

**A PROJECT REPORT
ON
EXPENSE TRACKER**

Submitted in the partial fulfilment

For the award of course in

CORE PYTHON

TO

<CODE INFINITE>

IT Training Service

Submitted by

K. KALAITHARUN (CIITS24066)

Under the guidance of

DR. R. MANSOOR AHAMAD

<CODE INFINITE> IT Training Service

#6c- 1st Floor, Chandrasekar Tower, Opp KGiSL,

Thudiyalur Rd, Saravanampatti, Coimbatore - 35

Ph : 0422-2666232 | Mobile : +91 99764 99765

info@codeinfinite.in | www.codeinfinite.in

DECEMBER 2024

Declaration

I hereby declare that the project titled "**Expense Tracker Application using Python, SQLite, and Tkinter**" is my original work, completed as part of my academic requirements. This project was undertaken to deepen my understanding of software development, particularly in Python programming, database management, and GUI design, under the guidance of my course instructor and mentor.

I affirm that the work presented in this project is entirely genuine and has not been copied or reproduced from any other source. All resources, references, and external contributions used during the development of this application have been appropriately acknowledged and cited.

I take full responsibility for the authenticity and accuracy of the information, design, and implementation details contained in this project report. This work has not been submitted for any other course or purpose elsewhere.

Name: K.KALAITHARUN

Course: CORE PYTHON

Institution: CODE INFINITE

Date: 9 - 12 - 24

Acknowledgement

I would like to extend my heartfelt gratitude to my course trainer, **Mr. Prakash V**, for his invaluable guidance and unwavering support throughout the development of this **Expense Tracker Application**. His expertise and advice were instrumental in deepening my understanding of Python, SQLite, and Tkinter. This foundation allowed me to effectively integrate object-oriented programming principles and database management concepts into the project.

I am also deeply grateful to **Dr. Mansoor Ahamad R**, the institution management head, for fostering a supportive educational environment and providing the resources that facilitated my learning journey. His dedication to creating an atmosphere where students can explore and excel is truly inspiring.

Furthermore, I wish to thank my peers and mentors for their thoughtful feedback and collaborative input during this project. Their innovative ideas and constructive suggestions helped refine the functionality and user interface of the application, ensuring a more polished and user-friendly product.

I also acknowledge the authors, developers, and contributors of various technical resources that provided valuable insights and guidance. Online communities, such as Stack Overflow and GitHub, along with official documentation for Python, SQLite, and Tkinter, were indispensable in addressing challenges and expanding my understanding of advanced programming concepts.

Finally, I would like to express my gratitude to my family and friends for their moral support and encouragement. Their confidence in my abilities motivated me to persevere through challenges and strive for excellence.

To everyone who contributed directly or indirectly to the completion of this project, I am deeply thankful. Your support and inspiration have made this experience both enriching and rewarding.

Abstract

The **Expense Tracker Application** is a comprehensive software tool designed to assist users in effectively managing their personal finances by enabling them to record, monitor, and analyze their daily, weekly, and monthly expenses. The motivation for this project stems from the increasing complexity of modern financial management and the lack of simple yet effective tools to help users maintain financial discipline.

This application combines an intuitive user interface, developed using Python's Tkinter library, with robust back-end data management powered by SQLite. Core features include user authentication, expense categorization, customizable monthly budget allocation, and detailed visual reports such as pie charts and bar graphs for financial insights. Users can input and organize expenses across categories like food, travel, utilities, and entertainment, and filter expense data by specific time ranges to understand spending trends.

The project emphasizes security and usability, ensuring that users of all technical backgrounds can easily navigate the interface while keeping their financial data secure. Testing and user feedback played a significant role in refining the system, ensuring it meets real-world needs. The result is a practical and accessible tool that promotes better financial planning, awareness, and informed decision-making.

Table of Content

S.no	TOPIC	Page. No
01	Introduction <ul style="list-style-type: none"> ○ Financial Management Challenges ○ Motivation for the Project ○ Key Features and Objectives 	07
02	Implementation <ul style="list-style-type: none"> ○ Coding Approach <ul style="list-style-type: none"> ▪ Key Functions and Classes ▪ Challenges and Solutions ○ Database Design <ul style="list-style-type: none"> ▪ Schema, Tables, and Relationships ○ User Interface Design <ul style="list-style-type: none"> ▪ Layout, Colors, and Usability ○ Testing and Debugging <ul style="list-style-type: none"> ▪ Test Cases and Bug Fixes 	08
03	Hardware Requirements <ul style="list-style-type: none"> ○ Minimum, Recommended, and High-End Specifications ○ Peripheral and Network Requirements 	09
04	Technologies Used <ul style="list-style-type: none"> ○ Programming Language ○ Database Management ○ GUI Framework ○ Supporting Libraries and Tools 	12
05	Special Features	15

	<ul style="list-style-type: none"> ○ User-Friendly Interface ○ Expense and Income Management ○ Summary and Analytics ○ Cross-Platform Compatibility 	
06	Project File Structure	17
07	Sample Code	20
08	Sample Screen	43
09	Results and Analysis <ul style="list-style-type: none"> ○ Performance Metrics ○ Feedback and Refinements 	51
10	Conclusion <ul style="list-style-type: none"> ○ Summary of Application Benefits ○ Personal Learning and Growth ○ Future Prospects and Impact 	52
11	References <ul style="list-style-type: none"> ○ Documentation and Tutorials ○ Online Communities and Libraries ○ Development Tools and APIs 	53
12	Further Enhancements <ul style="list-style-type: none"> ○ Cloud Integration ○ Mobile App Development ○ AI Insights and API Integration ○ Advanced Reporting and Multi-Language Support 	54

Introduction

Financial management is an integral part of daily life, and the ability to track and control expenses is critical for achieving long-term financial stability. In today's fast-paced world, where individuals are juggling multiple sources of income and expenditure, the challenge of maintaining a clear and organized overview of one's financial status is more pressing than ever. Many individuals struggle with overspending due to a lack of awareness and effective tools to track expenses.

The **Expense Tracker Application** was developed to address these challenges by providing a user-friendly and efficient solution for personal financial management. This project aims to bridge the gap between complex, enterprise-level financial tools and the practical needs of everyday users.

The application offers several key functionalities:

- **User Authentication:** Secure login to ensure data privacy.
- **Expense Categorization:** Assign expenses to customizable categories such as food, transportation, and utilities.
- **Budget Management:** Allow users to set monthly budgets and monitor expenses against them.
- **Data Analysis and Reporting:** Generate detailed summaries and visualizations (e.g., bar graphs and pie charts) for better financial insights.
- **Expense Filtering:** Enable filtering by date range, category, or custom criteria for granular analysis.

The project employs Python's Tkinter library to create a clean and responsive graphical user interface (GUI), making it accessible to users with minimal technical expertise. SQLite serves as the back-end database, ensuring efficient and secure data storage. The lightweight design of the application makes it suitable for personal use on various devices without requiring extensive system resources.

The primary objective of this project is to promote financial literacy and discipline by empowering users to make informed decisions about their spending habits. By integrating intuitive design, robust functionality, and practical utility, the **Expense Tracker Application** aspires to become a reliable companion for personal finance management.

Implementation

1. Coding Approach:

- Provide an overview of how the application was coded. Include examples of key functions and classes.
- Highlight challenges faced during coding, such as debugging or integrating different modules, and how they were resolved.

1. Database Design:

- Present the database schema in detail, including tables, relationships, and attributes.
- Explain how the database ensures efficient storage and retrieval of data.

1. User Interface Design:

- Include screenshots or mockups of the application.
- Discuss design choices like layout, color scheme, and navigation flow.

1. Testing and Debugging:

- Describe the testing process, including test cases for features like login, expense addition, and report generation.
- Highlight any bugs encountered and how they were fixed.

