Criterion C: Development

List of Techniques Used:

- Additional libraries
- Database creation
- Inputting data into the database
- Data extraction from the database
- Parsing text
- Use of 2D arrays
- For loops
- Exception handling
- Data parsing

Additional Libraries

```
8  from flask import Flask, render_template, request, redirect
9  from pathlib import Path
10 import sqlite3
```

I used three additional libraries in this software. One of them was sqlite3. Sqlite3 is used to create and handle any databases that were to be created. Although an intense and intricate system of lists and arrays would have sufficed for this project, I decided to use a database feature to be able to store the data so that it can reappear if the user decides to rerun the program. It also ensures that if the user switches between different days, the tasks for any other day are not deleted and can be accessed again.

Flask is a library in python that is used to create a website in python. It links your HTML, CSS, and JavaScript files to the python code for manipulation. This type of software could easily have been created in just the python shell region on an IDE, removing the need to use Flask. However, for a program such as a to-do list, it is vital for the program to be intuitive. Incorporating a graphical user interface to the program was a better decision to provide easy navigation.

The Pathlib library was imported to detect if the file containing the databases already exists. If yes, the databases won't be created and if no, the databases will be created. The use of this library helped me avoid the "table already exists" error.

Database creation

```
79  def createTable():
80      global DB_NAME
81      CONNECTION = sqlite3.connect(DB_NAME)
82      CURSOR = CONNECTION.cursor()
83      CURSOR.execute("""
84          CREATE TABLE
85              tasks (
86                  task_name TEXT NOT NULL,
87                  task_day INTEGER NOT NULL,
88                  task_month TEXT NOT NULL,
89                  task_year INTEGER NOT NULL
90              )
91      ;""")
92      CONNECTION.commit()
93      CONNECTION.close()
```

The image above shows the database table “tasks” being created.

Since the program uses SQLite3 as its data storing module, it is important to define variables called CONNECTION which connects the SQLite library to your database. I also had to include a global variable which was the actual database file itself into my createTable() function. It is also required to define a CURSOR variable which allows you to perform the executive tasks in SQLite. After these have been defined, I used CURSOR.execute() to create a table in my SQLite database called “tasks”. Within this table I created 4 columns: the task name, task day, task month, and task year. The task_name and task_month are text values, whereas the text_day and text_year are integer values. This has been done intentionally to ensure that the program (or the user) doesn’t confuse the day with the month. After the CURSOR executes the creation of the table, it is important to commit changes to the SQLite database using CONNECTION.commit() and then to close the connection using CONNECTION.close(). This ensures that the function is executed with all systems closed to prevent any data leaks or errors.

Inputting data into the database

```
95  def createTask(TASK, DAY, MONTH, YEAR):
96      """
97      adds the task to the sql database
98      :param TASK: str
99      :param DAY: int
100     :param MONTH: str
101     :param YEAR: int
102     :return: sql
103     """
104     global DB_NAME
105     CONNECTION = sqlite3.connect(DB_NAME)
106     CURSOR = CONNECTION.cursor()
107     CURSOR.execute("""
108         INSERT INTO
109             tasks
110         VALUES (
111             ?, ?, ?, ?
112         )
113     ;""", [TASK, DAY, MONTH, YEAR])
114     CONNECTION.commit()
115     CONNECTION.close()
116
```

The image above represents the creation of an individual task and its insertion into the SQLite database.

The function createTask() takes in four parameters called “TASK”, “DAY”, “MONTH”, and “YEAR” respectively. This mimics the columns of the “tasks” table in the database. I use CURSOR.execute() to insert the task information into the “tasks” table in the database. For the values that the table is fed, I feed it values from a list that contains the same parameters of “TASK”, “DAY”, “MONTH”, and “YEAR”. This ensures that that data is directly being fed into the database and limits the possibility of an SQL injection attack.

Extracting data from the database

```
117 def getAllTasks(DAY, MONTH, YEAR):
118     """
119     Queries all the tasks to be displayed nicely
120     :param DAY: int
121     :param MONTH: str
122     :param YEAR: int
123     :return: 2D array
124     """
125     global DB_NAME
126     CONNECTION = sqlite3.connect(DB_NAME)
127     CURSOR = CONNECTION.cursor()
128     TASKS = CURSOR.execute("""
129         SELECT
130             *
131         FROM
132             tasks
133         WHERE
134             task_day = ?
135         AND
136             task_month = ?
137         AND
138             task_year = ?
139     ;""", [DAY, MONTH, YEAR]).fetchall()
140     CONNECTION.close()
141     return TASKS
```

This image shows how the program retrieves the tasks from the database to display on the HTML to-do list page.

