Simple Expense Tracker

Project Report

Student Project Documentation

Course: Python Programming
Date: November 2024
Project Type: Command-Line Application

---

1. Cover Page

PROJECT TITLE: Simple Expense Tracker
PROJECT TYPE: Personal Finance Management Tool
TECHNOLOGY STACK: Python 3.x
DEVELOPMENT APPROACH: Console-based Application

---

2. Introduction

Let's be real - keeping track of expenses is annoying. I wanted to build something that actually makes it less painful, so I created this command-line expense tracker.

The whole idea was to make a simple app where you can quickly add what you spent money on, see where your money's going, and maybe feel a little more in control of your finances. No complicated interfaces, no cloud sync, no subscriptions - just a straightforward Python program that does what it needs to do.

I decided to make it friendly with emojis and casual messages because financial stuff is already stressful enough. Why make the interface boring too?

What it does:

- Tracks your expenses with descriptions, amounts, and categories
- Shows you totals and breakdowns
- Lets you filter by spending category
- Gives you a simple way to manage and delete entries
- Doesn't judge you (too much) when you spend $500 on stuff

---

3. Problem Statement

The Problem:

Most people don't track their expenses consistently because:

- Fancy apps require too many steps (sign up, sync, categorize with 50 options)
- Spreadsheets are boring and feel like work
- Mobile apps have ads and want your data
- Nobody wants another subscription service

The Gap:
There's a need for something dead simple - something you can open, type a few things, and close. No frills, no data collection, no complexity.

My Solution:
A lightweight command-line tool that runs locally, has zero dependencies, stores data temporarily (you can extend it later), and makes expense tracking feel less like a chore with friendly messages and a clean interface.

Target Users:

- Students managing pocket money
- Anyone wanting to start tracking expenses without commitment
- People who prefer terminal apps
- Developers who like keeping things simple

---

4. Functional Requirements

Here's what the app needs to do:

FR1: Add Expenses

- User can input expense description, amount, and category
- System validates that amount is a valid positive number
- System assigns unique ID to each expense
- Provides confirmation message after adding

FR2: View All Expenses

- Display all expenses in a formatted table
- Show ID, description, amount, and category for each
- Handle empty list gracefully

FR3: Calculate Total

- Sum all expenses and display total

- Provide contextual feedback based on spending amount
- Handle zero expenses scenario

FR4: Filter by Category

- Allow user to specify category name
- Display only expenses matching that category
- Show category-specific total
- Handle case where category has no expenses

FR5: Delete Expenses

- Show current expenses with IDs
- Allow deletion by ID number
- Validate ID exists before deletion
- Confirm deletion to user
- Allow cancellation (ID = 0)

FR6: Menu System

- Display clear menu with all options
- Accept user choice (1-6)
- Route to appropriate function
- Handle invalid menu selections
- Allow graceful exit

---

5. Non-functional Requirements

NFR1: Usability

- Interface should be intuitive for non-technical users
- Error messages should be helpful, not cryptic
- Menu should be clear and emoji-enhanced for better UX
- Response time should feel instant (under 1 second)

NFR2: Reliability

- Input validation should prevent crashes from bad data

- No external dependencies means no version conflicts
- Pure Python ensures it runs anywhere Python does

NFR3: Maintainability

- Code should be modular with clear function names
- Each function should do one thing well
- Comments where logic might be unclear
- Easy to extend with new features

NFR4: Performance

- Should handle hundreds of expenses without lag
- Linear search is acceptable for small datasets
- No memory leaks from list operations

NFR5: Portability

- Must run on Windows, Mac, and Linux
- No OS-specific features
- Standard input/output only

NFR6: Security

- No data sent to external services
- All data stays local
- No file system access (current version)

---

6. System Architecture

Architecture Style: Monolithic Console Application

The app follows a simple procedural architecture:

Key Components:

1. **Main Controller** (`main()`)
   - Initializes expense list
   - Runs main program loop
   - Routes user choices to functions
2. **UI Layer** (`display_menu()`)
   - Presents options to user
   - Handles input/output
3. **Business Logic**
   - Six core functions for CRUD operations
   - Each function is independent
   - Direct list manipulation
4. **Data Layer**
   - Simple Python list of dictionaries
   - No persistence (currently)

