABASE SYSTEMS
ASSIGNMENT 4
UNIVERSITY OF VICTORIA

Due: Sunday, April 8th, 2018 at 11:55pm. **Late assignments will not be accepted.**

This assignment will be evaluated by interactive demos starting the day after the due date (during the examination period). If you will not be on campus during that time, please contact your instructor **before April 4th** to arrange a demo during the last week of classes. If you do not schedule and attend a demo, you will receive a mark of zero.

1 Overview

This assignment covers applied database design and the use of database interfaces in a high level programming language (Python 3). Your submission for this assignment will consist of a database definition (`CREATE` statements and accompanying constraint definitions) in SQL (submitted as a `.txt` file due to conneX limitations) and a set of short Python 3 programs to act as database front-ends. Some of the Python programs will read comma-separated input data and perform database insertion and update operations. The other programs will use `SELECT` statements to retrieve data from the DBMS and format it as a report for the user. This assignment is a small-scale example of the type of database interaction that you might normally encounter as a developer.

Although this assignment requires writing Python code, the core objective of the assignment is to **write as little Python as possible**: the Python programs you write should function as simple appliances to convert input data `INSERT` or `UPDATE` statements and convert query results to formatted text. For full marks, you will be expected to leave all of the processing logic (checking the validity of data, moderating conflicts which arise in the data entry, processing results, etc.) to the DBMS and perform no non-trivial data processing in your Python code.

Since you may not have used Python recently, some example programs have been posted which demonstrate Python's capabilities for reading comma-separated data and producing formatted output. You may also find the examples of the `psycopg2` interface from the lectures to be useful in designing your solution.

2 Computing Environment

Your Python programs will be marked based on their behavior when run in the Linux environment on `dblinux.csc.uvic.ca`. It is expected that you use your individual database on one of the CSC 370 database servers (either `studdb1.csc.uvic.ca` or `studdb2.csc.uvic.ca`) as the back-end DBMS for your assignment.

Since your programs will be marked on `dblinux.csc.uvic.ca`, you must ensure that all of the libraries and Python modules that you use are available and work correctly in that environment.

Since your programs will contain very little non-trivial code, it should suffice to use the installed `psycopg2` module (for PostgreSQL connectivity) and the Python standard library, but you are free to use any other modules that may be available.

You are required to use Python 3 for this assignment (if you use Python 2, or any other language, your submission will not be marked).

3 The Task: A Student Registration Database

Your task for this assignment is to design a database schema and a set of front-end programs for a student registration database (similar to several example databases we have seen over the semester, as well as the larger-scale database actually used at UVic). Your database will track students, the courses they take and the grades they receive.

You will submit the files below. They **must** be named as shown.

- `create_schema.txt`: A schema creation file, consisting of an SQL script with `DROP` and `CREATE` statements for each table, along with `DROP` and `CREATE` statements for all required constraints. It must be possible to completely drop/recreate your database schema by running all of the commands in this file.
- `add_drop.py`
- `assign_grades.py`
- `create_courses.py`
- `report_classlist.py`
- `report_enrollment.py`
- `report_transcript.py`

4 The Database

For this assignment, your database will focus on students, courses (and course offerings) and grades. Other information (like staffing and accounting) has been omitted, but would normally be present in such a database. The subsections below detail the requirements and restrictions of each facet. You are not required to use any particular data model for your database, but you are encouraged to start by sketching an E/R diagram for the requirements (the exact set of entities may vary, but it will likely include separate entities for students, courses, course offerings, enrollments and grades). Requirements which are marked by an asterisk (*) in the sections below are **deliberately unrealistic** and are present to help simplify the task for this assignment.

For clarity, the set of constraints to enforce (which must be actively checked and enforced by your database) and constraints to assume (which you may assume will always hold in every input dataset, and therefore do not need to be checked) are given in point form at the end of each section.

