FC308 Practical Programming Assignment – Part 2

I confirm that this assignment is my own work. Where I have referred to academic sources, I have provided in-text citations and included the sources in the final reference list.

KAPLAN INTERNATIONAL COLLEGE LONDON – IYO COMPUTER SCIENCE PROGRAMME Module Title : Information Technology Module Code : FC308

Module Title : Information Technology
Module Code : FC308
Pnumber: p422027
Name: Gabriel Paniroi Haposan Siallagan

Table of Contents

1. Analysis

The Budget Planner application addresses the challenge of helping users effectively manage and visualize their personal finances through a user-friendly platform. The program's main goal is to let users to set up personal accounts so they may enter, monitor, and evaluate budgetary data. The program guarantees the safe and effective management of personal financial information by storing each user's data in distinct JSON files. In order to maintain a clear and straightforward user experience, providing smooth data synchronisation across several frames in the main window posed a significant problem during the application's development. Moreover, handling JSON files for the first time introduced complexities in data storage and retrieval, requiring careful implementation to prevent data inconsistencies. The program's capacity to give users precise and understandable financial facts so they may make well-informed decisions regarding their spending and saving practices is critical to its success. The program efficiently handles numerous user accounts and budgets, but as the user base increases, scalability concerns and the requirement for more sophisticated financial analytics will likely restrict its usefulness. With an eye towards simplicity, the application aims to combine extensive budget management capabilities with an intuitive user interface that can be used by users with different degrees of financial literacy. Future enhancements may address these limitations by incorporating more sophisticated data analysis tools and exploring alternative data storage solutions to accommodate a larger audience.

2. Design

Program Budget Tracker

Initital Set Up

Import csv

Import json

Import os

Import tkinter library as Tk

Import messagebox from tkinter

Import ttk from tkinter

Import matplotlib.pyplot as plt

Import FigureCanvasTkAgg from matplotlib.backends.backend_tkagg

Define constant FILE as "/Users/gabrielsiallagan/Desktop/information systems/users.csv"

Preparation Classes

Class UserManager:

Initialize user data from CSV file

Function __init__():

self.data = reader()

Login with username and password

```
Function login(username, password):
    Return True if username exists in self.data and self.data[username] is equal to password
    Else return False

Sign up with a new username and password
Function signup(username, password):
    If username exists in self.data:
        Return False
    Else:
        self.data[username] equal to password
        writer(self.data)
        Return True

Read CSV file and return dictionary of users
Function reader():
    data = empty dictionary
    Open FILE in read mode as csvfile
    Initialize reader as csv.reader(csvfile)
    For each row in reader:
        If row has at least 2 elements:
            data[row[0]] = row[1] // Map username to password
    Return data

Write user data to CSV file
Function writer(data):
```

Open FILE in write mode as csv_file

Initialize writer as csv.writer(csv_file, delimiter=',')

For each username, password in data.items():

writer.writerow([username, password])

Class UserDataManager:

Initialize user data and file path

Function __init__(current_user):

self.filepath = "/Users/gabrielsiallagan/Desktop/information systems/" + current_user + "_data.json"

self.data = load_data()

self.subscribers = empty list

// Save updated data to JSON file and notify subscribers

Function save_data():

Open self.filepath in write mode as file

Dump self.data to file as JSON

Call notify_subscribers()

// Initialize categories for monthly expenses

Function initialize_month_expenses():

Return dictionary with keys as categories: "Food", "Rent", "Utilities", "Transportation", "Shopping", "Life and Entertainment", "Other" with each value as empty list

```
    Update user data with new income, balance, and savings goal
    Function update_user_data(income, balance, savings_goal):
        Set self.data["income"] = income
        Set self.data["balance"] = balance
        Set self.data["savings_goal"] = savings_goal
        Call save_data()

    Add an expense to the user's data
    Function add_expense(day, amount, category):
        If category exists in self.data["expenses"][day]:
            Append amount to self.data["expenses"][day][category]
        Else:
            Initialize self.data["expenses"][day][category] = [amount]
        Call save_data()

    Get total expenses for each category
    Function get_category_expenses():
        Initialize category_totals as dictionary with keys from self.data["expenses"]["1"]
set to 0
        For each day, categories in self.data["expenses"].items():
            For each category, amounts in categories.items():
                Add sum of amounts to category_totals[category]
        Return category_totals

    Load data from JSON file or create new data structure if file doesn't exist
    Function load_data():
```

If os.path.exists(self.filepath):

Open self.filepath in read mode as file

Return JSON-loaded data from file

Else:

Return dictionary with keys "income", "balance", "savings_goal" set to 0.0 and "expenses" as dictionary of days (1-28) with initialized monthly expenses

Calculate the total expenses for the month

Function total_expense():

Initialize categories list with "Food", "Rent", "Utilities", "Transportation", "Shopping", "Life and Entertainment", "Other"

Initialize monthly_expense as empty list

For each day from 1 to 28:

Initialize daily_expenses as empty list

For each category in categories:

Get sum of self.data['expenses'][day][category] and append to daily_expenses

Calculate daily_total as sum of daily_expenses

Append daily_total to monthly_expense

Calculate total_monthly_expense as sum of monthly_expense

Return total_monthly_expense

Subscribe a callback to data changes

Function subscribe(callback):

Append callback to self.subscribers

Notify all subscribers about data changes

Function notify_subscribers():

For each callback in self.subscribers:

Call callback()

Class CenterWindow:

Center a window on the screen

Function center_window(width, height):

Get screen_width and screen_height from window

Calculate x = (screen_width / 2) - (width / 2)

Calculate y = (screen_height / 2) - (height / 2)

Set window geometry to width x height at position x, y

Class Construction:

Create a frame with specified dimensions

Function create_frame(parent, dimension, side, row=None, fill=None, expand=False):

Initialize frame as Frame with parent, width=dimension[0], height=dimension[1], highlightbackground="#b09662", highlightthickness=2

Set frame to not resize based on content

If row is None:

Configure frame.column 0 with weight 1

Pack frame with pady=10, side=side, fill=fill, expand=expand

Else:

```
        For x in range(row):
            Configure frame.column x with weight 1
        Pack frame with pady=10, side=side, fill=fill, expand=expand
    Return frame

Create a label and entry pair on a specified row
Function create_label_entry_pair(parent, label_text, row, fontsize, show=None):
    Initialize label as Label with parent, text=label_text, font=('Arial', fontsize)
    Set label.grid position at row=row, column=0, sticky=W+E

    Initialize entry_button_frame as Frame with parent
    Set entry_button_frame.grid position at row=row, column=1, sticky=W+E

    Initialize entry as Entry within entry_button_frame with font=('Arial', fontsize),
show=show
    Pack entry with side=LEFT, fill=X, expand=True

    Return label, entry

Center a window with specified width and height
Function center_window(window, width, height):
    Get screen_width and screen_height from window
    Calculate x = (screen_width / 2) - (width / 2)
    Calculate y = (screen_height / 2) - (height / 2)
    Set window geometry to width x height at position x, y
```

Window Classes

class LOGIN(Tk, CenterWindow)

initialise (user_manager)

call super().__init__()

SET self.user_manager = user_manager

SET window title

DISABLE window resizing

call self.center_window with size

call self.create_widgets()

call self.mainloop()

function create_widgets()

call self.create_input_frame()

call self.create_buttons()

function create_input_frame()

call login_frame

configure columns

pack frame

call self.create_label_entry_pair for Username

call self.create_label_entry_pair for Password

function create_label_entry_pair(parent, label_text, row, show=None)

create label

place label in grid

```
    create entry
    place entry in grid
    if row is equal to 0 THEN
        set self.entry_username
    else
        set self.entry_password

function create_buttons()
    create login_button
    pack login_button
    create signup_button
    pack signup_button

function login()
    get username from entry
    get password from entry
    if self.user_manager.login(username, password) then
        show success message
        close window
        create main_window
        call main_window.mainloop()
    else
        show error message

function open_signup_window()
```

```
        create SignupWindow instance


class SignupWindow(Toplevel, CenterWindow)
    initialise (login_instance)
        call super().__init__()
        set self.login_instance = login_instance
        set window title
        disable window resizing
        call self.center_window with size
        call self.create_widgets()

    function create_widgets()
        create signup_frame
        configure columns
        pack frame
        call self.create_label_entry_pair for Username
        call self.create_label_entry_pair for Password
        create signup_button
        pack signup_button

    funciton create_label_entry_pair(parent, label_text, row, show=None)
        create label
        place label in grid
        create entry
        place entry in grid
```

```
        if row is equal to 0 THEN
            set self.entry_username
        else
            set self.entry_password

    funciton signup()
        get username from entry
        get password from entry
        if self.login_instance.user_manager.signup(username, password) then
            show success message
            close window
        else
            show error message

class FrameRight(Construction)
    initialise(parent, data_manager)
        set self.data_manager
        create frame
        configure frame
        call self.expense()
        subscribe self.update_table to data changes

    function expense(frame)
        create labels, entries, and buttons for expense
        call self.Table()
```

```
function add_expense()
    try
        get expense_amount from entry
        get expense_day from combobox
        get expense_category from combobox
        if expense_amount less or equal to 0 then
            show error message
        else
            call self.data_manager.add_expense()
            show success message
    except non-numerical values
        show error message

function expensepie()
    create pie chart window
    fetch data
    plot pie chart
    display pie chart

function Table(frame)
    create table with columns
    add scrollbar to table
    call self.table_data()
    insert data into table
```

```
        pack table

    function update_table()
        clear existing table entries
        get all expenses
        insert expenses into table

    function table_data()
        fetch data
        return formatted data

class FrameLeft(Construction)
    initialise(parent, data_manager)
        set self.data_manager
        create frame
        configure frame
        initialise current values
        create labels, entries, and buttons for balance, income, savings goals
        subscribe self.update_frame to data changes

    function calculate_daily_savings_expense()
        calculate daily savings expense

    function update_balance()
        try
```

```
        get balance_value from entry
        if balance_value less or equal 0 THEN
            show error message
        else
            update balance
            show success message
    except ValueError
        show error message

function update_income()
    try
        get income_value from entry
        if income_value <= 0 THEN
            show error message
        else
            update income
            show success message
    except ValueError
        show error message

function update_savings()
    try
        get savings_value from entry
        if savings_value less than or equal to 0 or more than current income then
            show error message
```

```
            else
                update savings goal
                show success message
        except ValueError
            show error message

    function update_displays()
        update display labels

    function get_feasibility(daily_expense)
        return feasibility string based on daily expense

    function update_frame()
        update current values
        update displays

class GraphFrame(Construction)
    initialise(parent, data_manager)
        set self.data_manager
        create frame
        subscribe self.update_graph to data changes
        call self.plot_graph()

    function plot_graph(frame)
        initialise figure and axes
```

plot data

plot data

create canvas for figure

pack canvas

function get_monthly_balance(data)

calculate net balance for each day

function update_graph()

clear figure

call self.plot_graph()

class WINDOW(Tk, Construction)

initialise (current_user)

call super().__init__()

set window title to 'Budget Planner'

set window background color to '#0b0b0b'

call self.center_window with dimensions 1512x982

create label with text "{current_user.capitalize()}'s Budget Planner"

configure label font to 'Arial', size 30, foreground color '#b09662', background color 'black'

pack label with padding 3

create self.user_manager with current_user

create FrameRight with self and self.user_manager

create FrameLeft with self and self.user_manager

create GraphFrame with self and self.user_manager

call self.mainloop()

Main Program Flow

Define main function to initialize the program

create user_manager instance

create login_app instance with user_manager

call login_app.mainloop()

Call main()

3. Technical Overview

3.1 Libraries/Modules used:

csv: Used in my code to save and load user data, budget details, or any tabular data in a simple text format that is easy to read and write.

json: Used in my code to save and load user preferences, settings, or any structured data in a lightweight, text-based format that is easy to read and write.

os: Used in my code to manage file paths, create directories, and handle file operations in a platform-independent way.

* from tkinter: Used in my code to create the main window, labels, buttons, frames, and other GUI elements for my budget planner application.

messagebox from tkinter: Used in my code to display messages to the user, such as confirmation dialogs, error messages, or information alerts.

ttk from tkinter: Provides themed widgets (Treeview, Combobox, Notebook, etc.) that can be used in my code to create more visually appealing and interactive widgets in your GUI application.

matplotlib.pyplot: Used in my code to create plots, charts, and graphs to visualize budget data, such as spending trends, income vs. expenses, etc.

FigureCanvasTkAgg from matplotlib.backends.backend_tkagg:

Used in my code to display Matplotlib plots within the Tkinter GUI, allowing you to integrate graphs directly into your budget planner application.

