Criteria C: Development

Table of Contents

1. Libraries used

A collection of linked modules is frequently referred to as a Python library. It includes code packages that may be utilized by several programs. A library might also include configuration information, message templates, classes, values, and other things aside from pre-compiled scripts. These external libraries can be imported into any program and its features can be used (Parthmanchanda81, 2021). Therefore, I used the following libraries in the development of my application.

```
# Import the sqlite3 module for working with SQLite databases
import sqlite3

# Import necessary modules for the GUI interface
from tkinter import *
from tkinter import messagebox
import tkinter as tk

# Import the matplotlib module for creating plots
import matplotlib.pyplot as plt

# Import the tempfile module for working with temporary files
import tempfile

# Import the sys module for interacting with the Python interpreter
import sys

# Import the subprocess module for running external commands
import subprocess
```

Figure 1: Screenshot of the libraries used

- sqlite3: a library for working with SQLite databases.

- tkinter: a library for creating GUI applications.

- messagebox: a module within tkinter that provides a simple way to display message boxes.

- tk: an alias for tkinter.

- matplotlib.pyplot: a library for creating data visualizations.

- tempfile: a library for creating temporary files and directories.

- sys: a library that provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

- subprocess: a library for spawning new processes, connecting to their input/output/error pipes, and obtaining their return codes.

2. Database

The SQLite Database was used and was implemented in the program using the ‘sqlite3’ library. Two databases were created and connected to store data one for the expense tracker and another for the buying list.

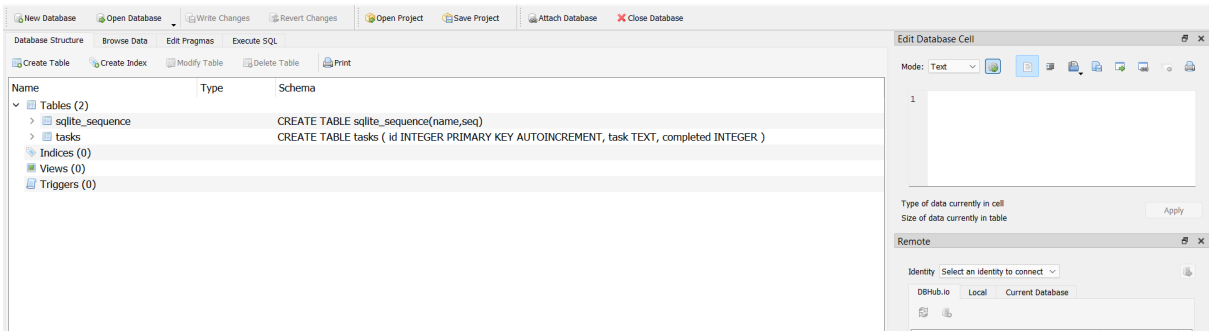

Figure 2: Screenshot of the tasks database

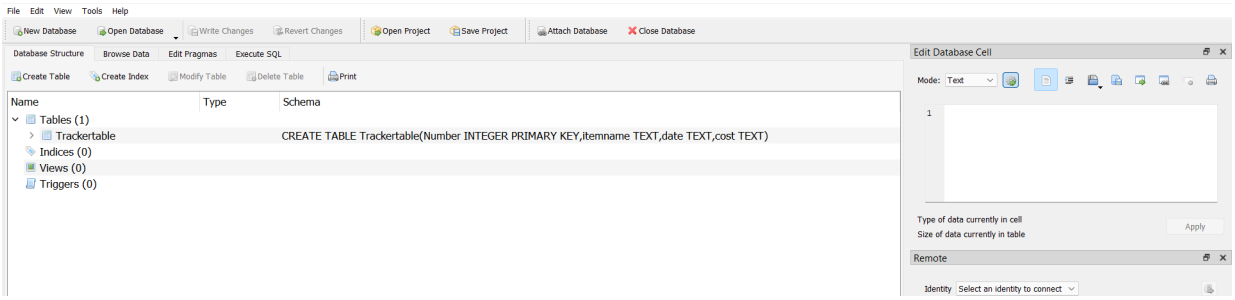

Figure 3: Screenshot of the Trackerdata database

2.1 Connections

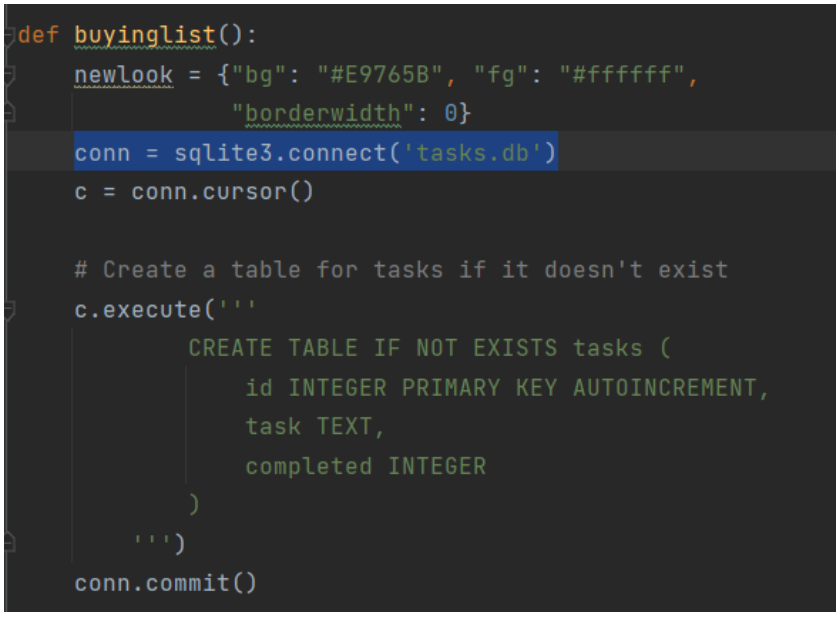

Figure 4: Screenshot of connection between butinglist() and database task.db

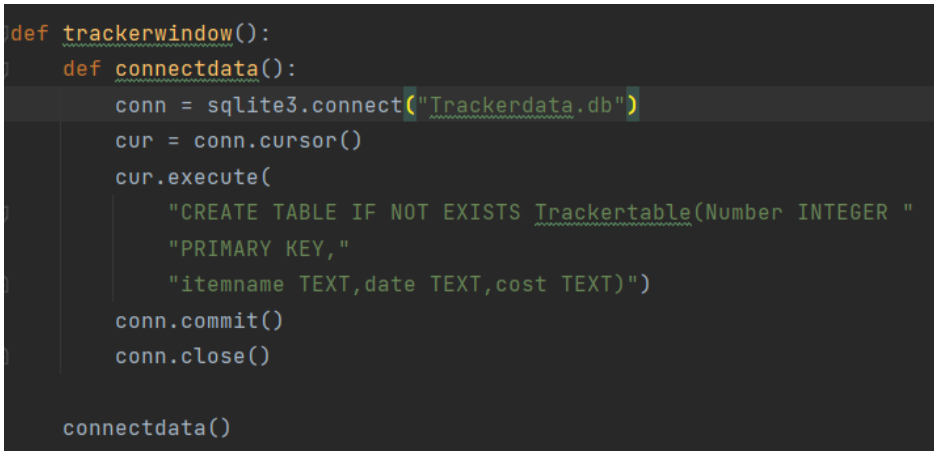

Figure 5: Screenshot of connection between trackerwindow() and database Trackerdata.db

