List of Techniques Used

- Additional Libraries
- Creating a database
- Normalization
- Inserting data into a database
- Reading data in a database
- Editing data in a database
- Deleting data in a database
- Multidimensional Arrays
- Joining tables
- Parallel arrays
- Parsing an URL link
- Front End Development, using HTML with CSS customization

Additional libraries

Algorithm Portion:

```
8  import sqlite3
9  from itertools import combinations
```

Web Application Portion:

```
6  # Import flask library
7  from flask import Flask, render_template, request, redirect
8  # Import file explorer library
9  from pathlib import Path
```

Using databases allows the client to save clothes and outfits. The library “sqlite3” is imported to create, read, update, and delete data on the database in Python.

The function “combinations” from the library “itertools” generates combinations with a specific length (“itertools.combinations() module in Python to print all possible combinations”). The following code generates all the combinations of accessories for generating outfits with length 0 to 5 from a preexisting list.

```
1038    # Find combinations of accessories
1039    for COMBINATION_LENGTH in range(min(len(ACCESSORIES) + 1, 5)):  # combinations from 0 length to 5 length
1040        ACCESSORIES_COMBINATIONS += list(combinations(ACCESSORIES, COMBINATION_LENGTH))
```

“Flask” implements Python on a web application with HTML. Using Flask is necessary for displaying data on a web application and for processing the inputs of the user on the web application.

“Pathlib” is a file explorer library, and checks if a database file already exists, and if not, to create one.

Creating a Database

I used databases because they prevent data anomalies; for example, their atomicity ensures that transactions happen fully or not at all, increasing integrity. The program will create a database if it does not exist already. Two important tables in this program are the clothing and outfit table. The first stores information about each piece of clothing, while the second stores the items for outfits. Their creation is found below.

Clothing:

```
23      CURSOR.execute("""
24      CREATE TABLE Clothing (
25          Clothing_ID INT PRIMARY KEY,
26          Name TEXT NOT NULL,
27          Color1 TEXT NOT NULL,
28          Style1 TEXT NOT NULL,
29          Fabric1 TEXT NOT NULL,
30          Weather INT NOT NULL,
31          Score INT NOT NULL,
32          Link TEXT NOT NULL,
33          Type TEXT NOT NULL
34      )
35      ;""")
```

Outfits:

```
60      CURSOR.execute("""
61      CREATE TABLE Outfits (
62          Outfit_ID INT PRIMARY KEY,
63          Name TEXT NOT NULL,
64          Top INT NOT NULL,
65          Bottom INT NOT NULL,
66          Shoes INT NOT NULL,
67          Comment INT NOT NULL,
68          Rating INT NOT NULL
69      )
70      ;""")
```

Normalization

All the tables in the program are in 3rd Normal Form (3NF), where all the columns cannot be empty, all columns relate to the primary key, and no columns are dependent on each other. Through implementing 3NF, SQL queries prevent data anomalies.

For the clothing, information about the name, colors, styles, fabrics, weather, comment, and a link to image are needed. These all relate to the clothing, so I made Clothing_ID the primary key. However, not every piece of clothing has multiple colors, whereas for others, each color is important. To implement 3NF, additional colors, styles, and fabrics are in separate tables, connected to the clothing table by their primary key, and created as so:

```
23      CURSOR.execute("""
24      CREATE TABLE Clothing (
25          Clothing_ID INT PRIMARY KEY,
26          Name TEXT NOT NULL,
27          Color1 TEXT NOT NULL,
28          Style1 TEXT NOT NULL,
29          Fabric1 TEXT NOT NULL,
30          Weather INT NOT NULL,
31          Score INT NOT NULL,
32          Link TEXT NOT NULL,
33          Type TEXT NOT NULL
34      )
35      ;""")
```

```
    CURSOR.execute("""
    CREATE TABLE Additional_Color_2 (
        Clothing_ID INT PRIMARY KEY,
        Data TEXT NOT NULL
    )
    ;""")
    CURSOR.execute("""
    CREATE TABLE Additional_Color_3 (
        Clothing_ID INT PRIMARY KEY,
        Data TEXT NOT NULL
    )
    ;""")
    CURSOR.execute("""
    CREATE TABLE Additional_Style_2 (
        Clothing_ID INT PRIMARY KEY,
        Data TEXT NOT NULL
    )
    ;""")
    CURSOR.execute("""
    CREATE TABLE Additional_Fabric_2 (
        Clothing_ID INT PRIMARY KEY,
        Data TEXT NOT NULL
    )
    ;""")
```

