



L OVELY
P ROFESSIONAL
U NIVERSITY

Transforming Education Transforming India

Python Project Final Report on EXPENSE TRACKER

GitHub: <https://github.com/JerardMFrancis/expense-tracker>

Name: Jerard M Francis

Registration No: 12310619

Section: K23UP, G2

Roll No: 41

Submitted to:

Mr. Aman Kumar

Acknowledgment

I would like to express my sincere gratitude to all those who supported and guided me throughout the development of this project, the **Expense Tracker** in Python. Their invaluable assistance and insights have been instrumental in the completion of this project.

Firstly, I would like to thank my Professor Mr. Aman Kumar, for their continuous support, encouragement, and constructive feedback. Their expertise in programming and guidance on structuring the project provided me with the foundation to approach and complete the project successfully.

I would also like to acknowledge the resources provided by Lovely Professional University, which offered valuable reference material and tutorials that significantly helped in understanding the concepts required for game development and implementing the game logic.

Finally, I am grateful to my peers, friends, and family, whose encouragement and belief in my abilities motivated me to overcome challenges and complete this project to the best of my ability.

Jerard M Francis

12310619

Table of Contents

1. Introduction
2. Objectives and Scope of the Project
3. Application Tools
4. Project Design
5. Flowchart
6. Project Implementation
7. Testing and Validation
8. Conclusion
9. References

1. Introduction

1.1 Project Overview

The Expense Tracker is a Python-based application designed to simplify the management of personal and household finances. Built using the Tkinter GUI library, this system enables users to effortlessly input, view, categorize, and manage their daily expenses. The application provides a comprehensive platform for tracking expenses such as groceries, utilities, transportation, and other categories, allowing users to maintain control over their spending habits. With a clean and user-friendly interface, the system allows users to add new expense entries, view all records in a tabular format, calculate the total expenditure, and delete unnecessary or outdated entries. Additionally, the system provides optional features like saving and loading data from files, ensuring data persistence. The Expense Tracker is a practical and effective tool for individuals seeking to enhance their financial discipline and manage their budgets efficiently.

1.2 Purpose and Significance

Purpose:

The primary objective of this project is to demonstrate an efficient and automated system for managing personal finances. It aims to simplify the process of recording, categorizing, and analysing daily expenses, thereby reducing the manual effort and errors often associated with traditional methods like pen-and-paper or spreadsheets. Effective expense tracking plays a critical role in understanding spending habits, setting financial goals, and creating budgets.

This project illustrates how a Python-based solution, with a user-friendly graphical interface (GUI), can make financial management more accessible and accurate.

Significance:

The significance of this system lies in its ability to centralize and automate the tracking of expenses. By providing users with an intuitive platform for adding, reviewing, and managing their financial data, the Expense Tracker ensures that financial records are kept up-to-date and easily accessible. This not only saves time but also enhances financial awareness and discipline, empowering users to take control of their financial health. The application helps users identify spending patterns, allocate budgets, and make informed financial decisions, which are crucial steps in achieving long-term financial stability.

Additionally, this project serves as a foundation for future expansions, including the integration of data visualization tools and the generation of detailed financial reports. These features will offer users deeper insights into their spending habits and enable them to plan better. As the system evolves, it will become increasingly adaptable to meet the needs of diverse users, providing a valuable resource for individuals and households committed to improving their financial well-being.

2. Objectives and Scope of the Project:

2.1 Project Objectives

The primary objective of this project is to design and implement an Expense Tracker application that efficiently manages personal finances using Python and Tkinter. The system aims to provide a user-friendly interface for tracking, categorizing, updating, and analyzing daily expenses. Specific objectives include:

1. Develop a Comprehensive Expense Management System:
 - Create a system that stores and manages various aspects of financial data, including expense name, amount, category, date, and description.
 - The system should allow users to efficiently add new expenses, update existing records, search for specific expenses, and delete records when necessary.
2. Implement an Organized Data Storage Solution:
 - Use Python's in-memory list data structure to store and manage expense records.
 - Future iterations can explore integrating a database (e.g., SQLite) for persistent storage and scalable data management.
3. Demonstrate Efficient Data Management and Interaction:
 - Showcase the system's functionality by allowing users to interact with expense data through a GUI.
 - Implement features for adding, searching, updating, and deleting expense records.
 - Provide a user-friendly way to display all expense records in a table format for easy viewing.
4. Highlight the Significance of Financial Management:
 - Illustrate the importance of tracking expenses and how it helps users stay within budgets, save money, and make informed financial decisions.
 - Provide insights into how the system can reduce manual record-keeping errors and improve financial awareness.
5. Design a Scalable and Extendable Solution:
 - Create a flexible architecture that allows for future enhancements, such as adding features like budgeting, financial reports, or integration with other financial tools.

