

VIT<sup>®</sup>  
—  
BHOPAL

---

*Personal Finance Tracker*  
*Project Report*

---



**Submitted by:** SHRUSHTI SMARANIKA SAHU (25MIP10113)

**Course / Subject:** *Introduction to Problem Solving and Programming*

**Date:** 22 November 2025

# INDEX

1. Cover Page
2. Introduction
3. Problem Statement
4. Objectives
5. Functional Requirements
6. Non-functional Requirements
7. System Architecture
8. Design Diagrams
  - o Use Case Diagram
  - o Workflow Diagram
  - o Sequence Diagram
  - o Class/Component Diagram
  - o ER Diagram
9. Design Decisions & Rationale
10. Implementation Details
11. Screenshots / Results
12. Testing Approach
13. Challenges Faced
14. Learnings & Key Takeaways
15. Future Enhancements
16. References

# **INTRODUCTION**

Personal finance management is an essential life skill, yet many people struggle to maintain consistent records of their income and expenses. Complex financial applications can feel overwhelming for users who simply want a lightweight method to record daily transactions and instantly view their financial position. This project addresses that need by implementing a straightforward, command-line-based Personal Finance Tracker using Python.

The system focuses on simplicity, clarity, and core functionality. Users can record income, log expenses, and view real-time financial summaries. Designed for beginners, students, and individuals seeking a minimal solution, this project demonstrates the fundamentals of software design, user interaction, and basic data handling in Python.

# **PROBLEM STATEMENT**

Managing personal finances is challenging without a structured way to track income and expenses. Existing tools can be overly complex for users who only need basic tracking. This project provides a simple Python-based command-line tool that allows users to record daily transactions and view financial summaries quickly.

# **OBJECTIVES**

- To create a minimalist personal finance tracker usable via the command line.
- To allow users to add income and expense records.
- To calculate and display total income, total expenses, and balance.
- To store transaction data temporarily during program execution.
- To provide a user-friendly text-based menu interface.

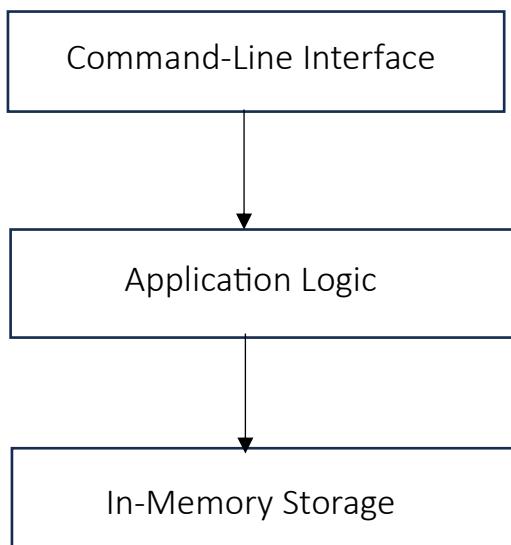
## **FUNCTIONAL REQUIREMENTS**

1. Add Income
  - o The system shall allow the user to input their income amount.
  - o The system shall store the income along with timestamp and type.
2. Add Expense
  - o The system shall allow the user to input their expense amount.
  - o The system shall store the expense along with timestamp and type.
3. View Financial Summary
  - o The system shall calculate total income.
  - o The system shall calculate total expenses.
  - o The system shall compute current balance = (income – expenses).
  - o Summary must be displayed on the screen.
4. Exit the Application

## **NON-FUNCTIONAL REQUIREMENTS**

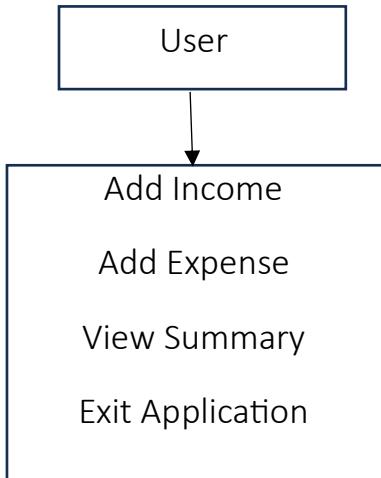
- Usability: Easy-to-navigate command-line interface
- Reliability: Error-resistant input handling
- Performance: Near-instant calculations
- Portability: Runs on any Python-supported OS
- Maintainability: Simple structure for easy updates

## **SYSTEM ARCHITECTURE**



## **DESIGN DIAGRAMS**

### **1. Use Case Diagram**



### **2. Workflow Diagram**

Start → Display Menu → User Selects Option →

- Add Income → Store Transaction → Back to Menu
- Add Expense → Store Transaction → Back to Menu
- View Summary → Show Totals → Back to Menu
- Exit → End Program

