

Criterion C: Development

List of Techniques Used

- Additional libraries
- Parsing a CSV file
- Database creation
- Inputting data into the database
- Data extraction from the database
- Joining databases
- Use of 2D arrays
- Recursive functions
- For loops
- Exception handling

Use of additional libraries

```
7 import sqlite3
8 import pathlib
9 from tabulate import tabulate # need to install prior to running
10 import sys
```

The library "sqlite3" was imported so databases could easily be created and handled.

"Pathlib" was imported to detect if the file containing databases already exists, if yes, databases won't be created, if no, databases will be created. This helps avoid the table already exists error.

"Tabulate" was imported to create nicely formatted tables which contain data for the user to clearly view.

"Sys" was imported so the program could be exited upon user request using "sys.exit()". Python already has a function to exit the program, but since that doesn't work for all python versions, it was better to exit using the "sys" library

Parsing a CSV file

The client provided a CSV file with information of people who run the club. Information from the file needs to be extracted and then inputted into the database(s).

```

61 def readData(FILENAME):
62     '''
63     read data from CSV file and extract into 2D array
64     :param FILENAME: the CSV file to be read
65     :return: (list) 2D array of data
66     '''
67     FILE = open(FILENAME)
68     CONTENT = FILE.readlines()
69     FILE.close()
70
71     # sanitize array
72     for i in range(len(CONTENT)):
73         CONTENT[i] = CONTENT[i].rstrip() # removes \n at the end
74         CONTENT[i] = CONTENT[i].split(",")
75
76     # remove headings
77     CONTENT.pop(0)
78
79     # change datatype to meet DB constraints
80     for i in range(len(CONTENT)):
81         CONTENT[i][0] = int(CONTENT[i][0]) # ID
82         CONTENT[i][2] = int(CONTENT[i][2]) # grade
83         CONTENT[i][6] = float(CONTENT[i][6]) # money_owed
84
85     return CONTENT

```

First the CSV file is opened in line 67, and then we use ".readlines()" as opposed to the ".read()" function. This is because ".readlines()" extracts data into a list, whilst the latter extracts data into a string. The file is then closed, and the list is converted into a **2D array** which is easily mutable. The client provided CSV file contains headings for the data, which is why the 1st (index 0) list in the 2D array was removed at line 77. Before returning, the data types are changed to either integer or float to match the database constraints.

Database Creation

Two databases were created in this program. The first one being the member database which stores all member information; the second one is the attendance database which stores the attendance records for each member.

```

87 def createMemberDB(DATA):
88     '''
89     create member info database and import DATA from CSV
90     :param DATA: (list) 2D array of data to import
91     :return: None
92     '''
93     global CURSOR, CONNECTION
94
95     # create DB
96     CURSOR.execute('''
97         CREATE TABLE
98             members(
99                 id INTEGER NOT NULL PRIMARY KEY,
100                 name TEXT NOT NULL,
101                 grade INTEGER NOT NULL,
102                 status TEXT NOT NULL,
103                 email TEXT NOT NULL,
104                 registered_paid TEXT NOT NULL,
105                 money_owed FLOAT NOT NULL,
106                 payment_method TEXT NOT NULL
107             )
108     ;''')
109
110     # import data
111     for i in range(len(DATA)):
112         CURSOR.execute('''
113             INSERT INTO
114                 members
115             VALUES(
116                 ?, ?, ?, ?, ?, ?, ?, ?
117             )
118             ;''', DATA[i])
119
120     CONNECTION.commit()

```

This shows the creation of the member database. As the client said he needs "all the member information," there are "not null" constraints to ensure each column is filled. The client also included that each member is identified by their unique ID's, which is why the "id" is given the constraint "primary key". The columns which will contain text are given the "text" constraint, likewise, columns containing integers are given the "integer" constraint. "Money_owed" will have a float input which is why the "float" constraint has been included there.

Immediately after data is extracted from the CSV file, it's added to the member database. To ensure data sanitization, there are "?"'s and parameters.