2.1 Project Scope

The scope of this project is focused on the design, development, and demonstration of an **Expense Tracker** in Python, with the following key areas of focus:

1. **Expense Entity Structure:**
 - Define an **Expense** class with attributes such as expense name, amount, category, date, and description. This class will serve as the foundational data structure for managing expense records.
2. **In-Memory Data Storage:**

- Store expense data in a list of **Expense** objects. Each expense record is an instance of the **Expense** class, providing easy access to and management of individual financial information.
 - This approach is suitable for demonstration purposes and smaller datasets. Future improvements could involve transitioning to a database for persistent storage.
3. **Expense Record Management Operations:**
- Implement the ability to add, update, search, delete, and display expense records.
 - Provide functionalities to retrieve expense data by category, amount, or date, update any expense details, and delete expense records when needed.
4. **Graphical User Interface (GUI):**
- Use Tkinter to build a user-friendly GUI for interacting with the system.
 - The GUI allows users to enter expense information, display records in a table format, and perform actions like searching, updating, and deleting records.
5. **Limitations:**
- The system currently focuses on basic expense record management and does not support advanced features like financial reports, budget tracking, or integration with bank account systems.
 - Data persistence is limited to in-memory storage, with no database integration at this stage. This means that data will be lost once the application is closed.
 - Security measures for sensitive financial data (such as data encryption) are not implemented in this version.

3. Application Tools

3.1 Software Applications

The development of the Expense Tracker project utilizes several software tools to ensure an efficient and organized development environment. These tools are as follows:

1. Python (Version 3.x):
 - Python is the core language used for implementing the system. It is a versatile, easy-to-learn, and powerful language that provides extensive libraries, including Tkinter for building the graphical user interface (GUI). Python's simplicity and readability make it an ideal choice for this project, as it allows for rapid development and easy maintenance.
2. Integrated Development Environment (IDE) - Visual Studio Code (VS Code):
 - VS Code is used for writing, debugging, and testing Python code. It provides a rich development environment with features such as syntax highlighting, code

linting, version control integration, and debugging tools, which make the development process smoother and more efficient.

3. Git and GitHub:

- Git is used for version control, while GitHub serves as a platform for code management and collaboration. Git tracks code changes, making it easy to manage different versions of the project. GitHub facilitates sharing and collaboration, allowing multiple developers to work on the project simultaneously and keeping the codebase synchronized.

4. Tkinter:

- Tkinter is used for building the graphical user interface (GUI). It provides a simple and effective way to create windows, buttons, labels, text boxes, and other GUI elements, making the system user-friendly and accessible for users.

5. SQLite (Optional):

- While the current version of the project uses in-memory storage for expense records, SQLite can be integrated in future versions for persistent data storage. SQLite is a lightweight, serverless relational database that is easy to use with Python through the sqlite3 module.

3.2 Programming Languages of the Project

The project is primarily implemented in Python, chosen for its simplicity, readability, and support for rapid development of desktop applications.

1. Python:

- Python serves as the core language for implementing the Expense class and the ExpenseTrackerApp class. The Expense class encapsulates expense data, while the ExpenseTrackerApp class provides methods for managing and processing records.
- Python's extensive standard libraries, such as Tkinter for GUI development and sqlite3 for optional database integration, make it a robust choice for this project.

2. SQLite (Optional):

- SQLite can be used in future versions to store expense data persistently. The sqlite3 library in Python allows for database integration, enabling the system to

store, retrieve, and update expense records even after the application is closed, making the system more scalable.

4. Project Design

The project design for the Expense Tracker application is structured into several core components, each serving a specific function to ensure efficient management and tracking of expenses. Below is an outline of the primary elements and their interactions:

1. System Initialization:

The main file initializes the application, setting up the GUI environment and loading necessary resources. It configures the layout for the main interface, which includes the display of the expense list, category selections, and navigation buttons for adding, editing, or deleting expenses.

2. Classes and Functions:

1. Class: Expense

- The Expense class represents individual expenses. It stores details such as the name of the expense, amount, category, date, and a unique identifier. This class also includes methods to validate the expense data and update the details when necessary.

2. Class: ExpenseTracker

- The ExpenseTracker class acts as the central controller for managing all expenses. It handles adding, deleting, and categorizing expenses, and

maintains a list of all recorded expenses. This class also handles the summary calculations (total expenses, categories, etc.) and organizes the data for display.

3. Class: Category

- The Category class defines the different types of expenses (e.g., Food, Transportation, Entertainment) and manages the categories available in the tracker. It helps users group expenses by type and generate category-based reports.

4. Class: ExpenseAnalyzer

- The ExpenseAnalyzer class provides the logic for analyzing spending patterns, generating expense reports, and creating visualizations (e.g., pie charts or bar graphs). It calculates the total expenses for a given period and can offer insights into the user's spending habits.

5. Function: GUIManager

- The GUIManager class controls the application's user interface, handling input from the user, such as adding new expenses, selecting categories, and interacting with the application's buttons and menus. It translates user actions into commands that interact with the ExpenseTracker class and updates the display accordingly.