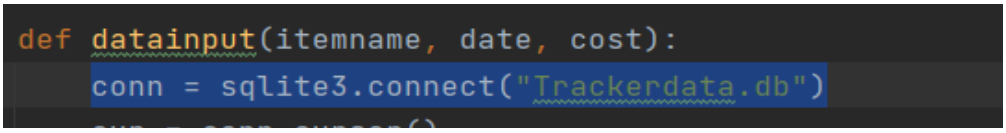

Figure 6: Screenshot of connection between datainput() and database Trackerdata.db

```
def view():
    conn = sqlite3.connect("Trackerdata.db")
```

Figure 7: Screenshot of connection between view() and database Trackerdata.db

```
def search(itemname="", date="", cost=""):
    conn = sqlite3.connect("Trackerdata.db")
```

Figure 8: Screenshot of connection between search() and database Trackerdata.db

```
def delete():
    selected_row = box1.curselection()
    if not selected_row:
        messagebox.showinfo('ERROR', "Select a row to delete")
        return

    selected_num = box1.get(selected_row).split()[0]

    conn = sqlite3.connect("Trackerdata.db")
    cur = conn.cursor()
    cur.execute("DELETE FROM Trackertable WHERE Number=?", (selected_num,))
    conn.commit()
    conn.close()
```

Figure 9: Screenshot of connection between delete() and database Trackerdata.db

```
def deletealldata():
    conn = sqlite3.connect("Trackerdata.db")
```

Figure 10: Screenshot of connection between deletealldata() and database Trackerdata.db

```
def update():

    date = vari1_date.get()
    count = date.count('-')
    selected_row = box1.curselection()
    if not selected_row:
        messagebox.showinfo('ERROR', "Select a row to update")
        return

    itemname = vari1_itemname.get()
    date = vari1_date.get()
    cost = vari1_cost.get()

    if not all([itemname, date, cost]):
        messagebox.showinfo('ERROR', "Fill in all fields")
        return

    try:
        cost = int(cost)
    except ValueError:
        messagebox.showinfo('ERROR', "Cost must be a number")
        return

    if len(date) != 10 or count != 2:
        messagebox.showinfo('ERROR', "Use dd-mm-yyyy format for date")
        return

    selected_num = box1.get(selected_row).split()[0]

    conn = sqlite3.connect("Trackerdata.db")
```

Figure 11: Screenshot of connection between update() and database Trackerdata.db

```
def totalsum():
    conn = sqlite3.connect("Trackerdata.db")
```

Figure 12: Screenshot of connection between totalsum() and database Trackerdata.db

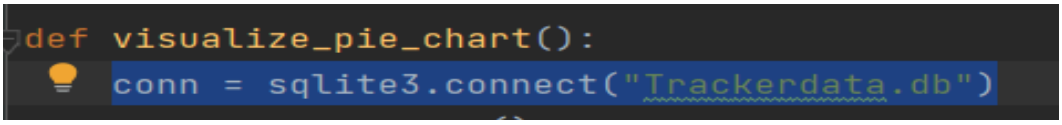

Figure 13: Screenshot of connection between visualize_pie_chart() and database Trackerdata.db

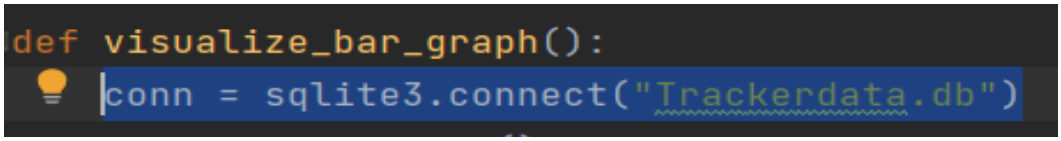

Figure 14: Screenshot of connection between visualize_pie_chart() and database Trackerdata.db

3. SQL queries used:

3.1 SELECT

This query was used in the following cases:

```
# Fetch tasks from the database and add them to the listbox
c.execute('SELECT task, completed FROM tasks')
tasks = c.fetchall()
for task in tasks:
    task_listbox.insert(tk.END, task[0])
    if task[1] == 1:
        task_listbox.itemconfig(tk.END, fg='grey')
```

Figure 15: Screenshot of SELECT used in buyinglist() to retrieve the tasks from the database to add them to the listbox

```
def view():
    conn = sqlite3.connect("Trackerdata.db")
    cur = conn.cursor()
    cur.execute("SELECT * FROM Trackertable")
    rows = cur.fetchall()
    conn.commit()
    conn.close()
    return rows
```

Figure 16:Screenshot of SELECT used in view() to retrieve all the records from the Trackertable in the Trackerdata.db

```
def search(itemname="", date="", cost=""):
    conn = sqlite3.connect("Trackerdata.db")
    cur = conn.cursor()
    cur.execute("SELECT *FROM Trackertable WHERE itemname=? OR date=? OR cost=?", (itemname, date, cost))
    rows = cur.fetchall()
    conn.commit()
    conn.close()
    return rows
```

Figure 17:Screenshot of SELECT used in search(itemname="",date="",cost="") to retrieve records from the Trackertable in the Trackerdata.db

```
def totalsum():
    conn = sqlite3.connect("Trackerdata.db")
    cur = conn.cursor()
    cur.execute("SELECT SUM(cost) FROM Trackertable")
    root = Tk()
    root.configure(bg='#2d2d2d')
    root.title("Amount Spent")
    root.geometry("350x300")
    tsum = cur.fetchone()
    newlook = {"bg": "#E9765B", "fg": "#ffffff",
               "borderwidth": 0}
    list1.delete(0, END)
    k = str(tsum[0])
    Label(root, width=60, font=("comic sans ms", 20), bg
```

Figure 18:Screenshot of SELECT used in totalsum() to retrieve all added cost from the Trackertable in the Trackerdata.db

```python
def visualize_pie_chart():
    conn = sqlite3.connect("Trackerdata.db")
    cur = conn.cursor()
    cur.execute("SELECT cost, itemname FROM Trackertable")
    rows = cur.fetchall()
    conn.commit()
    conn.close()
    costs = []
    products = []
    for row in rows:
        products.append(row[1])
        costs.append(int(row[0]))

    plt.pie(costs, labels=products, autopct='%1.1f%%', startangle=90)
    plt.axis('equal')
    plt.title('Distribution of expenses')
    plt.show()
```