3.2 Functions Used From Respective Libraries/Modules

csv Module

- **csv.reader()**
  - **Purpose**: Used to read data from CSV files into Python, producing an iterable reader object.
  - **Usage**: Parses CSV data into a list of rows or columns.
- **csv.writer()**
  - **Purpose**: Used to write data to CSV files.
  - **Usage**: Converts Python data into CSV format and writes it to a file.

json Module

- **json.dump()**
  - **Purpose**: Serializes Python objects and writes them as JSON to a file.
  - **Usage**: Used to save Python data structures like lists and dictionaries into JSON format files.
- **json.load()**
  - **Purpose**: Deserializes JSON data from a file into a Python object.
  - **Usage**: Reads JSON data from a file and converts it into corresponding Python data structures.

os Module

- **os.path.exists()**
  - **Purpose**: Checks if a given path exists.
  - **Usage**: Determines if a file or directory is present at the specified path.

Tkinter Library

tkinter Module

- **Tk()**
  - **Purpose**: Initializes a Tkinter application and creates a main window.
  - **Usage**: Acts as the root widget of the Tkinter GUI application.
- **Label()**
  - **Purpose**: Displays static text or images in the GUI.
  - **Usage**: Used to label different parts of the GUI with text or icons.
- **Button()**
  - **Purpose**: Creates a clickable button widget.
  - **Usage**: Used to perform actions when clicked, such as submitting forms or closing windows.
- **Entry()**
  - **Purpose**: Provides a text entry field for user input.
  - **Usage**: Collects single-line text input from the user.
- **Text()**
  - **Purpose**: Provides a multi-line text area for user input or display.
  - **Usage**: Allows users to input or display multiple lines of text.
- **Frame()**

    - **Purpose**: Acts as a container to hold and organize other widgets.
    - **Usage**: Used for grouping related widgets in the GUI for better layout management.
- **messagebox.showinfo()**
    - **Purpose**: Displays an information dialog box.
    - **Usage**: Shows information messages to the user with an OK button.
- **messagebox.showerror()**
    - **Purpose**: Displays an error dialog box.
    - **Usage**: Alerts users to errors with an OK button.
- **winfo_screenwidth()**
    - **Purpose**: Returns the width of the screen in pixels, used to calculate the center position for a window.
    - **Usage**: Calculate necessary dimensions and placement for the window for it to be centred on user screens
- **winfo_screenheight()**
    - **Purpose**: Returns the height of the screen in pixels, used to calculate the center position for a window.
    - **Usage**: Calculate necessary dimensions and placement for the window for it to be centred on user screens
- **Configure()**
    - **Purpose:** Change the properties of a widget after its initial creation.
    - **Usage:** Add colours and design of labels frames etc.

tkinter.ttk Module

- **ttk.Combobox()**
    - **Purpose**: Provides a dropdown list widget with a selection of items.
    - **Usage**: Allows users to select a single item from a predefined list of options.

Matplotlib Library

matplotlib.pyplot Module

- **plt.Figure()**
    - **Purpose**: Creates a new figure for plotting.
    - **Usage**: Serves as the main container for a plot and its components.
- **plt.pie()**
    - **Purpose**: Generates a pie chart.
    - **Usage**: Creates pie charts to represent data distribution visually.
- **plt.plot()**
    - **Purpose**: Plots data points on a 2D plane.
    - **Usage**: Creates line graphs and scatter plots.
- **plt.xlabel()**
    - **Purpose**: Sets the label for the x-axis.
    - **Usage**: Annotates the x-axis with a descriptive label.
- **plt.ylabel()**
    - **Purpose**: Sets the label for the y-axis.
    - **Usage**: Annotates the y-axis with a descriptive label.
- **plt.title()**

    - **Purpose**: Sets the title for the plot.
    - **Usage**: Provides a title for the chart or graph to convey its purpose.
- **plt.show()**
    - **Purpose**: Displays the plot in a window.
    - **Usage**: Renders the plot and opens it in a GUI window for viewing.

matplotlib.backends.backend_tkagg Module

- **FigureCanvasTkAgg()**
    - **Purpose**: Embeds a Matplotlib figure into a Tkinter widget.
    - **Usage**: Integrates Matplotlib plots into a Tkinter GUI, allowing for interactive plots within the application

3.3 Built-in Python Functions

- **Class**
    - **Purpose:** Creating objects that possesses certain properties and methods

- **if, else**
    - **Purpose**: Conditional statements used for control flow, determining which code blocks to execute based on conditions.
- **for loops**
    - **Purpose**: Looping statement used for iterating over sequences, such as lists, tuples, or dictionaries.
- **return**
    - **Purpose**: Exits a function and optionally returns a value to the caller.
- **with**
    - **Purpose**: Used for resource management and exception handling. It ensures that clean-up code is executed, for example, closing a file.
- **append**
    - **Purpose**: Adds an item to the end of a list. Used for adding expenses to lists and for managing subscribers.
- **items**
    - **Purpose**: Provides a view object that displays a list of dictionary's key-value tuple pairs, useful for iteration over dictionaries.
- **__init__**
    - **Purpose**: Special method that initializes newly created objects.
- **open**
    - **Purpose**: Opens a file and returns a corresponding file object, allowing for reading or writing operations.
- **super()**
    - **Purpose**: Returns a temporary object of the superclass that allows you to call its methods. Used to call the parent class's __init__ method.
- **get()**
    - **Purpose**: Retrieves the current value of a specified key in a dictionary or, in this context, gets the text from Tkinter's Entry widget. Though not a standalone Python function, it's a method used on a variety of built-in collections.

- **strip()**
    - **Purpose**: Returns a copy of the string with leading and trailing whitespace removed.
- **lower()**
    - **Purpose**: Returns a copy of the string with all characters converted to lowercase.

3.4 Classes

UserManager Class

Purpose: The UserManager class is responsible for handling user authentication and account management. It provides functionalities for user login and signup, as well as reading from and writing to a CSV file that stores user credentials (username and password). It acts as a bridge between the application and the stored user data, allowing secure authentication and account management.

UserDataManager Class

Purpose:

The UserDataManager class is responsible for managing and persisting user-specific data related to financial information and expenses. It handles operations like loading user data from a JSON file, updating financial information, adding expenses, and notifying subscribers about data changes. This class allows users to track their financial status, monitor expenses, and make informed decisions based on up-to-date data.

```
def subscribe(self, callback):
        """Subscribe a callback to data changes."""
        self.subscribers.append(callback)
        ##This method allows other parts of the application to register a callback function that will be called whenever the data changes. A callback function is a piece of code that you want to run later in response to some event, such as a data update.



    def notify_subscribers(self):
        """Notify all subscribed components about data changes."""
        for callback in self.subscribers:
            callback()
        #This method is responsible for informing all subscribed components that the data has changed. When the data changes, this method will go through each subscribed callback and execute it, ensuring that every component is updated accordingly.
```

Special subscriber concept used in this class:

The concept of subscribers in my code, specifically within the UserDataManager class, is an implementation of the **Observer Pattern**. This design pattern is used to allow an object to notify other interested objects (subscribers) about changes in its state. This concept was used to synchronise data in all frames within my main window such that the widgets in my code will update according to any data change made in other frames.

CenterWindow Class

```
class CenterWindow:
    def center_window(self, width, height):
        screen_width = self.winfo_screenwidth()
        screen_height = self.winfo_screenheight()
        x = (screen_width / 2) - (width / 2)
        y = (screen_height / 2) - (height / 2)
        self.geometry(f"{width}x{height}+{int(x)}+{int(y)}")
```

Purpose:

The CenterWindow class provides a utility function for centering windows on the screen. This is particularly useful for graphical user interfaces (GUIs) where you want to position windows centrally for better user experience.

Construction Class

Purpose:

The Construction class is a utility class designed to simplify the creation and layout of GUI components, such as frames, labels, and entry fields, in a Tkinter application. It provides methods to consistently style and position these elements, making the GUI development process more streamlined.

The classes listed above were created for convenience in construction of the main windows which proved to be important in making the main window codes easier to read.

LOGIN Class

Purpose:

creates the login window. It inherits the Tk which is the main window class in Tkinter. Additionally, it inherits the CenterWindow class to be centered on the users screens.

```
def login(self):
        username = self.entry_username.get().strip().lower()
        password = self.entry_password.get()
        if self.user_manager.login(username, password):
            messagebox.showinfo("Login", "Login Successful!")
            self.destroy()
            main_window = WINDOW(username)
            main_window.mainloop()
        else:
            messagebox.showerror("Login", "Invalid username or password.")
```

The method login() is used for error handling for when users type in an invalid password or a username that does not exist within the file. .strip() and .lower() makes the username case insensitive and removes any additional spaces that users may have accidentally typed in. Message pop ups will advise users on what went wrong in the login accordingly. Main window will only open when login is successful.

SignupWindow Class

Purpose:

creates the signup page when users want to register an account. It inherits TopLevel, a tkinter class used to create new windows separate from the main

application window. This allows the signup process to occur in a distinct interface, enhancing user experience.

```
def signup(self):
        username = self.entry_username.get().strip().lower()
        password = self.entry_password.get()
        if self.login_instance.user_manager.signup(username, password):
            messagebox.showinfo("Sign Up", "Sign Up Successful!")
            self.destroy()
        else:
            messagebox.showerror("Sign Up", "Username already exists.")
```

The method signup() is used for error handling for when users type in a username that already exists in the file. Message pop ups will inform users on what went wrong in the login accordingly. Once user has successfully made an account, signup window will close immediately.

FrameRight Class

Purpose:

The FrameRight class is a part of the window and is responsible for managing a right-hand frame within the GUI. This frame is designed for managing and displaying expense data. The class provides functionality for adding new expenses, visualizing expenses in a pie chart, and displaying expense data in a tabular format.

```
# Subscribe to data changes
        self.data_manager.subscribe(self.update_table)#if there is a change in data
from any other frame, table will update
```

```
def update_table(self):
        """Refresh the expense table with current data."""
        # Clear existing entries in the table
        for item in self.tree.get_children():
            self.tree.delete(item)

        # Get all expense data
        all_expenses = self.table_data()

        # Insert data into the table
        for expense in all_expenses:
            day, category, amount = expense
            self.tree.insert("", END, values=(day, category, amount))
```

As mentioned previously, within this frame, subscribe function is used to callback the update_table() function. This is so that the treeview Table would be updated after any changes in the expenses and would express the expenses in order of the date.

FrameLeft Class

Purpose:

The FrameLeft class is a part of the window and is responsible for managing a left-hand frame within the GUI. This frame is designed for managing and displaying expense data. The class provides functionality for adding new expenses, visualizing expenses in a pie chart, and displaying expense data in a tabular format.

```
self.data_manager.subscribe(self.update_frame)
```

```
def update_displays(self):
        """Updates all display labels to reflect the current state."""
        # Calculate the new balance and net balance
        self.new_balance = self.current_balance + self.current_income
        self.total_expense = self.data_manager.total_expense()
        self.net_balance = self.new_balance - self.total_expense

        # Update display labels with new values
        self.new_balance_display.config(text=f"Balance after Income:
${self.new_balance:.2f}")
        self.total_expense_display.config(text=f"Total Expense:
${self.total_expense:.2f}")
        self.net_balance_display.config(text=f"Net Balance:
${self.net_balance:.2f}")

        # Update the daily savings expense
        daily_savings_expense = self.calculate_daily_savings_expense()
        self.daily_savings_expense_display.config(text=f"Daily Savings Expense:
${daily_savings_expense:.2f}")

        # Update feasibility display
        if self.get_feasibility(daily_savings_expense) == "Not Feasible":

self.feasibility_display.config(text=self.get_feasibility(daily_savings_expense),fg
= '#f00e29', bg = '#797d7f')
        elif self.get_feasibility(daily_savings_expense) == "Possible but
difficult":

self.feasibility_display.config(text=self.get_feasibility(daily_savings_expense),fg
= '#f27c1b', bg = '#797d7f')
        elif self.get_feasibility(daily_savings_expense) == "Possible":

self.feasibility_display.config(text=self.get_feasibility(daily_savings_expense),
fg = '#14d368', bg = '#797d7f')
```

As mentioned previously, within this frame, subscribe function is used to callback the update_displays() function. This is so that the display showing the users data would be updated after any new expenses.

