Expense Tracker Application Documentation

By Rupesh Kumar Mistri

Expense Tracker Application

INTRODUCTION

Expense tracker application which will keep a track of Expenses of a user on a day- to-day basis. This application keeps a record of your expenses and also will give you a category wise distribution of your expenses. With the help of this application user can track their daily/weekly/monthly expenses.

Setup Steps:

1. clone repository
2. make .venv using command python -m venv .venv
3. activate virtual environment by command .venv\scripts\activate
4. install requirement.txt by command pip install -r "requirements.txt"
5. go inside src director and run command python expense_tracker.py
6. Follow instruction given in application

ARCHITECTURE

The architecture of the Expense Tracker application is structured as a command-line interface (CLI) for managing expenses. It consists of three main Python modules:

1. Modules and Class Structure

- **Category Module** (category.py):
    - Contains the Category class, representing an expense category with a single attribute, name.
    - Provides a method, to_dict, which formats category data for storage in CSV format.
- **Expense Module** (expense.py):
    - Contains the Expense class, representing an individual expense with attributes: amount, date, category, and description.
    - Also has a to_dict method, converting an Expense instance to a dictionary format for easy CSV storage.
- **Expense Tracker Module** (expense_tracker.py):
    - Central module managing application logic and interactions.

    - Contains the ExpenseTracker class, which handles reading, writing, updating, and deleting both expenses and categories.
    - Manages the data storage files (category.csv for categories and expense.csv for expenses) and their persistence.

2. Class Descriptions and Responsibilities

- **Category Class**:
    - Represents a category, such as "Food" or "Transport".
    - Responsible for storing and returning its data in dictionary format.
- **Expense Class**:
    - Represents an individual expense entry with attributes for the amount, date, category, and optional description.
    - Converts itself into dictionary format for ease of integration with pandas and CSV storage.
- **ExpenseTracker Class**:
    - Acts as the core controller for the application, handling all user commands.
    - Responsible for loading data from CSV files, managing in-memory data frames (using pandas), and providing CRUD operations for both categories and expenses.
    - Includes the following methods:
        - add_category, read_categories, update_category, and delete_category: For managing categories.
        - add_expense, read_expenses, update_expense, and delete_expense: For managing expenses.
        - generate_report: Generates daily, weekly, or monthly expense summaries.

3. Data Management with CSV Files

- The application uses **CSV files** for data storage:
    - **category.csv**: Stores the list of categories.
    - **expense.csv**: Stores expense records, with columns for amount, date, category, and description.
- pandas data frames are used to load, modify, and save data to these files.

4. Command-Line Interface (CLI)

- The **main program (main.py)** initiates an instance of ExpenseTracker and provides a simple CLI menu for interacting with the application.
- Users can navigate through options to add, view, update, and delete categories and expenses, as well as generate reports.

5. Data Flow and Execution

- **Initialization**: ExpenseTracker loads existing data from CSV files into data frames.
- **User Commands**: Each user action (e.g., adding an expense) interacts with the appropriate class methods and modifies the data frame.
- **Data Persistence**: Changes in data frames are saved back to CSV files after each operation, ensuring data is retained between sessions.

6. Error Handling and Data Validation

- The application checks for the existence of files before attempting to read/write to them.
- User inputs are validated for correctness (e.g., valid date formats) within each method.
- Provides feedback for invalid operations, like attempting to delete a non-existent category or expense.

This modular architecture enables easy maintenance and future expansion, such as adding more detailed reporting options, integrating a database for data persistence, or implementing a graphical user interface (GUI).

FEATURES

The architecture of the Expense Tracker application is structured as a command-line interface (CLI) for managing expenses. It consists of three main Python modules:

1. Modules and Class Structure

- **Category Module** (category.py):
    - Contains the Category class, representing an expense category with a single attribute, name.
    - Provides a method, to_dict, which formats category data for storage in CSV format.
- **Expense Module** (expense.py):
    - Contains the Expense class, representing an individual expense with attributes: amount, date, category, and description.
    - Also has a to_dict method, converting an Expense instance to a dictionary format for easy CSV storage.
- **Expense Tracker Module** (expense_tracker.py):
    - Central module managing application logic and interactions.
    - Contains the ExpenseTracker class, which handles reading, writing, updating, and deleting both expenses and categories.
    - Manages the data storage files (category.csv for categories and expense.csv for expenses) and their persistence.