- In-memory only

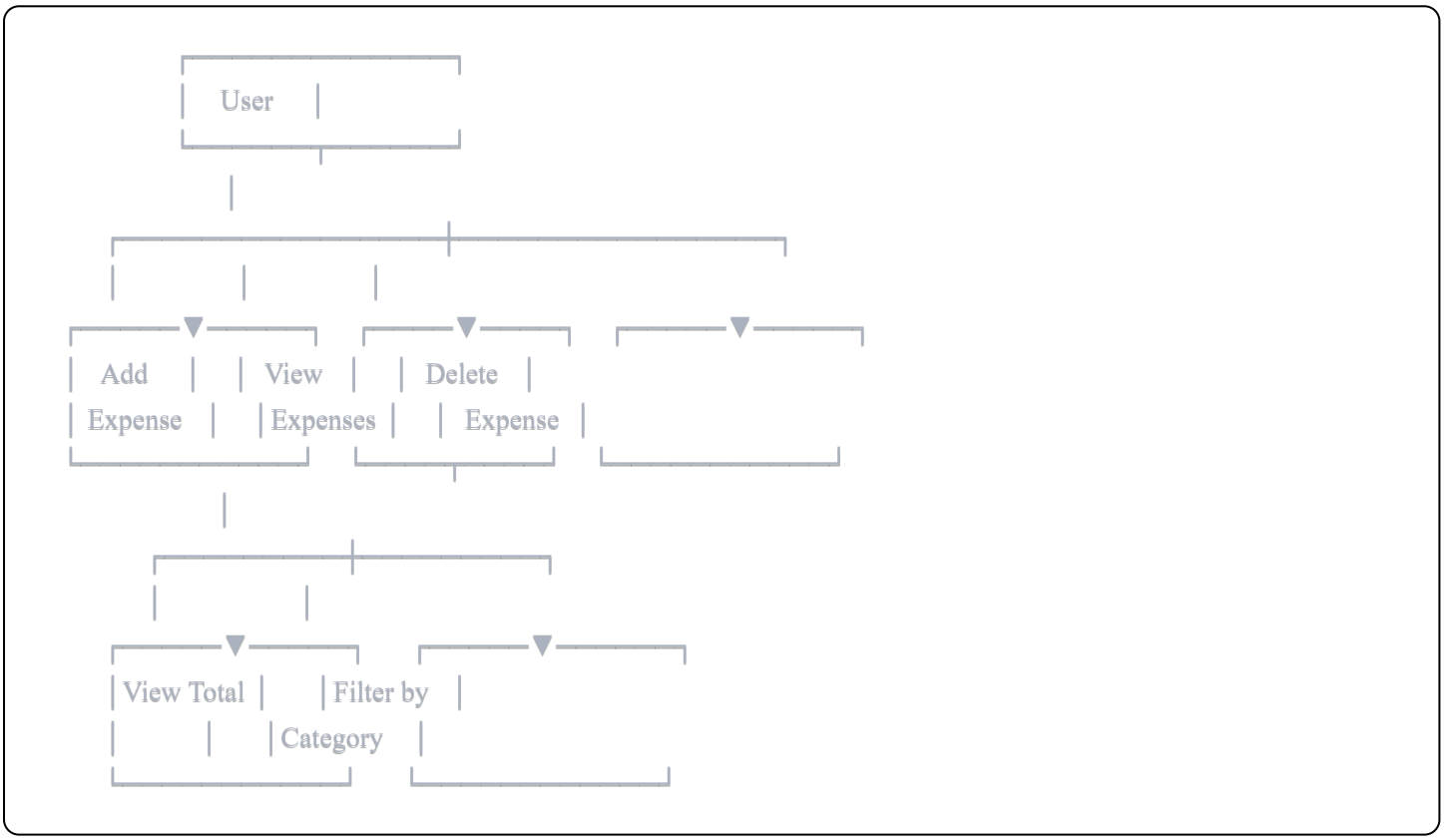
Why this architecture?

- Simple enough for a beginner project
- Easy to understand and modify
- No over-engineering
- Can be extended later (add file I/O, database, etc.)