4.1 Students

A student is uniquely identified by a student ID, which will be a string of the form `V00xxxxxx`, where each `x` is a digit (0-9). Students also have a name, which is not necessarily unique and will

be a string of text. You may assume that student names are at most 255 characters long. Student names are not required to have any particular format (that is, there is no requirement that a student have a first and last name). However, you may assume that student names do not contain comma or semicolon characters, or any form of space (e.g. tab, newline) except regular space characters. Student names may contain digits.

Students can enroll in course offerings and be assigned grades for course offerings in which they are enrolled. See Section 4.3 below for details on when enrollment is permitted.

Due to a requirement of `add_drop.py` (see Section 5.2), you may find it helpful to add a trigger that allows a student to be added to the database more than once, with every duplicate insertion ignored as long as the name and student ID match the existing record (note that the database should never actually contain duplicate student records).

Constraints to enforce:

- Only one name string may be associated with a particular student ID.
- Student IDs are unique.
- Every student has a student ID.

Constraints to assume:

- The maximum length for student names will be obeyed.
- Student numbers will always follow the format above.
- Student names will not contain commas, semicolons or any whitespace besides the regular space character.

4.2 Courses and Course Offerings

A **course** is identified by a course code, which will be a string of text (e.g. 'CSC 370' or 'SENG 265'). You may assume that a course code will never be longer than 10 characters.

A **course offering** is a particular instance of a course. A course offering is associated with a valid course code, a course name, a term code, a maximum capacity and an instructor name. Every offering of a particular course will share the same course code, but each offering may have a distinct course name. Course names will be at least 1 character and at most 128 characters in length. There may be at most one offering of a particular course in a particular term. The term code, which can be treated like either an integer or string, and will be a six-digit number of the form `yyyytt`, where `yyyy` is a year (e.g. 2018) and `tt` is a semester number: either `01` (for Spring), `05` (for Summer) or `09` (for Fall). Notice that two term codes in this format can be compared with an ordinary $<$ comparator to determine chronological order (for example, $201709 < 201801$, so Fall 2017 is earlier than Spring 2018). The maximum capacity will be a non-negative integer. It is possible for a course offering to have a capacity of zero. Instructors are identified only by name (and there is no need to introduce a surrogate key for instructors); if the same instructor name is used for multiple courses, that name is defined to refer to the same person (*).

A course offering will have an associated collection of zero or more prerequisites. The set of prerequisites may differ between offerings of the same course. For example, the 201709 offering of CSC 999 might require CSC 187, but the 201801 offering of CSC 999 might require CSC 188. The prerequisite list for a particular course offering will contain only course codes, not term codes, since

any offering of a prerequisite course (in an earlier term) is sufficient to satisfy the prerequisite. See Section 4.3 below for more details.

Every prerequisite in the prerequisite list must be a valid course code, and at least one offering of the prerequisite course must exist, but there is no requirement that offerings of prerequisite courses occur in earlier terms than course offerings which require them.

Constraints to enforce:

- Every course offering is associated with a valid course code.
- Every course offering has a non-empty name.
- The maximum capacity of the course offering is at least 0 (that is, the capacity is not negative).
- The instructor's name is non-empty.
- Each prerequisite has a valid course code.

Constraints to assume:

- The maximum lengths for course codes, course names and instructor names will be obeyed.
- Term codes will have the format described above.
- Course codes, course names and instructor names will not contain commas, semicolons or any whitespace besides the regular space character.

4.3 Enrollment

Suppose that S is a student and F is an offering of course C in term T. The student S may be **enrolled** into the course offering F if and only if all of the conditions below are met.

1. The student S is not already enrolled in F (since there is no need to do anything if they are already in the course)
2. The number of students already enrolled in F is less than its maximum capacity (so there is at least one free space for S)
3. Either the course offering F has no prerequisites, or one (or both) of the following conditions applies for **each** prerequisite course P:
   - The student is enrolled in any offering of P in a term **before** term T and has not yet received a grade.
   - The student is enrolled in any offering of P in a term **before** term T and has received a grade of at least 50.