3. Interaction Between Components:

1. The ExpenseTracker class interacts with the Expense and Category classes to manage expenses and categorize them.

2. The ExpenseAnalyzer class works closely with the ExpenseTracker class to retrieve data for analysis and visualization. It processes the expense data and generates summaries, charts, and reports.
3. The GUIManager captures user interactions, such as button clicks for adding or deleting expenses, and passes those commands to the appropriate classes (e.g., ExpenseTracker or Category) for execution. It also ensures that the interface is updated with the most current expense information and analytics.

4. Modular Structure and Data Flow:

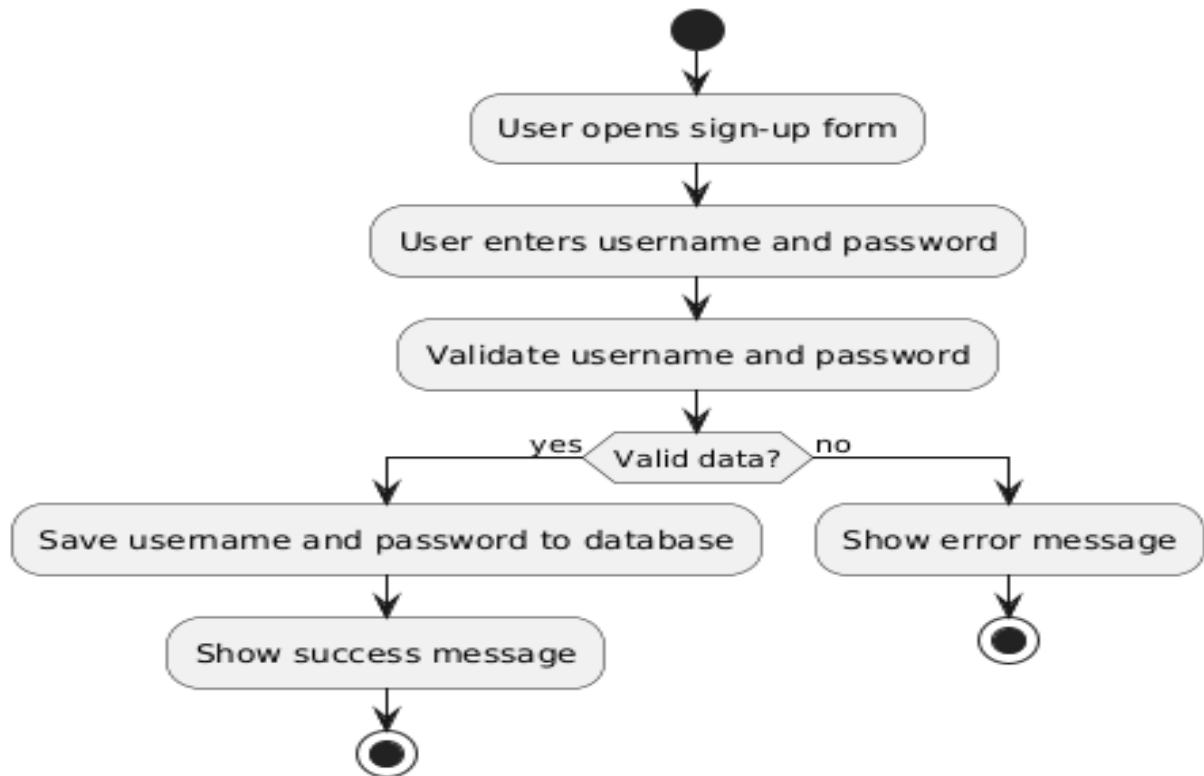
This modular design ensures that each component of the application has a clear and defined role, promoting separation of concerns. The ExpenseTracker class handles all expense-related data management, while the ExpenseAnalyzer is responsible for insights and reporting. The GUIManager connects all components, providing a seamless interface for user interactions.

- Expense management (adding, deleting, categorizing) is handled by the ExpenseTracker class.
- User interaction is managed through the GUIManager.
- Expense analysis and reporting is done by the ExpenseAnalyzer.

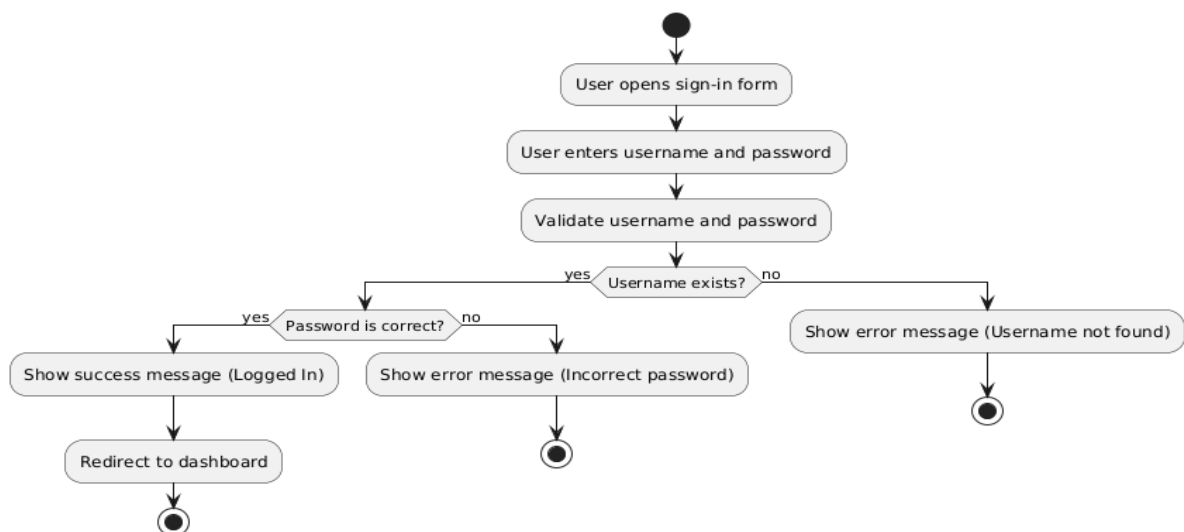
This structured approach ensures that each part of the application functions independently but integrates smoothly, creating a robust, scalable system for personal finance tracking.