---

7. Design Diagrams

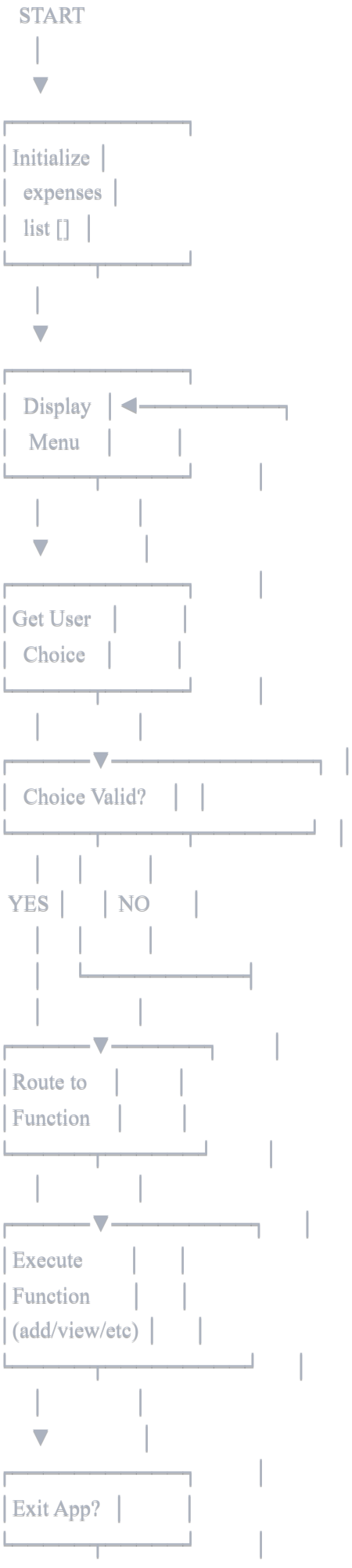
Use Case Diagram

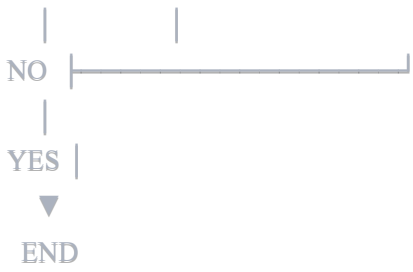


Use Cases:

1. Add Expense - User inputs expense details
2. View All Expenses - User sees complete list
3. View Total - User sees sum of all expenses
4. Filter by Category - User filters expenses
5. Delete Expense - User removes an expense
6. Exit - User closes the application