   Note that it is possible for one of the above conditions to apply (and the prerequisite to be satisfied) even if the student has also taken the course P and received a failing (< 50) grade if they have retaken the course.

The conditions above only need to be checked when the enrollment occurs. As long as all conditions are met, the enrollment should be allowed. If the conditions later become false (for example, if the student is later assigned a failing grade in a prerequisite course or drops a prerequisite course), there is no need to automatically remove that student from the course offering F (*).

For additional clarity: Students may enroll in multiple offerings of the same course and receive different grades in each. Students may retake a course even if they have previously passed that course.

Students may be removed ("dropped") from a course offering in which they are enrolled, but only if no grade has been assigned. **Once a grade has been assigned for a student, that student is not permitted to drop that course offering.**

Constraints to enforce:

- All of the conditions (1-3) for enrollment above.
- Enrollments are only permitted if the student and course offering are valid.
- Once a grade has been assigned to a student S in a course offering F, student S cannot drop course offering F.

4.4 Grades

If a student S is enrolled in a course offering F, they may be assigned a final grade. Final grades are integers and **must** be in the range 0 through 100 (inclusive). A particular student may **not** be assigned more than one grade for a single course offering. However, one student may take multiple offerings of the same course and receive a grade for each of them. Additionally, it is not required for a student to be assigned any grade for a particular course offering (so the database must permit grades for a course to be absent).

Constraints to enforce:

- To receive a grade for a course offering, the student must **already be enrolled** in that course offering (and, by extension, all of the constraints governing enrollment must have already been checked and enforced).
- The grade must be in the range 0 to 100 (inclusive).

4.5 Deliverable: Schema creation script

The submitted `create_schema.txt` file will be an SQL script (which can be run, in its entirety, on the DBMS) containing `CREATE` statements (and any other necessary statements, including `INSERT`, `UPDATE` and `ALTER` if required) for all tables an constraints to allow the programs described below to function correctly. To be clear, your Python programs should **not** contain the SQL `CREATE` statements.

You **must** also include, at the top of your file, `DROP` statements for each database object (table, function, etc.) created in your script. You should use the 'if exists' notation to ensure that the `DROP` statements continue to work even if the tables/functions being dropped do not already exists.

The goal of this format is to allow the `create_schema.txt` script to be run as a way of clearing out the database and preparing it for a fresh testing sequence. A similar format has been followed in the posted SQL scripts from the modifications, constraints and transactions lectures.

Include a comment with your name and student number at the beginning of the file.

In addition to implementing the requirements described in the previous sections, your database design must also obey the following style constraints.

- All tables have a primary key.
- String fields must use the `varchar` type, not a Postgres-specific string type like `text`.

- Every definition of a foreign key constraint **must** include `ON DELETE` and `ON UPDATE` policies (you can set the policies to `RESTRICT` if needed, but you must explicitly include the policy directives instead of allowing the default to be used).

Advice: If you want to run your entire `create_schema.txt` script (to refresh your database), you can try one of the three options below.

- Open a `psql` session and use the `\i` command to run the script.
- Pipe the contents of `create_schema.txt` into `psql` on the command line. For example,
  ```
  psql -h studdb1.csc.uvic.ca dbname username < create_schema.txt
  ```
- Open the script in DBeaver and use Alt-X to run the entire file.