5. DFD

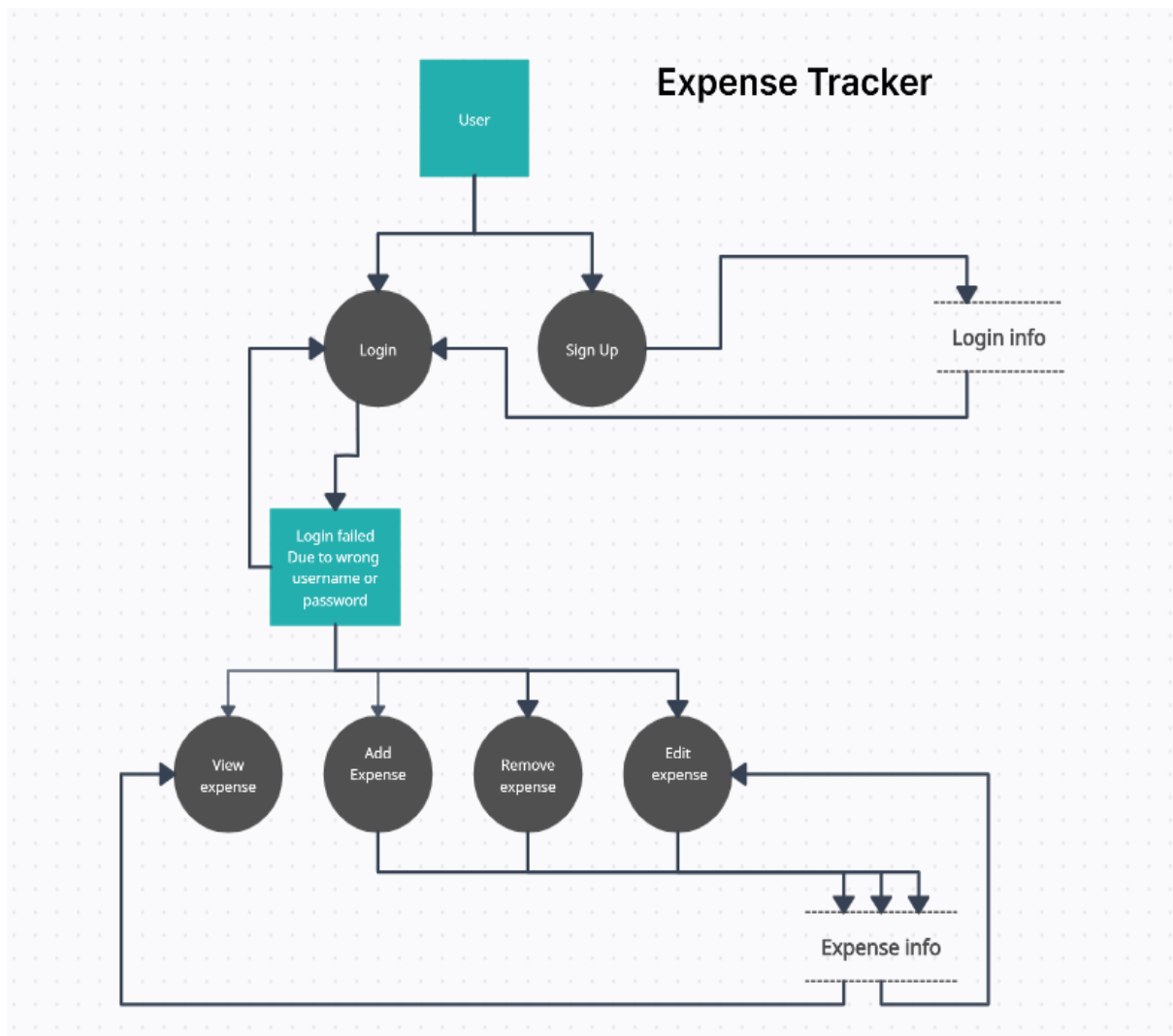
For Sign Up:




For Sign in:



Full DFD:



6. Project Implementation



Sign up


Username

Password

Conform Password

[Sign up](#)

[I have an account?](#) [Sign in](#)