GraphFrame class

The main purpose of the GraphFrame class is to visually represent and track financial data over a month, displaying trends in expenses, income, and savings goals through a dynamic and interactive graph. This class is designed to be a part of a larger application that helps users manage and analyze their financial status, providing insights into their spending habits and financial goals.

```
self.data_manager.subscribe(self.update_graph)
```

```
def update_graph(self):
        """Updates the graph with new data when notified of data changes."""
        # If the figure exists, clear the axes
        if self.fig:
            self.fig.clear()

            # Create new axes
            ax = self.fig.add_subplot(111)

            # Retrieve updated data
            self.days = list(range(1, 29))  # Assuming 28 days for simplicity
            self.numbers = self.get_monthly_balance(self.data_manager.data)

            # Update income and savings goal for visualization
            monthly_income = self.data_manager.data["income"]
            monthly_savings_goal = self.data_manager.data["savings_goal"] +
(self.data_manager.data["balance"])
            self.income = [monthly_income] * 28  # Distribute monthly income
equally
            self.savings_goal = [monthly_savings_goal] * 28  # Distribute savings
goal equally

            # Plot the updated data
            ax.plot(self.days, self.numbers, color='blue', linewidth=2, marker='o',
markerfacecolor='blue', markersize=3, label='Expenses')
            ax.plot(self.days, self.income, color='green', linewidth=1,
linestyle='dotted', label='Income')
            ax.plot(self.days, self.savings_goal, color='red', linewidth=1,
linestyle='dotted', label='Savings Goal')

            # Update labels and title
            ax.set_xlabel('Days')
            ax.set_ylabel('Amount ($)')
```

```
            ax.set_ylim(bottom=0)
            ax.set_title('Expenses Graph')
            ax.legend(loc='upper right', fontsize = 10)  # Add legend

            # Adjust ticks
            ax.tick_params(axis='both', which='major', labelsize=10)  # Adjust the
labelsize as needed

            # Redraw the canvas with the new figure
            self.canvas.draw()
```

within this frame, subscribe function is used to callback the update_graph() function. This is so that the display showing the users data would be updated after any adjustments to the data. This is done by deleting the previous graph using .clear() function and plotting a brand new graph.

Window Class

```
class WINDOW(Tk, Construction):
    def __init__(self, current_user):
        super().__init__()
        self.title('Budget Planner')
        self.configure(bg = '#0b0b0b')
        self.geometry('1512x982')
        self.label = Label(self, text= f"{current_user.capitalize()}'s Budget
Planner", font=('Arial', 30), fg = '#b09662', bg = 'black')
        self.label.pack(pady=3)

        # Create a single data manager for the whole application
        self.user_manager = UserDataManager(current_user)


        # Pass the data manager to each frame
        FrameRight(self, self.user_manager)
        FrameLeft(self, self.user_manager)
        GraphFrame(self, self.user_manager)
```

Purpose:

Creates the main window that sets up all the widgets and functions from the respective frames(found in FrameRight, FrameLeft, GraphFrame). It sets the window title, background color, and size. Addtionally it Instantiate UserDataManager with the current_user to manage user-specific data and passes the user_manager to different frames for managing different parts of the UI.

Mainline

```
if __name__ == "__main__":
    #define UserManager class to call in LOGIN class
    user_manager = UserManager()
    login_app = LOGIN(user_manager)
```

if __name__ == “__main__” function was used to ensure that the code is being run as the mainline.

