CS457 Project 2

Yifeng Qin

10/15/20

1 How does my program organize multiple databases (Project 1)

The design of my program follows the example in the Assignment Overview.
One Directory corresponds to a database.
For example the parent directory will be /your_home/cs457
If you were to create a new database, it would create a new directory inside the parent directory.
Example: /your_home/cs457/db_1 (is what it would look like after creating db_1)
Then if you want to remove this database it would remove that directory with the db name.
Example: /your_home/cs457 (is what it would look like after removing db_1)
You can create as many distinct databases within the parent directory because they will all be represented by a directory. This means you can also delete a database by deleting the corresponding directory.

2 How does my program organize multiple tables (Project 1)

One file corresponds a table in a directory.
For example the parent directory will be /your_home/cs457
To create a table you have to already created a database because you need to have a directory to place the table file in.
You also have to USE a database to create a table in that database.
When wanting to create a table we join the parent directory with database directory and create a file with the table name.
So each database can have the same distinct names.

3 How does my program store data (Project 2)

So all data that is inserted into a table is written to the corresponding file for that table. Meaning that everything is written to the file as a string including numbers. Each line is separated by a newline and each column is separated by a '|'. I created a helper function that reads all the data from the file and returns the all the data in a list format. All the data that is read from the file is still in string format. Then I take each line and make it it's own list making the data structure a list of lists. Each list represents a line and each index in that line represents a specific column. I decided against a tuple because specific operations like '==' and '!=' still compare numbers the same in string and number format. If I see a '>' or '<' operation that means I have to compare numbers, meaning that I will cast the string to a float to perform the logic operation.

4 Functional Requirements

List the functional requirements that were listed in the assignment. Each functionality correlates to a function, which can be found at the end of the document.

4.1 Project 1

Database Creation - 4.2.1 create_database(self, db):
Joins the parent directory with the entered database name. Checks if the database already exists. If it exists already it will print out and error, if not it will create a directory of the database name.

Deletion - 4.2.2 drop_database(self, db) :
Joins the parent directory with the entered database name. Checks if the database already exists. If it does not exist it will print out and error, else it will delete a directory of the database name.

Table Creation - 4.2.4 create_table(self, tbl, inp) :
Gets the path to that table and will check if that already exists. Then if it does not exist, it will change the directory and create a file of that table name in that directory. If there are inputs to within the command it will write those variables to that file.

Deletion - 4.2.7 drop_table(self, tbl):
Joins the parent directory, database directory, and the name of the table, and if it exists delete that path. If it does not exist it will error and print the error message.

Update - 4.2.6 alter_table(self, tbl, inp):
Joins the parent directory, database directory, and the name of the table, and if it exists open the file and append the update values to file. If the table does not exist it will print and error message.

Query - 4.2.5 select_all(self, table):
Joins the parent directory, database directory, and the name of the table, and if it exits it will open the file and read all the contents into a list. Then it will print out all the contents to the terminal. If it does not exits it will print and error out to the terminal.

4.2 Project 2

Insert - 5.2.8 insert_table(self, table, new_data):
Checks if there the table is a valid path, then it will read all the current data from that table and store it in a list of lists. We take the new data that needs to be inserted and place them in the respective columns. We append that new data to the new current data. Then then call a helper function to write the new data back to the file.

Delete - 5.2.12 delete_items(self, table, data):
Checks if the table is a valid path. Reads all the data from the current table and stores it in a list of list. Checks the logic operation to determine how to compare the values that need to be deleted. Then calls a helper function to delete the data from the current list of lists. This helper function will find the index of the variable we are comparing. Then we use the eval function to turn the string operation into a logic operation and perform it on the index of the variable we are comparing. If the variable we are comparing satisfies the condition we will delete that line from the current data. When all lines have been compared, it will call the insert helper function to write the new data back to the file.