Sign in

Username

Password

[Sign in](#)

[Don't have an account?](#) [Sign up](#)

EXPENSE TRACKER - JAVATPOINT

EXPENSE TRACKER

View Selected Expense's Details

Edit Selected Expense

Convert Selected Expense to a Sentence

Delete Selected Expense

Delete All Expense

Date: 11/22/24

Description:

Amount: 0.0

Payee:

Mode of Payment: Cash

Edit Expense

Convert to Text before Adding

Reset the fields

S.No.	Date	Payee	Description	Amount	Mode of Payment
3	2024-11-21	joe	food	250.0	Cash
11	2024-11-13	wdw	food	1200.0	Debit Card
13	2024-11-14	Johns	For food	120.0	Google Pay
			Total	1570.0	

Code Implementation:

```
from tkinter import *

from tkinter import ttk

from tkinter import messagebox as mb

import datetime

import sqlite3

from tkcalendar import DateEntry

# Function to list all expenses from the database

def listAllExpenses():

    global dbconnector, data_table

    data_table.delete(*data_table.get_children()) # Clear existing entries

    all_data = dbconnector.execute('SELECT * FROM ExpenseTracker') # Fetch
all records

    for val in all_data.fetchall():

        data_table.insert('', END, values=val) # Insert each record into
the table

# Function to view selected expense details

def viewExpenseInfo():

    global data_table, dateField, payee, description, amount, modeOfPayment
```



```

if not data_table.selection():

    mb.showerror('No expense selected', 'Please select an expense from
the table to view its details')

    return

val = data_table.item(data_table.focus())['values'] # Get selected
item

expenditureDate = datetime.date(int(val[1][:4]), int(val[1][5:7]),
int(val[1][8:])) # Parse date

dateField.set_date(expenditureDate) # Set date in the entry field

payee.set(val[2])

description.set(val[3])

amount.set(val[4])

modeOfPayment.set(val[5])

# Function to clear input fields

def clearFields():

    global description, payee, amount, modeOfPayment, dateField, data_table

    todayDate = datetime.datetime.now().date()

    description.set('')

    payee.set('')

    amount.set(0.0)

    modeOfPayment.set('Cash')

    dateField.set_date(todayDate) # Reset date to today

```

```

        data_table.selection_remove(*data_table.selection()) # Deselect any
selected item

# Function to remove the selected expense

def removeExpense():

    if not data_table.selection():

        mb.showerror('No record selected!', 'Please select a record to
delete!')

        return

    valuesSelected = data_table.item(data_table.focus())['values']

    if mb.askyesno('Are you sure?', f'Are you sure that you want to delete
the record of {valuesSelected[2]}'):

        dbconnector.execute('DELETE FROM ExpenseTracker WHERE ID=?',
(valuesSelected[0],)) # Delete record

        dbconnector.commit()

        listAllExpenses() # Refresh the table

        mb.showinfo('Record deleted successfully!', 'The record has been
deleted successfully')

# Function to remove all expenses

def removeAllExpenses():

    if mb.askyesno('Are you sure?', 'Are you sure that you want to delete
all the expense items from the database?'):

```

```

        dbconnector.execute('DELETE FROM ExpenseTracker') # Delete all
records

        dbconnector.commit()

        listAllExpenses() # Refresh the table

# Function to add a new expense

def addAnotherExpense():

    global dateField, payee, description, amount, modeOfPayment,
dbconnector

    if not dateField.get() or not payee.get() or not description.get() or
not amount.get() or not modeOfPayment.get():

        mb.showerror('Fields empty!', "Please fill all the missing fields
before pressing the add button!")

        return

    dbconnector.execute(

        'INSERT INTO ExpenseTracker (Date, Payee, Description, Amount,
ModeOfPayment) VALUES (?, ?, ?, ?, ?)',

        (dateField.get_date(), payee.get(), description.get(),
amount.get(), modeOfPayment.get())

    )

    dbconnector.commit()

    clearFields() # Clear input fields after adding

    listAllExpenses() # Refresh the table

```

```

        mb.showinfo('Expense added', 'The expense has been added to the
database')

# Function to edit an existing expense

def editExpense():

    global data_table

    if not data_table.selection():

        mb.showerror('No expense selected!', 'Please select an expense to
edit')

        return

    viewExpenseInfo() # Show current details

def editExistingExpense():

    global dateField, amount, description, payee, modeOfPayment,
dbconnector, data_table

    content = data_table.item(data_table.focus())['values']

    dbconnector.execute(

        'UPDATE ExpenseTracker SET Date = ?, Payee = ?, Description =
?, Amount = ?, ModeOfPayment = ? WHERE ID = ?',

        (dateField.get_date(), payee.get(), description.get(),
amount.get(), modeOfPayment.get(), content[0])

    )

    dbconnector.commit()