5 Front-end Programs: Data Entry

You are required to write three Python 3 programs which read data from a text file (in a simple comma-separated spreadsheet format) and make insertions or modifications to the database. You are permitted to use any features of Python 3 available on `dblinux.csc.uvic.ca`, but, in general, you should avoid as much processing as possible in the front-end programs. Even simple processing like input validation (for example, checking if a grade is within the required range) should be implemented with constraints in the DBMS, not on the front-end. For full marks, your front-end programs must be as simple as possible and leave all non-trivial data processing to the DBMS: a perfect solution will consist of a simple read loop to read each line of the input file and a few statements to run SQL on the server (and possibly handle any exceptions that may occur). You will be deducted marks if your solutions for these programs contain `SELECT` statements in any form: either stand-alone (that is, a query that is retrieved by your program, inspected, and used to determine what to insert/delete) or nested within an `INSERT` statement (although nested `SELECT` statements inside `INSERT` statements will be subject to a smaller deduction).

Several example programs have been posted which detail how to read comma-separated data and use the `psycopg2` module to interact with the DBMS. The total number of lines of Python needed to implement each part should be relatively small, and it should be possible to reuse most of the code for reading input between the programs.

5.1 Creating courses: `create_courses.py`

The `create_courses.py` script is invoked from the command line as follows.

```
python3 create_courses.py <input file>
```

Where '`<input file>`' is a file name (e.g. '`courses_to_create.txt`').

Each line of the provided input file will either be blank (or consist entirely of spaces), in which case it will be ignored, or will contain 4 or more values, with a single comma character between each field, in the form

```
<course code>,<course name>,<term code>,<instructor name>,<maximum capacity>,<prereq1>,<prereq2>,...
```

The fields after `<maximum capacity>` are optional and list the course codes of prerequisites. There may be any number of such fields (including zero).

For example, the input sample below contains records for seven courses.

```
CSC 110,Fundamentals of Programming: I,201709,Jens Weber,200
CSC 110,Fundamentals of Programming: I,201801,LillAnne Jackson,150
CSC 115,Fundamentals of Programming: II,201709,Tibor van Rooij,100,CSC 110
CSC 115,Fundamentals of Programming: II,201801,Mike Zastre,200,CSC 110
MATH 122,Logic and Fundamentals,201709,Gary McGillivray,100
MATH 122,Logic and Fundamentals,201801,Gary McGillivray,100
CSC 225,Algorithms and Data Structures: I,201805,Bill Bird,100,CSC 115,MATH 122
```

Notice that some courses have zero prerequisites, while others have one or two prerequisites.

Some starter code has been posted which uses the `csv` module to parse the input data. If you use the provided code as the basis for your submission, you should not need to worry about reading the input format.

The `create_courses.py` program will read each line from the input file and insert records into the database for each course offering. The set of all modifications made by the program must be atomic with respect to the database: If a database error occurs at any point during execution, the set of all modifications must be rolled back. As a result, a particular run of `create_courses.py` will either successfully read an input file and modify the database, or fail and make no modifications whatsoever.

The `create_courses.py` program must catch any exceptions that occur as a result of database errors and print an error message. As stated above, an error should result in the database transaction being rolled back such that no modification occurs, but it is important that your Python program does not crash: instead, it should catch the exception, display some kind of error message (ideally a descriptive one) and exit gracefully.

Records should be added to the database in the same order as they appear in the file. Although the entire file should be added as one large transaction, constraints should not be deferred. For example, if a course is created with references to prerequisites that do not yet exist (but will be added by a later line), a database error should occur.

The `create_courses.py` program may be run multiple times on the database: do not assume that courses are only created once.

In the case where the data entry is successful, the program should not generate any output to standard output or standard error. In cases where an error of any kind occurs, the program should print an error message to indicate that the data entry failed. If the program crashes due to an unhandled exception, you will lose marks.

5.2 Enrollment management: `add_drop.py`

The `add_drop.py` script is invoked from the command line as follows.

```
python3 add_drop.py <input file>
```

Where '`<input file>`' is a file name (e.g. '`adds_and_drops.txt`').

Each line of the provided input file will either be blank (or consist entirely of spaces), in which case it will be ignored, or will contain exactly 5 text fields, with a single comma character between each field, in the form

```
<ADD_or_DROP>,<student id>,<student name>,<course code>,<term>
```

The first field, `<ADD_or_DROP>`, will either be the word "ADD" or the word "DROP", indicating which operation to perform. Notice that the name of each student appears every time that student is added or dropped from a course.

