Final Project - Check-In 2
INST326
Group 32: John Bursie, Nathaniel Ball, Nguyen Tran
Professor Cruz
Date: May 11, 2025

What has been working?

The core functionality is finished and stable. The code is able to read and write to a csv expenses the user enters. We have an operational command-line interface (CLI), which supports adding new expenses as well as generating three types of financial reports that will help the user manage their finances. The three reports focus on:

- Summary of total aggregate costs classified according to category. The report will look at each category and display the total expenses for that category. This gives the user the ability to see where they are spending the most amount of money.
- Total expense breakdown for each month. It adds up all the expenses in the month and displays the total expenses by month.
- A listing that illustrates the users' highest purchases to his/her lowest expense. The listing displays the date in which the entry was made as well as the category and description.

Every report performs as intended and accurately interprets data from our CSV file.

- Persistent data storage in a CSV file is reliable. When you add a new transaction, it's properly appended to the file expenses.csv. The script can read from that file as well, to produce reports, with both initial file creation and incremental updates supported.
- The Transaction class is complete and tested. It holds each transaction's date, amount, category, and description, and they are printed neatly. We have validated that it does data types correctly (i.e., interpreting amounts as floats).
- Argument parsing with argparse is functioning nicely. The user may use arguments such as --amount, --category, and --description, and the program correctly handles input and gives meaningful error messages when any required argument is missing.
- The unit tests have been completed and pass for all principal parts. The transaction class, file writing/reading, and summary reporting have been tested. These all help assure that the alterations to the program do not cause problems with the existing functionality.
- Collaboration and teamwork have gone very smoothly. Each of us has owned one of the essential parts of the project, and we have debugged and assisted each other with individual parts as necessary.

What has not been functioning?

- ASCII visual charts are still being developed. We originally hoped to add ASCII bar charts to visually display category and month budgets, but getting these to fit into the terminal nicely, in terms of both scaling and aligning them horizontally so the data is clearly represented, has taken longer than we hoped.
- There are edge cases with bad or absent CSV data which are not yet managed. If the file is corrupted or manually modified (i.e., column missing), the application crashes rather than offering graceful fallback or an informative warning. We need to enhance data checking when reading files.
- Monthly formatting logic does not always deal with edge dates or future/past ranges correctly. Currently, the summary relies upon string slicing of the date, which is acceptable under regular circumstances but may produce inconsistencies when dealing with malformed or improperly formatted dates.

What can you do differently to make improvements where things haven't been working?

- Add try/except statements to our file reading functions. This will allow us to catch and report problems such as missing headers or wrong field types, instead of allowing the script to fail or crash.
- Break down the implementation of the ASCII chart into incremental steps. John can create an MVP that displays simple bar lengths expressed as dollar values. When that's functional, we can add labels and alignment refinement.
- Let's add malformed or invalid CSV entries to our unit tests. Nguyen can prepare defective test files with missing fields and ascertain how the read_expense_data() function handles them. The function should skip invalid rows but read good ones.
- Enhance our coordination through at least one additional scheduled Zoom session. During this time, we'll also go over any outstanding bugs, verify with real-world inputs, and complete chart logic.

GitHub Repository URL

https://github.com/jabursie/326FinalProject

https://github.com/njbschool-git/326FinalProject.git

Instructions to Run the Code

1. Clone the repository:
Open your terminal window and execute:
git clone https://github.com/jabursie/326FinalProject.git
cd 326FinalProject

2. Ensure you are working with Python 3.x:
Verify your copy with the following:
python --version
If necessary, use python3 instead of python throughout the steps below.

3. Include an additional cost:
To add an expense, use this command:
python planner.py ADDEXPENSE --amount 25 --category food --description "Lunch with friends"
It will log an additional transaction with the present timestamp.

4. Review expense reports:

To see an overview by category:
python planner.py REPORT1

Report 1: Summary by Category
groceries: $25.99
transportation: $15.00
rent: $1200.00
entertainment: $45.50
utilities: $241.00
To see total spending by month:
python planner.py REPORT2
Report 2: Monthly Expenses
2025-04: $1527.49

To see the 5 largest costs:
python planner.py REPORT3

Report 3: Largest Expenses
2025-04-18 11:42:10 | $1200.00 | rent | April apartment rent
2025-04-19 16:18:46 | $100.50 | utilities | Paid for food
2025-04-19 15:11:36 | $55.50 | utilities | Paid electricity bill
2025-04-18 11:45:30 | $45.50 | entertainment | Concert tickets
2025-04-18 22:04:54 | $42.50 | utilities | Paid electricity bill

5. Execute the unit tests (optional but suggested):
From the project's root folder, execute:
python -m unittest tests\test_planner.py

This will execute all your test cases to check that your code behaves as desired

python -m unittest tests\test_planner.py
Expense added: $75.0 - travel - Train ticket

```
----------------------------------------------------------------------
Ran 7 tests in 0.016s

OK
```