2. Class Descriptions and Responsibilities

- **Category Class**:
    - Represents a category, such as "Food" or "Transport".
    - Responsible for storing and returning its data in dictionary format.
- **Expense Class**:
    - Represents an individual expense entry with attributes for the amount, date, category, and optional description.
    - Converts itself into dictionary format for ease of integration with pandas and CSV storage.
- **ExpenseTracker Class**:
    - Acts as the core controller for the application, handling all user commands.
    - Responsible for loading data from CSV files, managing in-memory data frames (using pandas), and providing CRUD operations for both categories and expenses.
    - Includes the following methods:
        - add_category, read_categories, update_category, and delete_category: For managing categories.
        - add_expense, read_expenses, update_expense, and delete_expense: For managing expenses.
        - generate_report: Generates daily, weekly, or monthly expense summaries.

3. Data Management with CSV Files

- The application uses **CSV files** for data storage:
    - **category.csv**: Stores the list of categories.
    - **expense.csv**: Stores expense records, with columns for amount, date, category, and description.
- pandas data frames are used to load, modify, and save data to these files.

4. Command-Line Interface (CLI)

- The **main program (main.py)** initiates an instance of ExpenseTracker and provides a simple CLI menu for interacting with the application.
- Users can navigate through options to add, view, update, and delete categories and expenses, as well as generate reports.

5. Data Flow and Execution

- **Initialization**: ExpenseTracker loads existing data from CSV files into data frames.
- **User Commands**: Each user action (e.g., adding an expense) interacts with the appropriate class methods and modifies the data frame.

- **Data Persistence**: Changes in data frames are saved back to CSV files after each operation, ensuring data is retained between sessions.

6. Error Handling and Data Validation

- The application checks for the existence of files before attempting to read/write to them.
- User inputs are validated for correctness (e.g., valid date formats) within each method.
- Provides feedback for invalid operations, like attempting to delete a non-existent category or expense.

This modular architecture enables easy maintenance and future expansion, such as adding more detailed reporting options, integrating a database for data persistence, or implementing a graphical user interface (GUI)

TESTING

Testing the Expense Tracker application involves verifying that each feature works as expected, ensuring data integrity, and checking edge cases. Given this is a command-line application, tests can be divided into unit tests for individual functions and integration tests to check the application as a whole. Below is a structured approach to testing and mock inputs/outputs.

CHALLENGES

Creating an Expense Tracker application involves several challenges, especially if it's intended to be scalable, user-friendly, and reliable. Here are some notable challenges and considerations:

1. Data Integrity and Consistency

- **Challenge**: Ensuring data remains consistent when multiple operations (such as adding, updating, or deleting categories and expenses) occur. The app needs to handle situations where data might become corrupted or inconsistent due to crashes or improper file handling.
- **Consideration**: Implementing checks to validate data types (e.g., amounts are numeric, dates are valid) and handling unexpected inputs can prevent data corruption.

2. File Handling and Concurrency

- **Challenge**: The application relies on CSV files (category.csv and expense.csv) for data storage, which presents challenges in file access, especially if multiple users or processes try to access the files simultaneously.
- **Consideration**: Using locks or upgrading to a database (like SQLite or PostgreSQL) for better concurrency control could mitigate these issues.

3. Error Handling and User Input Validation

- **Challenge**: The application depends on user inputs, which can be prone to errors, such as entering invalid dates, empty values, or non-numeric amounts.
- **Consideration**: Implementing robust input validation, such as checking date formats and verifying that amounts are positive numbers, helps prevent errors and improve user experience.

4. Data Aggregation and Reporting

- **Challenge**: Generating reports (daily, weekly, monthly) requires aggregating and formatting data, which can become complex as the dataset grows. Misconfigurations in date handling can also affect report accuracy.
- **Consideration**: Using optimized data operations and libraries like pandas helps, but performance could still degrade with large datasets. A database solution could support more efficient querying and aggregation.

5. Modularization and Code Organization

- **Challenge**: Structuring code for readability, maintainability, and scalability is crucial but can be challenging, especially as the application grows with additional features.
- **Consideration**: Adopting principles like Object-Oriented Programming (OOP) and modularization can help manage complexity. Ensuring clear separation between data handling, business logic, and user interface functions is also essential.