Modify: - 5.2.10 update_table(self, table, data)
Checks if the table is valid. Then it will read all the data from the current table and store it in a list of lists. It will then pass all that information the update helper function to update the current data. This helper function will first find the index of the variable we want to compare and the variable we want to

change. Uses the eval function to perform a logic operation based on the operation in the command. It will compare the index of the variable compared and if it satisfies the condition it will change the value at the index of the desired variable. Then it will call the insert helper function to write all the data back the file.

Query - 5.2.14 select_specific(self, var, table, where):
Checks if the table is valid. Then it will read all the data from the current table and store it in a list of lists. It find all the missing variables that are not needed to be shown. It will take all those unnecessary indexes and delete the columns from the print. Then it will call the delete helper function to get rid of rows that don't satisfy the where clause. This will return the final result to be printed to the terminal.

5 Functions

This section contains all the code that runs the functionality of the program. Broken down into python files and functions.

5.1 main.py

Main driver that takes in the input from the file and calls functions in the run_script.py file to execute those commands.

5.1.1 read_file()

```
    """
    Opens the file with standard input, and reads every line and checks if it is a valid
    command to be
    appended to the list. If there is a command on multiple lines then it will store it in a
     temp variable to be appended when it finds
    the semicolon.
    :return: Returns a List of the Commands
    """
    commands = []
    temp = []
    for line in sys.stdin:
        if line == '\r\n' or line == '\n' or line[0:2] == '--':  # checks if it is a
    relevant line to read
            continue
        else:
            if ';' not in line:
                temp.append(line.rstrip()+' ')
            else:
                temp = "".join(temp)
                line = re.sub(r"[\n\t]*", "", line)  # removes random special characters
    like tabs
                commands.append(temp + line.rstrip())  # removes newline from line to append
     to list
                temp = []
    return commands

```

5.1.2 run_commands()

```
    """
    Calls the helper function read_file to get all the valid commands from the file. It will
     run through the list and
    splice the commands to get rid of special characters.
    :return: None
```

```
    """
    commands = read_file()  # calls helper function to initialize to list commands
    for command in commands:
        l = command.split(' ')  # splits the string command into a list based on spaces
        command = command.upper()  # converts the command to all uppercase so it can cover case sensitivity
        size = len(l)  # gets length to handle missing spaces
        if 'CREATE DATABASE' in command:
            if size == 3:  # checks if all arguments are present
                script.create_database(l[2][:-1].upper())  # only gets the database name and removes the ';' from the back
            else:
                print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
        elif 'DROP DATABASE' in command:
            if size == 3:  # checks if all arguments are present
                script.drop_database(l[2][:-1].upper())   # only gets the database name and removes the ';' from the back
            else:
                print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
        elif 'DROP TABLE' in command:
            if size == 3:  # checks if all arguments are present
                script.drop_table(l[2][:-1].upper())   # only gets the database name and removes the ';' from the back
            else:
                print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
        elif 'USE' in command:
            if size == 2:  # checks if all arguments are present
                script.use_database(l[1][:-1].upper())   # only gets the database name and removes the ';' from the back
            else:
                print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
        elif 'CREATE TABLE' in command:
            if size >= 3:  # checks the the minimum amount of arguments are present
                command = " ".join(l[3:])  # gets all the variables after the table name and converts it into a string
                command = command[1:-2]  # then slice off the beginning '(' and the ');' at the end
                script.create_table(l[2].upper(), command)  # passes in the name of the table and the sliced variables to input
            else:
                print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
        elif 'SELECT * FROM' in command:
            if size == 4:  # checks if all arguments are present
                script.select_all(l[3][:-1].upper()) # only gets the table name and removes the ';' from the back
            else:
                print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
        elif 'ALTER TABLE' in command:
            if size >= 4:  # checks if all arguments are present
                command = " ".join(l[4:])  # gets all the variables after the table name and converts it into a string
                command = command[:-1]  # removes the ';' from the back of string
                script.alter_table(l[2].upper(), command)  # passes in the name of table and sting of variables
            else:
                print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
        elif 'INSERT' in command:
            if size >= 5:  # checks if all arguments are present
                script.insert_table(l[2].upper(), l[3:])
            else:
                print('Syntax Error:', command)  # if size does not match there has to be a
```

```
    syntax error with cmd
        elif 'UPDATE' in command:
            if size >= 8:  # checks if the minimum amount of variables are present
                script.update_table(l[1].upper(), l[2:])
            else:
                print('Syntax Error:', command)  # if size does not match there has to be a
    syntax error with cmd
        elif 'DELETE' in command:
            if size >= 7:  # checks if the minimum amount of variables are present
                script.delete_items(l[2].upper(), l[3:])
            else:
                print('Syntax Error:', command)  # if size does not match there has to be a
    syntax error with cmd
        elif 'SELECT' in command:
            if size >= 7:  # checks if the minimum amount of variables are present
                from_idx = l.index('from')
                script.select_specific(l[1:from_idx], l[from_idx+1].upper(), l[from_idx+2:])
            else:
                print('Syntax Error:', command)  # if size does not match there has to be a
    syntax error with cmd
        elif '.EXIT' in command:
            print('All Done')
            return
        else:  # if the command is not recognised it's and unknown command or there is
    something wring with the syntax
            print('Syntax Error | Unknown Command')
            print(command)