```

```

        clearFields() # Clear fields after editing

        listAllExpenses() # Refresh the table

        mb.showinfo('Data edited', 'The data has been updated in the
database')

        editSelectedButton.destroy() # Remove the edit button ``python

        editSelectedButton.destroy() # Remove the edit button after editing


        editSelectedButton = Button(frameL3, text="Edit Expense",
font=("Bahnschrift Condensed", "13"), width=30,

                                bg="#90EE90", fg="#000000", relief=GROOVE,
activebackground="#008000",

                                activeforeground="#98FB98",
command=editExistingExpense)

        editSelectedButton.grid(row=0, column=0, sticky=W, padx=50, pady=10)

# Function to convert selected expense details to a readable format
def selectedExpenseToWords():

    global data_table

    if not data_table.selection():

        mb.showerror('No expense selected!', 'Please select an expense to
read')

        return

    val = data_table.item(data_table.focus())['values']

```

```

    msg = f'You paid {val[4]} to {val[2]} for {val[3]} on {val[1]} via
{val[5]}'

    mb.showinfo('Expense Details', msg)  # Display the formatted message

# Function to display expense details before adding to the database
def expenseToWordsBeforeAdding():

    global dateField, description, amount, payee, modeOfPayment

    if not dateField.get() or not payee.get() or not description.get() or
not amount.get() or not modeOfPayment.get():

        mb.showerror('Incomplete data', 'The data is incomplete, please
fill all fields first!')

        return

    msg = f'You are about to add: \n"You paid {amount.get()} to
{payee.get()} for {description.get()} on {dateField.get_date()} via
{modeOfPayment.get()}'

    if mb.askyesno('Confirm Addition', f'{msg}\n\nShould I add it to the
database?'):

        addAnotherExpense()  # Call function to add expense

    else:

        mb.showinfo('Cancelled', 'Please take your time to add this
record')

# Main function to run the application

if __name__ == "__main__":

```

```
dbconnector = sqlite3.connect("Expense_Tracker.db") # Connect to the
database

dbcursor = dbconnector.cursor()

dbconnector.execute(

    'CREATE TABLE IF NOT EXISTS ExpenseTracker (ID INTEGER PRIMARY KEY
AUTOINCREMENT NOT NULL, Date DATETIME, Payee TEXT, Description TEXT, Amount
FLOAT, ModeOfPayment TEXT) '

)

dbconnector.commit() # Commit the table creation


main_win = Tk() # Create main window

main_win.title("EXPENSE TRACKER - JAVATPOINT") # Set window title

main_win.geometry("1415x650+400+100") # Set window size and position

main_win.resizable(0, 0) # Disable resizing

main_win.config(bg="#FFFAF0") # Set background color

main_win.iconbitmap("./piggyBank.ico") # Set window icon


# Create frames for layout

frameLeft = Frame(main_win, bg="#FFF8DC")

frameRight = Frame(main_win, bg="#DEB887")

frameL1 = Frame(frameLeft, bg="#FFF8DC")

frameL2 = Frame(frameLeft, bg="#FFF8DC")

frameL3 = Frame(frameLeft, bg="#FFF8DC")
```

```
frameR1 = Frame(frameRight, bg="#DEB887")

frameR2 = Frame(frameRight, bg="#DEB887")


# Pack frames into the main window

frameLeft.pack(side=LEFT, fill="both")

frameRight.pack(side=RIGHT, fill="both", expand=True)

frameL1.pack(fill="both")

frameL2.pack(fill="both")

frameL3.pack(fill="both")

frameR1.pack(fill="both")

frameR2.pack(fill="both", expand=True)


# Adding widgets to the frames

headingLabel = Label(frameL1, text="EXPENSE TRACKER",
font=("Bahnschrift Condensed", "25"), width=20, bg="#8B4513", fg="#FFFAF0")

subheadingLabel = Label(frameL1, text="Data Entry Frame",
font=("Bahnschrift Condensed", "15"), width=20, bg="#F5DEB3", fg="#000000")

headingLabel.pack(fill="both")

subheadingLabel.pack(fill="both")


# Create labels for data entry

dateLabel = Label(frameL2, text="Date:", font=("consolas", "11",
"bold"), bg="#FFF8DC", fg="#000000")
```



```

        descriptionLabel = Label(frameL2, text="Description:",
font=("consolas", "11", " bold"), bg="#FFF8DC", fg="#000000")

        amountLabel = Label(frameL2, text="Amount:", font=("consolas", "11",
"bold"), bg="#FFF8DC", fg="#000000")

        payeeLabel = Label(frameL2, text="Payee:", font=("consolas", "11",
"bold"), bg="#FFF8DC", fg="#000000")

        modeLabel = Label(frameL2, text="Mode of \nPayment:", font=("consolas",
"11", "bold"), bg="#FFF8DC", fg="#000000")

# Position labels in the grid

dateLabel.grid(row=0, column=0, sticky=W, padx=10, pady=10)

descriptionLabel.grid(row=1, column=0, sticky=W, padx=10, pady=10)

amountLabel.grid(row=2, column=0, sticky=W, padx=10, pady=10)

payeeLabel.grid(row=3, column=0, sticky=W, padx=10, pady=10)

modeLabel.grid(row=4, column=0, sticky=W, padx=10, pady=10)

# Create input fields

description = StringVar()

payee = StringVar()

modeOfPayment = StringVar(value="Cash")

amount = DoubleVar()

dateField = DateEntry(frameL2, date=datetime.datetime.now().date(),