4. Code with Comments

```
##########################IMPORTED LIBRARIES#################################
import csv
import json
import os
from tkinter import *
from tkinter import messagebox
from tkinter import ttk
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

##########################PREPERATION_CLASSES#################################
# Define FILE to for convenience when calling
FILE = "/Users/gabrielsiallagan/Desktop/information systems/users.csv"

#used to retrieve and update data in login and sign-up class
class UserManager:
    #function that is automatically called when creating the object
    def __init__(self):
        self.data = self.reader()
        #reads file into dictionary

    def login(self, username, password):
        return username in self.data and self.data[username] == password
        #return either 'true' or 'false' if user has an account and password is
valid


    def signup(self, username, password):
        if username in self.data:
            return False
            #return false if user already has an account
        self.data[username] = password
        self.writer(self.data)
        #adds new account into csv file
        return True

    #main reader function that turns csv data into a dictionary
    def reader(self):
        data = {}
        with open(FILE, mode='r') as csvfile:
            reader = csv.reader(csvfile)
            #reader object
            data = dict((row[0], row[1]) for row in reader if len(row) >= 2)
            #creates the dictionary
        return data

    def writer(self, data):
        with open(FILE, 'w', newline='') as csv_file:
            writer = csv.writer(csv_file, delimiter=',')
```

```
            #writer object
            for username, password in data.items():
                writer.writerow([username, password])
                #splits list variable into rows

class UserDataManager:
    def __init__(self, current_user):
        self.filepath = "/Users/gabrielsiallagan/Desktop/information systems/" +
current_user + "_data.json"
        self.data = self.load_data()
        #reads data

        # Subscribers for data changes
        self.subscribers = []

    #re-writes data according to updated data
    def save_data(self):
        with open(self.filepath, 'w') as file:
            json.dump(self.data, file, indent=2)
        self.notify_subscribers()

    #set up dictionary with categories
    def initialize_month_expenses(self):
        """Initializes a dictionary for monthly expenses with categories."""
        return {
            "Food": [],
            "Rent": [],
            "Utilities": [],
            "Transportation": [],
            "Shopping": [],
            "Life and Entertainment": [],
            "Other": []
        }

    #get updated data and saves it into file
    def update_user_data(self, income, balance, savings_goal):
        """Updates user income, balance, and savings goal."""
        self.data["income"] = income
        self.data["balance"] = balance
        self.data["savings_goal"] = savings_goal
        self.save_data()

    #adds new expense with date and category into the file
    def add_expense(self, day, amount, category):
        """Adds an expense to a specific user for a given month."""
        if category in self.data["expenses"][str(day)]:
            self.data["expenses"][str(day)][category].append(amount)
        else:
            self.data["expenses"][str(day)][category] = [amount]
        self.save_data()
```

```
    #get total expense of each category for pie chart
    def get_category_expenses(self):
        """Calculates total expenses per category."""
        category_totals = {category: 0 for category in self.data["expenses"]["1"]}
# Initialize totals
        for day, categories in self.data["expenses"].items():
            for category, amounts in categories.items():
                category_totals[category] += sum(amounts)
        return category_totals
    #reads or creates file
    def load_data(self):
        if os.path.exists(self.filepath):
            with open(self.filepath, 'r') as file:
                return json.load(file)
        else:
            return {
                "income": 0.0,
                "balance": 0.0,
                "savings_goal": 0.0,
                "expenses": {str(day): self.initialize_month_expenses() for day in
range(1, 29)}
            }
    #calculates total expense
    def total_expense(self):

        categories = ["Food", "Rent", "Utilities", "Transportation", "Shopping",
"Life and Entertainment", "Other"]

        # Initialize list to hold monthly expenses
        monthly_expense = []

        # Iterate over each day
        for day in range(1, 29):  # Use the actual range of days as integers
            day_str = str(day)    # Convert day to string to use as a key in the
JSON data
            daily_expenses = []   # List to store daily expenses per category

            # Check if the day exists in the data
            for category in categories:
                # Check if the category exists for the day, and sum the expenses if
it does
                category_sum = sum(self.data['expenses'][day_str].get(category,
[]))
                daily_expenses.append(category_sum)

            # Calculate the total expenses for the day
            daily_total = sum(daily_expenses)

            # Append daily total to monthly expenses
            monthly_expense.append(daily_total)
            total_monthly_expense = sum(monthly_expense)
```

```
        # Print the list of monthly expenses
        return total_monthly_expense

    def subscribe(self, callback):
        """Subscribe a callback to data changes."""
        self.subscribers.append(callback)
        ##This method allows other parts of the application to register a callback function that will be called whenever the data changes. A callback function is a piece of code that you want to run later in response to some event, such as a data update.



    def notify_subscribers(self):
        """Notify all subscribed components about data changes."""
        for callback in self.subscribers:
            callback()
        #This method is responsible for informing all subscribed components that the data has changed. When the data changes, this method will go through each subscribed callback and execute it, ensuring that every component is updated accordingly.

#to be called in login and signup to center window
class CenterWindow:
    def center_window(self, width, height):
        screen_width = self.winfo_screenwidth()
        screen_height = self.winfo_screenheight()
        x = (screen_width / 2) - (width / 2)
        y = (screen_height / 2) - (height / 2)
        self.geometry(f"{width}x{height}+{int(x)}+{int(y)}") ## The following lines of code were adapted from https://www.youtube.com/watch?v=TdTks2eSx3c&ab_channel=Codemy.com


#class helps with the creation of frames
class Construction:
    #frame creation function
    def create_frame(self, parent, dimension, side, row=None, fill=None, expand=False):
        frame = Frame(parent, width=dimension[0], height=dimension[1], highlightbackground="#b09662", highlightthickness=2)
        frame.pack_propagate(False)  # Prevent the frame from resizing based on its content
        if row is None:
            frame.columnconfigure(0, weight=1)
            frame.pack(pady=10, side=side, fill=fill, expand=expand)
        else:
            for x in range(row):
                frame.columnconfigure(x, weight=1)
            frame.pack(pady=10, side=side, fill=fill, expand=expand)
```

```
        return frame
    #creates a label and an entry side by side
    def create_label_entry_pair(self, parent, label_text, row, fontsize,
show=None):
        label = Label(parent, text=label_text, font=('Arial', fontsize))
        label.grid(row=row, column=0, sticky=W+E)

        # Frame to contain both Entry and Button
        entry_button_frame = Frame(parent)
        entry_button_frame.grid(row=row, column=1, sticky=W+E)

        entry = Entry(entry_button_frame, font=('Arial', fontsize), show=show)
        entry.pack(side=LEFT, fill=X, expand=True)  # Use pack for entry in this
frame

        return label, entry  # Return the entry widget for later use

    # Add 'window' parameter to specify the window to be centered
    def center_window(self, window, width, height):
        screen_width = window.winfo_screenwidth()
        screen_height = window.winfo_screenheight()
        x = (screen_width / 2) - (width / 2)
        y = (screen_height / 2) - (height / 2)
        window.geometry(f"{width}x{height}+{int(x)}+{int(y)}") ## The following
lines of code were adapted from
https://www.youtube.com/watch?v=TdTks2eSx3c&ab_channel=Codemy.com


######################################################################


##########################LOGIN_WINDOW#################################
class LOGIN(Tk, CenterWindow):
    def __init__(self, user_manager):
        super().__init__()
        self.user_manager = user_manager
        self.title("Login/Signup")
        self.resizable(False, False)
        self.center_window(500, 200)
        self.create_widgets()
        self.mainloop()

    def create_widgets(self):
        self.create_input_frame()
        self.create_buttons()

    def create_input_frame(self):
        self.login_frame = Frame(self)
        self.login_frame.columnconfigure(0, weight=1)
        self.login_frame.columnconfigure(1, weight=1)
        self.login_frame.pack(padx=20, pady=(20, 0), fill=BOTH, expand=False)
```

```
        self.create_label_entry_pair(self.login_frame, "Username: ", 0)
        self.create_label_entry_pair(self.login_frame, "Password: ", 1, show='*')

    def create_label_entry_pair(self, parent, label_text, row, show=None):
        label = Label(parent, text=label_text, font=('Arial', 18))
        label.grid(row=row, column=0, sticky=W+E, pady=10)

        entry = Entry(parent, font=('Arial', 18), show=show)
        entry.grid(row=row, column=1, sticky=W+E, pady=10)

        if row == 0:
            self.entry_username = entry
        else:
            self.entry_password = entry

    def create_buttons(self):
        self.login_button = Button(self, text='Login', font=('Arial', 15),
command=self.login)
        self.login_button.pack()

        self.signup_button = Button(self, text='Sign Up', font=('Arial', 15),
command=self.open_signup_window)
        self.signup_button.pack()

    def login(self):
        username = self.entry_username.get().strip().lower()
        password = self.entry_password.get()
        if self.user_manager.login(username, password):
            messagebox.showinfo("Login", "Login Successful!")
            self.destroy()
            main_window = WINDOW(username)
            main_window.mainloop()
        else:
            messagebox.showerror("Login", "Invalid username or password.")

    def open_signup_window(self):
        SignupWindow(self)
########################################################################


##########################SIGN-UP_WINDOW################################
#toplevel used to create a window on top of a window
class SignupWindow(Toplevel, CenterWindow):
    def __init__(self, login_instance):
        super().__init__()
        self.login_instance = login_instance
        self.title("Sign Up")
        self.resizable(False, False)
        self.center_window(500, 200)
        self.create_widgets()
```

```
    #widgets within this window
    def create_widgets(self):
        self.signup_frame = Frame(self)
        self.signup_frame.columnconfigure(0, weight=1)
        self.signup_frame.columnconfigure(1, weight=1)
        self.signup_frame.pack(padx=20, pady=(20, 0), fill=BOTH, expand=False)

        self.create_label_entry_pair(self.signup_frame, "Username: ", 0)
        self.create_label_entry_pair(self.signup_frame, "Password: ", 1, show='*')

        self.signup_button = Button(self, text='Sign Up', font=('Arial', 15),
command=self.signup)
        self.signup_button.pack()

    #creates a label and entry specific to login page
    def create_label_entry_pair(self, parent, label_text, row, show=None):
        label = Label(parent, text=label_text, font=('Arial', 18))
        label.grid(row=row, column=0, sticky=W+E, pady=10)

        entry = Entry(parent, font=('Arial', 18), show=show)
        entry.grid(row=row, column=1, sticky=W+E, pady=10)

        if row == 0:
            self.entry_username = entry
        else:
            self.entry_password = entry

    #function for users to make an account
    def signup(self):
        username = self.entry_username.get().strip().lower()
        password = self.entry_password.get()
        if self.login_instance.user_manager.signup(username, password):
            messagebox.showinfo("Sign Up", "Sign Up Successful!")
            self.destroy()
        else:
            messagebox.showerror("Sign Up", "Username already exists.")
##############################################################################

############################MAIN_WINDOW_FRAMES#################################

#to be added in the main window class
class FrameRight(Construction):
    def __init__(self, parent, data_manager):
        self.data_manager = data_manager
        self.widget_frame_right = self.create_frame(parent, (300, 300), RIGHT,
row=None, fill='y')
        self.widget_frame_right.configure(bg='#1c1d20')
        self.expense(self.widget_frame_right)

        # Subscribe to data changes
```

```
        self.data_manager.subscribe(self.update_table)#if there is a change in data
from any other frame, table will update

    #expense widgets
    def expense(self, frame):
        Label(frame, text='Add Expense', fg = '#b09662', bg =
'#421d59').grid(row=0, column=0, columnspan=2)
        self.expense_amount = self.create_label_entry_pair(frame, 'Amount:', 1, 13)
        self.expense_amount_label, self.expense_amount_entry = self.expense_amount
        self.expense_amount_label.configure(fg = '#b09662', bg = '#421d59')
        Label(frame, text="Date:", fg = '#b09662', bg = '#421d59').grid(row=2,
column=0, sticky=W)
        self.expense_day = ttk.Combobox(frame, values=list(range(1, 29))) # More
concise way to list days
        self.expense_day.grid(row=2, column=1)
        Label(frame, text="Category:", fg = '#b09662', bg = '#421d59').grid(row=3,
column=0, sticky=W)
        self.expense_category = ttk.Combobox(frame, values=["Food", "Rent",
"Utilities", "Transportation", "Shopping", "Life and Entertainment", "Other"])
        self.expense_category.grid(row=3, column=1)
        Button(frame, text="Add Expense", font=('Arial', 15),
command=self.add_expense, highlightbackground='#b09662').grid(row=4, column=0,
columnspan=2, pady=10)
        Button(frame, text="View Expense", font=('Arial', 15),
command=self.expensepie, highlightbackground='#b09662').grid(row=5, column=0,
columnspan=2, pady=10)
        self.Table(frame)

    #gets input from entries and adds to file
    def add_expense(self):
        try:
            # Validate inputs
            expense_amount = float(self.expense_amount[1].get())
            expense_day = int(self.expense_day.get())
            expense_category = self.expense_category.get()
            #error handling
            if expense_amount <= 0:
                messagebox.showerror("Invalid Input", "Please enter a postive
amount.")
            else:

                # Update the data manager
                self.data_manager.add_expense(expense_day, expense_amount,
expense_category)#add to file

                messagebox.showinfo("Success", "Expense added successfully.")
        except ValueError:
            messagebox.showerror("Invalid Input", "Please enter valid numbers for
amount and day.")

    #piechart widget
```

```
    def expensepie(self):
        # Create a new Toplevel window
        pie_window = Toplevel(self.widget_frame_right)
        pie_window.title("Expense Distribution")

        # Center the Toplevel window instead of the main window
        self.center_window(pie_window, 700, 500)

        # Create the pie chart
        fig, ax = plt.subplots()

        # Get the total expenses per category
        category_expenses = self.data_manager.get_category_expenses()
        categories = list(category_expenses.keys())
        amounts = list(category_expenses.values())

        ax.pie(amounts, labels=categories, textprops={'fontsize': 10},
autopct='%1.1f%%')
        ax.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a
circle
        plt.title('Monthly Expenses Distribution')

        # Create a canvas for the figure and add it to the Toplevel window
        canvas = FigureCanvasTkAgg(fig, master=pie_window)
        canvas.draw()
        canvas.get_tk_widget().pack(fill=BOTH, expand=True)

        # Ensure that the window is closed properly
        pie_window.protocol("WM_DELETE_WINDOW", pie_window.destroy)

    #table widget
    def Table(self, frame):
        self.entry_frame = self.create_frame(frame, (500, 600), BOTTOM)
        self.tree = ttk.Treeview(self.entry_frame, columns=("Day", "category",
"Amount"), show='headings', height = 40)

        # Define the column headings
        self.tree.heading("Day", text="Day")
        self.tree.heading("category", text="Category")
        self.tree.heading("Amount", text="Amount")

        # Define the column width
        self.tree.column("Day", width=50)
        self.tree.column("category", width=150)
        self.tree.column("Amount", width=50)



        # Add a scrollbar to the Treeview
        scrollbar = ttk.Scrollbar(self.entry_frame, orient=VERTICAL,
command=self.tree.yview)
```

```
        self.tree.configure(yscroll=scrollbar.set)
        scrollbar.pack(side=RIGHT, fill=Y)
        self.table_data()
        for expense in self.all_expenses:
            day, category, amount = expense
            self.tree.insert("", END, values=(day, category, amount))


        self.tree.pack(side=TOP)

    #updates treeview table after every expense entry
    def update_table(self):
        """Refresh the expense table with current data."""
        # Clear existing entries in the table
        for item in self.tree.get_children():
            self.tree.delete(item)

        # Get all expense data
        all_expenses = self.table_data()

        # Insert data into the table
        for expense in all_expenses:
            day, category, amount = expense
            self.tree.insert("", END, values=(day, category, amount))

    #to get data to insert into treeview table
    def table_data(self):
        categories = ["Food", "Rent", "Utilities", "Transportation", "Shopping",
"Life and Entertainment", "Other"]
        days = list(self.data_manager.data['expenses'].keys())

        self.all_expenses = [
            [day, category, expense]
            for day in days
            for category in categories
            for expense in self.data_manager.data['expenses'][day][category]
        ]

        return self.all_expenses


#to be added in the main window class
class FrameLeft(Construction):
    def __init__(self, parent, data_manager):
        self.data_manager = data_manager
        self.widget_frame_left = self.create_frame(parent, (300, 300), LEFT,
row=None, fill='y')
        self.widget_frame_left.configure(bg='#1c1d20')

        # Initialize the current values
        self.current_balance = data_manager.data["balance"]
```

```
        self.current_income = data_manager.data["income"]
        self.current_savings_goal = data_manager.data["savings_goal"]
        self.total_expense = self.data_manager.total_expense()
        self.new_balance = self.current_balance + self.current_income
        self.net_balance = self.new_balance - self.total_expense

        # Create Balance Entry and Button
        balance_label, self.balance_entry =
self.create_label_entry_pair(self.widget_frame_left, "Balance", 0, 18)
        balance_label.configure(fg = '#b09662', bg = '#421d59')
        self.balance_button = Button(self.widget_frame_left, text="Update",
font=('Arial', 15), command=self.update_balance, highlightbackground='#b09662')
        self.balance_button.grid(row=0, column=2, sticky=W, padx=(5, 0))

        # Create Income Entry and Button
        income_label, self.income_entry =
self.create_label_entry_pair(self.widget_frame_left, "Income", 1, 18)
        income_label.configure(fg = '#b09662', bg = '#421d59')
        self.income_button = Button(self.widget_frame_left, text="Update",
font=('Arial', 15), command=self.update_income, highlightbackground='#b09662')
        self.income_button.grid(row=1, column=2, sticky=W, padx=(5, 0))

        # Create Savings Goals Entry and Button
        savings_label, self.savings_entry =
self.create_label_entry_pair(self.widget_frame_left, "Monthly Savings Goals", 2,
18)
        savings_label.configure(fg = '#b09662', bg = '#421d59')
        self.savings_button = Button(self.widget_frame_left, text="Update",
font=('Arial', 15), command=self.update_savings, highlightbackground='#b09662')

        self.savings_button.grid(row=2, column=2, sticky=W, padx=(5, 0))


        # Create Display Labels for stats at the bottom of the frame
        self.balance_display = Label(self.widget_frame_left, text=f"Initial
Balance: ${self.current_balance:.2f}", font=('Arial', 20), fg = '#14d368', bg =
'#797d7f')
        self.balance_display.grid(row=3, column=0, columnspan=3, sticky=W,
pady=(100, 15))

        self.income_display = Label(self.widget_frame_left, text=f"Current Income:
${self.current_income:.2f}", font=('Arial', 20), fg = '#1559d7', bg = '#797d7f')
        self.income_display.grid(row=4, column=0, columnspan=3, sticky=W, pady=15)

        self.new_balance_display = Label(self.widget_frame_left, text=f"Balance
after Income: ${(self.new_balance):.2f}", font=('Arial', 20), fg = '#14d368', bg =
'#797d7f')
        self.new_balance_display.grid(row=5, column=0, columnspan=3, sticky=W,
pady=15)
```

```
        self.total_expense_display = Label(self.widget_frame_left, text=f"Total
Expense: ${(self.total_expense):.2f}", font=('Arial', 20), fg = '#f00e29', bg =
'#797d7f')
        self.total_expense_display.grid(row=6, column=0, columnspan=3, sticky=W,
pady=15)

        self.net_balance_display = Label(self.widget_frame_left, text=f"Net
Balance: ${(self.net_balance):.2f}", font=('Arial', 20), fg = '#14d368', bg =
'#797d7f')
        self.net_balance_display.grid(row=7, column=0, columnspan=3, sticky=W,
pady=15)

        self.savings_display = Label(self.widget_frame_left, text=f"Monthly Savings
Goal: ${self.current_savings_goal:.2f}", font=('Arial', 20), fg = '#14d368', bg =
'#797d7f')
        self.savings_display.grid(row=8, column=0, columnspan=3, sticky=W, pady=15)

        self.daily_savings_expense_display = Label(self.widget_frame_left,
text=f"Daily Savings Expense: ${self.calculate_daily_savings_expense():.2f}",
font=('Arial', 20), fg = '#14d368', bg = '#797d7f')
        self.daily_savings_expense_display.grid(row=9, column=0, columnspan=3,
sticky=W, pady=15)

        #feasibility font colour is dependent on feasiblity it self
        if self.get_feasibility(self.calculate_daily_savings_expense()) == "Not
Feasible":
            self.feasibility_display = Label(self.widget_frame_left,
text=self.get_feasibility(self.calculate_daily_savings_expense()), font=('Arial',
20), fg = '#f00e29', bg = '#797d7f')
        elif self.get_feasibility(self.calculate_daily_savings_expense()) ==
"Possible but difficult":
            self.feasibility_display = Label(self.widget_frame_left,
text=self.get_feasibility(self.calculate_daily_savings_expense()), font=('Arial',
20), fg = '#f27c1b', bg = '#797d7f')
        elif self.get_feasibility(self.calculate_daily_savings_expense()) ==
"Possible":
            self.feasibility_display = Label(self.widget_frame_left,
text=self.get_feasibility(self.calculate_daily_savings_expense()), font=('Arial',
20), fg = '#14d368', bg = '#797d7f')
        self.feasibility_display.grid(row=10, column=0, columnspan=3, sticky=W,
pady=15)

        # Subscribe to data changes
        self.data_manager.subscribe(self.update_frame)

    def calculate_daily_savings_expense(self):
        """Calculates the daily amount available after the savings goal is
subtracted from the income."""
        #self.current_income and self.current_savings_goal cannot be 0 when
calculating
        if self.current_income and self.current_savings_goal:
```

```
            return (self.current_income - self.current_savings_goal) / 28
        return 0.0

    #get balance input and updates file data and in UI
    def update_balance(self):
        """Updates the balance and reflects the changes in the UI."""
        #error handling
        try:
            balance_value = float(self.balance_entry.get())
            if balance_value <=0:
                messagebox.showerror("Invalid Input", "Please enter a positive
number for balance.")
            else:

                self.current_balance = balance_value

                # Update the data manager
                self.data_manager.data["balance"] = self.current_balance
                self.data_manager.save_data()

                self.balance_display.config(text=f"Current Balance:
${self.current_balance:.2f}")
                self.update_displays()
                messagebox.showinfo("Update Successful", "Balance updated
successfully.")
        except ValueError:
            messagebox.showerror("Invalid Input", "Please enter a valid number for
balance.")

    #get income input and updates file data and in UI
    def update_income(self):
        """Updates the income and reflects the changes in the UI."""
        try:
            income_value = float(self.income_entry.get())

            if income_value <=0:
                messagebox.showerror("Invalid Input", "Please enter a positive
number for Income.")

            else:
                self.current_income = income_value

                # Update the data manager
                self.data_manager.data["income"] = self.current_income
                self.data_manager.save_data()

                self.income_display.config(text=f"Current Income:
${self.current_income:.2f}")
                self.update_displays()
                messagebox.showinfo("Update Successful", "Income updated
successfully.")
```

```
        except ValueError:
            messagebox.showerror("Invalid Input", "Please enter a valid number for
income.")

    def update_savings(self):
        #Updates the savings goal and reflects the changes in the UI.
        try:
            savings_value = float(self.savings_entry.get())

            if savings_value <= 0:
                messagebox.showerror("Invalid Input", "Please enter a positive
number for Savings Goal.")

            elif savings_value > self.current_income:
                messagebox.showerror("Invalid Input", "Savings Goal must be less
than Income")
            else:
                self.current_savings_goal = savings_value

                # Update the data manager
                self.data_manager.data["savings_goal"] = self.current_savings_goal
                self.data_manager.save_data()

                self.savings_display.config(text=f"Monthly Savings Goal:
${self.current_savings_goal:.2f}")
                self.update_displays()
                messagebox.showinfo("Update Successful", "Savings Goal updated
successfully.")
        except ValueError:
            messagebox.showerror("Invalid Input", "Please enter a valid number for
savings goal.")

    #updates widgets after change in data from within the window
    def update_displays(self):
        """Updates all display labels to reflect the current state."""
        # Calculate the new balance and net balance
        self.new_balance = self.current_balance + self.current_income
        self.total_expense = self.data_manager.total_expense()
        self.net_balance = self.new_balance - self.total_expense

        # Update display labels with new values
        self.new_balance_display.config(text=f"Balance after Income:
${self.new_balance:.2f}")
        self.total_expense_display.config(text=f"Total Expense:
${self.total_expense:.2f}")
        self.net_balance_display.config(text=f"Net Balance:
${self.net_balance:.2f}")

        # Update the daily savings expense
        daily_savings_expense = self.calculate_daily_savings_expense()
```

```
        self.daily_savings_expense_display.config(text=f"Daily Savings Expense:
${daily_savings_expense:.2f}")

        # Update feasibility display
        if self.get_feasibility(daily_savings_expense) == "Not Feasible":

self.feasibility_display.config(text=self.get_feasibility(daily_savings_expense),fg
= '#f00e29', bg = '#797d7f')
        elif self.get_feasibility(daily_savings_expense) == "Possible but
difficult":

self.feasibility_display.config(text=self.get_feasibility(daily_savings_expense),fg
= '#f27c1b', bg = '#797d7f')
        elif self.get_feasibility(daily_savings_expense) == "Possible":

self.feasibility_display.config(text=self.get_feasibility(daily_savings_expense),
fg = '#14d368', bg = '#797d7f')

    #funciton to access the feasibility of saving goal
    def get_feasibility(self, daily_expense):
        """Determines the feasibility of saving based on daily expenses."""
        if daily_expense <= 10:
            return "Not Feasible"
        elif daily_expense <= 20:
            return "Possible but difficult"
        return "Possible"

    #updates frame if there is any change in data within the window
    def update_frame(self):
        """Callback to update the frame when data changes."""
        self.current_balance = self.data_manager.data["balance"]
        self.current_income = self.data_manager.data["income"]
        self.current_savings_goal = self.data_manager.data["savings_goal"]

        # Update displays to reflect any changes
        self.update_displays()

        # Update the balance, income, and savings goal displays
        self.balance_display.config(text=f"Current Balance:
${self.current_balance:.2f}")
        self.income_display.config(text=f"Current Income:
${self.current_income:.2f}")
        self.savings_display.config(text=f"Monthly Savings Goal:
${self.current_savings_goal:.2f}")

#to be added in the main window class
class GraphFrame(Construction):
    def __init__(self, parent, data_manager):
        self.data_manager = data_manager
        self.graph_frame = self.create_frame(parent, (750, 750), TOP)
```

```
        # Store references to the figure and canvas
        self.fig = None
        self.canvas = None

        # Subscribe to data changes
        self.data_manager.subscribe(self.update_graph)

        # Initial plot
        self.plot_graph(self.graph_frame)

    def plot_graph(self, frame):
        """Initializes and draws the graph with existing data."""
        # Initialize figure and axes
        self.fig, ax = plt.subplots(figsize = (4, 8))  # Adjust figsize as needed

        # Fetch existing data from the data manager
        self.days = list(range(1, 29))  # Assuming 28 days for simplicity
        self.numbers = self.get_monthly_balance(self.data_manager.data)

        # Retrieve income and savings goal
        income = self.data_manager.data['income']
        balance = (self.data_manager.data["balance"]) + income
        monthly_savings_goal = (self.data_manager.data["savings_goal"])
+(self.data_manager.data["balance"])

        # Distribute income and savings goal equally for visualization
        self.balance = [balance] * 28  # Distribute monthly income equally
        self.savings_goal = [monthly_savings_goal] * 28  # Distribute savings goal
equally

        # Plot the initial data
        ax.plot(self.days, self.numbers, color='blue', linewidth=2, marker='o',
markerfacecolor='blue', markersize=3, label='Net Balance')
        ax.plot(self.days, self.balance, color='green', linewidth=1,
linestyle='dotted', label='Balance After Income')
        ax.plot(self.days, self.savings_goal, color='red', linewidth=1,
linestyle='dotted', label='Savings Goal')

        # Set labels and title
        ax.set_xlabel('Days')
        ax.set_ylabel('Balance ($)')
        ax.set_ylim(bottom=0)
        ax.set_title('Balance Graph')
        ax.legend(loc='upper right', fontsize = 10)  # Add legend

        # Adjust ticks
        ax.tick_params(axis='both', which='major', labelsize=10)  # Adjust the
labelsize as needed
```

```
        # Create canvas for the figure
        self.canvas = FigureCanvasTkAgg(self.fig, master=frame)
        self.canvas.draw()
        self.canvas.get_tk_widget().pack(side=TOP, fill=X, expand=False)

    #updates graph according to new set of data
    def update_graph(self):
        """Updates the graph with new data when notified of data changes."""
        # If the figure exists, clear the axes
        if self.fig:
            self.fig.clear()

            # Create new axes
            ax = self.fig.add_subplot(111)

            # Retrieve updated data
            self.days = list(range(1, 29))  # Assuming 28 days for simplicity
            self.numbers = self.get_monthly_balance(self.data_manager.data)

            # Update income and savings goal for visualization
            monthly_income = self.data_manager.data["income"]
            monthly_savings_goal = self.data_manager.data["savings_goal"] +
(self.data_manager.data["balance"])
            self.income = [monthly_income] * 28  # Distribute monthly income
equally
            self.savings_goal = [monthly_savings_goal] * 28  # Distribute savings
goal equally

            # Plot the updated data
            ax.plot(self.days, self.numbers, color='blue', linewidth=2, marker='o',
markerfacecolor='blue', markersize=3, label='Expenses')
            ax.plot(self.days, self.income, color='green', linewidth=1,
linestyle='dotted', label='Income')
            ax.plot(self.days, self.savings_goal, color='red', linewidth=1,
linestyle='dotted', label='Savings Goal')

            # Update labels and title
            ax.set_xlabel('Days')
            ax.set_ylabel('Amount ($)')
            ax.set_ylim(bottom=0)
            ax.set_title('Expenses Graph')
            ax.legend(loc='upper right', fontsize = 10)  # Add legend

            # Adjust ticks
            ax.tick_params(axis='both', which='major', labelsize=10)  # Adjust the
labelsize as needed

            # Redraw the canvas with the new figure
            self.canvas.draw()

    def get_monthly_balance(self, data):
```

```
        categories = ["Food", "Rent", "Utilities", "Transportation", "Shopping",
"Life and Entertainment", "Other"]

        # Initialize list to hold monthly expenses
        monthly_balance = []
        balance = data['balance'] + data['income']

        # Iterate over each day
        for day in range(1, 29):  # Use the actual range of days as integers
            day_str = str(day)    # Convert day to string to use as a key in the
JSON data
            daily_expenses = []   # List to store daily expenses per category


            # Check if the day exists in the data
            for category in categories:
                # Check if the category exists for the day, and sum the expenses if
it does
                category_sum = sum(data['expenses'][day_str].get(category, []))
                daily_expenses.append(category_sum)

            # Calculate the total expenses for the day
            daily_total = sum(daily_expenses)
            balance = balance - daily_total

            # Append daily total to monthly expenses
            monthly_balance.append(balance)

        # Print the list of monthly expenses
        return monthly_balance
#################################################################


######################MIAN WINDOW#############################
class WINDOW(Tk, Construction):
    def __init__(self, current_user):
        super().__init__()
        self.title('Budget Planner')
        self.configure(bg = '#0b0b0b')
        self.geometry('1512x982')
        self.label = Label(self, text= f"{current_user.capitalize()}'s Budget
Planner", font=('Arial', 30), fg = '#b09662', bg = 'black')
        self.label.pack(pady=3)

        # Create a single data manager for the whole application
        self.user_manager = UserDataManager(current_user)


        # Pass the data manager to each frame
        FrameRight(self, self.user_manager)
        FrameLeft(self, self.user_manager)
```

```
        GraphFrame(self, self.user_manager)
#########################################################################


############################MAIN LINE####################################
if __name__ == "__main__":
    #define UserManager class to call in LOGIN class
    user_manager = UserManager()
    login_app = LOGIN(user_manager)
```

