

Programming 1 — Formative Project (Week 7)

Weighting: 50 % of module grade (formative)

Mode: Individual Submission

Language: Python 3

Project Overview

In this project, you'll build a **Budget Tracker** (runs on the command line) that allows users to record income and expenses, view and filter transactions, and display summaries within a single terminal session. This project specification will be available in the repository (https://github.com/ypoorun/prog1FormativeProject_F25). Students will be required to clone this repository on their local machine.

Project Requirements

Your program must:

1. **Add Transactions**
 - Accept date, amount, category, description, and transaction type (income or expense).
 - Store the transactions in a list or dictionary.
2. **List Transactions**
 - Display all transactions in a clean, readable format.
3. **Filter Transactions**
 - Filter by type (income/expense), category, or month (e.g., 2025-10).
4. **Summarize Budget**
 - Display total income, total expenses, balance, and per-category totals.
5. **Validate Input**
 - Handle invalid menu choices and amounts gracefully.
6. **Session Only**
 - All data remains in memory during execution. No saving or loading to files.

Technical Requirements

- **Strings & Conditionals:** For menu control, validation, and flow.
- **Loops:** Main program loop and transaction iteration.
- **Functions:** Break down features logically (e.g., `add_income()`, `filter_transactions()`, `show_summary()`).

- **Collections:** Use lists or dictionaries for transactions and summaries.
- **OOP:**
 - *Transaction* class (attributes: *date*, *amount*, *category*, *description*, *ttype*).
 - *BudgetTracker* class (methods for add/list/filter/summary).
- **Inheritance:** Implement *Income* and *Expense* as subclasses of *Transaction*.
- **Robustness:** Input validation (e.g., numeric amount, menu selection), graceful handling of empty datasets.

Sample Menu

- 1) Add income
- 2) Add expense
- 3) List transactions
- 4) Filter (by category / type / month)
- 5) Show summary
- 0) Exit

Deliverables

- **GitHub repository** with regular commits
- **Python source code** (.py files)
- **README.md** (max 2 pages) with:
 - Project overview & features
 - Instructions to run the program
 - Menu structure
 - Sample interactions
- **Screenshots** showing *add*, *list*, *filter*, and *summary*
- **Short reflection** (max 1 page):
 - What did you learn?
 - Challenges you faced?
 - How do you intend to improve it, given more time?

Suggested Class Skeleton

e.g class Transaction

```
class Transaction:
    def __init__(self, date, amount, category, description, ttype):
        self.date = date
        self.amount = float(amount)
```

```

self.category = category.lower().strip()
self.description = description
self.type = ttype # 'income' or 'expense'

```

Expectations

- Clean, user-friendly menu
- Program runs without errors or crashes
- Logical structure with functions and OOP
- Inheritance applied purposefully
- Proper Git history — no single final commit dump

Optional Features

- Budget threshold warnings
- Top spending categories
- Undo last transaction
- Basic test assertions

Assessment Rubric (50 %)

Section	Detailed Criteria	Weight
1. Environment & Setup	<ul style="list-style-type: none"> • Git environment correctly configured and repository cloned from GitHub. • Initial setup tested with a working print statement. • At least 3+ incremental commits showing progressive development (not a single final push). • Clear folder structure (main.py, supporting files). 	8%
2. Strings & Conditionals	<ul style="list-style-type: none"> • Menu system implemented with clear, readable text and consistent structure. • User inputs handled using input(); appropriate use of string formatting for output messages. • Conditional statements (if/elif/else) used to navigate between menu options correctly. • Invalid inputs handled gracefully with appropriate feedback messages. 	8%

3. Functions & Modularity	<ul style="list-style-type: none"> Code divided logically into functions that each perform a single task (e.g., <code>add_income()</code>, <code>add_expense()</code>, <code>show_summary()</code>). Functions use parameters and return values effectively instead of relying solely on global variables. Demonstrates understanding of code reuse and avoids repetition. Each function is well-commented and contributes to an overall modular program flow. 	8%
4. Loops & Collections	<ul style="list-style-type: none"> Main program loop correctly manages user interactions until exit option is chosen. Proper use of for or while loops for iterating through data collections. Transactions stored in appropriate data structures (e.g., list of dictionaries or list of class objects). Operations such as filtering or calculating totals are implemented using loops efficiently and without logic errors. 	8%
5. Classes (OOP)	<ul style="list-style-type: none"> Transaction class correctly defined with attributes (<code>date</code>, <code>amount</code>, <code>category</code>, <code>description</code>, <code>type</code>). BudgetTracker class implemented with methods to add, list, filter, and summarize transactions. Evidence of understanding encapsulation—attributes and methods logically grouped. Demonstrates correct object creation and method calls. Code is readable and aligns with Python naming conventions. 	9%
6. Inheritance & Correctness	<ul style="list-style-type: none"> Income and Expense subclasses extend Transaction and correctly inherit properties. Appropriate use of <code>super()</code> constructor to initialize inherited attributes. Program runs reliably without crashes during all menu operations. Input validation ensures that only valid transactions are stored (e.g., positive amounts). Output values and summaries are logically correct (e.g., total income minus total expenses = balance). 	9%
TOTAL		50%