Results and Analysis

1. Key Findings:

- Discuss how the application helps users manage their finances. Provide hypothetical or real examples of usage scenarios.

1. Performance Analysis:

- Include metrics such as response time for adding or retrieving data.
- Mention user feedback received during testing and how it influenced refinements.

1. Visual Examples:

- Use charts or screenshots generated by the application to demonstrate its reporting capabilities.

Hardware Requirements

The **Expense Tracker Application** is a lightweight software that primarily runs on personal computers. Below is a detailed list of the hardware requirements based on different system configurations to ensure smooth development, testing, and execution of the application.

1. Minimum Hardware Requirements

- **Processor:**
 - Intel Core i3 (3rd generation or later) or AMD equivalent with at least 2 cores.
 - Rationale: Handles basic Python operations and GUI rendering efficiently.
- **RAM:**
 - 4 GB
 - Rationale: Provides sufficient memory for running lightweight applications and basic development tools like Python IDEs.
- **Storage:**
 - 256 GB HDD or SSD (with at least 5 GB free for software and project files).
 - Rationale: Stores project files, SQLite database, and Python installation.
- **Display:**
 - 1366x768 resolution monitor.
 - Rationale: Suitable for viewing the Tkinter-based GUI and running development tools.
- **Graphics:**
 - Integrated GPU (e.g., Intel HD Graphics).
 - Rationale: Tkinter applications don't require advanced graphics rendering.
- **Input Devices:**
 - Standard keyboard and mouse.

2. Recommended Hardware Requirements (For Optimal Performance)

- **Processor:**
 - Intel Core i5 (6th generation or later) or AMD Ryzen 3 (latest generation).
 - Rationale: Enhances performance when running multiple programs like Python IDEs, database tools, and testing environments simultaneously.
- **RAM:**
 - 8 GB
 - Rationale: Enables smooth multitasking, especially if you are using modern IDEs like PyCharm or Visual Studio Code alongside debugging tools.
- **Storage:**
 - 512 GB SSD
 - Rationale: Faster read/write speeds for project files and application execution, along with sufficient storage for additional resources.

- **Display:**
 - Full HD (1920x1080) monitor.
 - Rationale: Provides a more comfortable workspace for coding, debugging, and designing GUIs.
- **Graphics:**
 - Integrated or entry-level discrete GPU (e.g., NVIDIA GeForce GTX 1050).
 - Rationale: Ensures fluid GUI rendering and allows for smoother testing of other potential Python libraries with graphical components.

3. High-End Hardware Requirements (For Advanced Development and Future Scalability)

- **Processor:**
 - Intel Core i7 (10th generation or later) or AMD Ryzen 5/7 (latest generation).
 - Rationale: Ideal for running advanced simulations, performance testing, or integrating AI/ML components in future application updates.
- **RAM:**
 - 16 GB or higher
 - Rationale: Supports resource-intensive tasks like data analysis, running virtual environments, or multitasking across multiple tools.
- **Storage:**
 - 1 TB SSD (NVMe preferred).
 - Rationale: Provides ample storage for large datasets, libraries, and future extensions of the application.
- **Display:**
 - 4K monitor.
 - Rationale: Offers a premium coding and GUI design experience, particularly helpful for detailed visual debugging.
- **Graphics:**
 - NVIDIA RTX series (e.g., RTX 3060) or AMD Radeon RX series.
 - Rationale: Allows for integration of advanced GUI libraries or 3D graphical interfaces if planned in future updates.
- **Peripheral Devices:**
 - High-quality mechanical keyboard and ergonomic mouse for long development sessions.

Additional Hardware Considerations

1. **External Backup Storage:**
 - A 1 TB external HDD or SSD to back up project files and database snapshots.
1. **Networking Requirements:**

- Reliable internet connection (broadband or fiber) for accessing resources, documentation, and cloud services.

1. **Power Backup:**

- UPS (Uninterruptible Power Supply) with a minimum of 600 VA for uninterrupted development sessions during power outages.

1. **Printer (Optional):**

- Useful for printing documentation, project reports, or user manuals if required.

1. **Testing Device (Optional):**

- A basic laptop or PC with lower specifications to test application performance on less powerful machines.

By adhering to these hardware specifications, you can ensure a seamless development and user experience for the **Expense Tracker Application**.

Technologies Used in the Expense Tracker Application

The Expense Tracker Application leverages a combination of programming languages, frameworks, libraries, and tools to provide an efficient, user-friendly, and maintainable solution. Below is a detailed breakdown of the technologies used in the project:

1. Programming Language

- **Python**
 - **Purpose:**
 - Primary language for developing the application.
 - Used for back-end logic, database interaction, and GUI integration.
 - **Features:**
 - Readable and easy-to-learn syntax.
 - Extensive standard library and support for third-party modules.

2. Database

- **SQLite**
 - **Purpose:**
 - Acts as the back-end database for storing and managing expense-related data (e.g., income, expenses, categories, and dates).
 - **Features:**
 - Lightweight and serverless database.
 - Ideal for applications with moderate data requirements.
 - Simple integration with Python using the sqlite3 module.

3. GUI Framework

- **Tkinter**
 - **Purpose:**
 - Provides the graphical user interface (GUI) for the application.
 - Allows users to interact with the application through windows, buttons, labels, entry fields, and other GUI components.
 - **Features:**
 - Built into Python's standard library, eliminating the need for external installation.
 - Simple and customizable.
 - Supports event-driven programming.

4. Libraries and Modules

- **sqlite3**
 - **Purpose:**

- Facilitates database operations such as CRUD (Create, Read, Update, Delete) with SQLite.
- **tkinter.ttk**
 - **Purpose:**
 - Provides enhanced and modern-looking widgets, such as Treeview for displaying tabular data (e.g., expense history).
- **datetime**
 - **Purpose:**
 - Handles date and time for recording transactions and filtering expenses by date.
- **os**
 - **Purpose:**
 - Manages file operations, such as creating or accessing SQLite database files.

5. Development Tools

- **IDE/Text Editors:**
 - **PyCharm, Visual Studio Code, or IDLE**
 - **Purpose:**
 - Used for writing, debugging, and testing Python code.
 - **Features:**
 - Code completion, debugging tools, and version control integration.
- **Version Control:**
 - **Git and GitHub**
 - **Purpose:**
 - Tracks changes in the project files and collaborates with other developers.

6. Operating System Compatibility

- **Windows, macOS, and Linux**
 - **Purpose:**
 - The application is cross-platform and runs on all major operating systems due to Python's portability.

7. Optional Technologies (for Advanced Features)

- **Matplotlib or Seaborn (Optional):**
 - **Purpose:**
 - Provides data visualization, such as graphs and charts for expense trends.
- **Pandas (Optional):**
 - **Purpose:**

- Allows for advanced data manipulation and analysis, making it easier to handle complex queries or export data to files like CSV.
- **Excel/CSV Integration** (Optional):
 - **Purpose:**
 - Exports expense data to external files for sharing or backup.

By utilizing this robust technology stack, the **Expense Tracker Application** offers an intuitive interface, reliable performance, and scalability for future enhancements, such as data visualization and cloud integration.

Special Features of the Expense Tracker Application

The **Expense Tracker Application** is designed to offer a user-friendly and efficient way to manage personal finances. Below is a detailed list of its special features:

1. User-Friendly Interface

- Built using **Tkinter**, the application features an intuitive and clean graphical user interface (GUI) with well-organized windows, buttons, and input fields.
- The design prioritizes simplicity, making it accessible to users with minimal technical knowledge.

2. Expense and Income Management

- **Add Expenses and Income:**
 - Easily log expenses and income by specifying details such as category, amount, and date.
- **Categorization:**
 - Classify transactions into predefined categories like *Food*, *Transport*, *Entertainment*, or custom categories.