5. Testing

5.1 Testing for development

```python
class FrameRight(Construction):
    def expense(self, frame):
        self.expense_amount = self.create_label_entry_pair(frame, 'Amount:', 1, 13)
        Label(frame, text="Date:").grid(row=2, column=0, sticky=W)
        self.income_date = DateEntry(frame)
        self.income_date.grid(row=2, column=1)
        Label(frame, text="Category:").grid(row=3, column=0, sticky=W)
        self.expense_category = ttk.Combobox(frame, values=["Food", "Rent", "Utilities", "Transportation", "Shopping", "Life and Entertainment", "Other"])
        self.expense_category.grid(row=3, column=1)
        Button(frame, text="Add Expense", font=('Arial', 15), command=self.add_expense).grid(row=4, column=0, columnspan=2, pady=10)
        Button(frame, text="View Expense", font=('Arial', 15), command=self.expensepie).grid(row=5, column=0, columnspan=2, pady=10)

    def add_expense(self):
        pass

    def expensepie(self):
        self.labels = ['Food', 'Rent', 'Utilities', 'Transportation', 'Shopping', 'Entertainment', 'Other']
        self.sizes = [20, 30, 10, 10, 15, 10, 5]  # Percentage distribution

        # Create the pie chart
        fig, ax = plt.subplots()
        ax.pie(self.sizes, labels=self.labels, autopct='%1.1f%%', startangle=90)

        # Equal aspect ratio ensures that pie is drawn as a circle
        ax.axis('equal')

        # Title of the pie chart
        plt.title('Monthly Expenses Distribution')

        # Display the pie chart
        plt.show(block=False)  # Use block=False to avoid blocking the Tkinter event loop
```

