Personal Finance Tracker

Keziah Blossom Pereira

November 2025
Course: Basic Toolkit for Data Science

Team Members:
Samrudhi Shikerkar, Falak Sardar, Anusha Shrivastava, Keziah Blossom Pereira

Contents

1 Introduction 4

2 Project Overview and Objectives 4

2.1 Objectives . . . 4

2.2 System Architecture . . . 4

3 Work Distribution and Team Roles 5

3.1 Team Members and Responsibilities . . . 5

3.2 Branching Strategy . . . 5

4 Tools and Technologies Used 5

5 Detailed Description of My Contribution 6

5.1 Design Goals for `history.py` . . . 6

5.2 Final Code: `history.py` . . . 6

5.3 Integration and Example Usage . . . 7

5.4 Testing Results . . . 8

6 Git Version Control and Commit History 8

6.1 Branching and Collaboration Practices . . . 8

6.2 Representative Commits for `history.py` . . . 9

6.3 Sample Git Commands Used . . . 9

7 Project Results and Output 9

7.1 Functional Results . . . 9

7.2 Sample Output (Console) . . . 10

8 Code Explanation and Design Choices 10

8.1 Why methods return data rather than printing . . . 10

8.2 Date handling . . . 10

8.3 Extensibility . . . 10

9 Challenges and Lessons Learned 11

9.1 Challenges . . . 11

9.2 Lessons Learned . . . 11

10 Personal Reflection and Contribution Summary 11

11 Conclusion 12

12 References 12

Abstract

This report documents the development and collaboration workflow of the *Personal Finance Tracker* project, created for the course *Basic Toolkit for Data Science*. The project used Git and GitHub for version control, Python for implementation, Markdown for lightweight documentation, and LaTeX for academic reporting. The system is modular: each team member developed an independent module on a feature branch, then integrated the modules into a single application. This document focuses on the project's goals, team organization, the tools and methods used, and a detailed account of my contribution (the `history.py` module). The report presents implemented code, testing results, Git commit history, challenges, and reflections on teamwork and version control best practices.

1 Introduction

In data science and software engineering, reproducible collaboration and version control are essential skills. The *Personal Finance Tracker* project was designed to practice these skills by implementing a modular Python application that allows users to log income and expenses, set budgets, view historical transactions, and visualize spending patterns. The project was completed by a four-member team using GitHub to coordinate changes.

This report documents:

- project goals and architecture,
- team roles and the distribution of work,
- tools and technologies used,
- a detailed description of my contribution (History module),
- test results and sample outputs,
- the Git workflow and commit history, and
- reflections and conclusions.

2 Project Overview and Objectives

2.1 Objectives

The main objectives were:

1. Build a simple, modular Personal Finance Tracker in Python.
2. Use Git and GitHub for collaborative development and version control.
3. Maintain clear documentation using Markdown for repository-level docs and LaTeX for the academic report.
4. Split work among four members so each implements and tests a dedicated module.

2.2 System Architecture

The project uses a modular architecture where each module has a clear responsibility:

- `transactions.py` – transaction recording and management (Member 1).
- `budgeting.py` – budget setting and monitoring (Member 2).
- `history.py` – transaction history viewing and filtering (Member 3 — Keziah).

- visualization.py – charts and notifications (Member 4).
- main.py – integration script to demonstrate module interaction.
- data.json – local JSON-based storage for transactions and budgets.

3 Work Distribution and Team Roles

3.1 Team Members and Responsibilities

Member	**Module**	**Responsibility**
Samrudhi (Member 1)	transactions.py	Implement TransactionManager for adding and returning transactions; maintain input validation.
Falak (Member 2)	budgeting.py	Implement BudgetManager to set and check budgets against spending.
Keziah Pereira (Member 3)	history.py	Implement HistoryManager to view and filter transaction history by date and category.
Anusha (Member 4)	visualization.py	Implement visualization (pie charts) and notification stubs.

3.2 Branching Strategy

Each member worked on a separate feature branch:

- transactions (Samrudhi)
- budgeting (Falak)
- history (Keziah)
- visualization (Anusha)

A lead (project owner) integrated the branches into main after peer review and basic testing.