3. Expense Summary and Analytics

- View summarized data such as total income, total expenses, and remaining balance.
- Optional integration with **matplotlib** to generate bar charts or pie charts showing expense distribution across categories.

4. Transaction History

- **Tabular View:**
 - Display all past transactions in a **Treeview widget** (from `tkinter.ttk`) for a structured overview.
- **Search and Filter:**
 - Search transactions by keywords (e.g., category, date, amount).
 - Filter transactions by specific time periods (e.g., today, this week, or custom date range).

5. Data Security and Persistence

- All data is stored securely in a local **SQLite database**, ensuring data persistence across application sessions.
- The application uses database validation to prevent duplicate entries or invalid data.

6. Cross-Platform Compatibility

- The application is platform-independent and can run on **Windows, macOS, and Linux** without modification.

7. Customization Options

- **Custom Categories:**
 - Add, edit, or delete custom categories to tailor the application to specific needs.
- **Currency Selection:**
 - Option to set and display amounts in the user's preferred currency.

8. Expense Tracking by Date

- Supports date-specific tracking to monitor daily, weekly, or monthly spending habits.
- Automatic categorization of transactions by date for easy analysis.

9. Data Export (Optional)

- Export transaction data to **CSV** or **Excel** files for external use, such as sharing or creating backups.

10. Notifications and Reminders (Advanced Feature)

- Set budget limits for specific categories and receive alerts when nearing or exceeding the limit.
- Notifications for overdue payments or recurring expenses (optional).

11. Lightweight and Offline Operation

- The application is designed to work offline, ensuring functionality without requiring an internet connection.
- Minimal system requirements make it ideal for low-resource devices.

12. Easy Installation and Portability

- No complex installation process.
- The SQLite database and the Python application are packaged together, ensuring easy portability.

13. Future Scalability

- Designed with modular architecture to accommodate future enhancements such as:
 - Cloud synchronization for remote access to data.
 - Integration with bank APIs for automatic expense logging.
 - Advanced AI-based spending recommendations.

Project File Structure

Below is the suggested file structure for the Expense Tracker Application using Python, SQLite, and Tkinter. This structure organizes the code, resources, and database for better readability and maintainability.

ExpenseTracker/

```
|
|
|—— main.py          # Entry point of the application
|—— database/
|   |—— expense_tracker.db # SQLite database file
|   |—— db_setup.py       # Script for initializing the database schema
|   |—— db_operations.py  # Module for database interactions (CRUD operations)
|
|—— gui/
|   |—— __init__.py      # Initializes the GUI package
|   |—— main_window.py   # Main GUI window (dashboard)
|   |—— add_expense.py    # GUI for adding expenses
|   |—— view_expenses.py  # GUI for viewing expense history
|   |—— filter_expenses.py # GUI for filtering expenses
|   |—— settings.py      # GUI for application settings and customization
|   |—— themes/          # Theme files for the application
|       |—— light_theme.css
|       |—— dark_theme.css
|
|—— utils/
|   |—— __init__.py      # Initializes the utilities package
|   |—— helpers.py       # Helper functions (e.g., date formatting, validations)
```

```

|   |—— data_export.py    # Module for exporting data to CSV/Excel
|   |—— notifications.py  # Module for reminders and notifications
|
|—— assets/
|   |—— icons/            # Icons for buttons and UI elements
|   |   |—— add.png
|   |   |—— report.png
|   |—— charts/          # Temporary folder for chart image generation
|   |—— fonts/           # Custom fonts for the UI
|
|—— tests/
|   |—— test_database.py  # Unit tests for database operations
|   |—— test_gui.py       # Unit tests for GUI interactions
|   |—— test_helpers.py   # Unit tests for utility functions
|   |—— test_end_to_end.py # End-to-end tests for the application
|
|—— requirements.txt      # Python libraries required for the project
|—— README.md            # Project overview and setup instructions
|—— LICENSE              # License for the application
|—— .gitignore           # Git ignore file to exclude unnecessary files
|—— docs/
|   |—— user_manual.pdf   # User guide for the application
|   |—— project_report.pdf # Detailed project report
|   |—— changelog.txt     # Record of updates and changes to the application

```

Explanation of Key Components

1. **main.py**

The starting point of the application that initializes the GUI and connects the components.

2. **database/**

- Contains the SQLite database file (expense_tracker.db) and scripts for database schema setup (db_setup.py).
- db_operations.py abstracts database queries to ensure modularity.

1. **gui/**

- Contains all GUI components split by functionality (e.g., main window, add expense, view expense).
- themes/ stores theme-related CSS files for customizing the application's look.

1. **utils/**

- Contains utility functions for tasks like data validation, date manipulation, and notifications.
- data_export.py handles exporting expenses to external formats like CSV or Excel.

1. **assets/**

- Stores static resources like icons, fonts, and chart images used by the application.

1. **tests/**

- Contains test scripts for unit testing, integration testing, and end-to-end testing.

1. **requirements.txt**

- Lists all third-party libraries used in the project (e.g., tkinter, sqlite3, matplotlib).

1. **README.md**

- Provides an overview of the project, instructions for installation, and usage details.

1. **docs/**

- Stores documentation, including a user manual and the project report.

This structured approach makes the application modular, scalable, and easier to maintain or enhance in the future.

Sample Code

import tkinter as tk

```
from tkinter import messagebox, ttk
```

```
from tkinter import filedialog
```

```
import sqlite3
```

```
class ExpenseTracker:
```

```
    def __init__(self, root):
```

```
        self.root = root
```

```
        self.root.title("Expense Tracker")
```

```
        self.root.geometry("400x450")
```

```
        self.root.resizable(False, False)
```

```
        self.root.config(bg="#D88DE7")
```

```
        # SQLite connection
```

```
        self.conn = sqlite3.connect("expenses.db")
```

```
        self.create_table()
```

```
        self.create_table_budget()
```

```
        ""
```

```
        # UI Components
```

```
        self.budget_label = tk.Label(root, text="Enter Monthly Budget (₹):", bg="#D88DE7", fg="#E0FFFF",  
font=("Helvetica", 12, "bold", "italic"))
```

```
        self.budget_label.grid(column=0, row=6, pady=(20, 0), padx=(20, 0))
```

```
        self.budget_entry = tk.Entry(root, bg="#E0FFFF")
```

```
        self.budget_entry.grid(column=1, row=6, pady=(20, 0), padx=(10, 20))
```

```
        ""
```

```
self.add_expense_btn = tk.Button(root, text=" Add Expense ", command=self.add_expense_window,
bg="#D88DE7", fg="#E0FFFF", font=("Helvetica", 10, "bold", "italic"))
```

```
self.add_expense_btn.place(relx=0.5, rely=0.4, anchor="center") # Centered
```

```
self.add_expense_btn = tk.Button(root, text=" Add Budget ", command=self.add_budget_window,
bg="#D88DE7", fg="#E0FFFF", font=("Helvetica", 10, "bold", "italic"))
```

```
self.add_expense_btn.place(relx=0.5, rely=0.5, anchor="center") # Centered
```

```
self.view_summary_btn = tk.Button(root, text=" View Summary ", command=self.view_summary,
bg="#D88DE7", fg="#E0FFFF", font=("Helvetica", 10, "bold", "italic"))
```

```
self.view_summary_btn.place(relx=0.5, rely=0.6, anchor="center") # Centered
```

```
def create_table(self):
```

```
    """Create SQLite table for storing expenses."""
```

```
    try:
```

```
        query = """
```

```
        CREATE TABLE IF NOT EXISTS expenses (
```

```
            id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
            date TEXT NOT NULL,
```

```
            amount REAL NOT NULL,
```

```
            category TEXT NOT NULL,
```

```
            description TEXT
```

```
        )
```

```
    """
```

```
    self.conn.execute(query)
```

```
    self.conn.commit()
```

```
except sqlite3.Error as e:
```

```
    messagebox.showerror("Database Error", f"Error creating table: {e}")
```

```
def create_table_budget(self):
```

```
    """Create SQLite table for storing expenses."""
```

```
    try:
```

```

query = """
CREATE TABLE IF NOT EXISTS budget (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    date TEXT NOT NULL,
    amount REAL NOT NULL,
    category TEXT NOT NULL,
    description TEXT
)
"""

self.conn.execute(query)

self.conn.commit()

except sqlite3.Error as e:

    messagebox.showerror("Database Error", f"Error creating table: {e}")

def add_expense_window(self):

    """Open a new window to add an expense."""

    add_window = tk.Toplevel(self.root)

    add_window.title("Add Expense")

    add_window.geometry("400x400")

    add_window.resizable(False, False)

    add_window.config(bg="#D88DE7")

    # Date Label and Entry

    tk.Label(add_window, text="Date (DD-MM-YYYY):", bg="#D88DE7", font=("Helvetica", 10,
"bold")).grid(row=0, column=0, padx=10, pady=10, sticky="w")

    date_entry = tk.Entry(add_window)

    date_entry.grid(row=0, column=1, padx=10, pady=10, sticky="w")

    # Amount Label and Entry

    tk.Label(add_window, text="Amount Spent (₹):", bg="#D88DE7", font=("Helvetica", 10, "bold")).grid(row=1,
column=0, padx=10, pady=10, sticky="w")

    amount_entry = tk.Entry(add_window)

    amount_entry.grid(row=1, column=1, padx=10, pady=10, sticky="w")