The function getAllTasks() accepts parameters of “DAY”, “MONTH”, and “YEAR”. These parameters are inputted into the program when the user uses the HTML form to select their desired date. The function defines another variable called “TASKS” which essentially consists of a list of all the retrievals that the CURSOR retrieves from the database. The CURSOR selects all data from the database where the inputs match with the data and inserts it into “TASKS”. The function then returns TASKS which essentially is a 2D array.

Use of 2D arrays

```
128     TASKS = CURSOR.execute("""
129         SELECT
130             *
131         FROM
132             tasks
133         WHERE
134             task_day = ?
135         AND
136             task_month = ?
137         AND
138             task_year = ?
139     ;""", [DAY, MONTH, YEAR]).fetchall()
```

This image shows the usage of 2D arrays in the program.

In the getAllTasks() function, the .fetchall() command is used which retrieves all instances and puts it inside the list of TASKS, essentially creating a 2D array out of it. For example: a sample set of what TASKS could contain looks like this:
[(task1, day1, month1, year1), (task2, day2, month2, year2)...]
The task data itself is stored as a tuple because there is no function to edit a task. The only functions one can do with tasks are to add tasks and delete tasks. To ensure data integrity, I have made all tasks tuples so they can't be edited, but they can be deleted. The TASKS variable is a list of tuples. When iterating through the TASKS variable

Use of for loops

```
50          {% for task in tasks %}
51              <tr>
52                  <td>{{task[0]}}</td>
53                  <td><a href="/delete/{{task[0]}}/{{task[1]}}/{{task[2]}}/{{task[3]}}" class="btn btn-danger" role="button">Delete Task</a>
54              </tr>
55          {% endfor %}
```

The above image represents the use of for loops in the jinja code.
The HTML code displays a table of all the tasks. It loops the display of a table row that consists of two main elements. One of these elements is the name of the task. The task consists of a tuple and the first element is the name, therefore the part being printed is task[0] (index 0). Next, there is a button being printed that allows the user to delete the task. The ‘tasks’ variable is generated from the flask command ‘render_template’. In this command, a variable called ‘tasks’ is defined as the 2D array of tasks that the getAllTasks() python function returns. This ‘tasks’ variable then connects to the ‘tasks’ variable in jinja to display all the tasks onto the HTML to-do list page.

Exception handling

```
203 if __name__ == "__main__":
204     if FIRST_RUN:
205         createTable()
206     app.run(debug=True)
```

This image shows how the database table is only created if the program is run for the first time.
The goal for this feature is to only have the SQLite database create the table if it is the first time the program is being run. The program determines if it is the first time being run by using the pathlib library and checking if a .db file already exists in the project folder. If it already exists, then the app will run normally. If a .db file doesn’t exist, then it will create a table using the createTable() function.

Data parsing

```python
def parseDate(DATE):
    """
    parsing the date into 3 distinct variables
    :param DATE: yyyy-mm-dd
    :return: integer variables
    """
    DAY = DATE[8:]
    if DAY[0] == '0':
        DAY = DATE[-1]
    DAY = int(DAY)
    MONTH = DATE[5:7]
    if MONTH[0] == '0':
        MONTH = DATE[6]
    MONTH = int(MONTH)
    YEAR = int(DATE[:4])
    MONTH_TO_TEXT = {
        1: 'January',
        2: 'February',
        3: 'March',
        4: 'April',
        5: 'May',
        6: 'June',
        7: 'July',
        8: 'August',
        9: 'September',
        10: 'October',
        11: 'November',
        12: 'December'
    }
    MONTH = MONTH_TO_TEXT[MONTH]
    DATE = [DAY, MONTH, YEAR]
    return DATE
```

The above image shows how the program uses a function to parse the date that the user inputs from the HTML form.

When the user selects the date using the HTML form, it is generated in all numerical values in the format of yyyy-mm-dd. For example if the user selects March 5, 2023, then the program will register it as 2023-03-05. This is a confusing format for the file paths as well as the user. Therefore, a function that I created called parseDate() rectifies that situation. It accepts a text parameter of DATE which is the date that the HTML form generates. It then divides that into three distinct variables called DAY, MONTH, and YEAR. Using index slicing, it divides the values into those three variables. I decided to convert the month into a text format to very clearly separate it from the day to remove confusing instances. I did this by passing the numerical month value through a dictionary to generate a text based format for the date. I then organized the three variables into a list in a variable called DATE and the function returns that list.
For example, instead of 2023-03-05, it will register as [5, ‘March’, 2023].

Word Count: 1114