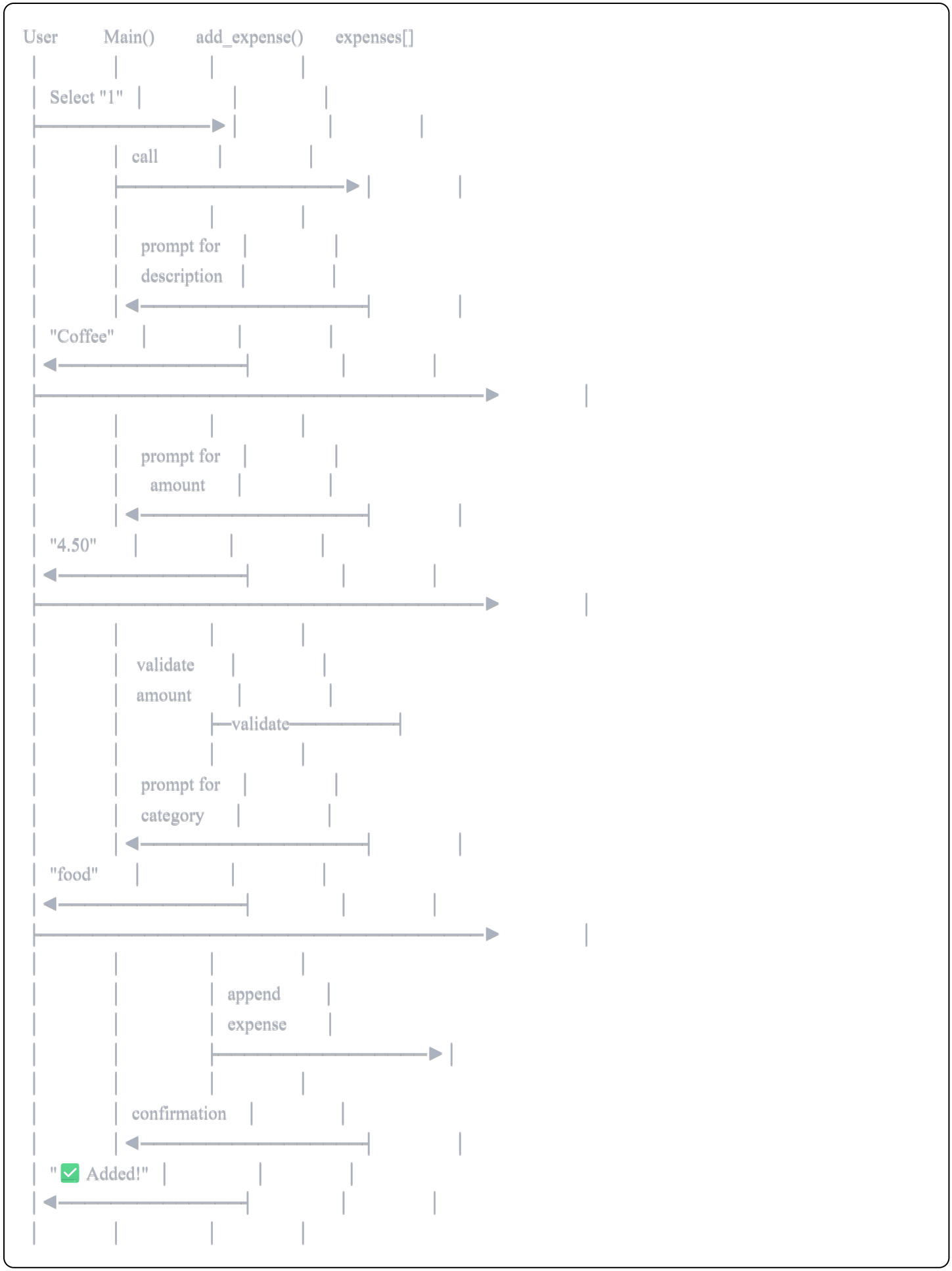
Workflow Diagram





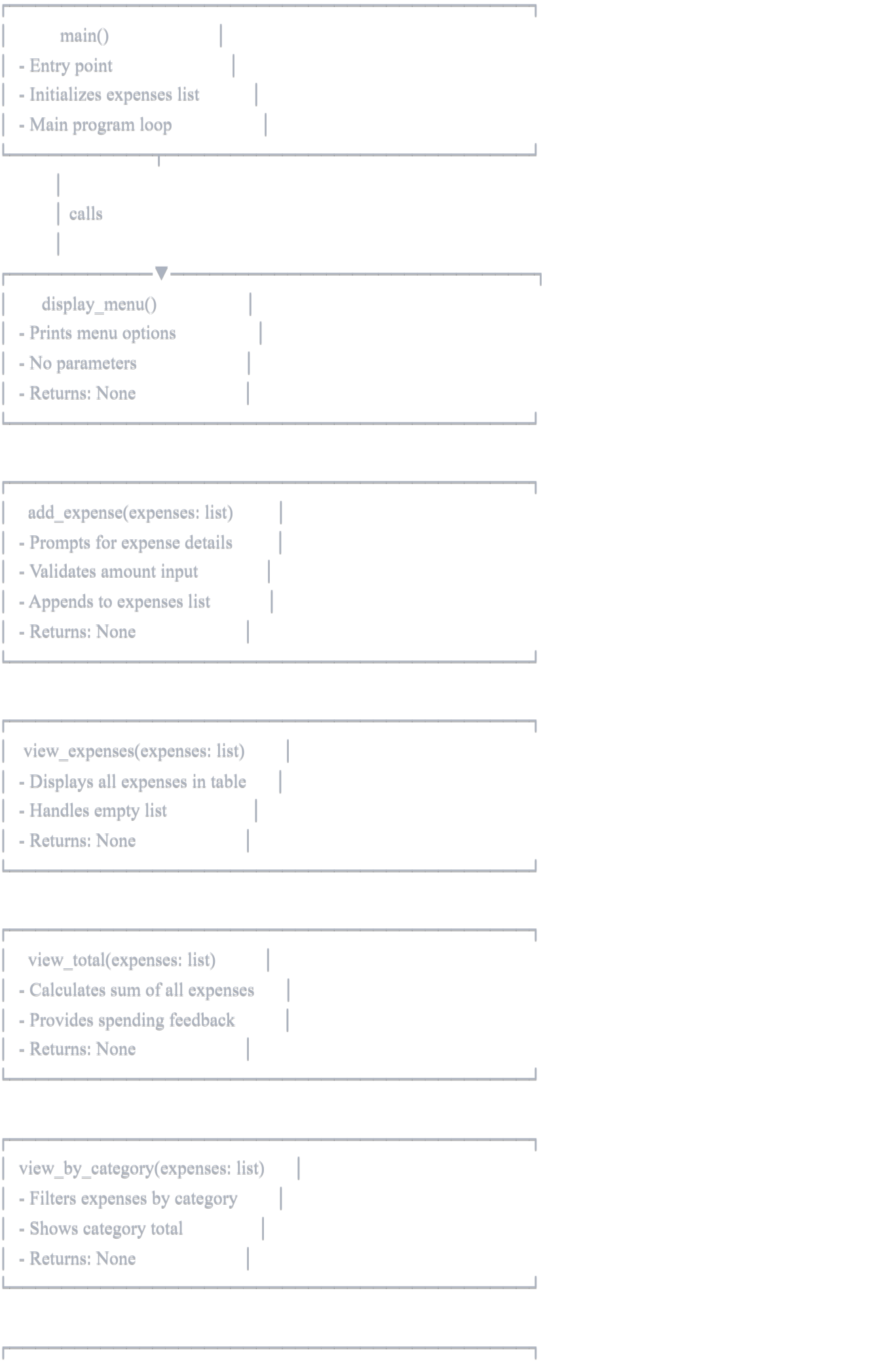
Sequence Diagram

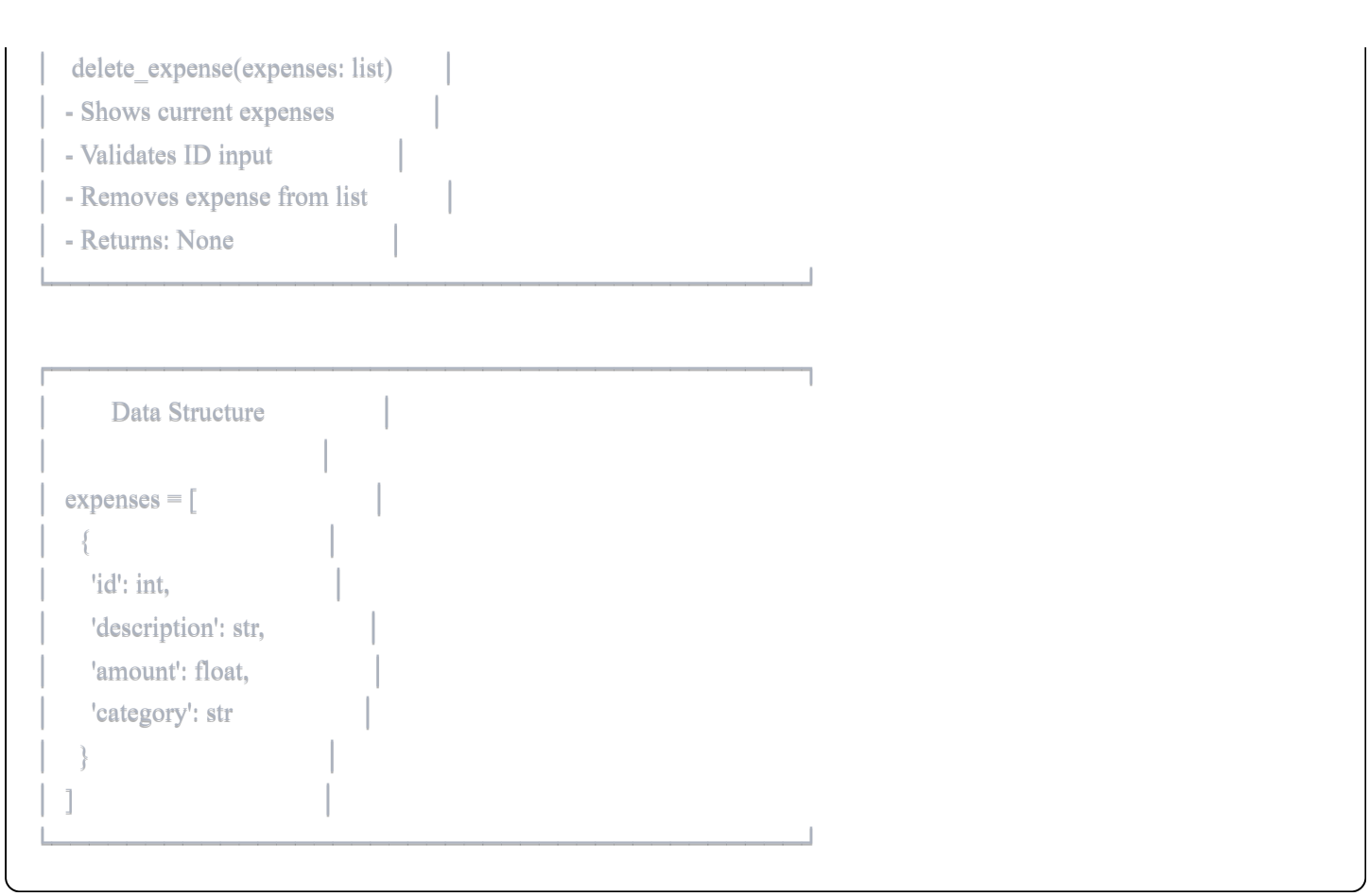
Scenario: Adding an Expense



Class/Component Diagram

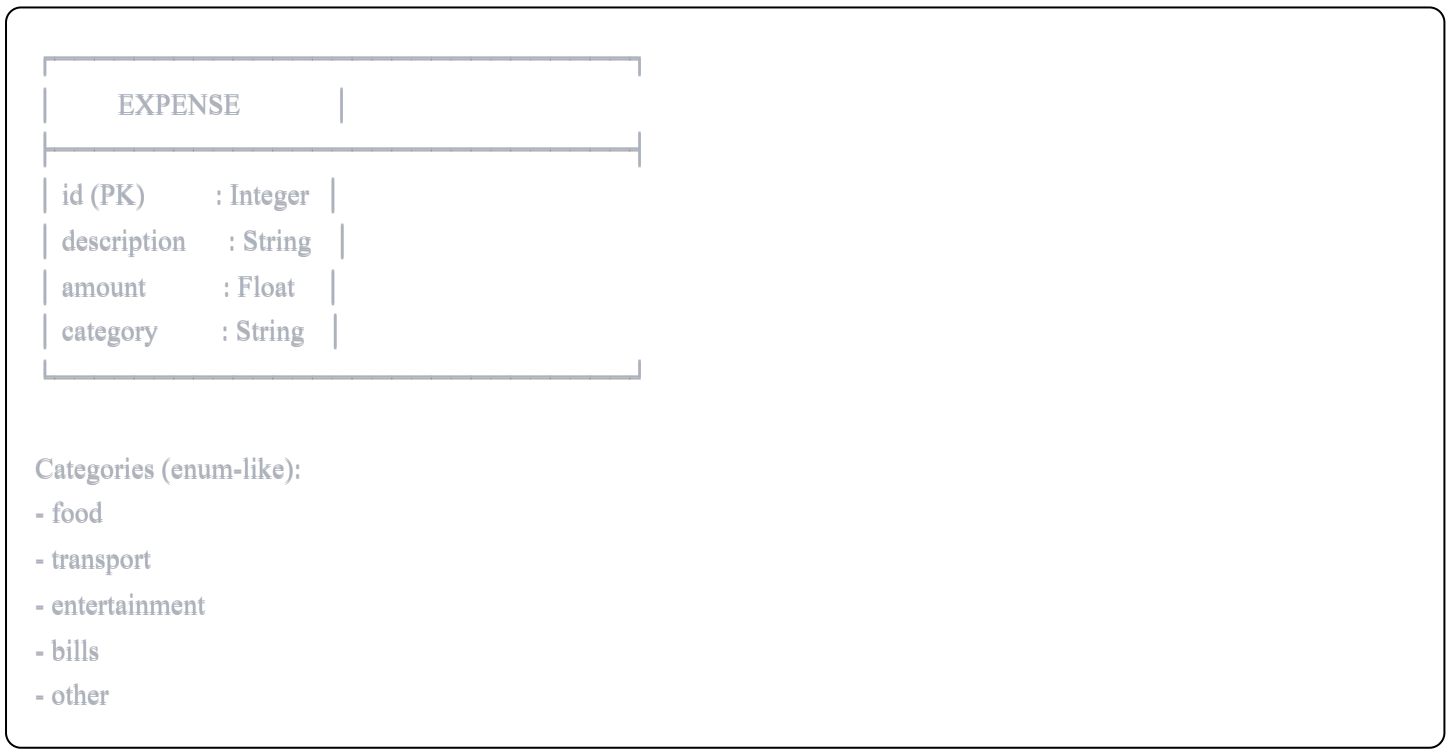
Since this is procedural Python, here's the component structure:


main()
- Entry point
- Initializes expenses list
- Main program loop
calls
display_menu()
- Prints menu options
- No parameters
- Returns: None
add_expense(expenses: list)
- Prompts for expense details
- Validates amount input
- Appends to expenses list
- Returns: None
view_expenses(expenses: list)
- Displays all expenses in table
- Handles empty list
- Returns: None
view_total(expenses: list)
- Calculates sum of all expenses
- Provides spending feedback
- Returns: None
view_by_category(expenses: list)
- Filters expenses by category
- Shows category total
- Returns: None



ER Diagram

Currently no database, but here's what the data model looks like:



Relationships:

- None currently (single entity)
- Future: Could add User entity for multi-user support

- Future: Could add Budget entity for budget tracking

---

8. Design Decisions & Rationale

Decision 1: No External Libraries

- **Why:** Keep it simple and portable
- **Trade-off:** Had to write custom validation
- **Worth it?** Yes - no installation hassles

Decision 2: In-Memory Storage

- **Why:** Simplifies the initial version
- **Trade-off:** Data doesn't persist
- **Worth it?** For learning, yes. Real use? Needs upgrade

Decision 3: Dictionary-Based Data Structure

- **Why:** Easy to understand, flexible
- **Alternative:** Could use classes
- **Rationale:** Dictionaries are more beginner-friendly

Decision 4: Manual Input Validation

- **Why:** Learn how validation actually works
- **Alternative:** Could use try-except with float()
- **Rationale:** Better control over error messages

Decision 5: Emoji-Enhanced UI

- **Why:** Makes it less boring
- **Trade-off:** Might not work in all terminals
- **Worth it?** Totally - personality matters

Decision 6: Procedural vs OOP

- **Why:** Simpler for a small project
- **Trade-off:** Less scalable
- **Rationale:** Can refactor to OOP later if needed

Decision 7: Linear Search for Deletion

- **Why:** Simple to implement
- **Trade-off:** O(n) complexity
- **Rationale:** Fine for expected data size

---

9. Implementation Details

Core Technologies

- **Language:** Python 3.x
- **Paradigm:** Procedural Programming
- **I/O:** Standard input/output
- **Data Storage:** In-memory list

Key Implementation Choices

Input Validation (Custom) Instead of using try-except, I manually validated numeric input:

```python
valid = True
dot_count = 0
for char in amount_str:
    if char == '.':
        dot_count += 1
    elif char < '0' or char > '9':
        valid = False
        break
```

This gives better control over error messages and teaches character iteration.