```

```

# Category Label and Combobox

tk.Label(add_window, text="Category:", bg="#D88DE7", font=("Helvetica", 10, "bold")).grid(row=2,
column=0, padx=10, pady=10, sticky="w")

categories = ["Grocery Items", "Food", "Essentials", "Fun", "Others"]

category_combobox = ttk.Combobox(add_window, values=categories )

category_combobox.grid(row=2, column=1, padx=10, pady=10, sticky="w")


# Description Label and Entry

tk.Label(add_window, text="Description:", bg="#D88DE7", font=("Helvetica", 10, "bold")).grid(row=3,
column=0, padx=10, pady=10, sticky="w")

description_entry = tk.Entry(add_window )

description_entry.grid(row=3, column=1, padx=10, pady=10, sticky="w")


# Save Expense Button

def save_expense():

    """Save the expense to the database."""

    date = date_entry.get()

    try:

        # Validate date format

        day, month, year = map(int, date.split("-"))

        if len(date) != 10 or date[2] != "-" or date[5] != "-":

            raise ValueError

    except (ValueError, IndexError):

        messagebox.showerror("Invalid Date", "Please enter the date in DD-MM-YYYY format (e.g., 15-11-
2024).")

        return

    try:

        amount = float(amount_entry.get())

    except ValueError:

        messagebox.showerror("Invalid Input", "Amount must be numeric!")

        return

```

```

category = category_combobox.get()

description = description_entry.get()

if not date or not category:

    messagebox.showerror("Invalid Input", "Date and Category are required!")

    return

try:

    query = "INSERT INTO expenses (date, amount, category, description) VALUES (?, ?, ?, ?)"

    self.conn.execute(query, (date, amount, category, description))

    self.conn.commit()

    messagebox.showinfo("Success", "Expense added successfully!")

    add_window.destroy()

except sqlite3.Error as e:

    messagebox.showerror("Database Error", f"Error saving expense: {e}")


save_button = tk.Button(add_window, text="Save Expense", command=save_expense, bg="#D88DE7",
fg="white", font=("Helvetica", 10, "bold"))

save_button.grid(row=4, column=0, columnspan=2, pady=20)


def add_budget_window(self):

    """Open a new window to add an budget."""

    add_window = tk.Toplevel(self.root)

    add_window.title("Add Budget")

    add_window.geometry("400x400")

    add_window.resizable(False, False)

    add_window.config(bg="#D88DE7")

    # Date Label and Entry

    tk.Label(add_window, text="Date (DD-MM-YYYY):", bg="#D88DE7", font=("Helvetica", 10,
"bold")).grid(row=0, column=0, padx=10, pady=10, sticky="w")

    date_entry = tk.Entry(add_window)

    date_entry.grid(row=0, column=1, padx=10, pady=10, sticky="w")

```



```

# Amount Label and Entry

tk.Label(add_window, text="Amount Spent (₹):", bg="#D88DE7", font=("Helvetica", 10, "bold")).grid(row=1,
column=0, padx=10, pady=10, sticky="w")

amount_entry = tk.Entry(add_window)

amount_entry.grid(row=1, column=1, padx=10, pady=10, sticky="w")

# Category Label and Combobox

tk.Label(add_window, text="Category:", bg="#D88DE7", font=("Helvetica", 10, "bold")).grid(row=2,
column=0, padx=10, pady=10, sticky="w")

categories = ["Grocery Items", "Food", "Essentials", "Fun", "Others"]

category_combobox = ttk.Combobox(add_window, values=categories )

category_combobox.grid(row=2, column=1, padx=10, pady=10, sticky="w")

# Description Label and Entry

tk.Label(add_window, text="Description:", bg="#D88DE7", font=("Helvetica", 10, "bold")).grid(row=3,
column=0, padx=10, pady=10, sticky="w")

description_entry = tk.Entry(add_window )

description_entry.grid(row=3, column=1, padx=10, pady=10, sticky="w" , )

# Save Expense Button

def save_budget():

    """Save the budget to the database."""

    date = date_entry.get()

    try:

        # Validate date format

        day, month, year = map(int, date.split("-"))

        if len(date) != 10 or date[2] != "-" or date[5] != "-":

            raise ValueError

    except (ValueError, IndexError):

        messagebox.showerror("Invalid Date", "Please enter the date in DD-MM-YYYY format (e.g., 15-11-2024).")

    return

```

```

try:
    amount = float(amount_entry.get())
except ValueError:
    messagebox.showerror("Invalid Input", "Amount must be numeric!")
    return

category = category_combobox.get()
description = description_entry.get()

if not date or not category:
    messagebox.showerror("Invalid Input", "Date and Category are required!")
    return

try:
    query = "INSERT INTO budget (date, amount, category, description) VALUES (?, ?, ?, ?)"
    self.conn.execute(query, (date, amount, category, description))
    self.conn.commit()
    messagebox.showinfo("Success", "Expense added successfully!")
    add_window.destroy()
except sqlite3.Error as e:
    messagebox.showerror("Database Error", f"Error saving expense: {e}")

save_button = tk.Button(add_window, text="Save Budget", command=save_budget, bg="#D88DE7",
fg="white", font=("Helvetica", 10, "bold"))

save_button.grid(row=4, column=0, columnspan=2, pady=20)

def view_summary(self):
    """Display expense summary with additional options."""
    summary_window = tk.Toplevel(self.root)
    summary_window.title("Summary")
    summary_window.geometry("600x500")
    summary_window.resizable(False, False)

```

```

summary_window.config(bg="#D88DE7")

# Date Range Filter
tk.Label(summary_window, text="Filter by Date Range:", bg="#D88DE7", fg="white", font=("Helvetica", 12,
"bold")).pack(pady=10)

date_frame = tk.Frame(summary_window, bg="#D88DE7")
date_frame.pack(pady=5)

tk.Label(date_frame, text="From (DD-MM-YYYY):", bg="#D88DE7", fg="white").grid(row=0, column=0,
padx=5, pady=5)

from_date_entry = tk.Entry(date_frame)
from_date_entry.grid(row=0, column=1, padx=5, pady=5, fg = "##908C91")

tk.Label(date_frame, text="To (DD-MM-YYYY):", bg="#D88DE7", fg="white").grid(row=0, column=2,
padx=5, pady=5)

to_date_entry = tk.Entry(date_frame)
to_date_entry.grid(row=0, column=3, padx=5, pady=5, fg = "##908C91")

# Placeholder for displaying summary
summary_frame = tk.Frame(summary_window, bg="#D88DE7")
summary_frame.pack(pady=20)

# Function to fetch and display the summary
def fetch_summary():

    """Fetch and display summary for the specified date range."""

    for widget in summary_frame.winfo_children():
        widget.destroy()

    from_date = from_date_entry.get()
    to_date = to_date_entry.get()

    # Validate date inputs
    try:

```

```

        if from_date:
            day, month, year = map(int, from_date.split("-"))

        if to_date:
            day, month, year = map(int, to_date.split("-"))

    except (ValueError, IndexError):
        messagebox.showerror("Invalid Date", "Please enter dates in DD-MM-YYYY format.")

    return

try:
    # Total expenses

    query = "SELECT SUM(amount) FROM expenses WHERE date BETWEEN ? AND ?" if from_date and
to_date else "SELECT SUM(amount) FROM expenses"

    params = (from_date, to_date) if from_date and to_date else ()

    cursor = self.conn.cursor()

    cursor.execute(query, params)

    total_expenses = cursor.fetchone()[0] or 0

    # Total budget

    query = "SELECT SUM(amount) FROM budget WHERE date BETWEEN ? AND ?" if from_date and
to_date else "SELECT SUM(amount) FROM budget"

    cursor.execute(query, params)

    total_budget = cursor.fetchone()[0] or 0

    # Category-wise expenses

    query = "SELECT category, SUM(amount) FROM expenses WHERE date BETWEEN ? AND ? GROUP
BY category" if from_date and to_date else "SELECT category, SUM(amount) FROM expenses GROUP BY
category"

    cursor.execute(query, params)

    category_wise = cursor.fetchall()

    # Display totals

    tk.Label(summary_frame, text=f"Total Expenses: ₹{total_expenses:.2f}", bg="#D88DE7", fg="white",
font=("Helvetica", 12)).pack(pady=5)

```

```
tk.Label(summary_frame, text=f"Total Budget: ₹{total_budget:.2f}", bg="#D88DE7", fg="white",
font=("Helvetica", 12)).pack(pady=5)
```

```
tk.Label(summary_frame, text=f"Remaining Budget: ₹{total_budget - total_expenses:.2f}",
bg="#D88DE7", fg="white", font=("Helvetica", 12)).pack(pady=5)
```

```
# Display category-wise summary
```

```
tk.Label(summary_frame, text="Expenses by Category:", bg="#D88DE7", fg="white",
font=("Helvetica", 14, "bold")).pack(pady=10)
```

```
for category, amount in category_wise:
```

```
tk.Label(summary_frame, text=f"{category}: ₹{amount:.2f}", bg="#D88DE7", fg="white",
font=("Helvetica", 12)).pack(pady=2)
```

```
except sqlite3.Error as e:
```

```
messagebox.showerror("Database Error", f"Error fetching summary: {e}")
```

```
# Fetch Summary Button
```

```
fetch_button = tk.Button(summary_window, text="Show Summary", command=fetch_summary,
bg="#D88DE7", fg="white", font=("Helvetica", 10, "bold"))
```

```
fetch_button.pack(pady=10)
```

```
# Export as CSV
```

```
def export_summary():
```

```
    """Export the summary as a CSV file."""
```

```
    import csv
```

```
    file_name = tk.filedialog.asksaveasfilename(defaultextension=".csv", filetypes=[("CSV files", "*.csv")])
```

```
    if not file_name:
```

```
        return
```

```
    try:
```

```
        query = "SELECT * FROM expenses"
```

```
        cursor = self.conn.cursor()
```

```
        cursor.execute(query)
```

```
        rows = cursor.fetchall()
```

```

with open(file_name, mode="w", newline="", encoding="utf-8") as file:

    writer = csv.writer(file)

    writer.writerow(["ID", "Date", "Amount", "Category", "Description"])

    writer.writerows(rows)


messagebox.showinfo("Export Successful", f"Summary exported successfully to {file_name}")


except Exception as e:

    messagebox.showerror("Export Error", f"Error exporting data: {e}")


export_button = tk.Button(summary_window, text="Export as CSV", command=export_summary,
bg="#D88DE7", fg="white", font=("Helvetica", 10, "bold"))

export_button.pack(pady=10)


def __del__(self):

    """Ensure SQLite connection is closed."""

    self.conn.close()


# Run the application

if __name__ == "__main__":

    root = tk.Tk()

    app = ExpenseTracker(root)

    root.mainloop()

```

#Expense_tracker_2

```

from tkinter import *

from PIL import Image, ImageTk

import sqlite3

```

```

from expense_tracker import ExpenseTracker

# Database setup
conn = sqlite3.connect('users.db')
cursor = conn.cursor()
cursor.execute("""
    CREATE TABLE IF NOT EXISTS users (
        username TEXT PRIMARY KEY,
        password TEXT NOT NULL
    )
""")
conn.commit()

# Functions
def register_user():
    username = reg_username.get()
    password = reg_password.get()
    confirm_password = reg_confirm_password.get()

    if not username or not password:
        reg_message_label.config(text="All fields are required!", fg="#E0FFFF")
    elif password != confirm_password:
        reg_message_label.config(text="Passwords do not match!", fg="#E0FFFF")
    else:
        cursor.execute('SELECT * FROM users WHERE username = ?', (username,))
        if cursor.fetchone():
            reg_message_label.config(text="Username already exists!", fg="#E0FFFF")
        else:
            cursor.execute('INSERT INTO users (username, password) VALUES (?, ?)', (username, password))
            conn.commit()
            reg_message_label.config(text="Registration successful!", fg="#E0FFFF")
            reg_username.delete(0, END)

```

```

        reg_password.delete(0, END)

        reg_confirm_password.delete(0, END)

def login_user():

    username = login_username.get()
    password = login_password.get()

    cursor.execute('SELECT * FROM users WHERE username = ? AND password = ?', (username, password))
    user = cursor.fetchone()

    if user:

        window.destroy()

        root = Tk()

        app = ExpenseTracker(root)

        root.mainloop()

        login_message_label.config(text="Login successful!", fg="#E0FFFF")

        show_frame(page1) # Navigate to the first image page on successful login
    else:

        login_message_label.config(text="Invalid credentials!", fg="#E0FFFF")

def show_frame(frame):

    frame.tkraise()

# Main Application Window
window = Tk()

window.title("EXPENSE TRACKER")

window.geometry("450x600")

window.resizable(False, False)

window.config(bg="#D88DE7")

# Container Frame

```



```

container = Frame(window)

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

# Configure container for stacking pages
container.grid_rowconfigure(0, weight=1)
container.grid_columnconfigure(0, weight=1)

# Frames

login_frame = Frame(container, bg="#D88DE7")
login_frame.grid(row=0, column=0, sticky="nsew")

register_frame = Frame(container, bg="#D88DE7")
register_frame.grid(row=0, column=0, sticky="nsew")

page1 = Frame(container, bg="#D88DE7")
page1.grid(row=0, column=0, sticky="nsew")

page2 = Frame(container, bg="#D88DE7")
page2.grid(row=0, column=0, sticky="nsew")

page3 = Frame(container, bg="#D88DE7")
page3.grid(row=0, column=0, sticky="nsew")

page4 = Frame(container, bg="#D88DE7")
page4.grid(row=0, column=0, sticky="nsew")

# Login Page#E0FFFF").pack(pady=(60, 0))
Label(login_frame, text="L o g i n", font=("Helvetica", 15, "bold"), bg="#D88DE7", fg="#E0FFFF").pack()
Label(login_frame, text="U s e r n a m e", font=("Helvetica", 12), bg="#D88DE7", fg="#E0FFFF").pack(pady=(40,
10))
login_username = Entry(login_frame, font=("Helvetica", 12), width=30, fg = "#474547")
login_username.pack(pady=6)

Label(login_frame, text="P a s s w o r d", font=("Helvetica", 12), bg="#D88DE7", fg="#E0FFFF").pack(pady=5)