Figure 19:Screenshot of SELECT used in visualize_pie_chart() to retrieve records from Trackertable in the Trackerdata.db to visualize in pie chart

```python
def visualize_bar_graph():
    conn = sqlite3.connect("Trackerdata.db")
    cur = conn.cursor()
    cur.execute("SELECT cost, itemname FROM Trackertable")
    rows = cur.fetchall()
    conn.commit()
    conn.close()
    costs = []
    products = []
    for row in rows:
        products.append(row[1])
        costs.append(int(row[0]))

    plt.bar(products, costs)
    plt.title('Distribution of expenses')
    plt.xlabel('Items')
    plt.ylabel('Cost')
    plt.show()
```

Figure 20:Screenshot of SELECT used in visualize_bar_graph() to retrieve records from Trackertable in the Trackerdata.db to visualize in bar graph

3.2 CREATE

```
def buyinglist():
    newlook = {"bg": "#E9765B", "fg": "#ffffff",
               "borderwidth": 0}
    conn = sqlite3.connect('tasks.db')
    c = conn.cursor()

    # Create a table for tasks if it doesn't exist
    c.execute('''
            CREATE TABLE IF NOT EXISTS tasks (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                task TEXT,
                completed INTEGER
            )
        ''')
    conn.commit()
```

Figure 21:Screenshot of CREATE used in buyinglist() to create a table tasks with id, task and completed columns to store data in the tasks.db

```
def trackerwindow():
    def connectdata():
        conn = sqlite3.connect("Trackerdata.db")
        cur = conn.cursor()
        cur.execute(
            "CREATE TABLE IF NOT EXISTS Trackertable(Number INTEGER "
            "PRIMARY KEY,"
            "itemname TEXT,date TEXT,cost TEXT)")
        conn.commit()
        conn.close()

    connectdata()
```

Figure 22:Screenshot of CREATE used in trackerwindow() to create a table Trackertable with number, itemname, date and cost columns to store data in the Trackerdata.db

3.3 INSERT

```
def add_task():
    task = task_entry.get()
    if task:
        c.execute('INSERT INTO tasks (task, completed) VALUES (?, ?)', (task, 0))
        conn.commit()
        task_listbox.insert(tk.END, task)
        task_entry.delete(0, tk.END)
```

Figure 23:Screenshot of INSERT used in add_task() in buyinglist() to add the task to a database table called tasks.

```
def datainput(itemname, date, cost):
    conn = sqlite3.connect("Trackerdata.db")
    cur = conn.cursor()
    cur.execute("INSERT INTO Trackertable VALUES(NULL,?,?,?)", (itemname, date, cost))
    conn.commit()
    conn.close()
```

Figure 24:Screenshot of INSERT used in datainput() in trackerwindow() to add item, date and cost to a database table called Trackertable

3.4 UPDATE

```
def complete_task():
    selection = task_listbox.curselection()
    if selection:
        task = task_listbox.get(selection)
        c.execute('UPDATE tasks SET completed=1 WHERE task=?', (task,))
        conn.commit()
        task_listbox.itemconfig(selection, fg='grey')
```

Figure 25:Screenshot of UPDATE used in complete_task() in buyinglist() to update the table called tasks to set completed as 1 for the selected tasks

```
conn = sqlite3.connect("Trackerdata.db")
cur = conn.cursor()
cur.execute("UPDATE Trackertable SET itemname=?, date=?, cost=? WHERE Number=?",(itemname, date, cost, selected_num))
conn.commit()
conn.close()
```

Figure 26:Screenshot of UPDATE used in update() in trackerwindow() to update the row in the table called Trckertable for the selected item

3.5 DELETE

```
def delete_task():
    selection = task_listbox.curselection()
    if selection:
        task = task_listbox.get(selection)
        c.execute('DELETE FROM tasks WHERE task=?', (task,))
        conn.commit()
        task_listbox.delete(selection)
```

Figure 27:Screenshot of DELETE used in delete_task in buyinglist() to delete the selected task from the tasks table in the database.

```
def delete():
    selected_row = box1.curselection()
    if not selected_row:
        messagebox.showinfo('ERROR', "Select a row to delete")
        return

    selected_num = box1.get(selected_row).split()[0]

    conn = sqlite3.connect("Trackerdata.db")
    cur = conn.cursor()
    cur.execute("DELETE FROM Trackertable WHERE Number=?", (selected_num,))
    conn.commit()
    conn.close()

    box1.delete(selected_row)
    e1.delete(0, END)
    e2.delete(0, END)
    e3.delete(0, END)
    messagebox.showinfo('SUCCESS', "Selected item deleted")
    viewallitems()
```

Figure 28:Screenshot of DELETE used in delete() in trackerwindow() to delete the selected row from the Trackertable in the database.

```
def deletealldata():
    confirm = messagebox.askyesno("Confirmation", "Are you sure you want to delete all data?")
    if confirm:
        conn = sqlite3.connect("Trackerdata.db")
        cur = conn.cursor()
        cur.execute("DELETE FROM Trackertable")
        conn.commit()
        conn.close()
        box1.delete(0, END)
        messagebox.showinfo('Successful', 'Vanished!!')
        viewallitems()
```

Figure 29:Screenshot of DELETE used in deletealldata() in trackerwindow() to delete all item from the table called Trackertable

4. Object Oriented Programming Concepts(OOP)

The code uses tkinter, a GUI toolkit that is based on the OOP concepts. The following shows the significant OOP concepts used by this program:

4.1 Class

A class is a set of instructions for producing objects, setting up state's initial values, and implementing behavior (Moore et al., n.d.). tkinter provides classes such as Label, Entry, and Button that are used to create objects with specific attributes and methods.

For example:

```
b1 = Button(root, text="Login", font=("comic sans ms", 10, "bold"), width=12, command=loginpage, **newlook)
b1.place(x=270, y=350)
b3 = Button(root, text="Exit Window", font=("comic sans ms", 10, "bold"), width=12, command=root.destroy, **newlook)
b3.place(x=95, y=350)
Label(root, width=60, font=("comic sans ms", 30), bg="#2d2d2d", fg="#E9765B", text="LOGIN").place(x=-500,
                                                                                                  y=100)
```

Figure 30:Screenshot to show the use of class

From above Button is a class that represents a push button widget. Two instances have been created of this class, b1 and b3, to create the "Login" and "Exit Window" buttons.

4.2 Object

Object is an instance of a class. The code creates instances of tkinter objects, for example root is an object of the Tk class, which represents the main window of the GUI which is shown below:

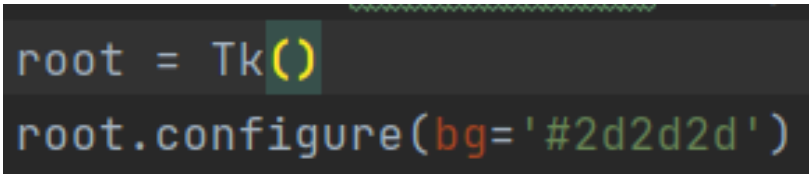

Figure 31:Screenshot to show the use of Object

4.3 Encapsulation

All the variables and functions are described inside of a function. This function can only be accessed from within the outer function[1], this shows the Encapsulation used in this code. As a result, the code has become more optimized and manageable.