List Manipulation for Deletion Used manual list reconstruction instead of `list.remove()` to practice list operations:

```python
new_expenses = []
for j in range(len(expenses)):
    if j != i:
        new_expenses.append(expenses[j])
```

ID Management Simple auto-increment based on list length:

```python
```

```
'id': len(expenses) + 1
```

Works fine until you start deleting entries (future improvement).

Category System No enforcement - user can type anything. Suggests five categories but accepts all strings. Flexible but could lead to typos.

Code Organization

- Each menu option has its own function
- `main()` controls program flow
- Functions modify the expense list directly (pass by reference)
- No return values needed - functions do their work in place

---

10. Screenshots / Results

Starting the Application:

```
👋 Hey there! Welcome to your personal Expense Tracker!
💡 Let's keep track of where your money goes...

💰 ====== YOUR EXPENSE TRACKER ====== 💰
1. 💸 Add a new expense
2. 📋 See all my expenses
3. 🧮 Show me the total damage
4. 🔍 Filter by category
5. 🗑️  Delete an expense
6. 👋 Exit
========================================

🤔 What would you like to do? (1-6):
```

Adding an Expense:

```
💸 Let's add a new expense!
What did you spend on? Coffee
How much did it cost? $4.50
Category (food/transport/entertainment/bills/other): food

✅ Got it! Expense added (ID: 1)
💭 'Coffee' for $4.50 - every penny counts!
```

Viewing All Expenses:

```
📋 Here's what you've spent so far:
ID   What you bought        Cost       Category
-----------------------------------------------------
1    Coffee                 $4.50      food
2    Bus ticket             $2.50      transport
3    Netflix                $15.99     entertainment
```

Viewing Total:

```
🧮 Total Money Spent: $22.99
💚 Not bad! You're keeping it under control.
```

Filtering by Category:

```
🔍 Which category do you want to check? food

💳 Expenses for 'food':
ID   Description            Amount
----------------------------------------
1    Coffee                 $4.50

💰 Total for 'food': $4.50
```

Deleting an Expense:

```
🗑️ Which expense ID do you want to remove? (0 to go back): 2

✅ Deleted! 'Bus ticket' is gone. More money for you! 🎉
```

---

11. Testing Approach

Manual Testing Strategy

Test Case 1: Adding Valid Expense

- Input: Description, valid amount, category
- Expected: Expense added, confirmation shown
- Result: ✅ Pass

Test Case 2: Invalid Amount Input

- Input: "abc", "25.50.30", "-5"

- Expected: Error message, prompt to retry
- Result: ✅ Pass

Test Case 3: Empty Expense List

- Action: View expenses with empty list
- Expected: Friendly message, no crash
- Result: ✅ Pass

Test Case 4: Delete Non-existent ID

- Input: ID that doesn't exist
- Expected: Error message, prompt to retry
- Result: ✅ Pass

Test Case 5: Category Filtering - No Matches

- Input: Category with no expenses
- Expected: "Nothing found" message
- Result: ✅ Pass

Test Case 6: Large Numbers

- Input: 999999.99
- Expected: Accepted and displayed correctly
- Result: ✅ Pass

Test Case 7: Menu Invalid Input

- Input: 7, "a", special characters
- Expected: Error message, show menu again
- Result: ✅ Pass

Test Case 8: Multiple Expenses Workflow

- Action: Add 10+ expenses, view, filter, delete
- Expected: All operations work smoothly
- Result: ✅ Pass

What I Tested

- All menu options work

- Input validation catches bad data
- Empty states handled gracefully
- Calculations are accurate
- IDs are unique and tracked correctly
- Deletion doesn't break the list

What Could Use More Testing

- Really large datasets (100+ expenses)
- Unicode characters in descriptions
- Very long strings
- Edge cases with floating point math

---

12. Challenges Faced

Challenge 1: Input Validation Without Libraries

- **Problem:** Needed to validate numeric input manually
- **Why it was hard:** Had to think through all edge cases (decimals, negatives, multiple dots)
- **Solution:** Wrote character-by-character validation
- **Learning:** Appreciate how much libraries handle for you