```

```

login_password = Entry(login_frame, font=("Helvetica", 12), width=30, show="*", fg = "#474547")
login_password.pack(pady=5)

Button(login_frame, text=" Login ", command=login_user, font=("Helvetica", 12), bg="#D88DE7",
fg="#E0FFFF").pack(pady=10)

Button(login_frame, text=" Register ", command=lambda: show_frame(register_frame), font=("Helvetica",
12), bg="#D88DE7", fg="#E0FFFF").pack(pady=5)

login_message_label = Label(login_frame, text="", font=("Helvetica", 10), bg="#D88DE7", fg="red")
login_message_label.pack(pady=10)

# Register Page

Label(register_frame, text="Register", font=("Helvetica", 16, "bold"), bg="#D88DE7",
fg="#6A636B").pack(pady=20)

Label(register_frame, text="Username", font=("Helvetica", 12), bg="#D88DE7", fg="#E0FFFF").pack(pady=5)
reg_username = Entry(register_frame, font=("Helvetica", 12), width=30, fg = "#474547")
reg_username.pack(pady=5)

Label(register_frame, text="Password", font=("Helvetica", 12), bg="#D88DE7", fg="#E0FFFF").pack(pady=5)
reg_password = Entry(register_frame, font=("Helvetica", 12), width=30, show="*", fg = "#474547")
reg_password.pack(pady=5)

Label(register_frame, text="Confirm Password", font=("Helvetica", 12), bg="#D88DE7",
fg="#E0FFFF").pack(pady=5)
reg_confirm_password = Entry(register_frame, font=("Helvetica", 12), width=30, show="*",fg = "#474547")
reg_confirm_password.pack(pady=5)

Button(register_frame, text=" Register ", command=register_user, font=("Helvetica", 12), bg="#D88DE7",
fg="#E0FFFF").pack(pady=10)

Button(register_frame, text=" Back To Login ", command=lambda: show_frame(login_frame),
font=("Helvetica", 12), bg="#D88DE7",fg = "#6A636B").pack(pady=5)

reg_message_label = Label(register_frame, text="", font=("Helvetica", 10), bg="#D88DE7", fg="#E0FFFF")
reg_message_label.pack(pady=10)

```

```

# Store PhotoImage references to prevent garbage collection
image_objects = []

# Image Pages
for i, (frame, text, image_path) in enumerate(
    zip(
        [page1, page2, page3, page4],
        [
            "Hello!..",
            "Control Your Cash Flow..",
            "Spend Wise, Save More",
            "Start To Track ",
        ],
        [
            "src/image1.jpg",
            "src/image2.jpg",
            "src/image3.jpg",
            "src/image4.jpg",
        ],
    ),
):
    Label(frame, text=text, font=("Helvetica", 12, "bold"), bg="#D88DE7", fg="#E0FFFF").pack(pady=20)

# Load image and store reference
image = Image.open(image_path).resize((250, 300))
tk_image = ImageTk.PhotoImage(image)
image_objects.append(tk_image) # Prevent garbage collection

# Display the image
Label(frame, image=tk_image, bg="#D88DE7").pack(pady=10)

```

```

# Navigation buttons

Button(
    frame,
    text=" Next.. " if i < 3 else " Start ",
    command=lambda f=[page2, page3, page4, login_frame][i]: show_frame(f),
    font=("Helvetica", 10, "bold", "italic"),
    bg="#D88DE7",
    fg="#E0FFFF",
).pack(pady=20)


# Start with the first image page
show_frame(page1)


# Run Application
window.mainloop()


# Close database connection
conn.close()


from tkinter import *
from PIL import Image, ImageTk
import sqlite3
from expense_tracker import ExpenseTracker


# Database setup
conn = sqlite3.connect('users.db')
cursor = conn.cursor()
cursor.execute("""
    CREATE TABLE IF NOT EXISTS users (
        username TEXT PRIMARY KEY,
        password TEXT NOT NULL

```

```

    )
    ")
conn.commit()

# Functions
def register_user():
    username = reg_username.get()
    password = reg_password.get()
    confirm_password = reg_confirm_password.get()

    if not username or not password:
        reg_message_label.config(text="All fields are required!", fg="#E0FFFF")
    elif password != confirm_password:
        reg_message_label.config(text="Passwords do not match!", fg="#E0FFFF")
    else:
        cursor.execute('SELECT * FROM users WHERE username = ?', (username,))
        if cursor.fetchone():
            reg_message_label.config(text="Username already exists!", fg="#E0FFFF")
        else:
            cursor.execute('INSERT INTO users (username, password) VALUES (?, ?)', (username, password))
            conn.commit()
            reg_message_label.config(text="Registration successful!", fg="#E0FFFF")
            reg_username.delete(0, END)
            reg_password.delete(0, END)
            reg_confirm_password.delete(0, END)

def login_user():
    username = login_username.get()
    password = login_password.get()

    cursor.execute('SELECT * FROM users WHERE username = ? AND password = ?', (username, password))

```

```

user = cursor.fetchone()

if user:
    window.destroy()

    root = Tk()

    app = ExpenseTracker(root)

    root.mainloop()

    login_message_label.config(text="Login successful!", fg="#E0FFFF")

    show_frame(page1) # Navigate to the first image page on successful login
else:
    login_message_label.config(text="Invalid credentials!", fg="#E0FFFF")


def show_frame(frame):
    frame.tkraise()


# Main Application Window
window = Tk()

window.title("EXPENSE TRACKER")

window.geometry("450x600")

window.resizable(False, False)

window.config(bg="#D88DE7")


# Container Frame
container = Frame(window)

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


# Configure container for stacking pages
container.grid_rowconfigure(0, weight=1)

container.grid_columnconfigure(0, weight=1)


# Frames

```

```

login_frame = Frame(container, bg="#D88DE7")
login_frame.grid(row=0, column=0, sticky="nsew")

register_frame = Frame(container, bg="#D88DE7")
register_frame.grid(row=0, column=0, sticky="nsew")

page1 = Frame(container, bg="#D88DE7")
page1.grid(row=0, column=0, sticky="nsew")

page2 = Frame(container, bg="#D88DE7")
page2.grid(row=0, column=0, sticky="nsew")

page3 = Frame(container, bg="#D88DE7")
page3.grid(row=0, column=0, sticky="nsew")

page4 = Frame(container, bg="#D88DE7")
page4.grid(row=0, column=0, sticky="nsew")

# Login Page#E0FFFF").pack(pady=(60, 0))
Label(login_frame, text="L o g i n", font=("Helvetica", 15, "bold"), bg="#D88DE7", fg="#E0FFFF").pack()
Label(login_frame, text="U s e r n a m e", font=("Helvetica", 12), bg="#D88DE7", fg="#E0FFFF").pack(pady=(40, 10))
login_username = Entry(login_frame, font=("Helvetica", 12), width=30, fg = "#474547")
login_username.pack(pady=6)

Label(login_frame, text="P a s s w o r d", font=("Helvetica", 12), bg="#D88DE7", fg="#E0FFFF").pack(pady=5)
login_password = Entry(login_frame, font=("Helvetica", 12), width=30, show="*", fg = "#474547")
login_password.pack(pady=5)

Button(login_frame, text=" Login ", command=login_user, font=("Helvetica", 12), bg="#D88DE7", fg="#E0FFFF").pack(pady=10)
Button(login_frame, text=" Register ", command=lambda: show_frame(register_frame), font=("Helvetica", 12), bg="#D88DE7", fg="#E0FFFF").pack(pady=5)

login_message_label = Label(login_frame, text="", font=("Helvetica", 10), bg="#D88DE7", fg="red")