4.2 Polymorphism

This code allows the user to use the same functions for different events like <Enter> <leave> using the bind method which shows the Polymorphism used in this code. One example of polymorphism is shown below:

[1] Refer Appendix B

```
def on_touch(button, color):
    button['bg'] = color


def on_alone(button, color):
    button['bg'] = '#E9765B'


def visualize():
    newlook = {"bg": "#E9765B", "fg": "#ffffff",
               "borderwidth": 0}
    gui = tk.Tk()
    gui.title("Visualize")
    gui.configure(bg='#2d2d2d')
    gui.geometry("300x300")

    pie_chart_button = tk.Button(gui, text="Pie Chart", height=1, width=18, command=visualize_pie_chart, **newlook)
    pie_chart_button.pack(side="left")
    pie_chart_button.bind("<Enter>", lambda event, b=pie_chart_button: on_touch(b, '#e81753'))  # <---------
    pie_chart_button.bind("<Leave>", lambda event, b=pie_chart_button: on_alone(b, '#E9765B'))  # <---------
```

Figure 32:Screenshot of polymorphism used in visualize()

5. Searching

The technique of locating the necessary information from a group of objects stored as components in the computer memory is known as searching in database systems.

```
def search(itemname="", date="", cost=""):
    conn = sqlite3.connect("Trackerdata.db")
    cur = conn.cursor()
    cur.execute("SELECT *FROM Trackertable WHERE itemname=? OR date=? OR cost=?", (itemname, date, cost))
    rows = cur.fetchall()
    conn.commit()
    conn.close()
    return rows
```

Figure 33:Screenshot of search function in trackerwindow()

A search function in the code enables looking for things in a SQLite database. The OR operator is used in the query to look for rows that match any of the specified parameters. The function retrieves all the rows that meet the search parameters after running the search query and delivers a list of those rows.

```
def search_item():
    box1.delete(0, END)
    box1.insert(END, "No.   NAME     DATE      COST")
    rows = search(vari1_itemname.get(), vari1_date.get(), vari1_cost.get())
    if len(rows) == 0:
        messagebox.showerror("Error", "Item not found in database.")
        viewallitems()
    else:
        for row in rows:
            j = str(row[0])
            k = str(row[1])
            l = str(row[2])
            m = str(row[3])
            o = j + "     " + k + "    " + l + "    " + m
            box1.insert(END, o)
    e1.delete(0, END)
    e2.delete(0, END)
    e3.delete(0, END)
```

Figure 34:Screenshot of search_item function in trackerwindow()

A wrapper function (mozilla, 2023) called search item runs the search function with the arguments vari1_itemname.get(), vari1_date.get(), and vari1_cost.get (). If the database does not contain any matching items, all items are shown along with an error message. The user input fields are cleaned and matched items are shown in box 1(the listbox) if there are any.

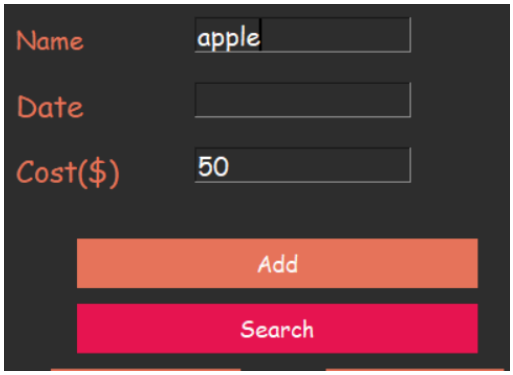

Figure 35:Screenshot of search feature in the Expense Tracker app

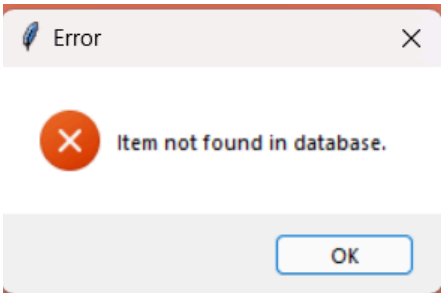

Figure 36:Screenshot of error message in search function

6. Updating

The update command is a data manipulation tool used to alter a table's records. It may be used to update a single row, all rows, or a group of rows depending on the user-provided condition.(Priya, 2021)

```
    viewallitems()
def update():

    date = vari1_date.get()
    count = date.count('-')
    selected_row = box1.curselection()
    if not selected_row:
        messagebox.showinfo('ERROR', "Select a row to update")
        return

    itemname = vari1_itemname.get()
    date = vari1_date.get()
    cost = vari1_cost.get()

    if not all([itemname, date, cost]):
        messagebox.showinfo('ERROR', "Fill in all fields")
        return

    try:
        cost = int(cost)
    except ValueError:
        messagebox.showinfo('ERROR', "Cost must be a number")
        return

    if len(date) != 10 or count != 2:
        messagebox.showinfo('ERROR', "Use dd-mm-yyyy format for date")
        return

    selected_num = box1.get(selected_row).split()[0]
```

```
conn = sqlite3.connect("Trackerdata.db")
cur = conn.cursor()
cur.execute("UPDATE Trackertable SET itemname=?, date=?, cost=? WHERE Number=?",(itemname, date, cost, selected_num))
conn.commit()
conn.close()

viewallitems()
e1.delete(0, END)
e2.delete(0, END)
e3.delete(0, END)
```

Figure 37:Screenshot of update function in the trackerwindow()

Based on user input, this code updates an item in an SQLite database and changes the display to reflect the updated entries. While updating it again verifies that all fields are completed and that the cost provided is a number and the data format is correct.

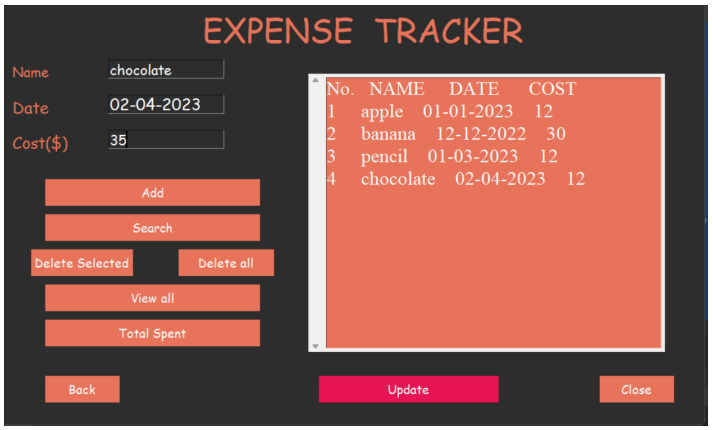

Figure 38:Screenshot of update feature in the trackerwindow()