Inserting Data into a Database

Data is inserted into the database multiple times in the program. This is one example:

```
def insertNewClothing(CLOTHING_INFORMATION):
    """
    Insert new clothing into the database
    :param CLOTHING_INFORMATION: list
    :return: none
    """
    global DATABASE_NAME
    CONNECTION = sqlite3.connect(DATABASE_NAME)
    CURSOR = CONNECTION.cursor()
    # FIND PRIMARY KEY
    # Find most recent primary key
    CLOTHING_PRIMARY_KEY = recentClothingID() + 1

    # Find information that is always not null
    NON_EMPTY_INFORMATION = [
        CLOTHING_PRIMARY_KEY,
        CLOTHING_INFORMATION[0],
        CLOTHING_INFORMATION[1][0],
        CLOTHING_INFORMATION[2][0],
        CLOTHING_INFORMATION[3][0],
        CLOTHING_INFORMATION[4],
        CLOTHING_INFORMATION[5],
        CLOTHING_INFORMATION[6],
        CLOTHING_INFORMATION[7],
    ]

    # Find information that may be null
    OPTIONAL_INFORMATION = [
        ["Additional_Color_2", CLOTHING_INFORMATION[1][1]],
        ["Additional_Color_3", CLOTHING_INFORMATION[1][2]],
        ["Additional_Style_2", CLOTHING_INFORMATION[2][1]],
        ["Additional_Fabric_2", CLOTHING_INFORMATION[3][1]]
    ]

    CURSOR.execute("""
        INSERT INTO
            Clothing
        VALUES (
            ?, ?, ?, ?, ?, ?, ?, ?, ?
        )
    ;""", NON_EMPTY_INFORMATION)

    # Get other colors if they exist
    for INFORMATION in OPTIONAL_INFORMATION:
        if not (INFORMATION[1] is None or INFORMATION[1] == ""):
            CURSOR.execute(f"""
            INSERT INTO
                {INFORMATION[0]}
            VALUES (
                ?, ?
            )
            ;""", [CLOTHING_PRIMARY_KEY, INFORMATION[1]])
    CONNECTION.commit()
    CONNECTION.close()
```

The above function inserts clothing into the database. First, it generates a primary key, to insert into additional tables if needed. Then, it finds data that is always filled, and inserts it into the Clothing table. After, it finds the optional information and puts it into a list. To reduce code, I used a for loop to insert the optional information and used an f string to find the tables to insert into. Although this method does not sanitize the data, the user does not interact with the table fields. For fields the user does input, I used “?” for data sanitization, preventing SQL injections.

Reading Data in a Database

```
477     # Find information that is always filled
478     FILLED_INFORMATION = CURSOR.execute("""
479     SELECT
480         Clothing_ID,
481         Name,
482         Color1,
483         Style1,
484         Fabric1,
485         Weather,
486         Score,
487         Link,
488         Type
489     FROM
490         Clothing
491     WHERE
492         Clothing_ID = ?
493     ;""", [CLOTHING_PRIMARY_KEY]).fetchone()
```

This code reads information from the main clothing table for one piece of clothing. The SQlite function “fetchone()” returns one row from the table . One use of this function is to display the clothing information on the website.

Editing Data in a Database

```
658     # Update
659     CURSOR.execute("""
660         UPDATE
661             Clothing
662         SET
663             Name = ?,
664             Color1 = ?,
665             Style1 = ?,
666             Fabric1 = ?,
667             Weather = ?,
668             Score = ?,
669             Link = ?,
670             Type = ?
671         WHERE
672             Clothing_ID = ?
673     ;""", NON_EMPTY_INFORMATION)
```

The above query updates all the information for clothing in the main table with the information in NON_EMPTY_INFORMATION, allowing the user to change information in the database.

Deleting Data in a Database

```
445     # Main clothing
446     CURSOR.execute("""
447         DELETE FROM
448             Clothing
449         WHERE
450             Clothing_ID = ?
451     ;""", [CLOTHING_ID])
```

The above query deletes a row on the main clothing table with the primary key.