For example, the input sample below contains enrollment records for several students, using the set of course offerings in the example from the previous section.

```
ADD,V00123456,Alastair Avocado,CSC 110,201709
ADD,V00123456,Alastair Avocado,CSC 115,201801
ADD,V00123457,Rebecca Raspberry,CSC 110,201709
ADD,V00123457,Rebecca Raspberry,CSC 115,201801
ADD,V00123456,Alastair Avocado,MATH 122,201709
ADD,V00123457,Rebecca Raspberry,MATH 122,201801
ADD,V00123456,Alastair Avocado,CSC 225,201805
ADD,V00123457,Rebecca Raspberry,CSC 225,201805
DROP,V00123456,Alastair Avocado,CSC 110,201709
```

Notice that all of the prerequisite constraints are met at each 'ADD' line. The 'DROP' at the end of the input file will result in a prerequisite no longer being met, but, as mentioned in Section 4.3, this is permitted (and you should not add triggers to your database to automatically remove students from courses for which the prerequisite constraints are not met due to a drop).

Some starter code has been posted which uses the `csv` module to parse the input data. If you use the provided code as the basis for your submission, you should not need to worry about reading the input format.

The `add_drop.py` program will read each line from the input file and make the appropriate insertions/updates/removals for each operation. The set of all modifications made by the program must be atomic with respect to the database: If a database error occurs at any point during execution, the set of all modifications must be rolled back. As a result, a particular run of `add_drop.py` will either successfully read an input file and modify the database, or fail and make no modifications whatsoever.

An `ADD` operation must not succeed if the student is already enrolled in the course offering (that is, a student cannot be enrolled in the same course offering twice). However, the same student can enroll in multiple offerings of the same course.

There is no direct mechanism to 'add students' to the database. Instead, a student is created the first time they are added to a course. The name used for the first registration will become the student's permanent name. All future ADD/DROP operations involving that student must use **exactly** the same name as the first registration or the operation should fail. As mentioned in Section 4.1, one way to achieve this is to add an `INSERT` trigger that allows the same student to be inserted multiple times as long as the name and student ID match the existing record (with the understanding that all duplicate insertions are silently ignored). If you can't figure out a way to implement the constraint directly into the database, you can use Python logic to enforce it (but you will not receive full marks for doing so).

A student will not exist until the first time they are added to a course. However, once added, students will never be deleted from the database (even if they drop every course in which they are

enrolled).

Modifications should be made to the database in the same order as they appear in the file. Although the entire file should be added as one large transaction, constraints should not be deferred.

The `add_drop.py` program may be run multiple times on the database, including after grades are assigned.

In the case where the data entry is successful, the program should not generate any output to standard output or standard error. In cases where an error of any kind occurs, the program should print an error message to indicate that the data entry failed. If the program crashes due to an unhandled exception, you will lose marks.

5.3 Grading: `assign_grades.py`

The `assign_grades.py` script is invoked from the command line as follows.

```
python3 assign_grades.py <input file>
```

Where '`<input file>`' is a file name (e.g. '`grades_to_assign.txt`').

Each line of the provided input file will either be blank (or consist entirely of spaces), in which case it will be ignored, or will contain exactly 5 text fields, with a single comma character between each field, in the form

```
<course code>,<term>,<student ID>,<grade>
```

For example, the input sample below assigns grades to the set of students and courses used in the examples in the previous sections. Notice that not every student in every course is assigned a grade (and that some courses have a mixture of students with grades and students without grades).

```
CSC 110,201709,V00123457,80
MATH 122,201709,V00123456,67
CSC 225,201805,V00123457,75
CSC 225,201805,V00123456,79
CSC 115,201801,V00123456,83
```

Some starter code has been posted which uses the `csv` module to parse the input data. If you use the provided code as the basis for your submission, you should not need to worry about reading the input format.