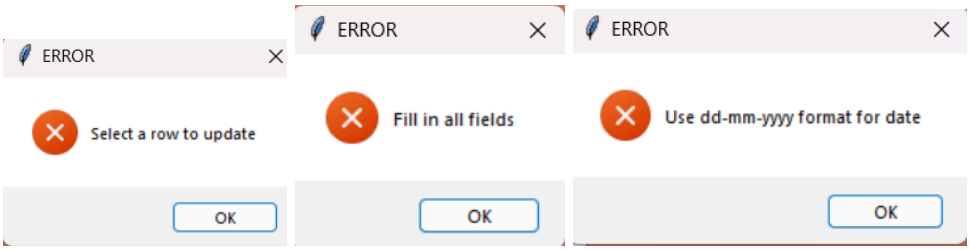

Figure 39:Screenshot of error message of update function in trackerwindow()

```
def complete_task():
    selection = task_listbox.curselection()
    if selection:
        task = task_listbox.get(selection)
        c.execute('UPDATE tasks SET completed=1 WHERE task=?', (task,))
        conn.commit()
        task_listbox.itemconfig(selection, fg='grey')
```

Figure 40:Screenshot of update feature in the buyinglist()

The code first determines if a task is chosen. A SQL UPDATE action is launched by the program once a task is selected, changing the completed column in the database(task.db) from 0 to 1 and locating the relevant task. This indicates that the task at hand has been completed. Basically, this code accurately records the accomplishment of a buying list/item in a SQLite database and uses a listbox display to show its completion with a gray highlighted color.

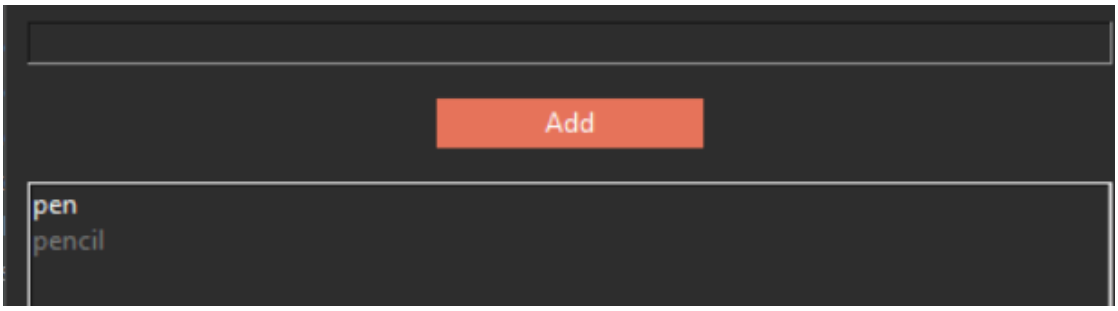

Figure 41:Screenshot showing the completion of a task with a gray highlighted color

7. Printing

```
def printdata():
    gui = Toplevel()
    gui.title("Print Data")
    gui.geometry("400x400")

    data_listbox = Listbox(gui)
    data_listbox.pack(expand=True, fill=BOTH)

    rows = view()
    for row in rows:
        data_listbox.insert(END, row)

    print_button = Button(gui, text="Print", command=lambda: print_to_printer(data_listbox))
    print_button.pack()
    back_button = Button(gui, text="Back", command=lambda: (report(), gui.destroy()))
    back_button.pack()

    def print_to_printer(data_listbox):

        temp_file = tempfile.NamedTemporaryFile(delete=False)

        for item in data_listbox.get(0, END):
            temp_file.write(bytes(str(item) + "\n", "utf-8"))

        temp_file.close()

        if sys.platform == "win32":
            subprocess.call(["notepad.exe", "/p", temp_file.name])
        else:
            print("Printing is not supported on this platform.")
```

Figure 42:Screenshot of printdata function in the trackerwindow()

The printdata() function creates a new window using the Tkinter library and adds a listbox to display data retrieved from a database by calling the view() function. By repeatedly going over the rows and placing them into the listbox, the data is presented there before being printed. Two buttons are also created by the function, one to print the information shown in the listbox and the other to return to the previous window.

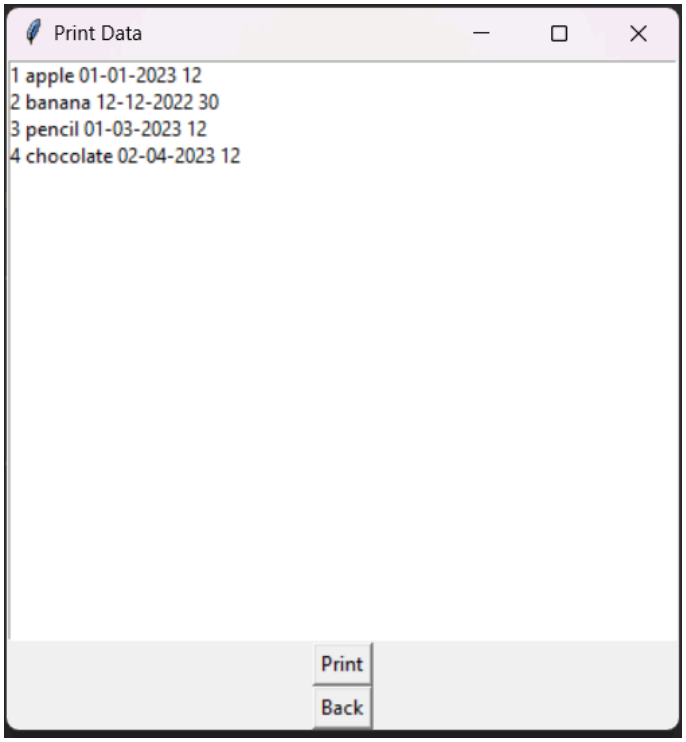

Figure 43: Screenshot of print feature in the Expense Tracker app

8. Hover feature

In response to the movement of the mouse pointer, the hover function is used to change the background color of the buttons . The config() method is used by the on enter() and on leave() functions to make the necessary adjustments to the button's background color depending on where the pointer is pointing. The bind() method is then used to associate these methods with the buttons <Enter> and <Leave> events. Hover feature has been used throughout the code (as the client really likes this feature[2])with different names to avoid the same name clashing for example on_leave is also used as on_alone.