```

login_message_label.pack(pady=10)

Register Page

```
Label(register_frame, text="Register", font=("Helvetica", 16, "bold"), bg="#D88DE7",  
fg="#6A636B").pack(pady=20)
```

```
Label(register_frame, text="Username", font=("Helvetica", 12), bg="#D88DE7", fg="#E0FFFF").pack(pady=5)
```

```
reg_username = Entry(register_frame, font=("Helvetica", 12), width=30, fg = "#474547")
```

```
reg_username.pack(pady=5)
```

```
Label(register_frame, text="Password", font=("Helvetica", 12), bg="#D88DE7", fg="#E0FFFF").pack(pady=5)
```

```
reg_password = Entry(register_frame, font=("Helvetica", 12), width=30, show="*", fg = "#474547")
```

```
reg_password.pack(pady=5)
```

```
Label(register_frame, text="Confirm Password", font=("Helvetica", 12), bg="#D88DE7",  
fg="#E0FFFF").pack(pady=5)
```

```
reg_confirm_password = Entry(register_frame, font=("Helvetica", 12), width=30, show="*",fg = "#474547")
```

```
reg_confirm_password.pack(pady=5)
```

```
Button(register_frame, text=" Register ", command=register_user, font=("Helvetica", 12), bg="#D88DE7",  
fg="#E0FFFF").pack(pady=10)
```

```
Button(register_frame, text=" Back To Login ", command=lambda: show_frame(login_frame),  
font=("Helvetica", 12), bg="#D88DE7",fg = "#6A636B").pack(pady=5)
```

```
reg_message_label = Label(register_frame, text="", font=("Helvetica", 10), bg="#D88DE7", fg="#E0FFFF")
```

```
reg_message_label.pack(pady=10)
```

Store PhotoImage references to prevent garbage collection

```
image_objects = []
```

Image Pages

```
for i, (frame, text, image_path) in enumerate(  
    zip(  
        [page1, page2, page3, page4],  
        [  

```



```

        "Hello!..",
        "Control Your Cash Flow..",
        "Spend Wise, Save More",
        "Start To Track ",
    ],
    [
        "src/image1.jpg",
        "src/image2.jpg",
        "src/image3.jpg",
        "src/image4.jpg",
    ],
)
):
    Label(frame, text=text, font=("Helvetica", 12, "bold"), bg="#D88DE7", fg="#E0FFFF").pack(pady=20)

# Load image and store reference
image = Image.open(image_path).resize((250, 300))
tk_image = ImageTk.PhotoImage(image)
image_objects.append(tk_image) # Prevent garbage collection

# Display the image
Label(frame, image=tk_image, bg="#D88DE7").pack(pady=10)

# Navigation buttons
Button(
    frame,
    text=" Next.. " if i < 3 else " Start ",
    command=lambda f=[page2, page3, page4, login_frame][i]: show_frame(f),
    font=("Helvetica", 10, "bold", "italic"),
    bg="#D88DE7",
    fg="#E0FFFF",
).pack(pady=20)

```

```
# Start with the first image page
```

```
show_frame(page1)
```

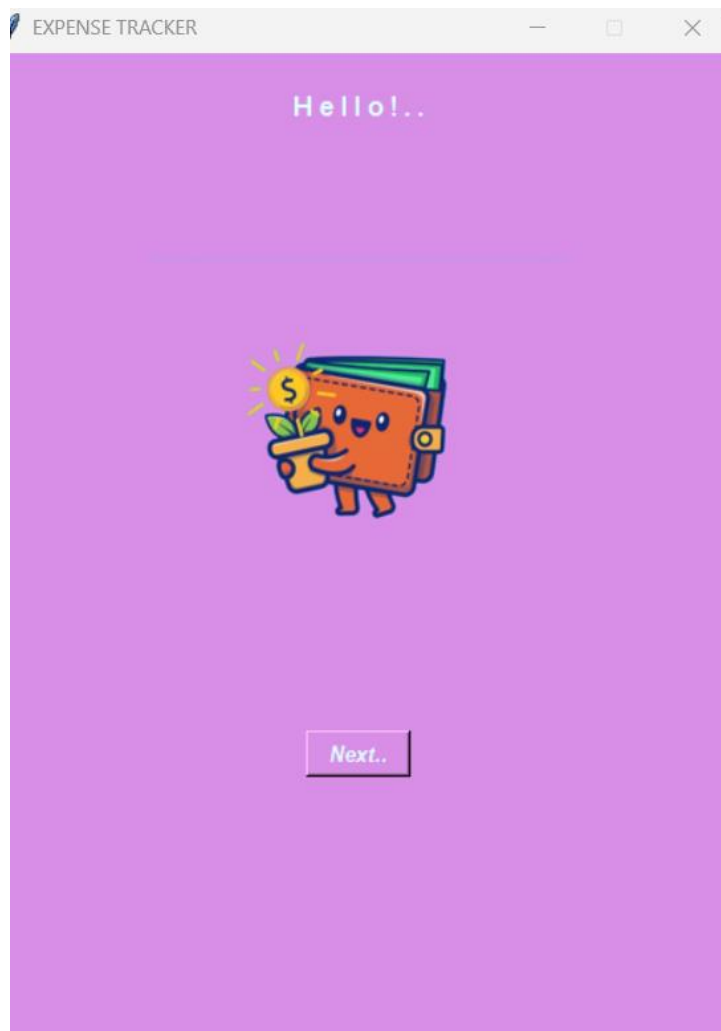
```
# Run Application
```

```
window.mainloop()
```