```

5.2 run_script.py

Contains all the functions that will execute the commands from the script.

5.2.1 create_database(self, db)

```
        """
        Joins the parent directory with the entered database name. Checks if the database
    already exists. If it exists
        already it will print out and error, if not it will create a directory of the
    database name.
        :param db: string that contains the name of the database
        :return: None
        """
        path = os.path.join(self.parentDir, db)  # joins cwd and db name
        if os.path.exists(path):  # check if path exists
            output = '!Failed to create database ' + db + ' because it already exists'
            print(output)
        else:
            os.mkdir(path)  # creates directory of path
            output = 'Database ' + db + ' created.'
            print(output)

```

5.2.2 drop_database(self, db)

```
        """
        Joins the parent directory with the entered database. Checks if the database already
     exists, and will either
        error out or delete that database.
        :param db: string that contains the name of the database
        :return: None
        """
        path = os.path.join(self.parentDir, db)  # check if path exists
        if os.path.exists(path):   # check if path exists
```

```
            cmd = 'rm ' + '-rf ' + path  # concatenate command to input
            os.system(cmd)  # runs the command
            output = 'Database ' + db + ' deleted.'
            print(output)
        else:
            output = '!Failed to delete ' + db + ' because it already exists.'
            print(output)

```

5.2.3 use_database(self, db)

```
    """
    Joins the parent directory with the entered database. Checks if the database already
    exists, and will either
    error out change the working directory to the database.
    :param db: string that contains the name of the database
    :return: None
    """
    path = os.path.join(self.parentDir, db)  # joins cwd and db name
    if os.path.exists(path):  # check if path exists
        os.chdir(path)  # changes cwd to this path
        self.dbDir = path
        output = 'Using database ' + db + '.'
        print(output)
    else:
        print('Cannot Use Database | Does Not Exist')

```

5.2.4 create_table(self, tbl, inp)

```
    """
    Gets the path to that table and will check if that already exists. Then if it does not
    exist, it will change the
    directory and call a helper function to append data to the table.
    :param tbl:
    :param inp: Contains all the data that will be entered into the table
    :return: None
    """
    path = os.path.join(self.dbDir, tbl)  # joins cwd and db name
    if os.path.exists(path):   # check if path exists
        output = '!Failed to create table ' + tbl + ' because it already exists.'
        print(output)
    else:
        os.mknod(path)  # creates file system of path
        out = inp.split(',')
        out = "|".join(out)
        f = open(path, "a")  # opens file
        f.write(out)  # write to file
        f.close()  # close file
        output = 'Table ' + tbl + ' created.'
        print(output)

