University of Toronto
CSC343, Fall 2022

Assignment 2

Due: Wed, November 2, before 4pm

Part 2: Embedded SQL

Imagine an Uber app that drivers, passengers and dispatchers log in to. The different kinds of users have different features available. The app has a graphical user-interface and is written in Python, but ultimately it has to connect to the database where the core data is stored. Some of the features will be implemented by Python methods that are merely a wrapper around a SQL query, allowing input to come from gestures the user makes on the app, like button clicks, and output to go to the screen via the graphical user-interface. Other app features will include computation that can't be done, or can't be done conveniently, in SQL.

For Part 2 of this assignment, you will write several methods that the app would need. It would need many more, but we'll restrict ourselves to just enough to give you practise with psycopg2 and to demonstrate the need to get Python involved, not only because it can provide a nicer user-interface than postgreSQL, but because of the expressive power of Python.

General requirements

- The methods we have asked you to write (`clock_in, pick_up, and dispatch`), and any helper methods they call, must not take input from the user or from a file. Doing so will result in the autotester timing out, causing you to receive a **zero** on that method. You can take input from the user in your main block and testing functions.
- Do not change any of the code provided. In particular, you may not change the header of any of the methods we've asked you to implement. Each method must have a try-except clause so that it cannot possibly throw an exception.
- You have been provided with methods called `connect()` and `disconnect()` that allow you to respectively connect to and disconnect from the database. You must **NOT** make any modifications to either method. You should also not modify the private method `_register_geo_loc()`.
- You should **NOT** call `connect()` and `disconnect()` in the other methods we ask you to implement; you can assume that they will be called before and after, respectively, any other method calls.
- All of your code must be written in `part2.py`. This is the only file you may submit for this part.
- You are welcome to write helper methods to maintain good code quality.
- Within any of your methods, you are welcome to define views to break your task into steps. Drop those views before the method returns, or otherwise a subsequent call to the method will raise an error when it tries to define a view that already exists. Alternatively, you can declare your view as temporary so that it is dropped automatically once the connection is closed. The syntax for this is `CREATE TEMPORARY VIEW name AS ...`
- Your methods should do only what the docstring comments say to do. In some cases there are other things that might have made sense to do but that we did not specify (in order to simplify your work). Don't do those extra things.
- If behaviour is not specified in a particular case, we will not test that case.

Your task

Complete the following methods in the starter code in `part2.py`:

1. `clock_in`: A method that would be called when the driver declares that they are ready to start a shift.
2. `pick_up`: A method that would be called when the driver declares that they have picked up a client.
3. `dispatch`: A method that would be called when the dispatcher chooses to dispatch drivers in response to clients' ride requests within a geographical area.

You will have to decide how much to do in SQL and how much to do in Python. You could use the database for very little other than storage: for each table, you could write a simple query to dump its contents into a data structure in Python and then do all the real work in Python. This is a bad idea. The DBMS was designed to be extremely good at operating on tables! You should use SQL to do as much as it can do for you. In particular, there is no need to use Python data structure such as dictionaries, sets or even simple lists for temporary storage.

We don't want you to spend a lot of time learning Python for this assignment, so feel free to ask lots of Python-specific questions as they come up.

Using values of type point

Some of the columns in our tables have type `geo_loc`, which is defined in terms the PostgreSQL type `point`. (See `schema.ddl` for the definition of type `geo_loc`.) In Part 2 of the assignment, you'll work with these kinds of values.

In SQL, here's how to access the x and y coordinates within a `point` (and therefore a `geo_loc`):

> "It is possible to access the two component numbers of a point as though the point were an array with indexes 0 and 1. For example, if t.p is a point column then `SELECT p[0] FROM t` retrieves the `X` coordinate and `UPDATE t SET p[1] = ...` changes the `Y` coordinate."

(Reference: `https://www.postgresql.org/docs/12/functions-geometry.html`)

When you need to store a point in your Python code, you will use a Python object of type `GeoLoc`. This class has been defined for you. Some additional code is necessary to enable the use of our custom class `GeoLoc` in `psycopg2`. That code is provided in method `_register_geo_loc()`.

Additional Python tips

Some of your SQL queries may be very long strings. You should write them on multiple lines, both for readability and to keep your code within an 80-character line length. You can achieve that by using Multi-line strings in Python. Multi-line strings are declared using triple quotes, and are allowed to span multiple lines.

```
sql_query = """
        SELECT client_id
        FROM Request r JOIN Billed b ON r.request_id = b.request_id
        WHERE amount > 50
   """
```

Alternatively, You can break the string into pieces and use `+` to concatenate them together. Don't forget to put a blank at the end of each piece so that when they are concatenated you will have valid SQL. Example:

```
sql_query =
   "SELECT client_id " +
   "FROM Request r JOIN Billed b ON r.request_id = b.request_id " +
   "WHERE amount > 50";
```