	A	B	C	D	E	F	G	H
1	Member id	Name	Grade	Status	Email	Registered/Paid	Money Owed	Payment Method
2	1	Matthew Swanson	12	President	mswanson@club	Yes	11.3	Cash
3	2	Angela Macintosh	12	President	amacintosh@club	Yes	11.3	E-transfer
4	3	William Taylor	11	Executive	wtaylor3@club	No	11.3	N/A
5	4	Andrew MacIsaac	11	Executive	amacisaac@club	No	11.3	N/A
6	5	Jeffrey Stone	11	Executive	jstone5@club	No	11.3	N/A

CSV File



Client provided CSV file extracted into the member DB

Member DB

ID	Name	Grade	Club Status	Email	Registered/Paid	Money Owed (\$)	Payment Method
1	<input type="text"/>	12	President	<input type="text"/>	Yes	11.3	Cash
2	<input type="text"/>	12	President	<input type="text"/>	Yes	11.3	E-transfer
3	<input type="text"/>	11	Executive	<input type="text"/>	No	11.3	N/A
4	<input type="text"/>	11	Executive	<input type="text"/>	No	11.3	N/A
5	<input type="text"/>	11	Executive	<input type="text"/>	No	11.3	N/A

Inputting data into the database

There are many instances in the program where at first inputs are collected from the user, and using those inputs, either both or one of the databases is updated.

```

241 def newMemberInfo(ID):
242     '''
243     get info on the new member to be added
244     :param ID: (int) member ID
245     :return: (list) of new member info
246     '''
247     NAME = input("New Member Name > ")
248     GRADE = checkInt(input("New Member Grade > "))
249     STATUS = input("New Member Club Status > ")
250
251     # get username and convert to email by adding domain
252     USERNAME = input("New Member EPSB Username > ")
253     EMAIL = USERNAME + "@share.epsb.ca"
254
255     REGISTERED_PAID = input("Has the member registered and paid? ")
256
257     MONEY_OWED = checkFloat(input("Money Owed By New Member ($) > "))
258
259     PAYMENT_METHOD = input("Payment Method > ")
260
261     INPUTS = [ID, NAME, GRADE, STATUS, EMAIL, REGISTERED_PAID, MONEY_OWED, PAYMENT_METHOD]
262
263     # check if inputs null or not to meet DB constraints
264     for i in range(len(INPUTS)):
265         if INPUTS[i] == "":
266             print("One or more pieces of information missing, please enter information again")
267             return newMemberInfo(ID)
268
269     return INPUTS

```

The image above shows the subroutine called upon to add the member. User inputs are taken to get information for the new member. After the data is inputted, an array is created with all user inputted data as well as the "ID", which is a parameter to this subroutine. To ensure the "Not Null" constraints of the member database are met, the variables are checked one by one using a **for loop** in line 264 to see if they have any data in them. If they do, there is no change, if not, an output message is displayed and this subroutine is called upon once again.

As this program ensures maximum efficiency, the user only needs to enter in the EPSB username of the member, which will be changed to an email at line 253.

```

32 def checkInt(NUM):
33     '''
34     checks if NUM is a number and converts to int
35     :param NUM: (str) user input
36     :return: (int) user input converted to integer
37     '''
38     if NUM.isnumeric():
39         NUM = int(NUM)
40         return NUM
41     else:
42         print('Enter a number please')
43         NEW_NUM = input("> ")
44         return checkInt(NEW_NUM)

```

To ensure the database constraint of "integer" for "GRADE", as soon as the user input is taken, a subroutine called "checkInt" is called upon, which checks if the input is an integer or not. If it is, the user input type is changed to integer and returned, if not, the user is asked to input the grade once again.

The class "checkInt" also uses **recursive functions** as the class calls upon itself if it's unable to convert the user input into a number.

```

46 def checkFloat(NUM):
47     '''
48     tries to convert NUM to float
49     :param NUM: (str) user input
50     :return: (float) user input converted to float
51     '''
52     try:
53         NUM = float(NUM)
54         return NUM
55     except ValueError:
56         print("This should be a number or decimal")
57         NEW_NUM = input("> ")
58         return checkFloat(NEW_NUM)

```

Likewise, "checkFloat" is called upon to ensure the database constraint of "float" for "MONEY_OWED".

