Overview

Ride Finder is a Python application that implements a simple command line interface and SQLite connectivity. Using SQLite databases, users are able to create new accounts (or login with existing ones) and navigate through a plethora of options to suit their purposes. Users may offer rides, search for existing rides, post ride requests, book other members on their rides, and even search to see what rides other members are requesting. Users may message each other in regards to particular rides and bookings and these messages are automatically displayed upon logging in to ensure the member is constantly kept up to date about the status of their rides, bookings or requests.

Ride Finder is a user-friendly and intuitive system that guides the users to where they want to go. Along every step of the way, the system gives the user a list of valid instructions that may be accepted as input; the user simply has to follow the instructions on the screen to reach the goal they are trying to achieve. There is sample error-handling to catch most (if not all) user errors.

Detailed Design

The design of *Ride Finder* may be broken down to 6 core functionalities. Each of these functionalities utilizes SQLite databases and queries in one form or another. They are as follows:

1. *Login/Account Creation*
2. *Offering a Ride*
3. *Book Members on Rides*
4. *Post Ride Requests*
5. *Search for Offered Rides*
6. *Search/Delete Ride Requests*

Login/Account Creation

Offer a Ride

Search for Offered Rides

The user is allowed to enter up to three keywords, and any rides that match any and all keywords is returned to the user along with any car information for that ride (if any). Each keyword may be a location code, city, province, or address. Location code keywords must match existing location codes letter-for-letter, however, keywords can by substrings of cities, provinces or addresses. All matching is case-insensitive. A ride is a match if the keyword is linked to either a ride's source location, destination location, or a enroute location.

The main function in the searchRides.py is searchForRide() which handles all querying and information output. The basic idea for how the SQLite queries were structured for this task is as follows: For each keyword, query for all rides that match that keyword and populate a list with

the rides. Once all rides have been queried, then find the intersection (i.e. find all rides that match all the keywords) of the lists. This leaves the rides that match all entered keywords - It is possible to not have any rides that match all the words. After printing the rides, the system then allows the user to select one of the rides and executes the messageOwner() function which gives the user the option to message the owner of the ride they selected. At any time the user may type in BACK or EXIT (case-insensitive) to leave the current screen.

Search/Delete Requests

There are two main streams to consider within this task. Either the user wants to view their own requests and possibly delete them, or they can search for all requests (excluding their own) based on source location. If they choose to look at their own request, the program executes the viewOwnRequests() function which queries for all the requests the user has posted. The user may select an index of the requests that were printed on the screen which then causes the program to call the deleteRequest() function.

If the user wants to search for other member's requests, the program then executes the viewOtherRequests() function and allows the user to input either a location code or city and search for requests with a pickup location matching the input. Unlike when searching for rides, the input must match letter-for-letter (still case-insensitive). The program prints out all requests (5 at a time, the user can choose to see more) and allows the user to select any request displayed on the screen and to message its owner using the messageRequestOwner() function. When a message is sent to a request owner, the message will be displayed for them next time they log in.

Book Members on Rides

The User is allowed to choose any of the following options: List all bookings on ride user offers, Cancel Bookings on rides user offers, List all rides user offers and the number of available seats for each ride and or Book Rides.

- When the user lists all bookings on his/her rides, the system queries both tables rides and bookings where rides.bno is the same as bookings.bno.
- When the user cancels bookings on rides user offers: an existing bno is entered then the system deletes the bno's tuple on the booking table.
- When the user lists all rides user offers and the number of available seats, the system displays all existing rides that have bookings then the system displays all rides that do not have bookings
- When the User books his/her rides, his/her rno is entered followed by their
- respective price, an existing email of another user, existing lcode for pick up and drop off and the number of seats the user wants to book. If the number of seats requested does not exceed the seats available then the System will insert a new tuple with the information entered in the booking tables. If the number of seats requested exceeds the seats available, the user has a choice to confirm. If the user confirms, a new booking with the information entered is inserted into the booking table. Otherwise, No booking will be inserted to the booking table and the user will return to the list of options. A unique

bno is created by taking the last unique bno of the booking table and adding one to the value.

Post Ride Requests

The User is allowed to post a Ride request by entering their return date, an existing pickup location, an existing drop of locations, and an amount they are willing to pay per seat. A new ride request with the information entered is then inserted into the ride requests tables. A unique rid is created by taking the last unique rid of the ride requests table then adding one to that value

Detailed Design

The functionality of Ride Finder is dependant on a given database (.db file) that has 8 tables: bookings, cars, enroute, inbox, locations, members, requests and rides. Bookings contains data regarding reservations for a driver to facilitate a ride, identified by a generated bno integer. Cars includes information on the registered vehicles and identified by a generated cno integer. Enroute contains a list of locations that were stopped at between the pickup and drop off location of a ride; these are identified by the combination of the corresponding rno (ride number) and the location. Inbox contains message information to a driver, including the viewed status of the message. Messages are identified by the combination of email they were sent to and the timestamp. Locations contain city, province, and address of locations and identifies them with a given lcode. Members are stored in the members database by unique email, and also contain name, phone number and entered password information. The requests table holds inquiry information by members for rides, each having a unique rid. Information in the rides table corresponds to rides offered/performed and identified with a unique, generated rno integer. When the program begins, it prompts the user for a database file name. If the entered file does not exist, a file is created by that name, however, it will experience runtime errors throughout the program's operation. The application works by performing inquiries on the given database using SQLite operations, and then performing updates to that database and committing those changes using Python functionalities including cursors and connections. Ride Finder is designed to take in user input and then construct SQL queries and send those queries to the specified database before presenting the results to the user.

Testing Strategy

Since the tasks were mostly independent, each member did their own individual testing for the modules they were responsible for with their own data set. Once we were confident enough in the integrity of our modules, we combined them together and tested the *Ride Finder* system as a cohesive whole. Part of that testing included using the marking rubric (posted on eClass) as a checklist to ensure all required functionality was satisfied.

Our test cases were scenario based and were meant to cover multiple functionalities in one test case. Some examples of test cases we used is as described below

Create New Account > Log In > Create New Ride > Search for (Newly Created)Ride Using Multiple Keywords > Message Owner of a Ride > Log In as Message Recipient > Verify Message was Received.

Offer Ride > Book Member on Newly Created Ride > Search Requests > Message User in Regards to Newly Created Ride

Group Work Breakdown

Warren Thomas

- Completed Login, SQL Injection, System Functionalities part 1, “offer a ride function”, and ensured branches from other members merged properly with the master. Estimated Time 10 to 12 hours

Aaron Espiritu

- Created Data Tables, Completed System functionalities part2, “ Search for Rides” and part 5, “Search and Delete Ride Requests”. Estimated Time 10 to 12 hours

Jeffrey Baglinit

- Completed System Functionalities part 3, “Book Members or Cancel Bookings” and part 4 “Post Ride Requests”. Estimated Time Spent is between 10 to 15 hours

All members contributed to the work of ensuring each of their project parts worked together in the master branch. Also, All Members contributed to the Design Document and Read Me File.

Meetings were taken place throughout the progression of the project to establish the group work, determine the progress of each member and to ensure each member was on the same page with the project objectives.

We used GitHub as our method of coordination. To keep track of the project, each member created their own branch and sent their pull requests as they progressed through their respective project parts.