Criterion C: Development

List of Techniques Used:

1. Additional libraries
2. Saving a .csv file from a file input
3. Parsing a .csv file
4. Database creation
5. Inputting data into a database
6. Data extraction from a database
7. Searching for data within a database
8. Exception handling
9. For loops
10. 2D arrays

Additional Libraries:

```
5   # Need to download Jinja and Flask beforehand
6   from flask import Flask, render_template, request
7   from werkzeug.utils import secure_filename # security measure for file upload
8   from pathlib import Path
9   import os
10  import sqlite3
```

The “Flask” library was imported to connect the program to a web page and render webpages and requests from those webpages.

The “secure_filename” function from “werkzeug.utils” was imported to check for potentially malicious files in file uploads. This helps to secure the program from malware.

“Pathlib” was imported to check if the database file already exists, which helps avoid a file exists error from the database.

“Os” was imported to save files to the program folder as well as check if files are in the correct directory. Since “Pathlib” is less compatible with older versions of Python, “os” was a better option for most of the program.

The library “sqlite3” was imported so that data could be easily organized and handled from databases.

Saving a .csv File from a File Input:

The “index.html” file prompts the user to upload .csv files, which are then saved to the project folder.

```
29  @app.route("/", methods=["GET", "POST"])
30  def index():
31      """
32      renders the index.html file in flask, uploads files into file folder
33      :return: renders file
34      """
35      global FIRSTRUN, REGULARFILENAME, OVERTIMEFILENAME, PRODUCTIONFILENAME, SALESFILENAME, SUMMARYFILENAME, TOTALFILENAME
36      ALERT = ""
37      if request.method == 'POST':
38          # inputs #
39          REGULARFILE = request.files['inputRegularHours']

45          # processing #
46          if REGULARFILE.filename == '' or OVERTIMEFILE.filename == "" or SALESFILE.filename == "" or PRODUCTIONFILE.
            filename == "" or SUMMARYFILE.filename == "" or TOTALFILE.filename == "":
47              ALERT = "Please select all files!"
48          elif allowed_file(REGULARFILE.filename) and allowed_file(OVERTIMEFILE.filename) and allowed_file(SALESFILE.
            filename) and allowed_file(PRODUCTIONFILE.filename) and allowed_file(SUMMARYFILE.filename) and allowed_file
            (TOTALFILE.filename):
49
50              # checks if all files have been selected and uploads files
51              REGULARFILENAME = secure_filename(REGULARFILE.filename)
52              REGULARFILE.save(os.path.join(app.config['UPLOADFOLDER'], REGULARFILENAME))
```

The function first checks if the form has been submitted, then it checks if all files have been uploaded using the “request.files” function. If not all files have been uploaded, an alert displays that tells the user to select all files. If all files have been uploaded, then the program checks if the file extension is either “.txt” or “.csv” using the “allowed_file” function. This is another way to check for malware and confirms that the uploaded file can be parsed for data.

If the file passes all these checks, the program gets the file name, using the "secure_filename" function to check it for malware, and saves the file to the program folder. This process is repeated for each file and then the files are parsed for use in the calculator.

Parsing a .csv File:

```
192     REGULARFILENAME = open(os.path.join(app.config['UPLOADFOLDER'], REGULARFILENAME))
193     REGULARDATA = REGULARFILENAME.readlines()
```

```
205         # REGULAR HOURS DATA
206         for i in range(len(REGULARDATA)):
207             if REGULARDATA[i][-1] == "\n":
208                     REGULARDATA[i] = REGULARDATA[i][:-1]
209             REGULARDATA[i] = REGULARDATA[i].split(",")
210             for j in range(len(REGULARDATA[i])):
211                 if checkFloat(REGULARDATA[i][j]):
212                     REGULARDATA[i][j] = float(REGULARDATA[i][j])
```

```
269     return REGULARDATA, OVERTIMEDATA, SUMMARYDATA, TOTALDATA, PRODUCTIONDATA, SALESDATA
```

The "extractFiles" function parses the .csv files so they can be inputted into the database. First, each file is opened using the file path to reduce errors, and then the "readlines()" dot function is used to extract the data from the .csv file into a list. This is used instead of the "read()" function because the "read()" function extracts data into a string, which is not as easily modifed. The "\n" character is removed from the end of each line before the list is converted to a **2D array** using the "split()" dot function so that it is easily mutable.