Fig 5.1.1: Code of piechart before update

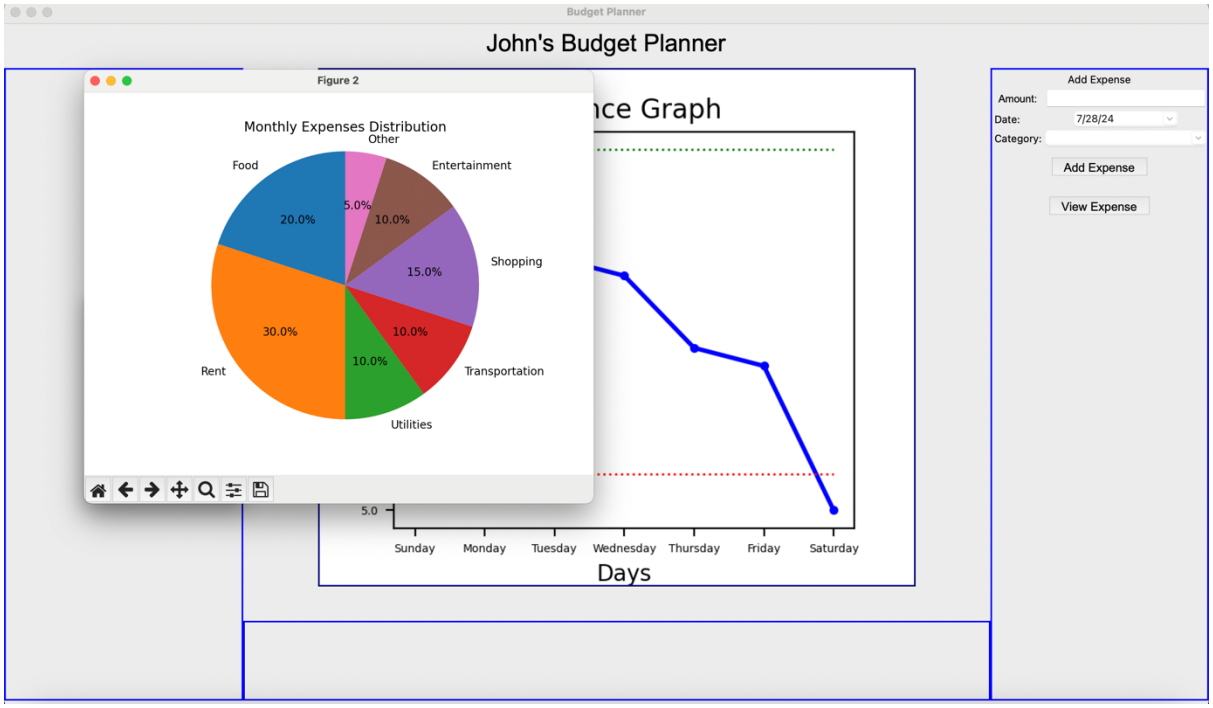

Fig 5.1.2: Intial GUI with piechart

Initially, as I was integrating a piechart to illustrate the proportion of expense in different categories, I had simply used the show() function thinking that the piechart would show when the button was pressed. However, it was seen for the whole programme to crash immediately after closing the piechart window. After much research, I found that the code crashes because the plt.show() function, used to display the pie chart, cannot run properly within the Tkinter event loop. This function blocks the execution and creates a new window for the plot, which interferes with Tkinter's mainloop.

```
def expensepie(self):
    # Create a new Toplevel window
    pie_window = Toplevel(self.widget_frame_right)
    pie_window.title("Expense Distribution")

    # Set the size of the Toplevel window
    pie_window.geometry("400x400")

    # Create the pie chart
    fig, ax = plt.subplots()
    self.labels = ['Food', 'Rent', 'Utilities', 'Transportation', 'Shopping', 'Entertainment', 'Other']
    self.sizes = [20, 30, 10, 10, 15, 10, 5]  # Percentage distribution
    ax.pie(self.sizes, labels=self.labels, autopct='%1.1f%%', startangle=90)
    ax.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle
    plt.title('Monthly Expenses Distribution')

    # Create a canvas for the figure and add it to the Toplevel window
    canvas = FigureCanvasTkAgg(fig, master=pie_window)
    canvas.draw()
    canvas.get_tk_widget().pack(fill=BOTH, expand=True)

    # Ensure that the window is closed properly
    pie_window.protocol("WM_DELETE_WINDOW", pie_window.destroy)
```

Fig 5.1.3: Revised code of piechart

After much thought, I decided to create a Toplevel window, make the piechart into an image using FigureCanvasTkAgg and embed that picture onto the window to prevent any disruption with the tkinter loop.

5.2 Testing for Evaluation

5.2.1Login Window

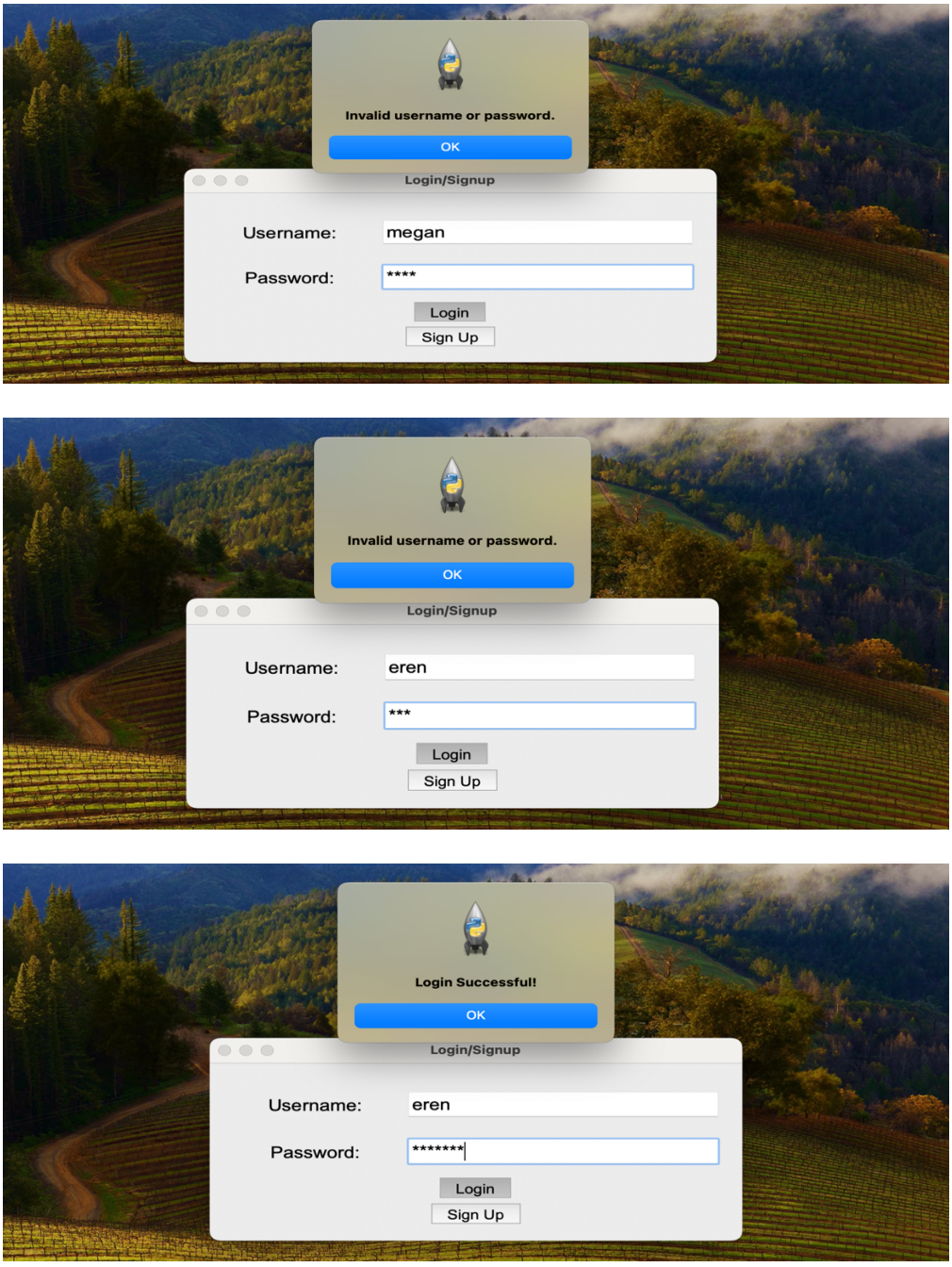

Fig 5.2.1a: Login GUI with different entries

Tested for invalid inputs for user and password Code was successful handling invalid inputs and would only proceed if input is valid.

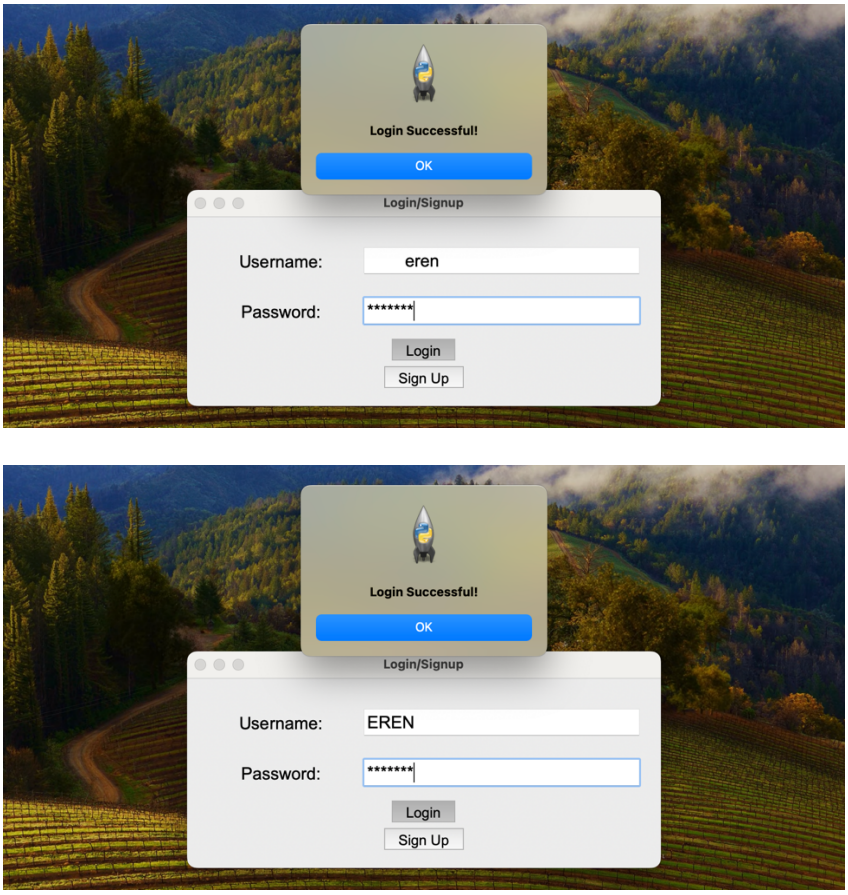

Fig 5.2.1b: Login GUI with different valid entries

Tested case insensitivity of username entry. Code was successful in handling these inputs and would still proceed.

5.2.2 Sign-up Window

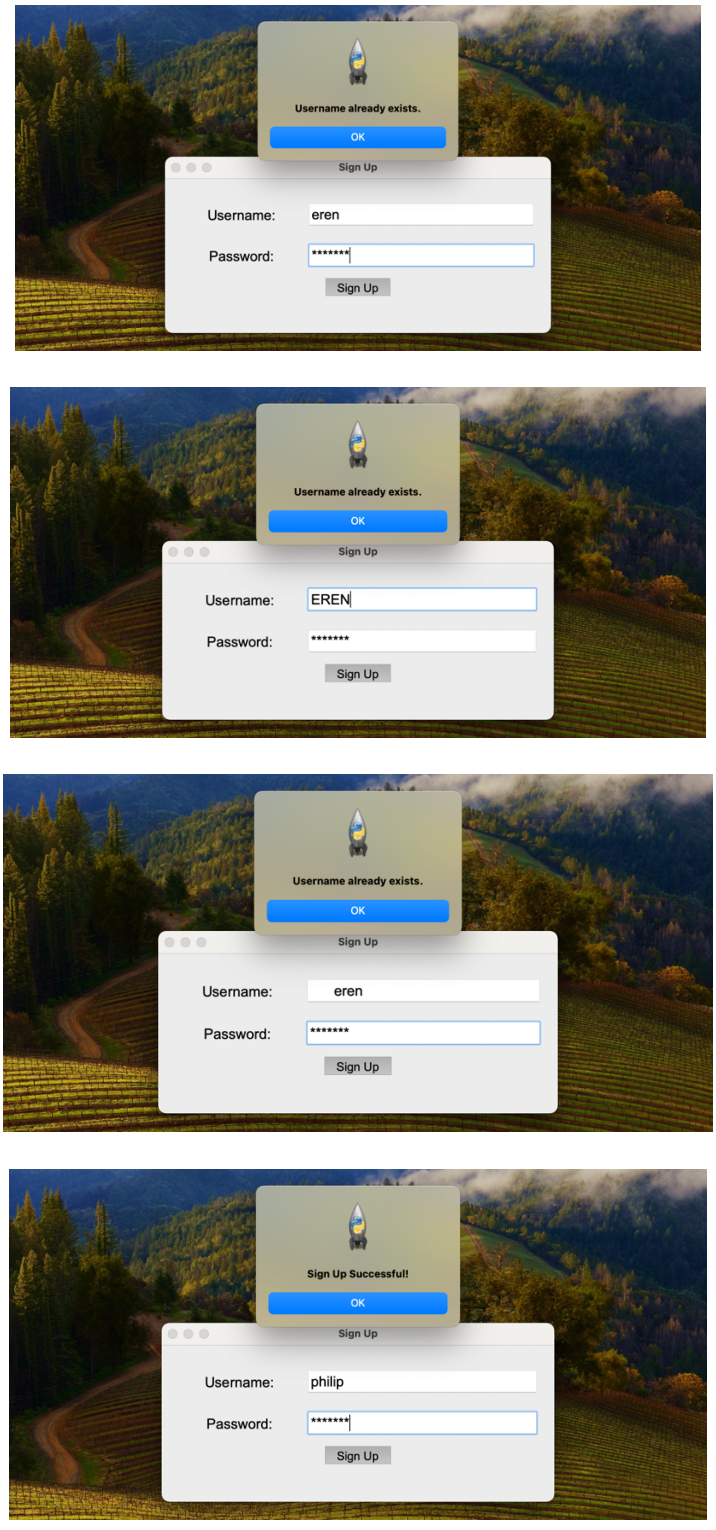

Fig 5.2.2a: Sign-up GUI with different entries

Tested for existing usernames. Code was successful in catching these usernames and would not add users unless they have a username that does not already exist within the csv file.