### **3. Sequence Diagram**

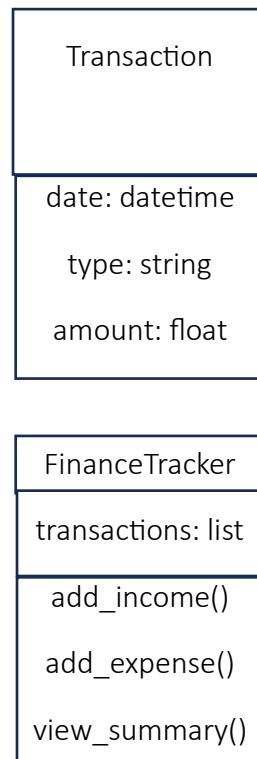
User → System: Select Option

User → System: Input Amount (if applicable)

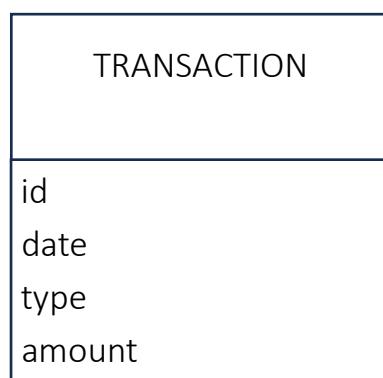
System → Storage: Create & Append Transaction

System → User: Confirmation / Summary Output

## **4. Class / Component Diagram**



## **ER DIAGRAM**



## DESIGN DECISIONS & RATIONALE

- Chose a CLI for simplicity and platform independence.
- Used in-memory storage to avoid complexity for beginner users.
- Kept architecture minimal to reflect project scope.

## IMPLEMENTATION DETAILS

- Python script using loops, lists, and dictionaries
- datetime for timestamps
- Simple calculations performed using Python list comprehensions

## SCREENSHOTS/RESULTS

```
[1] ✓ 19s
  # Very Simple Personal Finance Tracker (No Graphs, No Monthly Reports)
  # Save as: simple_finance.py

  from datetime import datetime

  transactions = []

  while True:
    print("\n1. Add Income")
    print("2. Add Expense")
    print("3. View Summary")
    print("4. Exit")

    choice = input("Choose an option: ")

    if choice == "1":
      amount = float(input("Enter amount: "))
      transactions.append({"date":datetime.now(),"type":"income","amount":amount})
      print("Income added!\n")

    elif choice == "2":
      amount = float(input("Enter amount: "))
      transactions.append({"date":datetime.now(),"type":"expense","amount":amount})
      print("Expense added!\n")

    elif choice == "3":
      income = sum(t["amount"] for t in transactions if t["type"] == "income")
      expenses=sum(t["amount"] for t in transactions if t["type"] == "expense")
      print(f"Total Income: ${income:.2f}")
      print(f"Total Expenses: ${expenses:.2f}")
      print(f"Balance: ${income - expenses:.2f}\n")

    elif choice == "4":
      print("Goodbye!")
      break

    else:
      print("Invalid choice! Try again.\n")
```

```
1. Add Income  
2. Add Expense  
3. View Summary  
4. Exit  
Choose an option: 1  
Enter amount: 50000  
Income added!
```

```
1. Add Income  
2. Add Expense  
3. View Summary  
4. Exit  
Choose an option: 2  
Enter amount: 40000  
Expense added!
```

```
1. Add Income  
2. Add Expense  
3. View Summary  
4. Exit  
Choose an option: 3  
Total Income: $50000.00  
Total Expenses: $40000.00  
Balance: $10000.00
```

---

```
1. Add Income  
2. Add Expense  
3. View Summary  
4. Exit  
Choose an option: 4  
Goodbye!
```

---

## **TESTING APPROACH**

- Manual testing with various income/expense values.
- Boundary testing (e.g., zero or negative numbers).
- Menu navigation testing.
- Error-handling testing for invalid inputs.

## **CHALLENGES FACED**

- Preventing crashes from invalid user input.
- Maintaining clarity in a menu-driven interface.
- Ensuring the program remains minimal and easy to understand.

## **LEARNINGS & KEY TAKEAWAYS**

- Importance of user-friendly interfaces.
- Benefits of modularizing even simple scripts.
- Value of UML diagrams in planning small projects.

## **FUTURE ENHANCEMENTS**

- Add file-based storage.
- Add categories for transactions.
- Add monthly reports.
- Add simple GUI version.
- Allow data export.

## **REFERENCES**

<https://www.geeksforgeeks.org/system-design/unified-modeling-language-uml-introduction/>

<https://www.geeksforgeeks.org/python/python-programming-language-tutorial/>

[https://en.wikipedia.org/wiki/Expense\\_management](https://en.wikipedia.org/wiki/Expense_management)

[https://colab.research.google.com/drive/1nqpWAeb\\_9qO6gZ6rvuhofi0qPVAAjsr0?usp=chrome\\_ntp](https://colab.research.google.com/drive/1nqpWAeb_9qO6gZ6rvuhofi0qPVAAjsr0?usp=chrome_ntp)

