Lab 1: Repo organization

Fabio Pizzati

January 2026

Introduction

You are provided with a simple command-line application written in Python. The program takes as input a directory of receipt images and processes each file independently. For every receipt, it calls a language model to extract a fixed set of fields: the receipt date, the total amount spent, the vendor name, and a category chosen from a predefined list. The output is a single JSON dictionary mapping each receipt filename to the extracted information. No validation, normalization, or post-processing is performed at this stage.
The codebase is intentionally minimal and organized into a small number of source files, each with a specific responsibility: file input handling (`file_io.py`), interaction with the language model (`gpt.py`), and orchestration through a command-line interface (`main.py`). The repository deliberately omits organization and comments, which you are expected to add as part of the exercise.
Before starting the laboratory exercise, you need to prepare your working environment and familiarize yourself with the provided codebase. Now, follow the steps below to complete the exercise:

Preparation

Download the starter code as a ZIP archive from the following address:

```
<LINK TO BE PROVIDED>
```

Once downloaded, unpack the archive in a directory of your choice on your local machine. This directory will become the working directory for the entire laboratory session.

Step 1: Running the Provided Program

The provided application relies on the OpenAI API to extract information from receipt images. To run the program, you must first configure an OpenAI API key in your environment. You can use the following key:

```
OPENAI_API_KEY = <API KEY TO BE PROVIDED>
```

Set the API key as an environment variable. Once the key is configured, run the program from the command line by passing the path to the directory containing receipt images. Verify that the program executes correctly and prints a JSON object containing the extracted information for each receipt.

Step 2: Repository Reorganization and Environment Setup

1. After confirming that the program runs successfully, reorganize the repository to follow standard Python project conventions. Move all source code files into a `src` directory, adjusting imports as necessary so that the application continues to run correctly.

2. Next, create a virtual environment (`venv`) dedicated to this project and use it to run the application. Inside the repository, add a `pyproject.toml` file that declares the project metadata and all required dependencies needed to execute the code. Ensure that the project can be installed and run using this configuration.

3. Create a simple Makefile where there is a single instruction `run` that allows to run the program with the `--print` option enabled, after setting the key in an environment variable.

4. Finally, include a `README.md` where you describe the software functionality and how to run it, and a license of your choice.

Step 3: Code Understanding and Documentation

1. Carefully read the provided source code and identify the role of each module and function. Your goal is to understand how data flows through the system, from reading files on disk, to calling the language model, to producing the final JSON output.

2. Write documentation for each function using the Google-style docstring format introduced during the lecture. The documentation should clearly describe the purpose of the function, its parameters, return values, and any relevant assumptions.

3. Once the docstrings are written, use the documentation generation tools presented in class to automatically generate user-facing documentation from the source code in a `docs` folder.

Step 4: Version Control Setup

1. Initialize a local Git repository in the project directory.

2. Create a `.gitignore` file that excludes unnecessary or generated files from version control, including the receipts directory, the virtual environment directory, the `docs` folder, `__pycache__` folders, and any other artifacts produced when running the program.

3. Next, create a remote repository on GitHub and link it to your local repository.

4. Push the complete project to GitHub using a single commit with the commit message `initial commit`. At the end of this step, your local and remote repositories should be synchronized and ready for further development.

The next phase of the exercise will focus on extending the application by implementing additional features. *Each feature must be developed on its own Git branch.*

Step 5: Adding a functionality (easy)

1. Create a new branch called "`feat/sanity_checks`"

2. The language model output is not always perfect. Sometimes it parses the correct amount without any indication of the currency. Sometimes, it also includes the currency symbol, such as "$", in the parsed value. Add a small function that processes the output of the language model, removes the "$" symbol if present, converts the parsed value to float, and replaces the one in the dictionary.

3. Once the function is ready, test it and push the new branch to the remote Github repository.

4. Create a pull request describing the additional feature, ask a classmate to review it, and if they believe that it is describing correctly the feature, merge it. Otherwise, modify it.

The following sections are optional.

Step 6: Adding a functionality (medium)

1. Create a new branch called `feat/expenses`.

2. Extend the command-line application by adding a new optional mode called `--expenses`. When the `--expenses` option is selected, require two additional command-line arguments: a start date and an end date, both provided in the ISO format `YYYY-MM-DD`. Process only those receipts whose extracted date falls within the specified date range, including both the start and end dates, computing the total amount spent by summing the numeric amounts associated with the selected receipts. Create a corresponding Makefile entry for the new option.

3. Ignore receipts with missing or invalid dates, as well as receipts with missing or invalid amounts.

4. Once the feature is implemented, test it locally, commit the changes, push the branch to GitHub, open a pull request, and merge it after review.

Step 7: Adding a functionality (hard)

1. Create a new branch called `feat/plot_by_category`.

2. Extend the command-line interface by adding a new option (for example `--plot`) that triggers the generation of a visualization instead of printing raw JSON output. Add a corresponding entry to the Makefile.

3. Add the `matplotlib` library to the project dependencies in `pyproject.toml` and install it in your virtual environment. You may consult the official documentation for reference: `https://matplotlib.org/stable/contents.html`.

4. When the new option is selected, aggregate the total amount spent for each expense category using the data already extracted from the receipts, and generate a pie chart showing the distribution of expenses across categories. The pie chart must include category labels, percentage values for each slice, and a clear title. Save the generated figure to a file (for example `expenses_by_category.png`).

5. Explore a small number of visualization options provided by `matplotlib` (such as colors, slice separation, legend placement, or start angle) and choose a configuration that improves readability. Briefly describe your choices in the README.

6. Commit your changes, push the branch to GitHub, open a pull request, ask a classmate to review it, and merge it once approved.