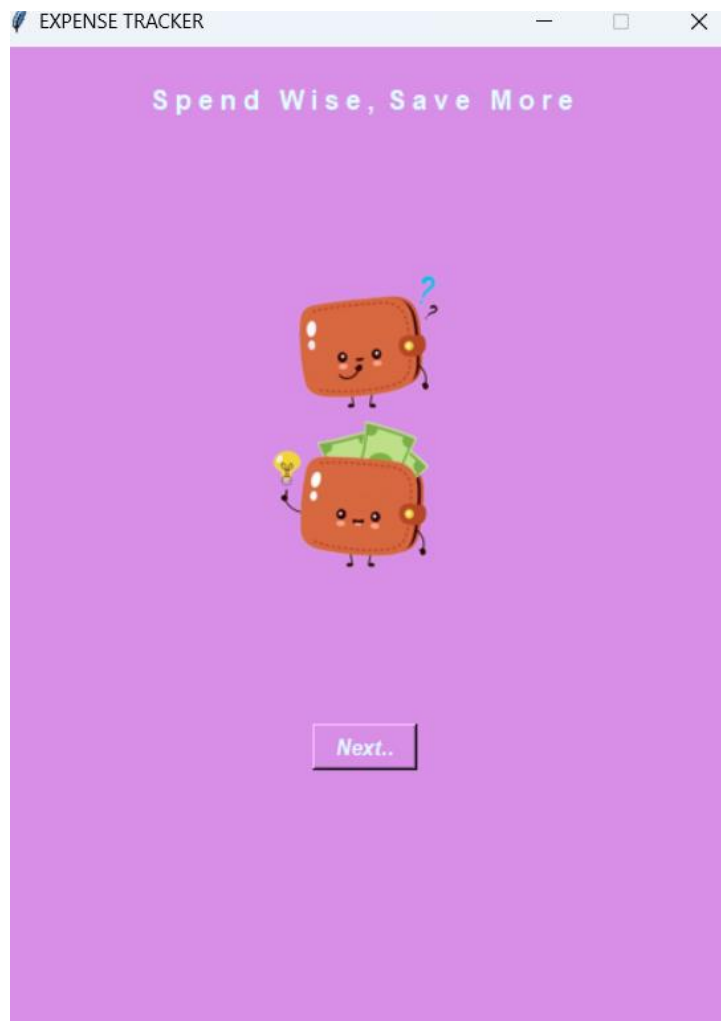
```
# Close database connection
```

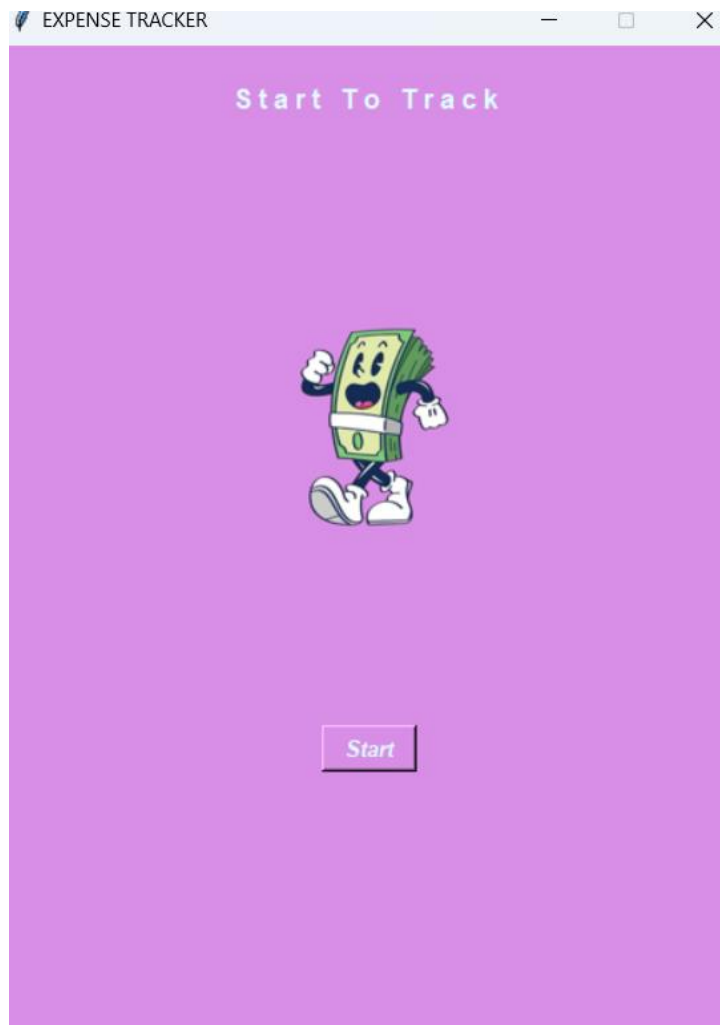
```
conn.close()
```


Sample Screen









 EXPENSE TRACKER

Register

Username

Password

Confirm Password

Register

Back To Login

EXPENSE TRACKER

— □ ×

Login

Username

Password

Login

Register

Expense Tracker

Add Expense

Add Budget

View Summary

Add Expense


Date (DD-MM-YYYY):

Amount Spent (₹):

Category:

Description:

Save Expense

 Add Budget


Date (DD-MM-YYYY):

Amount Spent (₹):

Category:

Description:

Save Budget

 Summary

Filter by Date Range:

From (DD-MM-YYYY):

Results and Analysis

2. Key Findings:

- Discuss how the application helps users manage their finances. Provide hypothetical or real examples of usage scenarios.

2. Performance Analysis:

- Include metrics such as response time for adding or retrieving data.
- Mention user feedback received during testing and how it influenced refinements.

2. Visual Examples:

- Use charts or screenshots generated by the application to demonstrate its reporting capabilities.

Conclusion

The **Expense Tracker Application** serves as a practical and efficient tool for managing personal finances, providing users with features such as income and expense logging, transaction history, and data visualization. Built using Python, SQLite, and Tkinter, the project demonstrates the integration of programming, database management, and graphical user interface design.

This project not only fulfills its intended objective of simplifying expense tracking but also serves as a testament to the versatility and power of Python as a development language. The use of SQLite ensures a lightweight yet robust database solution, while Tkinter provides an intuitive and user-friendly interface.

Through this project, I gained a deeper understanding of full-stack application development, including coding best practices, debugging, and user-centered design principles. Additionally, the project opened avenues for further enhancements, such as multi-user support, cloud-based storage, and advanced analytics.

Overall, the **Expense Tracker Application** has been a rewarding learning experience, blending theory with practical implementation. This endeavor has equipped me with valuable skills that are transferable to real-world software development and problem-solving scenarios.

References

To develop and enhance the Expense Tracker Application, the following resources and tools were utilized:

1. **Programming Language:**
 - Python documentation: <https://docs.python.org>
1. **Database:**
 - SQLite documentation: <https://sqlite.org>
1. **GUI Framework:**
 - Tkinter tutorial: <https://tkdocs.com>
1. **Data Visualization:**
 - Matplotlib documentation: <https://matplotlib.org>
1. **Online Communities:**
 - Stack Overflow: <https://stackoverflow.com>
 - GeeksforGeeks: <https://geeksforgeeks.org>
1. **Learning Resources:**
 - TutorialsPoint: <https://tutorialspoint.com>
 - W3Schools: <https://w3schools.com>
1. **Financial APIs (For Future Enhancements):**
 - Plaid: <https://plaid.com>
 - Open Exchange Rates: <https://openexchangerates.org>
1. **Version Control:**
 - GitHub: Used for code versioning and collaboration: <https://github.com>

These resources collectively supported the creation, testing, and improvement of the application.

Further Enhancements for the Expense Tracker Application

While the current Expense Tracker Application provides a robust set of features, it can be enhanced to improve functionality, usability, and scalability. Below are potential future enhancements:

1. Cloud Integration

- Enable cloud-based storage of transaction data using platforms like Firebase or AWS to allow users to access their records from multiple devices.

2. Mobile Application Version

- Develop a mobile version of the application for Android and iOS platforms using frameworks like **Kivy** or **Flutter**.

3. Multi-User Support

- Introduce account-based access to allow multiple users to have their own secure profiles with unique data.
- Implement secure authentication using **OAuth** or JWT for enhanced privacy.

4. AI-Powered Insights

- Use machine learning models to analyze user spending patterns and provide actionable insights, such as budgeting tips or recommendations for cost-saving.
- Example: Suggest a budget adjustment based on overspending trends in specific categories.

5. Integration with Financial APIs

- Integrate with APIs like **Plaid** or **Yodlee** to automatically fetch transaction data from bank accounts, credit cards, and digital wallets.

6. Multi-Language Support

- Add support for multiple languages to cater to a diverse user base.
- Leverage Python libraries like gettext for internationalization.

7. Advanced Reporting Tools

- Generate detailed financial reports in formats like **PDF** or **Excel**, with visualizations such as graphs and trendlines.
- Include options to compare monthly or yearly expenses.

8. Budget Planning and Goal Setting

- Allow users to set monthly or yearly budgets for different categories.

- Enable tracking of progress toward financial goals (e.g., saving for a vacation or reducing debt).

9. Recurring Transactions

- Add features for scheduling recurring income or expense entries (e.g., salary, rent, subscriptions).
- Automate reminders for upcoming due dates.

10. Expense Sharing and Collaboration

- Introduce expense-sharing functionality for groups or families.
- Provide real-time updates to shared users about group expenses and balances.

11. Cross-Currency Support

- Enable tracking expenses in multiple currencies and provide real-time currency conversion using APIs like **Open Exchange Rates** or **Forex API**.

12. Voice and Chatbot Assistance

- Add a voice assistant or chatbot feature to simplify interaction.
- Example: A user can ask, *"How much did I spend on food this week?"*

13. Dark Mode and Theme Customization

- Offer dark mode and customizable themes for enhanced user experience and accessibility.

14. Integration with External Tools

- Sync with third-party tools such as **Google Calendar** for reminders or **QuickBooks** for advanced accounting.