Multiple Dimensional Arrays

In this program, many multi dimensional lists were used. One usage is to store outfits. There is a list to store every outfit, a list containing each item's ID.

```
1237 def getAllOutfits():
1238     """
1239     Get all outfits in the database
1240     :return: list
1241     """
1242     global DATABASE_NAME
1243     CONNECTION = sqlite3.connect(DATABASE_NAME)
1244     CURSOR = CONNECTION.cursor()
1245 
1246     # Call all the outfits
1247     OUTFITS = CURSOR.execute("""
1248     SELECT
1249         Outfit_ID,
1250         Name,
1251         Top,
1252         Bottom,
1253         Shoes,
1254         Comment,
1255         Rating
1256     FROM
1257         Outfits
1258     ORDER BY
1259         Rating DESC
1260     ;""").fetchall()
1261 
1262     FULL_OUTFITS = []
1263     # Find side tables
1264     for OUTFIT in OUTFITS:
1265         ID = OUTFIT[0]
1266         # find side table information
1267         FILLED, OPTIONAL = getExistingOutfitInfo(ID)
1268         OUTFIT_COPY = list(OUTFIT)
1269         OUTFIT_COPY.insert(5, OPTIONAL[0])
1270         OUTFIT_COPY.insert(6, OPTIONAL[1])
1271         OUTFIT_COPY.insert(7, OPTIONAL[2:])
1272         FULL_OUTFITS.append(OUTFIT_COPY)
1273     CONNECTION.close()
1274     return FULL_OUTFITS
```

This function uses the SQLite function “fetchall()”, which gets every qualified row from a table, where each row is a list within another list. Then, it appends the optional information, like additional colors, for each outfit.

Joining Tables

```
885         COLOR_2_CLOTHING = CURSOR.execute("""
886             SELECT
887                 Additional_Color_2.Clothing_ID,
888                 Clothing.Type
889             FROM
890                 Additional_Color_2
891             JOIN
892                 Clothing
893             ON
894                 Additional_Color_2.Clothing_ID = Clothing.Clothing_ID
895             WHERE
896                 Additional_Color_2.Data = ?
897         ;""", [COLOR]).fetchall()
```

I needed a way to select the id of a piece of clothing with its type. This is to find specific shoes or jackets, with a specific color, to make an outfit. The above query gets the color with the type by joining tables together where they have the same Clothing_ID, since those pieces of information are in separate tables/

Parallel Arrays

Parallel arrays are arrays where the data in each index is related between them. Outfits contain information like the accessories and sweaters, which are stored in parallel arrays, allowing me to design algorithms to loop through each list and get related information.

```
1138        # Find optional information: [sweater, jacket, accessories]
1139        OPTIONAL_INFORMATION = [
1140            NEW_OUTFIT_INFORMATION[4],
1141            NEW_OUTFIT_INFORMATION[5],
1142        ]
1143
1144        # For accessories to be on one line
1145        for ACCESSORY in NEW_OUTFIT_INFORMATION[6]:
1146            if not (ACCESSORY is None or ACCESSORY == "" or ACCESSORY == 0):
1147                OPTIONAL_INFORMATION.append(ACCESSORY)
1148            else:
1149                # Make them none as placeholder
1150                OPTIONAL_INFORMATION.append(None)
```

This code shows the creation of a parallel array to store information about the non-essential elements of outfits. This list is parallel to this tuple:

```
1356 OPTIONAL_OUTFIT_DATA = ("Additional_Sweater", "Additional_Jacket", "Additional_Accessory_1",
1357                         "Additional_Accessory_2", "Additional_Accessory_3",
1358                         "Additional_Accessory_4", "Additional_Accessory_5")
```

```
    for i in range(len(OPTIONAL_OUTFIT_DATA)):
        if OPTIONAL_INFORMATION[i] is not None and OPTIONAL_INFORMATION[i] != "" and OPTIONAL_INFORMATION[i] != 0:
            # columns exist
            if EXISTING_OPTIONAL_INFORMATION[i] is not None:
                # Regular update
                CURSOR.execute(f"""
                    UPDATE
                        {OPTIONAL_OUTFIT_DATA[i]}
                    SET
                        Clothing_ID = ?
                    WHERE
                        Outfit_ID = ?
                    ;""", [OPTIONAL_INFORMATION[i], OUTFIT_PRIMARY_KEY])
            else:  # columns do not exist
                # Have to insert
                CURSOR.execute(f"""
                    INSERT INTO
                        {OPTIONAL_OUTFIT_DATA[i]} (
                            Outfit_ID, Clothing_ID
                        )
                    VALUES (
                        ?, ?
                    )
                    ;""", [OUTFIT_PRIMARY_KEY, OPTIONAL_INFORMATION[i]])
        else:  # no information provided for extra fields
            # columns exist
            if EXISTING_OPTIONAL_INFORMATION[i] is not None:
                CURSOR.execute(f"""
                    DELETE FROM
                        {OPTIONAL_OUTFIT_DATA[i]}
                    WHERE
                        Outfit_ID = ?
                    ;""", [OUTFIT_PRIMARY_KEY])
            # else, nothing happens
```