4 Tools and Technologies Used

- **Git** for distributed version control.
- **GitHub** for remote repository hosting, pull requests, and code review.
- **Python 3.x** for implementation of modules.
- **Matplotlib** for visualization (pie charts).

- **JSON** (local `data.json`) for simple persistence.
- **Markdown** (`README.md`, docs) for repository documentation.
- **LaTeX** (Overleaf) for the academic report and final documentation.
- Development environment: Visual Studio Code (recommended settings and extensions noted in repository README).

5 Detailed Description of My Contribution

This section explains the work I (Member 3, Keziah Pereira) completed: the History module. It includes design decisions, the final code, testing approach, and how the module integrates with other components.

5.1 Design Goals for `history.py`

- Provide easy access to the full transaction list.
- Support filtering by date range (inclusive).
- Support filtering by category.
- Keep the interface simple to integrate with the TransactionManager and the rest of the system.
- Write concise, testable functions that return data structures (not just print).

5.2 Final Code: `history.py`

```
# history.py
# Author: Keziah Pereira
# Provides methods for transaction history display and filtering.

class HistoryManager:
    def __init__(self, transaction_manager):
        """
        transaction_manager: an object that provides:
            - get_all_transactions() -> list of transaction dicts
            - get_transactions_by_category(category) -> filtered
               ↪ list
        Each transaction dict is expected to have at least:
            - ’date’ (ISO string or comparable string)
            - ’category’ (string)
            - ’amount’ (numeric)
        """
        self.tm = transaction_manager

    def show_all(self):
        """Return all transactions as a list. Returns [] if none."""
        transactions = self.tm.get_all_transactions()
        return transactions if transactions else []

    def filter_by_date(self, start_date, end_date):
        """
        Return transactions with start_date <= txn[’date’] <=
           ↪ end_date.
        Dates are expected to be comparable strings (ISO format
           ↪ recommended).
        """
        transactions = self.tm.get_all_transactions()
        return [txn for txn in transactions
                if start_date <= txn[’date’] <= end_date]

    def filter_by_category(self, category):
        """Return transactions for the provided category."""
        return self.tm.get_transactions_by_category(category)
```

5.3 Integration and Example Usage

To test the History module independently, a small mock TransactionManager was used:

```
# Mock transaction manager for testing
class MockTransactionManager:
    def __init__(self):
        self.transactions = [
            {’id’: 1, ’date’: ’2025-10-10’, ’category’: ’Food’, ’
               ↪ amount’: 120},
            {’id’: 2, ’date’: ’2025-10-12’, ’category’: ’Travel’, ’
               ↪ amount’: 300},
            {’id’: 3, ’date’: ’2025-10-15’, ’category’: ’Food’, ’
               ↪ amount’: 50}
        ]
    def get_all_transactions(self):
        return self.transactions
    def get_transactions_by_category(self, category):
        return [t for t in self.transactions if t[’category’] == category]

# Example test
tm = MockTransactionManager()
hm = HistoryManager(tm)

print("All:", hm.show_all())
print("Date filter (2025-10-11 to 2025-10-15):", hm.filter_by_date("2025-10-11", "2025-10-15"))
print("Category filter (Food):", hm.filter_by_category("Food"))
```

5.4 Testing Results

The mock test returns:

- `show_all()` returns all 3 transactions.
- `filter_by_date("2025-10-11","2025-10-15")` returns transactions with ids 2 and 3.
- `filter_by_category("Food")` returns transactions with ids 1 and 3.

These behaviors confirm that basic filtering works when dates are stored in ISO-like string format. If date objects (e.g., `datetime.date`) are used, minor adjustments to comparison logic would be needed.

6 Git Version Control and Commit History

6.1 Branching and Collaboration Practices

We used feature branches for each module. Typical steps taken by each member were:

1. `git clone <repo>`
2. `git checkout -b <feature-branch>`
3. Make code changes and test locally.
4. Stage and commit frequently with descriptive messages.
5. Push branch and create a Pull Request on GitHub.
6. Address review comments and merge after approval.

6.2 Representative Commits for `history.py`

Below is a condensed, representative commit log for the History module. Commits were designed to be small, atomic, and descriptive.