The `assign_grades.py` program will read each line from the input file and set the grade for each student accordingly.

The set of all modifications made by the program must be atomic with respect to the database: If a database error occurs at any point during execution, the set of all modifications must be rolled back. As a result, a particular run of `assign_grades.py` will either successfully read an input file and modify the database, or fail and make no modifications whatsoever.

Modifications should be made to the database in the same order as they appear in the file. Although the entire file should be added as one large transaction, constraints should not be deferred.

The `assign_grades.py` program may be run multiple times on the same database.

In the case where the data entry is successful, the program should not generate any output to standard output or standard error. In cases where an error of any kind occurs, the program should print an error message to indicate that the data entry failed. If the program crashes due to an unhandled exception, you will lose marks.

6 Front-end Programs: Reports

You are required to write three Python 3 programs which query the database (using whatever sequence of `SELECT` statements you find appropriate) and generate data 'reports' to standard output.

Sample Python scripts with output commands to produce mockups of the expected format have been posted to conneX, so you do not need to worry about the tedious process of designing the output formatting commands. You are expected to produce output in the format described in the mockup files (whether or not you actually use the mockup files as starter code).

6.1 Course Enrollment Summaries: `report_enrollment.py`

The `report_enrollment.py` script is invoked from the command line as follows.

```
python3 report_enrollment.py
```

The program takes no arguments. The output of the program (on standard output, not to a file) will be a list of all course offerings, showing their term, course code, course name, instructor name, enrollment and maximum capacity. The exact format should follow the formatting given in the mockup `report_enrollment.py` program posted to conneX (and in the example below).

When run on the database produced by the all of the input files in the examples of Section 5, the following result will be produced.

```
201709    CSC 110 Fundamentals of Programming: I      Jens Weber               1/200
201709    CSC 115 Fundamentals of Programming: II     Tibor van Rooij          0/100
201709   MATH 122 Logic and Fundamentals              Gary McGillivray         1/100
201801    CSC 110 Fundamentals of Programming: I      LillAnne Jackson         0/150
201801    CSC 115 Fundamentals of Programming: II     Mike Zastre              2/200
201801   MATH 122 Logic and Fundamentals              Gary McGillivray         1/100
201805    CSC 225 Algorithms and Data Structures: I   Bill Bird                2/100
```

6.2 Class Lists: `report_classlist.py`

The `report_classlist.py` script is invoked from the command line as follows.

```
python3 report_classlist.py <course code> <term>
```

The output of the program (on standard output, not to a file) will be a list of all students in the specified course offering, including their name and grade (if a grade has been assigned). The list of students will be sorted by student ID. The first few lines will contain a summary of the details of the course (course code, course name, term and instructor name), and the last line will show the total enrollment and maximum capacity.

If no course offering with the provided course code and term exists, the program will print an error and exit without generating any other output.

The exact format of the output should match the format produced by the mockup `report_classlist.py` program posted to conneX (and in the example below).

On the database produced by the all of the input files in the examples of Section 5, the following command will produce a class list for the 201801 offering of CSC 115.

```
python3 report_classlist.py "CSC 115" 201801
```

Notice that the course code must be given in quotes since it contains a space. The output of the program on the command above is shown below.

```
Class list for CSC 115 (Fundamentals of Programming: II)
  Term 201801
  Instructor: Mike Zastre
 V00123456 Alastair Avocado            GRADE: 83
 V00123457 Rebecca Raspberry
2/200 students enrolled
```

6.3 Class Lists: `report_transcript.py`

The `report_transcript.py` script is invoked from the command line as follows.

```
python3 report_transcript.py <student ID>
```

The output of the program (on standard output, not to a file) will be a list of all courses taken by the student, along with the grade assigned (if present). For each course, the report will show the term, course code and course name. The courses will be sorted by term and course code. The first line of the report will show the student's ID and name.