```

```

font=("consolas", "11"), relief=GROOVE)

    descriptionField = Entry(frameL2, text=description, width=20,
font=("consolas", "11"), bg="#FFFFFF", fg="#000000", relief=GROOVE)

    amountField = Entry(frameL2, text=amount, width=20, font=("consolas",
"11"), bg="#FFFFFF", fg="#000000", relief=GROOVE)

    payeeField = Entry(frameL2, text=payee, width=20, font=("consolas",
"11"), bg="#FFFFFF", fg="#000000", relief=GROOVE)

    modeField = OptionMenu(frameL2, modeOfPayment, *['Cash', 'Cheque',
'Credit Card', 'Debit Card', 'UPI', 'Paytm', 'Google Pay', 'PhonePe',
'Razorpay'])

    modeField.config(width=15, font=("consolas", "10"), relief=GROOVE,
bg="#FFFFFF")

# Position input fields in the grid

dateField.grid(row=0, column=1, sticky=W, padx=10, pady=10)

descriptionField.grid(row=1, column=1, sticky=W, padx=10, pady=10)

amountField.grid(row=2, column=1, sticky=W, padx=10, pady=10)

payeeField.grid(row=3, column=1, sticky=W, padx=10, pady=10)

modeField.grid(row=4, column=1, sticky=W, padx=10, pady=10)

# Create buttons for actions

insertButton = Button(frameL3, text="Add Expense", font=("Bahnschrift
Condensed", "13"), width=30, bg="#90EE90", fg="#000000", relief=GROOVE,

```

```

activebackground="#008000", activeforeground="#98FB98",
command=addAnotherExpense)

    convertButton = Button(frameL3, text="Convert to Text before Adding",
font=("Bahnschrift Condensed", "13"), width=30, bg="#90EE90", fg="#000000",
relief=GROOVE, activebackground="#008000", activeforeground="#98FB98",
command=expenseToWordsBeforeAdding)

    resetButton = Button(frameL3, text="Reset the fields",
font=("Bahnschrift Condensed", "13"), width=30, bg="#FF0000", fg="#FFFFFF",
relief=GROOVE, activebackground="#8B0000", activeforeground="#FFB4B4",
command=clearFields)


# Position buttons in the grid

insertButton.grid(row=0, column=0, sticky=W, padx=50, pady=10)

convertButton.grid(row=1, column=0, sticky=W, padx=50, pady=10)

resetButton.grid(row=2, column=0, sticky=W, padx=50, pady=10)


# Create buttons for viewing and editing expenses

viewButton = Button(frameR1, text="View Selected Expense's Details",
font=("Bahnschrift Condensed", "13"), width=35, bg="#FFDEAD", fg="#000000",
relief=GROOVE, activebackground="#A0522D", activeforeground="#FFF8DC",
command=viewExpenseInfo)

editButton = Button(frameR1, text="Edit Selected Expense",
font=("Bahnschrift Condensed", "13"), width=35, bg="#FFDEAD", fg="#000000",
relief=GROOVE, activebackground="#A0522D", activeforeground="#FFF8DC",
command=editExpense)

```

```

        convertSelectedButton = Button(frameR1, text="Convert Selected Expense
to a Sentence", font=("Bahnschrift Condensed", "13"), width=35,
bg="#FFDEAD", fg="#000000", relief=GROOVE, activebackground="#A0522D",
activeforeground="#FFF8DC", command=selectedExpenseToWords)

        deleteButton = Button(frameR1, text="Delete Selected Expense",
font=("Bahnschrift Condensed", "13"), width=35, bg="#FFDEAD", fg="#000000",
relief=GROOVE, activebackground="#A0522D", activeforeground="#FFF8DC",
command=removeExpense)

        deleteAllButton = Button(frameR1, text="Delete All Expense",
font=("Bahnschrift Condensed", "13"), width=35, bg="#FFDEAD", fg="#000000",
relief=GROOVE, activebackground="#A0522D", activeforeground="#FFF8DC",
command=removeAllExpenses)

# Position buttons in the grid

viewButton.grid(row=0, column=0, sticky=W, padx=10, pady=10)

editButton.grid(row=0, column=1, sticky=W, padx=10, pady=10)

convertSelectedButton.grid(row=0, column=2, sticky=W, padx=10, pady=10)

deleteButton.grid(row=1, column=0, sticky=W, padx=10, pady=10)

deleteAllButton.grid(row=1, column=1, sticky=W, padx=10, pady=10)

# Create a table to display all expenses

data_table = ttk.Treeview(frameR2, selectmode=BROWSE, columns=('ID',
'Date', 'Payee', 'Description', 'Amount', 'Mode of Payment'))

Xaxis_Scrollbar = Scrollbar(data_table, orient=HORIZONTAL,
command=data_table.xview)