The following shows the example of hover feature used in the secondwindow():

[2] See Appendix A.3

```
def secondwindow():
    gui = Tk()
    gui.title("My Window")
    gui.geometry("300x250")
    gui.configure(bg="#2c2f33")

    btn_style = {"bg": "#E9765B", "fg": "#ffffff",
                 "borderwidth": 0}

    def tracker_command():
        trackerwindow()
        gui.destroy()

    expense_tracker_btn = tk.Button(gui, text="Expense Tracker", width=20, height=2, command=tracker_command,
                                    **btn_style)
    expense_tracker_btn.pack(pady=10)

    def visualize_command():
        visualize()
        gui.destroy()

    visualize_btn = tk.Button(gui, text="Visualize", width=20, height=2, command=visualize_command, **btn_style)
    visualize_btn.pack(pady=10)

    def buyinglist_command():
        buyinglist()
        gui.destroy()
```

```
    buying_list_btn = tk.Button(gui, text="Buying List", width=20, height=2, command=buyinglist_command, **btn_style)
    buying_list_btn.pack(pady=10)

    def closeapp_command():
        gui.destroy()

    close_app_btn = tk.Button(gui, text="Close", width=20, height=2, command=closeapp_command, **btn_style)
    close_app_btn.pack(pady=10)

    def on_enter(btn):
        btn.config(bg="#e81753")

    def on_leave(btn):
        btn.config(bg="#E9765B")

    expense_tracker_btn.bind("<Enter>", lambda event, btn=expense_tracker_btn: on_enter(btn))
    expense_tracker_btn.bind("<Leave>", lambda event, btn=expense_tracker_btn: on_leave(btn))
    visualize_btn.bind("<Enter>", lambda event, btn=visualize_btn: on_enter(btn))
    visualize_btn.bind("<Leave>", lambda event, btn=visualize_btn: on_leave(btn))
    buying_list_btn.bind("<Enter>", lambda event, btn=buying_list_btn: on_enter(btn))
    buying_list_btn.bind("<Leave>", lambda event, btn=buying_list_btn: on_leave(btn))
    close_app_btn.bind("<Enter>", lambda event, btn=close_app_btn: on_enter(btn))
    close_app_btn.bind("<Leave>", lambda event, btn=close_app_btn: on_leave(btn))
```

Figure 44: Screenshot of code to implement hover feature

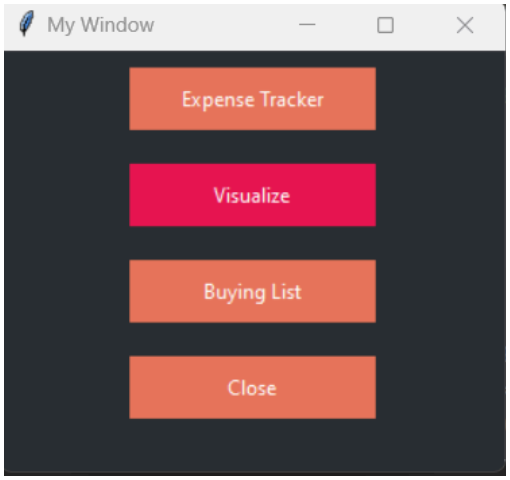

Figure 45: Screenshot of hover feature

9. Error handling and Data validation

To handle any errors that the program may have, this application has been created using several try and catch blocks and numerous validation checks, some are mentioned below. When an error happens, the program is intended to manage it rather than being terminated.

9.1 Error handling

```
def visualize_pie_chart():
    global cur, conn
    try:
        conn = sqlite3.connect("Trackerdata.db")
        cur = conn.cursor()
        cur.execute("SELECT cost, itemname FROM Trackertable")
        rows = cur.fetchall()
        conn.commit()
    except sqlite3.Error as e:
        messagebox.showinfo('error', f"Error connecting to database: {e}")
        return

    finally:
        cur.close()
        conn.close()

    if not rows:
        messagebox.showinfo('error', "No data found in the database.")
        return

    costs = []
    products = []
    for row in rows:
        products.append(row[1])
        costs.append(int(row[0]))

    plt.pie(costs, labels=products, autopct='%1.1f%%', startangle=90)
    plt.axis('equal')
    plt.title('Distribution of expenses')
    plt.show()
```

Figure 46:Screenshot showing the use of try except statements in visualize_pie_graph

```
def visualize_bar_graph():
    global conn, cur
    try:
        conn = sqlite3.connect("Trackerdata.db")
        cur = conn.cursor()
        cur.execute("SELECT cost, itemname FROM Trackertable")
        rows = cur.fetchall()
        conn.commit()
    except sqlite3.Error as e:
        messagebox.showinfo('error', f"Error connecting to database: {e}"
        return

    finally:
        cur.close()
        conn.close()

    if not rows:
        messagebox.showinfo('error',"No data found in the database.")
        return

    costs = []
    products = []
    for row in rows:
        products.append(row[1])
        costs.append(int(row[0]))

    plt.bar(products, costs)
    plt.title('Distribution of expenses')
    plt.xlabel('Items')
```

```
645     plt.ylabel('Cost')
646     plt.show()
```

Figure 47:Screenshot showing the use of try except statements in visualize_bar_graph

The code's try-except statements have been used to construct a strong error handling system. This statement's main goal is to find and fix any mistakes that could emerge when the code is being executed. The code is specifically made to deal with SQLite3 issues that might arise while connecting to or querying the database.

The code immediately alerts the user and halts the function in the case of an error. This method guarantees that the user is informed of any potential issues and may take the necessary corrective measures as required. The program also looks for data in the database, if none is discovered, an appropriate error message is shown. This guarantees that the visualization will only be produced when the data is in the database.

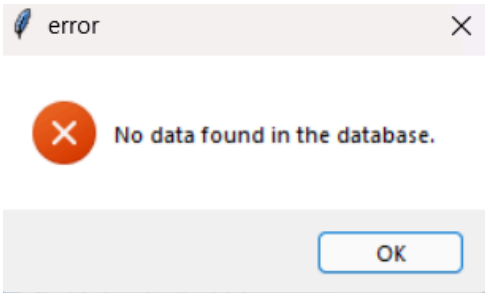

Figure 48:Screenshot of error message in visualizing pie chart and bar graph

9.2 Data validation

```
def loginpage():
    # Get the username and password entered by the user
    username = e1.get()
    password = e2.get()

    if username == "" and password == "":
        messagebox.showinfo("", "Try typing Admin for both username and password")

    elif username == "Admin" and password == "Admin":
        # If the credentials are correct, call the secondwindow function and destroy the root window
        secondwindow()
        root.destroy()

    else:
        # Clear the username and password fields
        e1.delete(0, END)
        e2.delete(0, END)
        messagebox.showerror('ERROR', 'Invalid credentials')
```

Figure 49:Screenshot showing the function created to perform data validation

The reliability of the user's entered credentials is verified during the validation procedure. So, if the entered credentials are wrong then an error message will pop up. Users won't be able to submit inaccurate login information due to this validation. Also, if there is no entry on the fields then the code will notify the user with a workable solution.

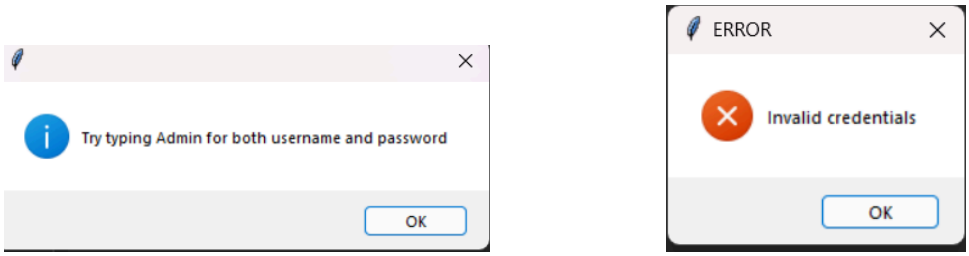

Figure 50:Screenshot of error message in loginpage

\

Overall, error handling and data validation improve the code's usability and robustness by delivering understandable error messages and averting possible problems brought on by improper database connections or missing data.