This subroutine uses **exception handling** by trying to convert the user input to float, if a "ValueError" were to occur, it would take user input once again and try to convert it to float once again.

```

381 def addMember(INFO):
382     '''
383     using data from INFO add new member
384     :param INFO: (list) of data for new member
385     :return: None
386     '''
387     global CURSOR, CONNECTION
388
389     # add to memberDB
390     CURSOR.execute('''
391         INSERT INTO
392             members(
393                 id,
394                 name,
395                 grade,
396                 status,
397                 email,
398                 registered_paid,
399                 money_owed,
400                 payment_method
401             )
402         VALUES(
403             ?, ?, ?, ?, ?, ?, ?, ?
404         )
405     ;''', INFO)
406
407     # add ID to attendanceDB
408     CURSOR.execute('''
409         INSERT INTO
410             attendance(
411                 id
412             )
413         VALUES(
414             ?
415         )
416     ;''', [INFO[0]])
417
418     CONNECTION.commit()

```

After user input is taken, the program next calls on the subroutine "addMember". This subroutine first adds all the inputs from the list into the member database. Again, to ensure sanitization, "?"s and parameters are used. Then the ID is added into the attendance database.

Data extraction from the database

When the member database needs to be viewed, data needs to be first extracted then displayed.

```

496 def memberDB():
497     '''
498     extract information and display the member database
499     :return: None
500     '''
501     global CURSOR
502
503     # extract data
504     MEMBERS = CURSOR.execute('''
505         SELECT
506             *
507         FROM
508             members
509     ;''').fetchall()
510
511     # table columns (needed for tabulate table)
512     COLUMNS = ["ID", "Name", "Grade", "Club Status", "Email", "Registered/Paid", "Money Owed ($)", "Payment Method"]
513
514     # display extracted data
515     print(tabulate(MEMBERS, headers=COLUMNS, tablefmt='fancy_grid'))
516

```

Using the SQLite3 command "fetchall()", each row from the member database is extracted into an array. Then using the external library "tabulate", all data is displayed in a nicely formatted table.

Joining databases

The attendance database contained member ID's, but not names. When the attendance database needs to be viewed, it must also display member names.


```

517 def attendanceDB():
518     '''
519     join the member and attendance databases and display attendance database
520     :return: None
521     '''
522     global CURSOR
523
524     # join both DB's (to get member name since that needs to be displayed)
525     ATTENDANCE = CURSOR.execute('''
526         SELECT
527             attendance.id,
528             members.name,
529             sep,
530             oct,
531             nov,
532             dec,
533             jan,
534             feb,
535             mar,
536             apr,
537             may,
538             jun
539         FROM
540             attendance
541         JOIN
542             members
543         ON
544             attendance.id = members.id
545     ;''').fetchall()
546
547     COLUMNS = ["ID", "Name", "Sep", "Oct", "Nov", "Dec", "Jan", "Feb", "Mar", "Apr", "May", "June"]
548
549     print(tabulate(ATTENDANCE, headers=COLUMNS, tablefmt='fancy_grid'))

```

The databases "members" and "attendance" are joined at ID. Everything from the attendance database is selected, while the "name" from the member database is selected. The rows are then saved into an array then displayed with tabulate.

Use of 2D arrays

```

347 def takeattendance():
348     '''
349     print out members one by one asking if attended
350     :return: list of attendance for meet
351     '''
352     global CURSOR
353
354     # join both DB's to get member name and ID
355     MEMBERS = CURSOR.execute('''
356         SELECT
357             attendance.id,
358             members.name
359         FROM
360             attendance
361         JOIN
362             members
363         ON
364             attendance.id = members.id
365     ;''').fetchall()
366
367     ATTENDANCE_LIST = []
368
369     for i in range(len(MEMBERS)):
370         ATTENDED = input(f"Did {MEMBERS[i][1]} attend today's meeting? Y/n ")
371         if ATTENDED == "" or ATTENDED == 'y':
372             ATTENDED = "Y"
373         elif ATTENDED == "n":
374             ATTENDED = 'N'
375         ATTENDANCE_LIST.append([ATTENDED, MEMBERS[i][0]])
376
377     return ATTENDANCE_LIST

```

When "fetchall()" is used, a **2D array** is created with the member ID, and their names. This 2D array is then traversed outputting the member name and asking if the member has attended the meeting. To make it easier for the user, "Y" is made a default where they can press enter and it'll be changed to "Y". The input is then added to another **2D array** with the member ID to add into the attendance database.

Over here we can also see the use of **for loops** in line 369, where the **2D array** containing members and their respective ID's are traversed.

Word Count: 994