Challenge 2: Deleting from List While Maintaining IDs

- **Problem:** After deleting, IDs don't match array indices
- **Solution:** Kept original IDs, searched by ID not index
- **Learning:** ID management is tricky - understand why databases use primary keys

Challenge 3: Formatting Output Tables

- **Problem:** Making columns align nicely
- **Solution:** Used string formatting with fixed widths (`:<25`)
- **Learning:** User experience matters, even in terminal apps

Challenge 4: List Modification During Iteration

- **Problem:** Can't modify list while looping through it
- **Solution:** Created new list, cleared old, copied back
- **Learning:** List manipulation has gotchas

Challenge 5: Keeping Functions Modular

- **Problem:** Functions getting too long, doing too much
- **Solution:** Broke down into smaller pieces, each with one job
- **Learning:** Clean code takes planning

Challenge 6: User-Friendly Error Messages

- **Problem:** Generic errors are confusing
- **Solution:** Wrote specific messages for each error type
- **Learning:** Good UX is about communication

---

13. Learnings & Key Takeaways

Technical Learnings

Python Fundamentals:

- Got really comfortable with lists and dictionaries
- Learned the difference between pass-by-value and pass-by-reference
- Practiced string manipulation and formatting
- Understood why validation matters

Code Organization:

- Breaking code into functions makes everything easier
- Each function should do ONE thing
- Good naming is half the documentation
- Comments help future-you understand past-you

User Experience:

- Even CLI apps need good UX
- Clear error messages save frustration
- Confirmation messages feel good
- A little personality goes a long way

Soft Skills

Problem Solving:

- Break big problems into smaller pieces
- Test as you go, not at the end
- Google is your friend (but understand what you copy)
- Sometimes the simple solution is the best solution

Project Management:

- Start with MVP (Minimum Viable Product)
- Add features incrementally
- Don't over-engineer early
- Know when "good enough" is actually good enough

Learning Process:

- Making mistakes is part of learning
- Debugging teaches you more than perfect code
- Reading your own code later shows how much you've grown
- Documentation helps, but doing helps more

Philosophical Takeaways

"Done is better than perfect" - I could've added a million features, but shipping v1 matters more.

"Keep it simple" - No need for a database, OOP, or fancy libraries for this project.

"User empathy" - Building something YOU would actually want to use makes it better.

"Fail fast" - Validate early, catch errors early, give feedback early.

---

14. Future Enhancements

Short-term (Easy Wins)

1. **File Persistence** - Save to JSON or CSV
2. **Date Tracking** - Add timestamps to expenses
3. **Edit Expense** - Modify existing entries
4. **Export Feature** - Generate expense reports
5. **Budget Warnings** - Set limits and get alerts

Medium-term (More Work)

6. **Recurring Expenses** - Track monthly bills

7. **Multiple Users** - Profile system
8. **Search Function** - Find expenses by description
9. **Income Tracking** - Not just expenses
10. **Visual Reports** - ASCII charts in terminal

Long-term (Major Features)

11. **Database Integration** - SQLite for real persistence
12. **Web Interface** - Flask/Django frontend
13. **Mobile App** - React Native version
14. **Cloud Sync** - Multi-device support
15. **AI Insights** - Spending pattern analysis

Technical Debt to Address

- Refactor to OOP for better scalability
- Add proper error handling with try-except
- Implement unit tests
- Fix ID system after deletion
- Add input sanitization
- Create config file for categories

---

15. References

Python Documentation:

- Python Lists: https://docs.python.org/3/tutorial/datastructures.html
- String Formatting: https://docs.python.org/3/library/string.html
- Input/Output: https://docs.python.org/3/tutorial/inputoutput.html

Learning Resources:

- Python basics and syntax fundamentals
- List manipulation techniques
- String validation methods
- CLI application design patterns

Inspiration:

- Personal need for simple expense tracking
- Various expense tracking apps (for features to include/avoid)
- Terminal-based app designs

Tools Used:

- Python 3.x
- Text editor / IDE
- Terminal / Command Prompt
- Manual testing

Conclusion

This was a fun first project! It taught me that building something useful doesn't require fancy tech - sometimes you just need Python, a good idea, and the willingness to figure things out as you go.

The app does what it's supposed to do: track expenses simply. Sure, there's room for improvement (lots of it), but that's the point. Version 1 is about learning and shipping, not perfection.

If I were to redo this, I'd probably use classes and add file saving from the start. But honestly? The simplicity taught me more than over-engineering would have.

Next steps: add persistence, maybe build a GUI version, and keep learning.

Final thought: The best project is the one you actually finish. ✅

Report compiled: November 2024
Project status: Complete and functional
Lines of code: ~200
Coffee consumed: Too much 🧉