```

5.2.5 select_all(self, table)

```
        """
        Checks if the table exists and then reads all the data from the file and prints it
    out.
        :param table: String that contains name of the table
        :return:
        """
        path = os.path.join(self.dbDir, table)  # joins cwd and db name
        if os.path.exists(path):  # check if path exists
```

```
            with open(path) as file_in:  # starts reading from file
                for line in file_in:
                    self.data.append(line.rstrip())
            for line in self.data:  # prints data to terminal
                print(line)
            self.data = []
        else:
            output = '!Failed to query table ' + table + ' because it does not exist.'
            print(output)

```

5.2.6 alter_table(self, tbl, inp)

```
        """
        Will check if the table exists, if it doesn't exist it will print out an error.
        If it exists it will then append the extra values to the file.
        :param tbl: String containing name of table
        :param inp: string that need to be inputted
        :return: None
        """
        path = os.path.join(self.dbDir, tbl)  # joins cwd and db name
        if os.path.exists(path):  # check if path exists
            out = inp.split(',')  # takes the string a separates all the values by comma's
    and storing it into a list
            out = "|".join(out)  # joins the list back together into a string with a '|' at
    value
            f = open(path, "a")  # opens file
            f.write('| ' + out)  # adds '|' to separate existing values and then writes the
    output string
            f.close()  # close file
            output = 'Table ' + tbl + ' modified.'
            print(output)
        else:
            output = '!Failed to alter table ' + tbl + ' because it does not exist'
            print(output)

```

5.2.7 drop_table(self, tbl)

```
        """
        Checks if the table exists and if that table exists it will delete that path. If it
    does not exist
        it will error and print the error message
        :param tbl: string that contains the name of the table
        :return: None
        """
        path = os.path.join(self.dbDir, tbl)  # check if path exists
        if os.path.exists(path):  # check if path exists
            cmd = 'rm ' + '-rf ' + path  # concatenate command to run
            os.system(cmd)  # runs the command
            output = 'Table ' + tbl + ' deleted.'
            print(output)
        else:
            output = '!Failed to delete ' + tbl + ' because it does not exists.'
            print(output)

```

5.2.8 insert_table(self, table, new_data)

```
        """
        Takes in the name of the table and inserts the new data to that data.
        :param table: string with name of the tale
        :param new_data: contains the new data that needs to be inserted
```

```
        :return: None
        """
        path = os.path.join(self.dbDir, table)
        if os.path.exists(path):  # check if path exists
            res = []
            data = self.read_all(path)  # reads all the current data from the file and stores it in a list
            for line in data:  # reads the lines and splits the data
                res.append(line.split('|'))

            new_data = "".join(new_data)[7:-2]  # gets rid of the excess data
            new_data = new_data.split(',')  # splits the data on commas
            res.append(new_data)  # adds the new list to the rest of the data
            self.insert_helper(path, res)  # calls the helper function to print the data to the file
            output = '1 New Record Inserted'
            print(output)
        else:
            output = '!Failed to insert into table ' + table + ' because it does not exist.'
            print(output)

```

5.2.9 insert_helper(self, path, inp)

```
        """
        Helper that writes the new list of values to the file
        :param path: Path that leads to the table file
        :param inp: the new data that needs to be written to the file
        :return: None
        """
        f = open(path, "a")  # opens file
        f.truncate(0)
        for line in inp:
            out = "|".join(line)
            out += '\n'  # adds the new line
            f.write(out)  # write to file
        f.close()  # close file

```

5.2.10 update_table(self, table, data)

```
        """
        Updates specific value if conditions match. Finds the index of the desired variable. Then it changes the
        variable that needs to be changed based on the index of the variable that needs to change.
        :param table: string with name of the table
        :param data: contains the data to update the table
        :return: None
        """
        path = os.path.join(self.dbDir, table)
        if os.path.exists(path):  # check if path exists
            var = data[1]  # gets the specific values that are in the list
            var2 = data[5]
            operation = data[2]
            changeto = data[3]
            change = data[7][:-1]
            new, count = self.update_helper(path, var, var2, operation, change, changeto)  # calls helper func. to get/
            # new data data and count of how many objects were updated
            self.insert_helper(path, new)  # calls helper function to print new data to file
            if count > 1:  # print plural of singular
                output = str(count) + ' Records Modified'
            else:
                output = str(count) + ' Records Modified'
```

```
        else:
            output = '!Failed to update table ' + table + ' because it does not exist.'
        print(output)