```

```
Yaxis_Scrollbar = Scrollbar(data_table, orient=VERTICAL,
command=data_table.yview)

# Configure scrollbars

data_table.config(yscrollcommand=Yaxis_Scrollbar.set,
xscrollcommand=Xaxis_Scrollbar.set)

Xaxis_Scrollbar.pack(side=BOTTOM, fill=X)

Yaxis_Scrollbar.pack(side=RIGHT, fill=Y)

# Set up table headings

data_table.heading('ID', text='S No.', anchor=CENTER)

data_table.heading('Date', text='Date', anchor=CENTER)

data_table.heading('Payee', text='Payee', anchor=CENTER)

data_table.heading('Description', text='Description', anchor=CENTER)

data_table.heading('Amount', text='Amount', anchor=CENTER)

data_table.heading('Mode of Payment', text='Mode of Payment',
anchor=CENTER)

# Set up table columns

data_table.column('#0', width=0, stretch=NO)

data_table.column('#1', width=50, stretch=NO)

data_table.column('#2', width=95, stretch=NO)

data_table.column('#3', width=150, stretch=NO)
```

```
data_table.column('#4', width=450, stretch=NO)

data_table.column('#5', width=135, stretch=NO)


# Place the table in the frame

data_table.place(relx=0, y=0, relheight=1, relwidth=1)


# Run the application

main_win.mainloop()
```

7. Testing and Validation

To ensure that the chess game functions correctly and provides a smooth user experience, the following testing methods were applied:

Unit Testing:

- Unit testing was performed to ensure that individual components of the expense tracker functioned as expected. Below is the table detailing the test cases:

Test Case ID	Description	Input	Expected Output	Actual Output	Status
TC01	Sign up function	Add new username and password	Sign up success	Sign up success	Passed
TC02	Login function	Username and password	Login success	Login success	Passed
TC03	Adding values	expense	Successfully added	Successfully added	Passed
TC04	Editing values	New value	Successfully added new value	Successfully added new value	Passed
TC05	Deleting value	Select value	Successfully deleted	Successfully deleted	Passed
TC06	Convert Selected Expense to a Sentence	Select the option	Convert successfully	Convert Successfully	Passed

System Testing:

System Testing for your **Expense Tracker** application ensures that the entire system works as expected when all components are integrated. It verifies that the application's features, GUI, and backend logic work together seamlessly under different scenarios.

Here's a structured approach to perform **System Testing** for your Expense Tracker:

Test Case ID	Description	Scenario	Expected Outcome	Actual Outcome	Status
TC001	Add a valid expense	Add a valid expense (e.g., name: "Groceries", amount: 50, category: "Food", date: "2024-11-22").	Expense is successfully added and appears in the expense list.	Expense added successfully and displayed in the list.	Pass
TC002	Add an invalid expense	Add an expense with a negative amount (e.g., name: "Invalid", amount: -20, category: "Food").	Application prevents the addition and displays an error message.	Error message displayed, and expense not added.	Pass
TC003	Edit an existing expense	Modify the amount of an existing expense from 50 to 60.	Expense is updated with the new amount, and changes are reflected in the list.	Expense updated correctly.	Pass
TC004	Delete an expense	Delete an expense with name "Groceries".	Expense is removed from the list and total is recalculated.	Expense removed, and total updated accurately.	Pass
TC005	Input date validation	Add an expense with an invalid date (e.g., "2024-02-30").	Application prevents the addition and displays an error message.	Error message displayed, and invalid date rejected.	Pass
TC006	Calculate total expenses accurately	Add expenses, delete one, and check the updated total.	Total recalculates correctly after addition and deletion.	Total updated accurately.	Pass

8. Conclusion

The Expense Tracker application successfully achieves its objective of providing a user-friendly and efficient platform for managing personal finances. By leveraging Python and a graphical user interface (GUI), the application simplifies the process of recording, categorizing, and analyzing daily expenses, addressing the limitations of traditional manual methods like pen-and-paper or spreadsheets.

The system's features, such as adding, editing, and categorizing expenses, along with generating summaries and visualizations, streamline expense management and foster financial discipline. The application's modular design ensures maintainability and scalability, allowing for future enhancements such as data visualization, detailed spending reports, or integration with external tools.

Through rigorous system testing, the application has demonstrated its reliability and responsiveness across various scenarios, ensuring accuracy in financial calculations and usability in handling large datasets. The project not only provides a practical tool for everyday expense tracking but also serves as an excellent learning experience in programming, GUI design, and system integration.

In conclusion, the Expense Tracker is a robust and adaptable solution for modern financial management needs, offering users a valuable resource to achieve their budgeting and financial goals while laying a strong foundation for further innovation and development.

9. References

1. Online Resources

- Python Programming Documentation: <https://docs.python.org/>
- Pygame Library Documentation: <https://www.pygame.org/docs>
- TutorialsPoint: Python Basics: <https://www.tutorialspoint.com/python/index.htm>

2. Documentation for Tools/Libraries

- Pygame Library Documentation: <https://www.pygame.org/docs>
- Python Standard Library: <https://docs.python.org/3/library/index.html>