Type	Commit message
feat	add basic HistoryManager class for managing transaction history
feat	implement show_all() method to return all transactions
feat	add filter_by_date() for date-range filtering
feat	add filter_by_category() for category filtering
docs	add author and module docstring; explain expected transaction schema
fix	handle empty transaction lists by returning [] instead of None
refactor	improve variable names and inline documentation for readability

6.3 Sample Git Commands Used

```
# Create feature branch and work
git checkout -b history
git add history.py
git commit -m "feat: add basic HistoryManager class for managing
   ↪ transaction history"
git commit -m "feat: implement show_all() method to return all
   ↪ transactions"
git commit -m "feat: add filter_by_date() for date-range filtering"
git commit -m "fix: handle empty transaction lists"
git push origin history

# Integration (performed by repository owner)
git checkout main
git merge history
git push origin main
```

7 Project Results and Output

7.1 Functional Results

When integrated with the other modules, the project offers:

- A functioning transaction pipeline: add transactions, set budgets, view history, and visualize expenses.
- Correct filtering of transactions in the History module for standard ISO-style dates.
- Basic visualizations (pie chart) that summarize expense distribution across categories.

7.2 Sample Output (Console)

```
All:
[{’id’: 1, ’date’: ’2025-10-10’, ’category’: ’Food’, ’amount’: 120},
 {’id’: 2, ’date’: ’2025-10-12’, ’category’: ’Travel’, ’amount’:
    ↪ 300},
 {’id’: 3, ’date’: ’2025-10-15’, ’category’: ’Food’, ’amount’: 50}]

Date filter (2025-10-11 to 2025-10-15):
[{’id’: 2, ’date’: ’2025-10-12’, ’category’: ’Travel’, ’amount’:
   ↪ 300},
 {’id’: 3, ’date’: ’2025-10-15’, ’category’: ’Food’, ’amount’: 50}]

Category filter (Food):
[{’id’: 1, ’date’: ’2025-10-10’, ’category’: ’Food’, ’amount’: 120},
 {’id’: 3, ’date’: ’2025-10-15’, ’category’: ’Food’, ’amount’: 50}]
```

8 Code Explanation and Design Choices

8.1 Why methods return data rather than printing

Returning data (lists/dicts) keeps functions testable and reusable. The presentation layer (CLI or GUI) should handle printing or visualization. This separation of concerns makes unit testing straightforward.

8.2 Date handling

The implementation assumes dates are stored as comparable strings (ISO format: YYYY-MM-DD or ISO datetime). This simplifies lexical comparison for inclusive ranges. If the project later uses `datetime` objects, the comparisons will be numeric/time-based and slightly more robust.

8.3 Extensibility

The HistoryManager is intentionally minimal and designed for extension:

- Add sorting (by date, amount).
- Add more filters (by amount range, transaction type).
- Add pagination for long histories.

9 Challenges and Lessons Learned

9.1 Challenges

- **Date formats:** Ensuring consistent date formatting across modules was crucial. We standardized on ISO-like strings during development.
- **Merge conflicts:** Occasional merge conflicts required careful resolution and communication via GitHub PR comments.
- **Testing across modules:** Integrating and testing modules required mock objects and small integration scripts to validate behavior.

9.2 Lessons Learned

- Commit frequently and write descriptive commit messages.
- Use feature branches and Pull Requests for safer integration.
- Keep modules small and focused to ease testing and reviews.
- Document expected input/output formats for each module (contract).

10 Personal Reflection and Contribution Summary

As Member 3, I took responsibility for the History module. My main contributions were:

- Implementing the `HistoryManager` with methods for showing all transactions and filtering by date and category.
- Writing module-level documentation and ensuring expected transaction schema compatibility.
- Performing unit-style tests using a mocked TransactionManager to validate logic.
- Participating in PR review and addressing any suggested changes related to the history module.
- Contributing to the project's Git workflow best practices and providing a clear commit history for evaluation.

11 Conclusion

The Personal Finance Tracker project provided hands-on experience in modular Python development and collaborative version control using Git and GitHub. My work on the History module produced a simple, tested, and extensible component that integrates cleanly with other modules. The project emphasized good software engineering practices: atomic commits, feature branching, and clear documentation using Markdown and LaTeX. These skills are essential for data science projects that require reproducibility and teamwork.

12 References

The source code for the Personal Finance Tracker project, including all modules, is publicly available on GitHub: Personal Finance Tracker.

— End of Report —