```

5.2.11 update_helper(self, path, var, var2, operation, change, changeto)

```
        """
        Helper function that finds all the indexes of the values that need to be change. it will compare those values
        using the operators specified and then update the specific value that matches up the the desired conditions.
        :param path: string that contains the path of the file
        :param var: contains the variable that needs to match
        :param var2: contains the variable that needs to change
        :param operation: the operation that needs to compare the values to satisfy the conditions.
        :param change: The variable that needs to be changed
        :param changeto: The variable that needs to be changed too
        :return:
        """
        data = self.read_all(path)  # reads in all the data from the current file
        res = [[data[0]]]  # creates the final list to be printed to file
        curr = self.read_all_list(data)  # turns all the lines into lists making it a list of lists
        count = 0  # initiates the count for the number of changes
        where_idx = self.find_idx(curr[0], var2)  # finds the index that needs to found
        set_idx = self.find_idx(curr[0], var)  # finds the index that equates to being changed
        if operation == '=':
            operation = '=='

        for line in curr[1:]:
            if operation == '==':
                if eval('line[where_idx]' + operation + 'change'):  # if the variable satisfies the condition to change
                    line[set_idx] = changeto  # changed the desired variable that needs to be changed
                    res.append(line)
                    count += 1
                else:
                    res.append(line)
            elif operation == '!=':  # checks if operation is not equal because of differences in float and strings
                if eval('line[where_idx]' + operation + 'change'):  # if the variable satisfies the condition to change
                    res.append(line)
                else:
                    line[set_idx] = changeto  # changed the desired variable that needs to be changed
                    res.append(line)
                    count += 1
            else:
                if eval('float(line[where_idx])' + operation + 'float(change)'):  # <, > can only compare numbers
                    line[set_idx] = changeto  # changed the desired variable that needs to be changed
                    res.append(line)
                    count += 1
                else:
                    res.append(line)

        return res, count

```

5.2.12 delete_items(self, table, data)

```
        """
        Takes in the table and deletes the desired rows that satisfy the conditions.
        :param table: string with name of table
        :param data: contains the data and conditions needed to delete a row
        :return: None
        """
        path = os.path.join(self.dbDir, table)
        if os.path.exists(path):  # check if path exists
            where = data[1]
            if data[2] == '=':  # turns it into and '==' to use as an operator
                operation = data[2] + data[2]
            else:
                operation = data[2]
            var = data[3][:-1]
            new, count = self.delete_helper(path, where, operation, var)  # calls helper to
    get new data and count
            self.insert_helper(path, new)  # calls helper to print new data to file
            if count >= 1:  # determines if singular or plural print
                output = str(count) + " Records Deleted"
            else:
                output = str(count) + " Record Deleted"
        else:
            output = '!Failed to delete items in table ' + table + ' because it does not
    exist.'
        print(output)

```

5.2.13 delete_helper(self, path, where, operation, var)

```
        """
        Helper to delete where it find the index of the variable that needs to compared.
    Then it will compare it with
        the operation specified to determine if that row needs to be deleted.
        :param path: string that has the path of the table
        :param where: the variable that needs to satisfy the condition
        :param operation: the logic operation
        :param var: the variable that needs to match the where to delete the row
        :return: the new list that needs to be printed and the count of the items deleted.
        """
        data = self.read_all(path)  # reads all the data that is stored in the current file
        res = [[data[0]]]  # initiates the new data with variable names
        curr = self.read_all_list(data)  # turns all the lines into a list
        where_idx = self.find_idx(curr[0], where)  # finds the index where the variable that
     needs to check is
        count = 0
        for line in curr[1:]:
            if operation == '==':  # checks if operation is equal because of differences in
    float and strings
                if eval('line[where_idx]' + operation + 'var'):  # eval turns string into a
    logic comparison
                    count += 1
                    continue
                else:
                    res.append(line)
            elif operation == '!=':  # checks if operation is not equal because of
    differences in float and strings
                if eval('line[where_idx]' + operation + 'var'):
                    res.append(line)
                else:
                    count += 1
            else:  # checks if operation is equal because of differences in float and
    strings
                if eval('float(line[where_idx])' + operation + 'float(var)'):  # greater or
    less than operations for num
```

```
                    count += 1
                    continue
                else:
                    res.append(line)
        return res, count

