Project Report: Console-Based Expense Tracker

1. Project Information

Field	**Detail**
Project Title	Simple Console-Based Expense Tracker
Submitted By	Srajal Gupta
Registration Number	**25BCE11126**
Date	November 23, 2025
Programming Language	Python

---

2. Executive Summary

The **Console-Based Expense Tracker** is a minimal, functional Python application designed to help users record and summarize their financial transactions. It provides a simple command-line interface (CLI) with options to add new expenses, view stored data, and calculate total spending. The project successfully demonstrates fundamental Python concepts, including function definition, dictionary usage, user input handling, and basic application loop control.

---

3. System Design and Data Structure

3.1. Data Storage Model

The core of the application relies on a single **Python dictionary** named expense.

$$expense = \{ \text{category} : \{ \text{details} \} \}$$

The **Category Name** (e.g., 'Food', 'Travel') serves as the **Key**, and the **Value** is another dictionary containing the expense details.

Key (Category)	Value (Details Dictionary)
"Food"	{"Date": "23-11-2025", "Description": "Lunch", "Amount": 500}
"Transport"	{"Date": "23-11-2025", "Description": "Bus fare", "Amount": 100}

3.2. Limitation of Current Design

A crucial point in the current implementation is that since the category is the dictionary key, **adding a new expense with an existing category name will overwrite the previous expense record** for that category.

For example, if the user adds "Food" (₹500) and then later adds "Food" (₹150), only the **₹150** expense will be stored for "Food".

---

4. Functional Modules

The system is organized into five main functions:

4.1. add_expense()

- **Purpose:** Collects user input for category, date, description, and amount.
- **Process:** Stores the new record as a nested dictionary within the main expense dictionary, using the category as the main key.

4.2. view_expenses()

- **Purpose:** Allows the user to view the details of a specific expense category.
- **Process:** Takes a category name as input and prints the Date, Description, and Amount associated with that category. **Note:** Due to the design limitation (Section 3.2), it only shows the last expense entered for that category.

4.3. total_expenses()

- **Purpose:** Calculates and prints the cumulative sum of all recorded expense amounts.
- **Process:** Iterates through all the expense entries in the expense dictionary and sums the values of the inner "Amount" keys.

4.4. categor_sum()

- **Purpose:** Prints the stored details for all categories.
- **Process:** Uses the .items() method to print every key-value pair in the expense dictionary, showing the category name and its full details dictionary.

4.5. Expense_tracker() (Main Function)

- **Purpose:** Serves as the application's entry point and menu handler.
- **Process:** Runs an infinite while True loop, displays the menu options, and calls the appropriate function based on the user's numeric choice (1-5).

---

5. Potential Improvements (Future Scope)

To enhance the application's functionality and practical utility, the following modifications are recommended:

1. **Multiple Expense Tracking:** Change the expense data structure from a dictionary ({}) to a **list of dictionaries** ([]). This would allow the application to record and store *all* transactions, regardless of the category, overcoming the current overwriting limitation.
2. **Input Validation:** Implement checks to ensure the amount input is an integer/float and handle non-numeric input gracefully (the current code assumes int() will succeed).
3. **Data Persistence:** Introduce functionality to save the expense data to an external file (e.g., a **CSV** or **JSON** file) before exiting, and load it upon startup, ensuring data is not lost when the program closes.
4. **Date Sorting:** Implement sorting logic in the view_expenses function to display records in chronological order.