10. GUI ELEMENT

The GUI Windows are given below[3]

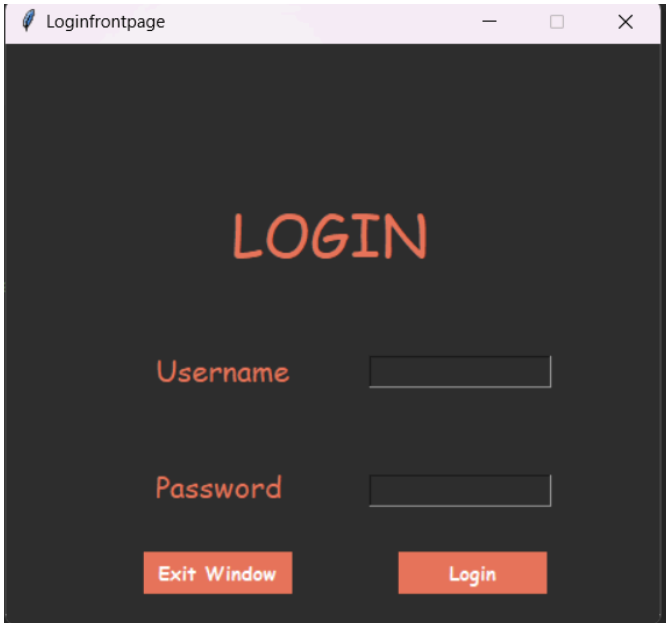

Figure 51:Screenshot of loginfrontpage

[3] See Appendix B for full code

```
def on_enter(btn):
    btn.config(bg="#e81753")
def on_leave(btn):
    btn.config(bg="#E9765B")
newlook = {"bg": "#E9765B", "fg": "#ffffff",
           "borderwidth": 0}
root = Tk()
root.configure(bg='#2d2d2d')
root.title("Loginfrontpage")
root.geometry("450x400")
l1 = Label(root, font=("comic sans ms", 15), bg="#2d2d2d", fg="#E9765B", text="Username")
l1.place(x=100, y=208)
l2 = Label(root, font=("comic sans ms", 15), bg="#2d2d2d", fg="#E9765B", text="Password")
l2.place(x=100, y=288)
login_name = StringVar()
e1 = Entry(root, font=("comic sans ms", 12), bg="#2d2d2d", fg="White", textvariable=login_name)
e1.place(x=250, y=215, height=22, width=125)
login_data = StringVar()
e2 = Entry(root, font=("comic sans ms", 12), bg="#2d2d2d", fg="White", textvariable=login_data, show="*")
e2.place(x=250, y=297, height=22, width=125)
b1 = Button(root, text="Login", font=("comic sans ms", 10, "bold"), width=12, command=loginpage, **newlook)
b1.place(x=270, y=350)
b3 = Button(root, text="Exit Window", font=("comic sans ms", 10, "bold"), width=12, command=root.destroy, **newlook)
b3.place(x=95, y=350)
Label(root, width=60, font=("comic sans ms", 30), bg="#2d2d2d", fg="#E9765B", text="LOGIN").place(x=-500,
                                                                                                  y=100)
b1.bind("<Enter>", lambda event, btn=b1: on_enter(btn))
b1.bind("<Leave>", lambda event, btn=b1: on_leave(btn))
b3.bind("<Enter>", lambda event, btn=b3: on_enter(btn))
b3.bind("<Leave>", lambda event, btn=b3: on_leave(btn))
```