```

5.2.14 select_specific(self, var, table, where)

```
        """
        Selects a specific row depending on the where condition and the variables it wants to see.
        First deletes the the values that depend on the where clause. Then goes through and deletes the columns
        that are not needed to be seen.
        :param var: contains the variable for the where clause
        :param table: contains the name of the table in string
        :param where: contains the string of the conditions that needs to be satisfied
        :return:
        """
        obj = where[1]
        op = where[2]
        w = where[3][:-1]
        path = os.path.join(self.dbDir, table)
        if os.path.exists(path):  # check if path exists
            data = self.read_all(path)
            missing = []
            variables = self.get_curr_var(data[0])
            for i in range(len(var)):
                var[i] = var[i].replace(',', '')  # gets rid of all commas
            for v in variables:
                if v not in var:  # checks if the variable needs to be seen
                    missing.append(v)
            res = self.select_specific_helper(path, missing, data, obj, op, w)  # calls helper to delete rows and cols
            for line in res:
                print("|".join(line))
        else:
            output = '!Failed to select items in table ' + table + ' because it does not exist.'
            print(output)

```

5.2.15 select_specific_helper(self, path, missing, data, obj, op, w)

```
        """
        It will call the delete helper to get rid of rows that don't have the specific value.
        Finds the missing values that are not needed to be displayed. It will delete those columns that are not
        needed.
        :param path: String with Path to the table
        :param missing: Contains the list of values you want
        :param data: list that has the data from the file
        :param obj: the object that wants to be found
        :param op: the logic operator
        :param w: the values that needs to be compared with the operator
        :return: return the list that needs to be inputted into the file
        """
        res = []
        data, count = self.delete_helper(path, obj, op, w)  # delete specific rows that satisfy condition
        for line in data:
            new = "| ".join(line)
            res.append(new.split('| '))
```

```
        for miss in missing:
            where_idx = self.find_idx(res[0], miss)
            for line in res:
                del line[where_idx]  # deletes the column from all the lines
        return res

```

5.2.16 read_all(self, path)

```
        """
        Checks if the table exists and then reads all the data from the file and prints it
    out.
        :param path: String that contains path of the table
        :return: None
        """
        self.data = []
        if os.path.exists(path):  # check if path exists
            with open(path) as file_in:  # starts reading from file
                for line in file_in:
                    self.data.append(line.rstrip())
            return self.data
        else:
            output = '!Failed to query table because it does not exist.'
            print(output)


```

5.2.17 read_all_list(self, data)

```
        """
        Turns a string into a list
        :param data: Data with strings in list
        :return: return a list with the strings separated into lists
        """
        curr = []
        for line in data:
            curr.append(line.split('|'))
        return curr

```

5.2.18 get_curr_var(self, data)

```
        """
        Splits the input data and removes commas in list leaving only the variable names
        :param data: List of data from the file
        :return: A new list that contains only the variable names
        """
        res = []
        new = data.split('| ')  # splits into list by '|'
        for var in new:
            s = var.split(' ')  # splits into a list generated by spaces
            s[0] = s[0].replace(',', '')  # removes commas
            res.append(s[0])
        return res

```

5.2.19 find_idx(self, inp, var)

```
        for i in range(len(inp)):
            if var in inp[i]:
                return i
        return -1

```