PROJECT REPORT (VITYARTHI)

Name- Arun Yadav

Registration Number- 25BCE11308

Subject- Introduction to Programming and Problem Solving

Project Title- Personal Financial Tracker and Analysis Tool (FinTrack)

1. Introduction

The Personal Financial Tracker and Analysis Tool (FinTrack) is a desktop application created using Python. Its main purpose is to give users an easy but effective way to record daily financial transactions, calculate account balances in real-time, and visualize spending habits. This project tackles the usual issue of manual tracking, which can be full of mistakes, by providing an automated and visual approach. It directly uses concepts of data structures, algorithms, and visualization learned in the course. The main implementation is based on modular programming principles, which separate data handling, business logic, and user interface components.

2. Problem Statement

Many people struggle to maintain a clear and updated view of their personal finances. Using paper ledgers or complex spreadsheets can be a hassle. This often leads to inconsistency, delays in balance calculations, and limited visual insight into spending habits. The aim is to develop a user-friendly digital system that automatically processes transactions, calculates current balances, and generates straightforward analytical charts.

3. Functional Requirements

The FinTrack application consists of three main functional modules:

- Transaction Management (CRUD): This lets users add new income or expense transactions, view the full list, update existing records, and remove outdated entries.
- Real-time Balance Calculation: The system calculates and displays the current total balance immediately after any transaction is added, updated, or deleted.
- Spending Analysis & Reporting: This creates a visual summary, such as a bar chart or pie chart, of expenses categorized by type, like Groceries, Rent, or Entertainment, for a specific time period.

4. Non-functional Requirements

The following non-functional requirements define the quality attributes of the system:

- Performance (Response Time): Transaction processing and chart generation must be nearly instant, with a target of under 2000ms.
- Reliability (Data Integrity): The transaction data must be permanently stored and can be recovered. A mechanism must stop saving corrupted or malformed data.
- Usability (Intuitive UI): The user interface must be clear, organized, and require minimal clicks for core tasks. The visualization must be easy to understand.
- Maintainability (Modularity): The codebase must be modular, separating concerns (Data, Logic, UI) to make future updates and debugging easier.

5. System Architecture

The system has a three-tier logical structure, built with Python libraries:

I. Data Layer (data_handler.py): This layer manages the transaction list, handles CRUD operations (add, delete, update), and stores data using a local JSON file.

II. Business Logic Layer (core_logic.py): This layer contains the main calculation functions, like calculate_balance and aggregate_spending. It works independently from the user interface.

III. Presentation Layer (main_ui.py): This layer manages the graphical user interface (GUI) using Tkinter. It calls the methods from the business logic and data layers based on what the user inputs.

6. Design Diagrams

The main application workflow starts with the user taking an action. The UI then calls the business logic. Next, the logic interacts with the data layer. Finally, the results go back to the UI.

Workflow Diagram: Adding a New Transaction

User enters data → UI validates → Logic calculates new balance → Data layer saves → UI → updates display

Figure 1: High-level workflow for transaction processing.

7. Design Decisions & Rationale

Python Language: It was chosen for its readability, strong library support for data processing and visualization, and ability to quickly create prototypes.

• Tkinter for UI: It was selected for its simplicity and its inclusion in the standard Python library, which avoids external dependencies for a cross-platform desktop application.

• JSON for Storage: A lightweight, human-readable format was chosen for persistent storage (transactions.json) instead of a full database. This decision reduces complexity and installation steps while still meeting the reliability requirement.

• Data/Logic Separation: The separation of CoreLogic and DataHandler was enforced to ensure that the business logic can be tested without depending on the UI and storage system, which improves maintainability.

8. Implementation Details

The application uses Python and the matplotlib library to generate charts. It employs a list of dictionaries to represent transactions.

Key Algorithm: Balance Calculation-

The calculate_balance function goes through the list of transactions. It adds up incomes and subtracts expenses.

9. Testing Approach

Tested the system with unit tests for the main logic and validation tests for user input.

9.1 Testing Plan

Unit Testing (Logic): The calculate_balance function in the CoreLogic component was tested using Python's unittest module. Test cases included:

i. An empty transaction list (expected balance: 0).
ii. A mix of incomes and expenses (checking for correct net calculation).
iii. Transactions with a zero amount (ensuring no impact on the balance).

Integration Testing: Tested the full flow of adding a transaction through the UI, checking its presence in the JSON file, and confirming the balance update on the screen.

10. Challenges Faced

The main challenge was to integrate three separate components smoothly. These were the Tkinter event loop for the user interface, file I/O operations through DataHandler, and analytical plotting using matplotlib. Managing the thread-blocking behaviour of the UI while conducting file operations required careful organization of function calls. This approach helped prevent the interface from freezing during data loads or saves.

11. Learnings & Key Takeaways

- Modular Design Value: This confirms the usefulness of the three-tier architecture in making debugging and testing of individual components easier.
- External Library Integration: We successfully added a strong third-party library, matplotlib, for visualization. This shows our skill in handling external dependencies.
- State Management: We found effective ways to manage the application's global state, including the transaction list and current balance, during different component interactions.

12. Future Enhancements

1. User Authentication: Set up user login and registration to support multiple users.

2. Cloud Storage: Move the JSON file storage to a cloud database, like Firebase or SQL, for better access across different devices.

3. Budgeting Feature: Include a function to establish monthly budgets for each category and send alerts when spending is close to or exceeds those limits.

13. References

• Python Documentation

• Tkinter Documentation

• Matplotlib Library Documentation