Figure 52:Screenshot of how loginfrontpage was created

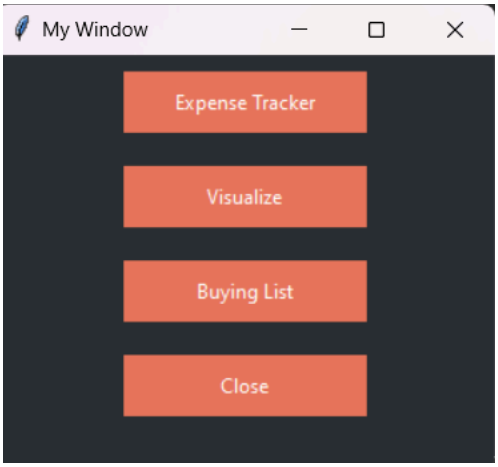

Figure 53:Screenshot of Mywindow

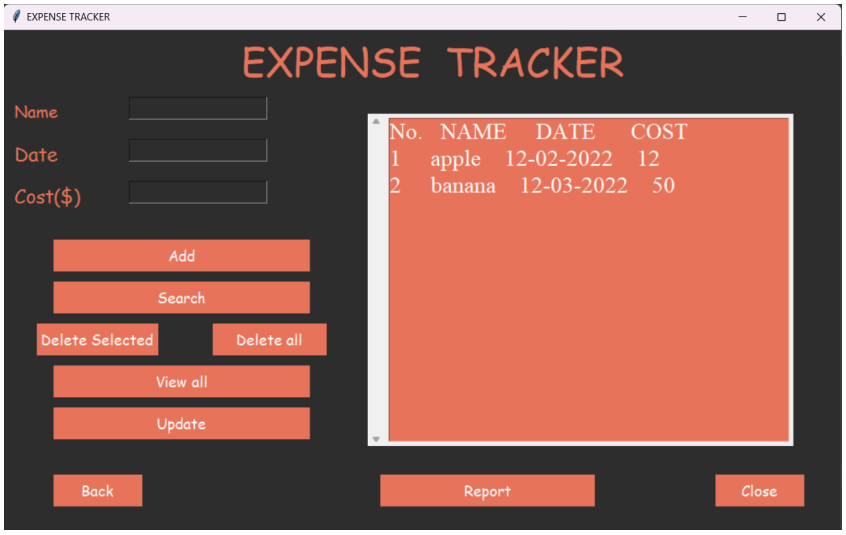

Figure 54:Screenshot of Expense Tracker

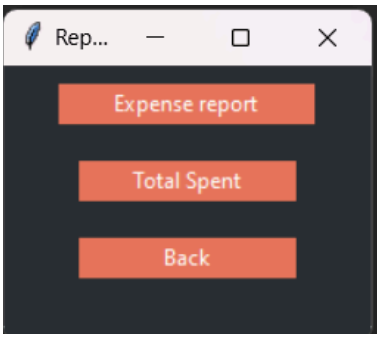

Figure 55:Screenshot of Report

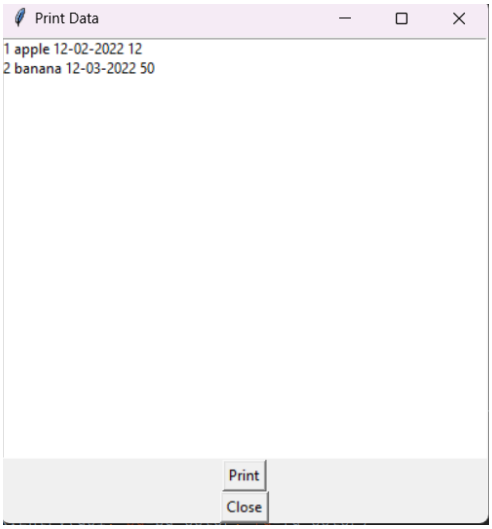

Figure 56:Screenshot of Print Data

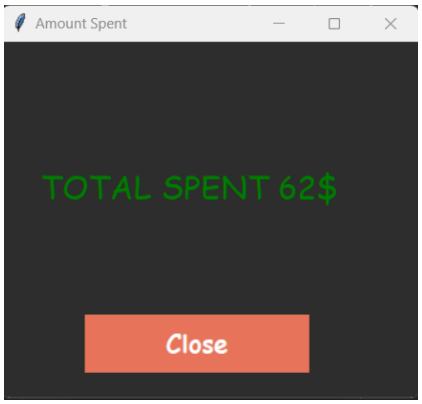

Figure 57:Screenshot of Amount Spent

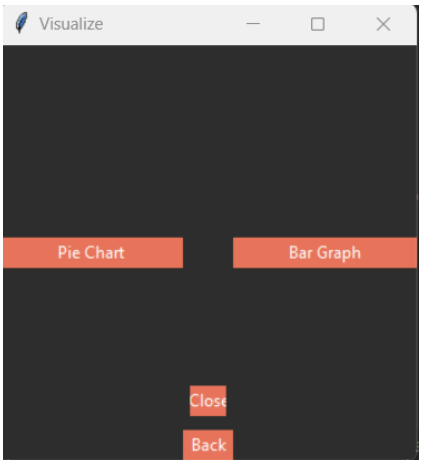

Figure 58:Screenshot of Visualize

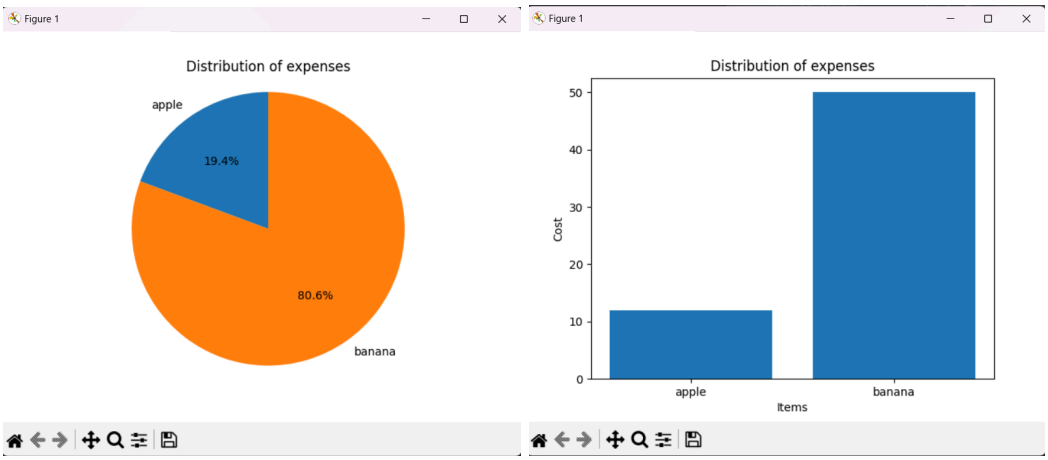

Figure 59:Screenshot of visualization of both pie chart and bar graph

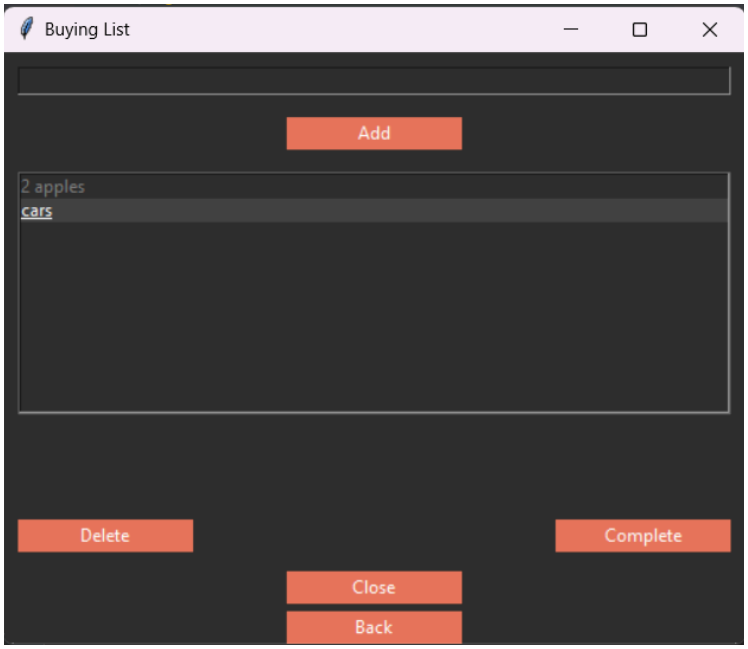

Figure 60:Screenshot of Buying List

The orange color theme buttons and the dark background has been updated upon client's request form the feedback session[4].

[4] See Appendix A.3

11. Extensibility

The code is expandable since it is modular and breaks down code into various functions. Without needing to completely rewrite the code, new features can be added. The procedure will go much more quickly as well. For future works if the code needs modification the user can easily add the code to the function that already exists , this makes it so the previous codes does not need modification.

Also, if the database schema is changed or modified with a new column, only the SQL query in the code needs to be updated so that it includes a new column and so that it can handle new data types.

Word Count: 1250

12. References

Harsha2580. (2018, June 12). *Ep 6: Living to Serve | SEARCH ON*. github. Retrieved March 1, 2023, from https://github.com/Harsha2580/ExpenseTracker/blob/main/ExpenseTracker.py

Moore, K., Njeru, E., & Pocevicius, M. (n.d.). *Classes (OOP)*. Brilliant. Retrieved March 10, 2023, from https://brilliant.org/wiki/classes-oop/

mozilla. (2023, February 21). *Wrapper - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*. MDN Web Docs. Retrieved March 25, 2023, from https://developer.mozilla.org/en-US/docs/Glossary/Wrapper

Parthmanchanda81. (2021, October 18). *Libraries in Python*. GeeksforGeeks. Retrieved February 27, 2023, from https://www.geeksforgeeks.org/libraries-in-python/

Priya, B. (2021, July 3). *What is the use of update command in SQL*. Tutorialspoint. Retrieved March 1, 2023, from https://www.tutorialspoint.com/what-is-the-use-of-update-command-in-sql

stackoverflow. (n.d.). *Python Try Except*. W3Schools. Retrieved March 3, 2023, from https://www.w3schools.com/python/python_try_except.asp