Together, it allows me to iterate through the accessories with a for loop, reducing redundancy. The above for loop updates tables containing additional items in outfits, which may involve inserting, updating, or deleting data. With this technique, I can access the table name from one array and the primary key from the other with one index.

Parsing URL Links

On my website, I had multiple pages on my website, and I had to pass information between pages. For example, to get from the outfit page to the editing outfit page, information about the outfit needs to be passed, through URLS, for the form to display existing information. However, URL links are strings, so the program needs to parse the string into a list.

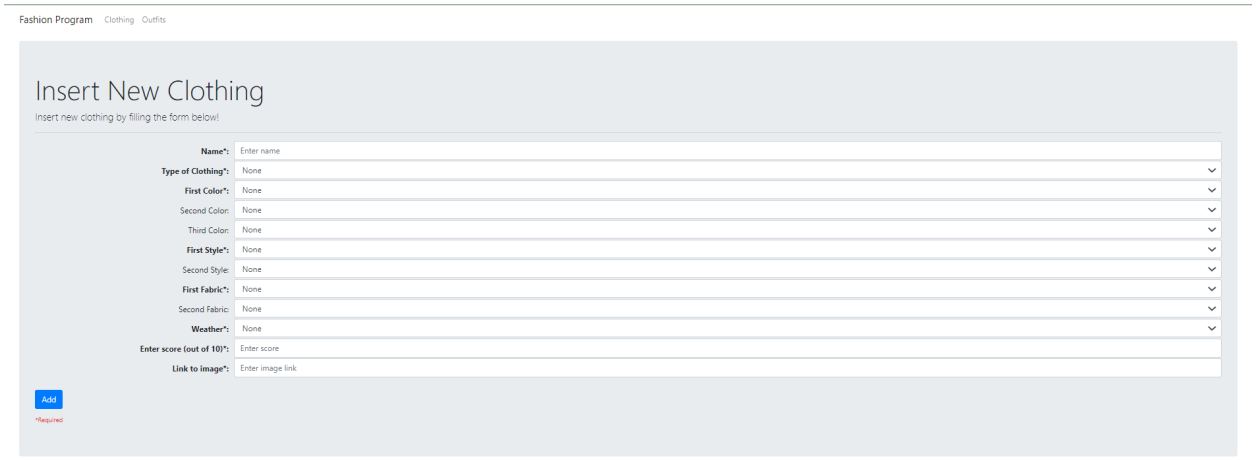
```
228 def parseChosenWithID(ID_and_chosen):
229     """
230     For parsing when there is an id as well
231     :return:
232     """
233     # Parse
234     # remove square brackets
235     print("raw", ID_and_chosen)
236     ID_and_chosen = ID_and_chosen.replace("[", "")
237     ID_and_chosen = ID_and_chosen.replace("]", "")
238     # split id and chosen
239     ID_and_chosen = ID_and_chosen.split(", ")
240     # find id
241     ID = int(ID_and_chosen[0])
242     # Get rid of first item (the id)
243     ID_and_chosen.pop(0)
244     # Make "None" into None
245     for i in range(len(ID_and_chosen)):
246         if ID_and_chosen[i] == "None":
247             ID_and_chosen[i] = None
248         # Get rid of quotations for names and comments
249         if ID_and_chosen[0] is not None:
250             if ID_and_chosen[0][0] == "'" or ID_and_chosen[0][0] == '"':
251                 ID_and_chosen[0] = ID_and_chosen[0][1:]
252             if ID_and_chosen[0][-1] == "'" or ID_and_chosen[0][-1] == '"':
253                 ID_and_chosen[0] = ID_and_chosen[0][:-1]
254         if ID_and_chosen[-2] is not None:
255             if ID_and_chosen[-2][0] == "'" or ID_and_chosen[-2][0] == '"':
256                 ID_and_chosen[-2] = ID_and_chosen[-2][1:]
257             if ID_and_chosen[-2][-1] == "'" or ID_and_chosen[-2][-1] == '"':
258                 ID_and_chosen[-2] = ID_and_chosen[-2][:-1]
259
```