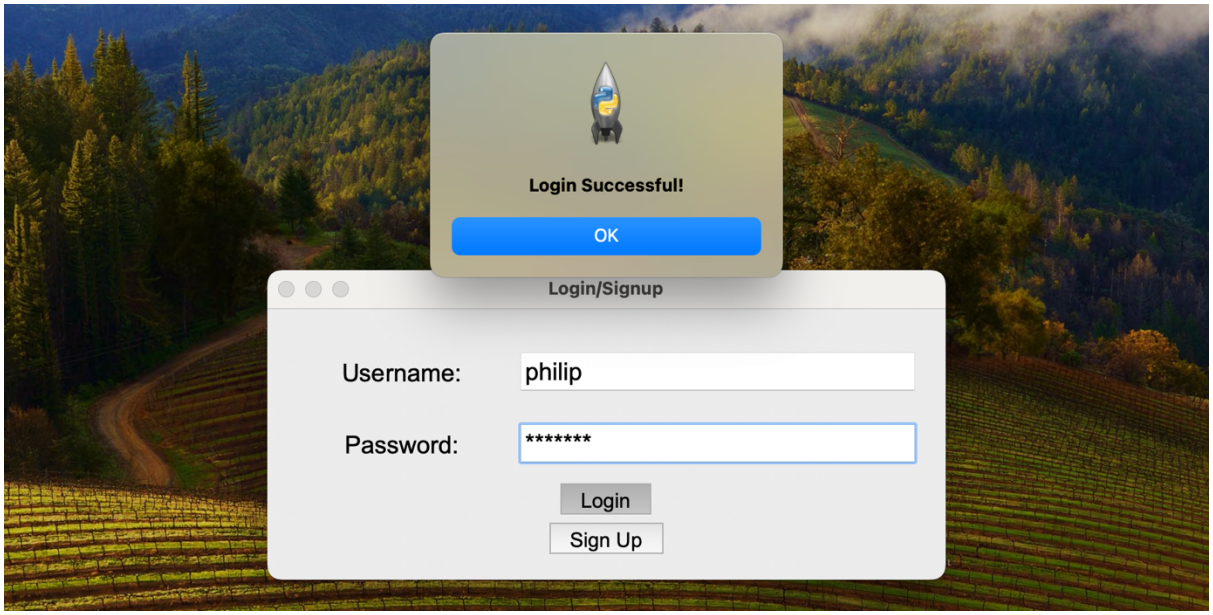

Fig 5.2.2a: Login GUI with a newly registered account

Code successfully stores data into the csv file via the sign up window.

5.2.3 Main Window

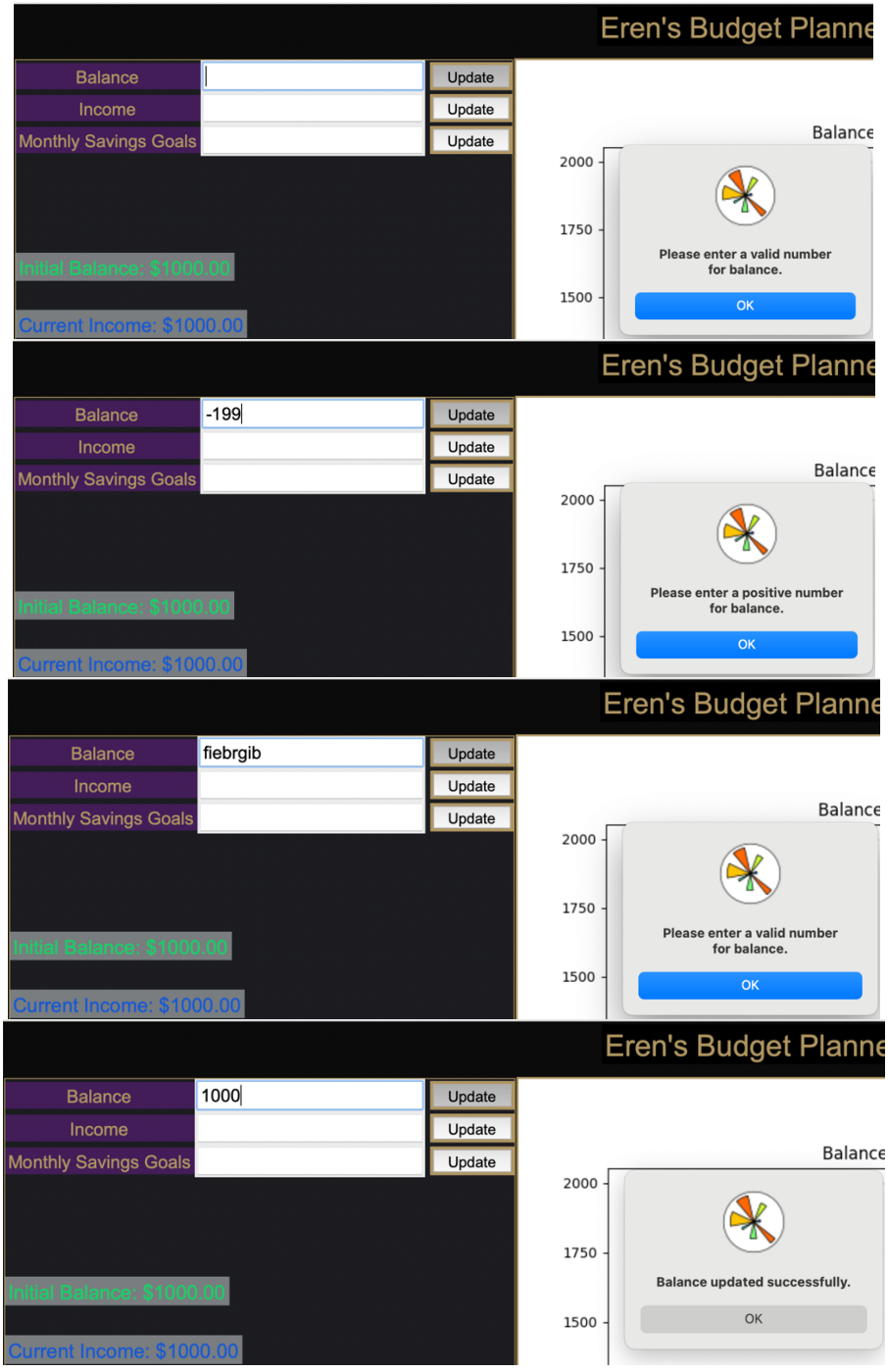

Fig 5.2.3a: Balance entry GUI with different entries

Tested for invalid entries for balance entry with various non integer and negative integers including a blank response. Code was successful handling invalid inputs and would only update user data if entry is valid.

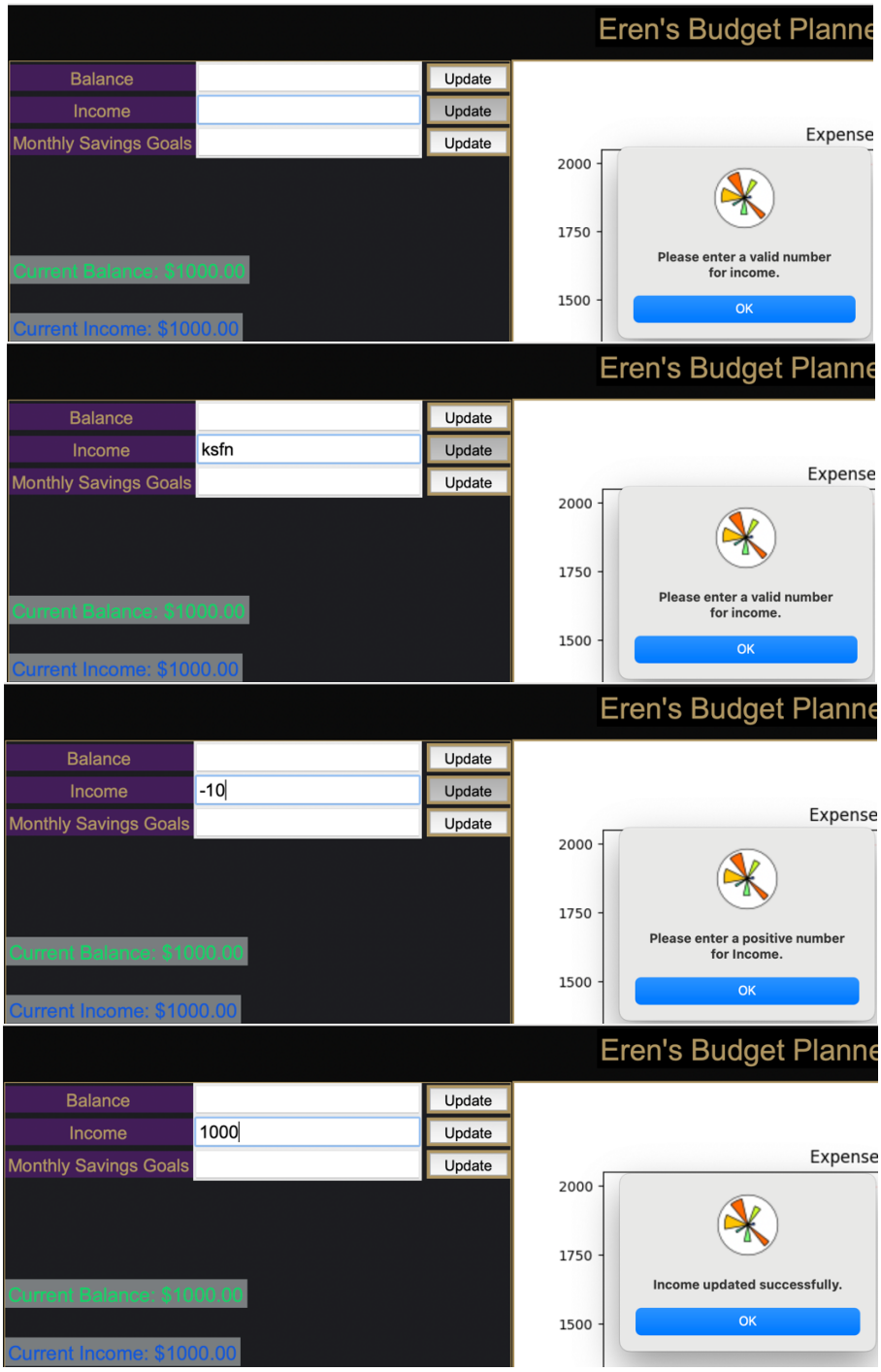

Fig 5.2.3b: Income entry GUI with different entries

Tested for invalid entries for income entry with various non integer and negative integers including a blank response. Code was successful handling invalid inputs and would only update user data if entry is valid.