Each item in the list is then checked to see if it can be converted to a float using the following function.

```
130   def checkFloat(VALUE) -> bool:
131       """
132       checks if a string contains a float
133       :param VALUE: str
134       :return: bool
135       """
136       try:
137           float(VALUE)
138           return True
139       except ValueError:
140           return False
```

This function uses **exception handling** to test if the string contains a float. If the string contains a float, the function returns true, and if a "ValueError" exception were to occur, then the function returns false.

If the item passes the check, then it is converted to a float. The "float" data type is used over the "integer" data type because the wage calculation requires precision.

Database Creation:

```
307       # create multiple tables for each row of data each time
308       for i in range(1, len(REGULARDATA[0])-1):
309           REGULARDATA[0][i] = checkTitle(REGULARDATA[0][i])
310           CURSOR.execute(f"""
311               CREATE TABLE
312                   {REGULARDATA[0][i]} (
313                       member_name TEXT NOT NULL PRIMARY KEY,
314                       {REGULARDATA[0][i]} TEXT NOT NULL
315                   );
316           """)
```

Multiple tables were created in the database file for the columns in the data using **for loops**. This is because the number of columns in each of the .csv files varies and they all have different names. The headers in the .csv files will become the table name and the name of the second column in the database. Before creating the table, each column name needs to be checked to see if they contain malicious data using the "checkTitle" function.

Because the client wants all the data to be in the database, "not null" constraints are used. Since all data connects to the member names, "member_name" is set as the primary key. Columns that contain text are given the "TEXT" constraint.

Inputting Data into a Database:

```
317             for j in range(1, len(REGULARDATA)):
318                 CURSOR.execute(f"""
319                     INSERT INTO
320                         {REGULARDATA[0][i]}
321                     VALUES (
322                         ?,
323                         ?
324                     );
325                 """, [REGULARDATA[j][0], REGULARDATA[j][i]])

452         CONNECTION.commit()
```

The function uses **for loops** to insert each row of data that was extracted from the .csv file into the table. The for loop starts from 1 instead of 0 because the first row of data is the header. The information is inserted into the database using "?" for sanitization to reduce the risk of SQL injection attacks. Lastly, the "CONNECTION.commit()" function saves the changes to the database.

Data Extraction from a Database:

```
454   def calculateWages() -> list:
455       """
456       calculates percentage wages for all members in the database
457       :return: list (each members wages in order)
458       """
459       global DBNAME
460       CONNECTION = sqlite3.connect(DBNAME)
461       CURSOR = CONNECTION.cursor()
462
463       # fetch all data from the database
464       TOTALHOURS = CURSOR.execute("""
465           SELECT
466               *
467           FROM
468               total_hours;
469       """).fetchone()
470
471       MEMBERREGULAR = CURSOR.execute("""
472           SELECT
473               *
474           FROM
475               regular_hours;
476       """).fetchall()
```

In order to calculate the wages, data needs to be extracted from the database so that it can be manipulated. The “fetchone()” dot function is used to fetch one item from the table and the “fetchall()” dot function is used to extract all items from the table into a **2D array** which can then be used to calculate the wages.

Searching for Data within a Database:

The program uses the "queryWages" function to search for a member's wage information in the database. First the "checkName" function is used to check if the name exists in the database.

```
142   def checkName(NAME) -> bool:
143       """
144       checks if the name is in the database
145       :param NAME: str
146       :return: bool
147       """
148       CONNECTION = sqlite3.connect(DBNAME)
149       CURSOR = CONNECTION.cursor()
150
151       try:
152           WAGE = CURSOR.execute(f"""
153               SELECT
154                   percent_wages
155               FROM
156                   wages
157               WHERE
158                   member_name = ?;
159           """, [NAME]).fetchone()
160           if checkFloat(WAGE[0]):
161               return True
162           else:
163               return False
164       except TypeError:
165           return False
```

This function uses **exception handling** in order to see if the name is in the database. If the member name can be queried from the database, and the wage is a float, then the function returns true. However, if there is a "TypeError" exception, the function returns false.

```
627    def queryWages(NAME) -> None:
628        """
629        queries the wages table for a members wages
630        :param NAME: str
631        :return: None
632        """
633        CONNECTION = sqlite3.connect(DBNAME)
634        CURSOR = CONNECTION.cursor()
635
636        WAGE = CURSOR.execute("""
637            SELECT
638                percent_wages
639            FROM
640                wages
641            WHERE
642                member_name = ?;
643        """, [NAME]).fetchone()
```

The "queryWages" function uses the "WHERE" operator to filter the database so that it only contains the wage information of that member. Then it uses the "fetchone()" dot function because only one value should exist for each name. The function uses "?" in place of the member name so that the data can be santizied; this also reduces the risk of SQL injection attacks.

Word Count: 931