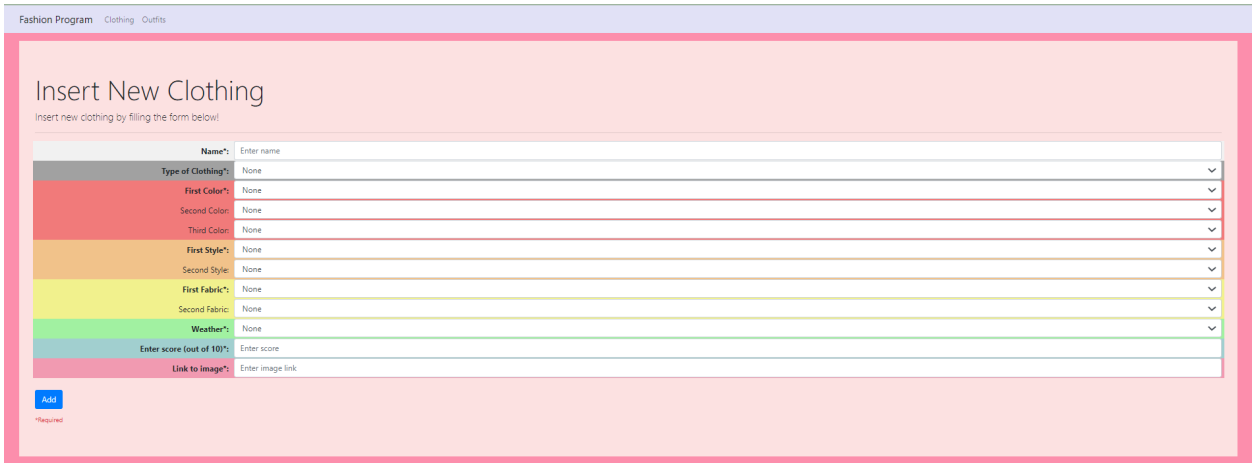
```
    # Regroup accessory list
    ACCESSORIES = ID_and_chosen[6:-2]

    ID_and_chosen[6] = ACCESSORIES
    while len(ID_and_chosen) > 9:
        ID_and_chosen.pop(7)

    OUTFIT_CHOSEN_CLOTHES = []

    i = 0
    for INFO in ID_and_chosen:
        if type(INFO) == list:
            OUTFIT_CHOSEN_CLOTHES.append([])
            for SUB_INFO in INFO:  # for the accessory list
                if SUB_INFO == "None" or SUB_INFO == "":
                    OUTFIT_CHOSEN_CLOTHES[i].append(None)
                elif type(SUB_INFO) == str:
                    if SUB_INFO.isnumeric():
                        OUTFIT_CHOSEN_CLOTHES[i].append(int(SUB_INFO))
                    else:
                        OUTFIT_CHOSEN_CLOTHES[i].append(SUB_INFO)
                else:
                    OUTFIT_CHOSEN_CLOTHES[i].append(SUB_INFO)
        elif INFO == "[]":
            OUTFIT_CHOSEN_CLOTHES.append([])
        elif INFO == "None" or INFO == "":
            OUTFIT_CHOSEN_CLOTHES.append(None)
        elif type(INFO) == str:
            if INFO.isnumeric():
                OUTFIT_CHOSEN_CLOTHES.append(int(INFO))
            else:
                OUTFIT_CHOSEN_CLOTHES.append(INFO)
        else:
            OUTFIT_CHOSEN_CLOTHES.append(INFO)
        i += 1
    return ID, OUTFIT_CHOSEN_CLOTHES
```

In this function, an outfit list as a string is the argument “ID_and_chosen”. This list contains the outfit ID with the information about the outfit. First, it removes square brackets, and splits per comma. The first item is the outfit id, which is stored and removed from the list.

Then, unnecessary quotation marks are removed. Next, the accessories are grouped into one single list. Finally, the data is type casted into integers where applicable.

Front End Development





For the front-end, I used HTML/CSS with bootstrap. Creating a website makes the user experience friendlier, like dropdowns in forms and being able to see pictures of the clothing. Bootstrap hastened the web development process by providing templates and shapes, like the navbar and containers. To give the website color, I created my own CSS stylesheet that overrides bootstrap.

For the back-end, I used Flask and Jinja, which integrates Python into a web application.

Word Count: 1017