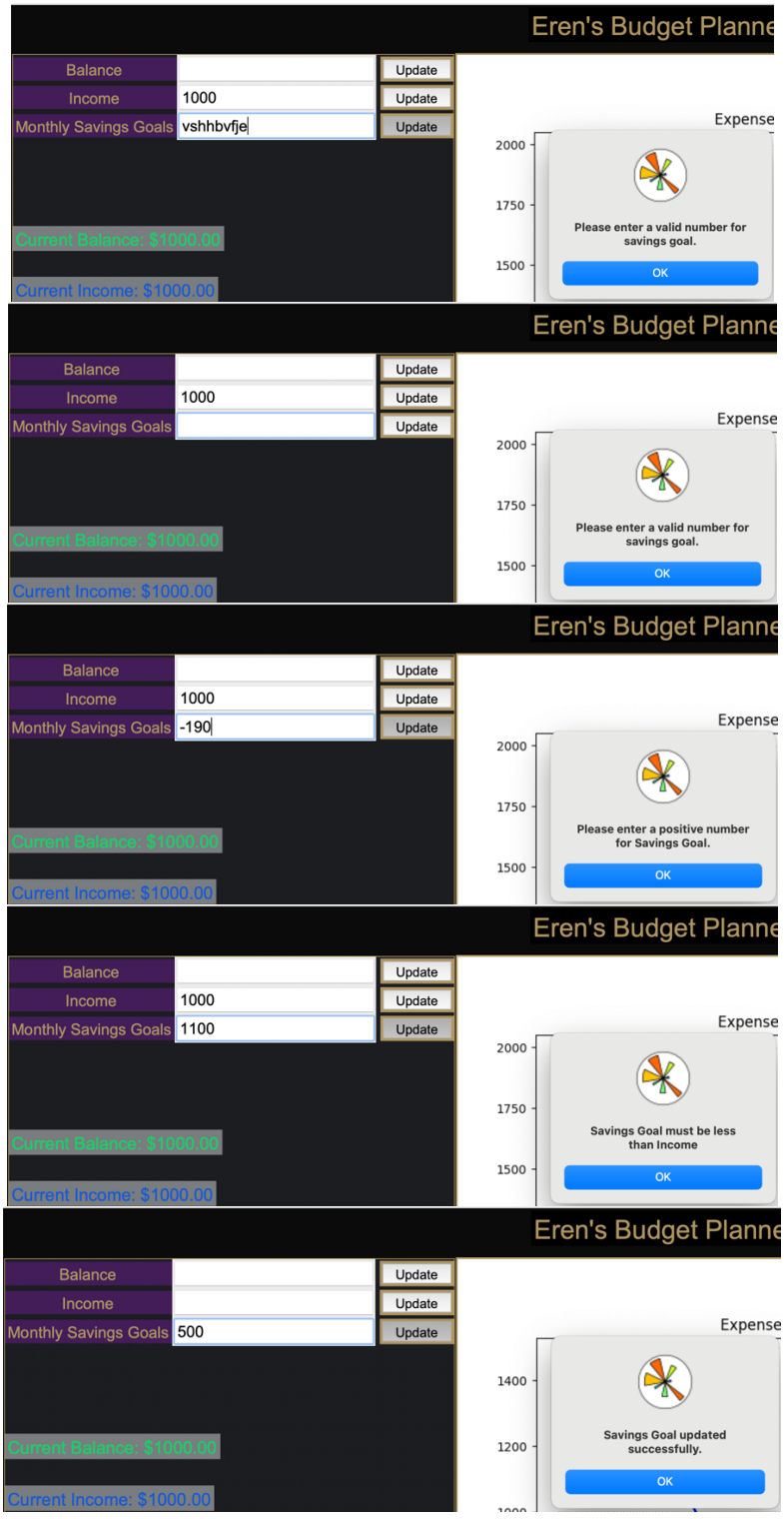

Fig 5.2.3c: Monthly Saving Goal entry GUI with different entries

Tested for invalid entries for Monthly Savings Goal entry with various non integer and negative integers including a blank response. Additionally, tested for the condition that the saving goal should be less than the income. Code was successful handling invalid inputs and would only update user data if entry is valid.

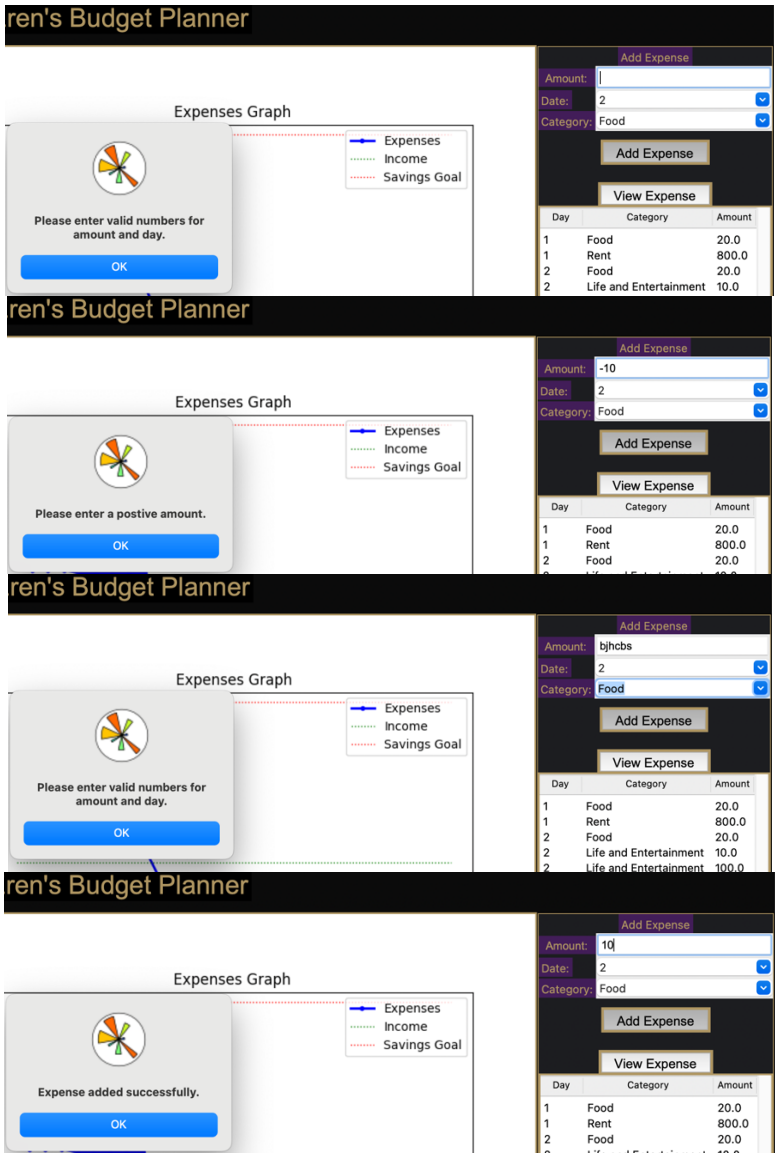

Fig 5.2.3d: Monthly Saving Goal entry GUI with different entries

Tested for invalid entries for expense entry with various non integer and negative integers including a blank response. Code was successful handling invalid inputs and would only update user data if entry is valid.

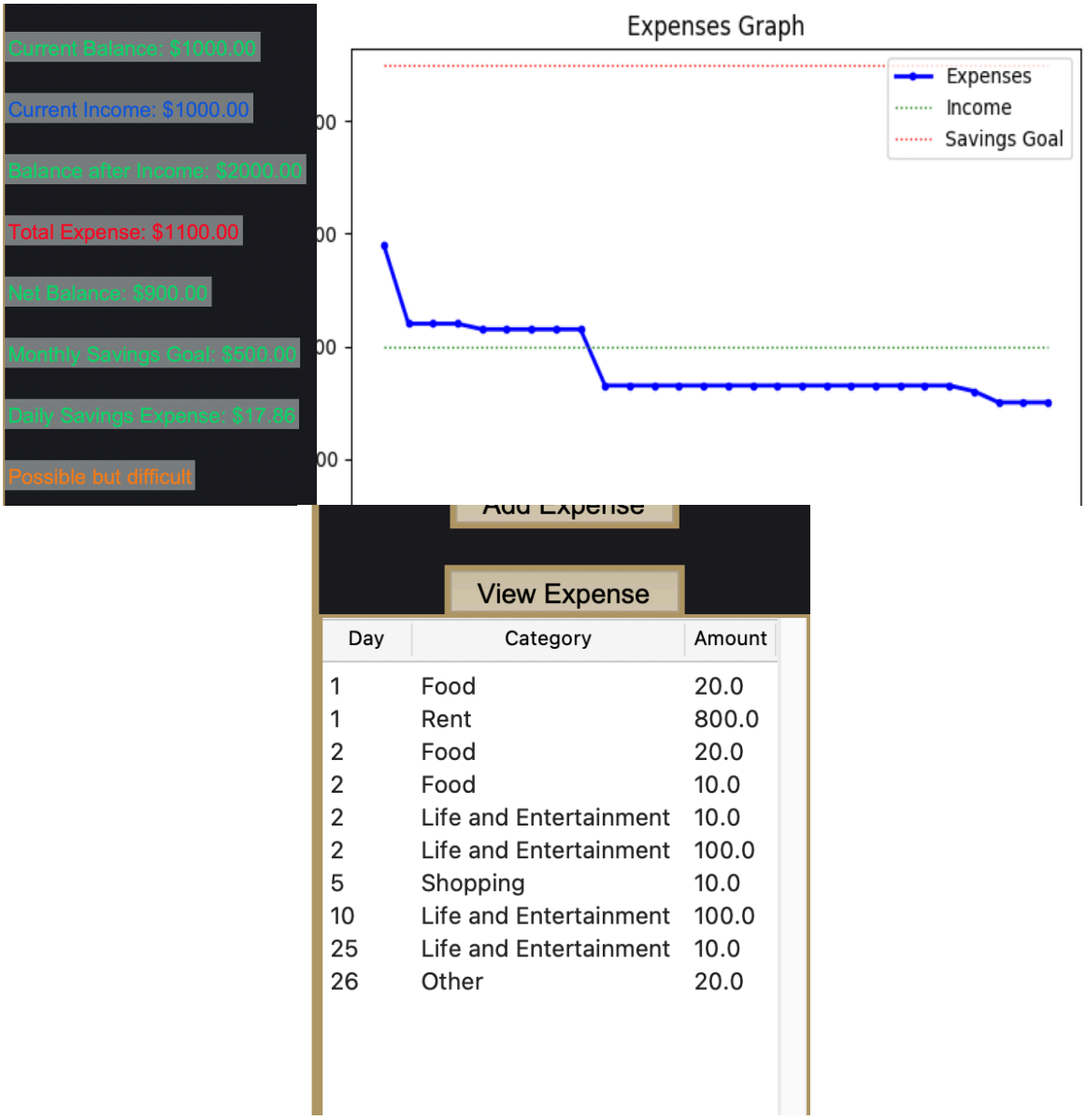

Fig 5.2.3e: Monthly Saving Goal entry GUI with different entries

From the successful entries, widgets were successful in updating data.

6. Evaluation and Summary

My Budget Tracker was an application aimed at offering users a platform for the setup of personal accounts, making entries, monitoring, and evaluation of their budgetary data. All these have been quite successfully realized with features such as user account management, data entry, and budget evaluation, all of which are facilitated by the Tkinter library in the creation of a user-friendly interface for the application. The first and foremost challenge was to learn Tkinter, which was unknown to me, including understanding how a responsive GUI is designed. Another challenge was handling the JSON data, for which I needed to learn proper data storage handling to keep its integrity. Overcoming these challenges deepened my understanding of these technologies and programming practices. Improvement areas for the next version include reporting and analytics with graphical tools, budget goals and alerts, and accessibility to mobile and web platforms. Overall, this project was an invaluable learning experience

Improvement ideas:

- improve design of Login/Sign-up window
- implement more functions in the main Window to enhance user experience (eg. Add User Settings Panel for users to change password, Add a clear data function if users want to start anew, etc.)
- Implement more graphs to illustrate other data

7. References

Built In. (2021). *__name__ in Python Explained*. [online] Available at: https://builtin.com/articles/name-python#:~:text=%E2%80%9CIf%20__name__%3D%3D%20 [Accessed 28 Jul. 2024].

Codemy.com (2020). *How to Center a Tkinter Window on the Screen - Python Tkinter GUI Tutorial #126*. [online] YouTube. Available at: https://www.youtube.com/watch?v=TdTks2eSx3c&ab_channel=Codemy.com [Accessed 28 Jul. 2024].

EDUCBA. (2021). *Tkinter Button Color | How to Color Button in Tkinter with examples?* [online] Available at: https://www.educba.com/tkinter-button-color/.

matplotlib.org. (n.d.). *matplotlib.axes.Axes.set_xticks — Matplotlib 3.8.0 documentation*. [online] Available at: https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.set_xticks.html.

OpenAI (2024). *ChatGPT* . [online] ChatGPT. Available at: https://chatgpt.com/.

W3Schools (2019). *Python Classes*. [online] W3schools.com. Available at: https://www.w3schools.com/python/python_classes.asp.

www.tutorialspoint.com. (n.d.). *Python - Tkinter pack() Method*. [online] Available at: https://www.tutorialspoint.com/python/tk_pack.htm.