If no student with the provided ID exists, the program will print an error and exit without generating any other output.

The exact format of the output should match the format produced by the mockup `report_transcript.py` program posted to conneX (and in the example below).

On the database produced by the all of the input files in the examples of Section 5, the following command will produce a transcript for the student with ID 'V00123456'.

```
python3 report_transcript.py V00123456
```

Notice that the course code must be given in quotes since it contains a space. The output of the program on the command above is shown below.

```
Transcript for V00123456 (Alastair Avocado)
201709   MATH 122 Logic and Fundamentals               GRADE: 67
201801    CSC 115 Fundamentals of Programming: II      GRADE: 83
201805    CSC 225 Algorithms and Data Structures: I    GRADE: 79
```

7 Evaluation

Your submitted schema creation script must run in a single pass (that is, as a script without the need for human intervention and without generating any errors) on your individual PostgreSQL database on the `studdb1.csc.uvic.ca` or `studdb2.csc.uvic.ca` server.

Your Python code must run correctly using the built-in installation of Python 3 (specifically, the `python3` command which resolves to `/usr/bin/python3`) on the `dblinux.csc.uvic.ca` login server. If your code does not meet that requirement, it will not be marked. All of your Python 3 programs must use the `studdb1.csc.uvic.ca` or `studdb2.csc.uvic.ca` database servers exclusively for all data storage; if you use local files to store data (or somehow connect to another database server), your code will not be marked.

When you hand in your code, it must be written to connect directly to the database server without prompting the user for a password. This means that you will have to hard-code your database account password into your scripts as text (so the members of the CSC 370 teaching team will be able to see the password). Therefore, **you must change your database password to something you feel comfortable sharing with us** before handing in the assignment. In particular, **do not use your Netlink, CSC account or conneX password** as your database password, and never divulge information about your Netlink password, CSC account password or conneX password to anyone, including your instructor or any other department personnel.

This assignment will be marked out of 28 during an interactive demo with a member of the CSC 370 teaching team. Demos must be scheduled in advance (through an electronic system available on conneX). More information about demo scheduling will be given closer to the due date. If you do not schedule a demo time, or if you do not attend your scheduled demo, you will receive a mark of zero.

The marks are distributed among the components of the assignment as follows.

Marks	Component
4	The database schema is well designed (using a normalized data model and obeying the style requirements/best practices covered by CSC 370) and consistent. In particular, primary keys are specified for every table and foreign keys are defined where applicable.
7	The suite of data entry and report programs functions correctly on test sequences containing strictly valid data (with no violations of any constraints). Note that you may still receive these marks even if some of the constraints are enforced on the client side.
7	The suite of data entry and report programs functions correctly on test sequences containing invalid data. Errors with input data must be properly handled, and inconsistent data must not be added to the database. Note that you may still receive these marks even if some of the constraints are enforced on the client side.
10	All constraints and data validation logic are integrated into the database schema instead of being enforced by the client-side programs. When this has been done correctly, the Python programs will perform the **bare minimum** amount of processing (all data validation, data processing and constraint enforcement is handled on the server-side), and it will be impossible to add invalid data to the database, even if manual insertion statements are used instead of the client-side data entry programs.

Submission Instructions

All submissions for this assignment will be accepted electronically. Submit all of your files through the Assignments tab on conneX. Do not use the `.sql` extension for any files, since conneX does not properly handle files with the `.sql` extension. Name each file as specified in the sections above. You are permitted to delete and resubmit your assignment as many times as you want before the due date, but no submissions or resubmissions will be accepted after the due date has passed.

Ensure that each file contains a comment with your name and student number.

After submitting your assignment, conneX will automatically send you a confirmation email to your `@uvic.ca` email address If you do not receive such an email, your submission was not received. If you have problems with the submission process, send an email to the instructor **before** the due date.