9-1 Project: Scenario 1: Process Summary

Marie-Chantal Foster

Cybersecurity, Southern New Hampshire University

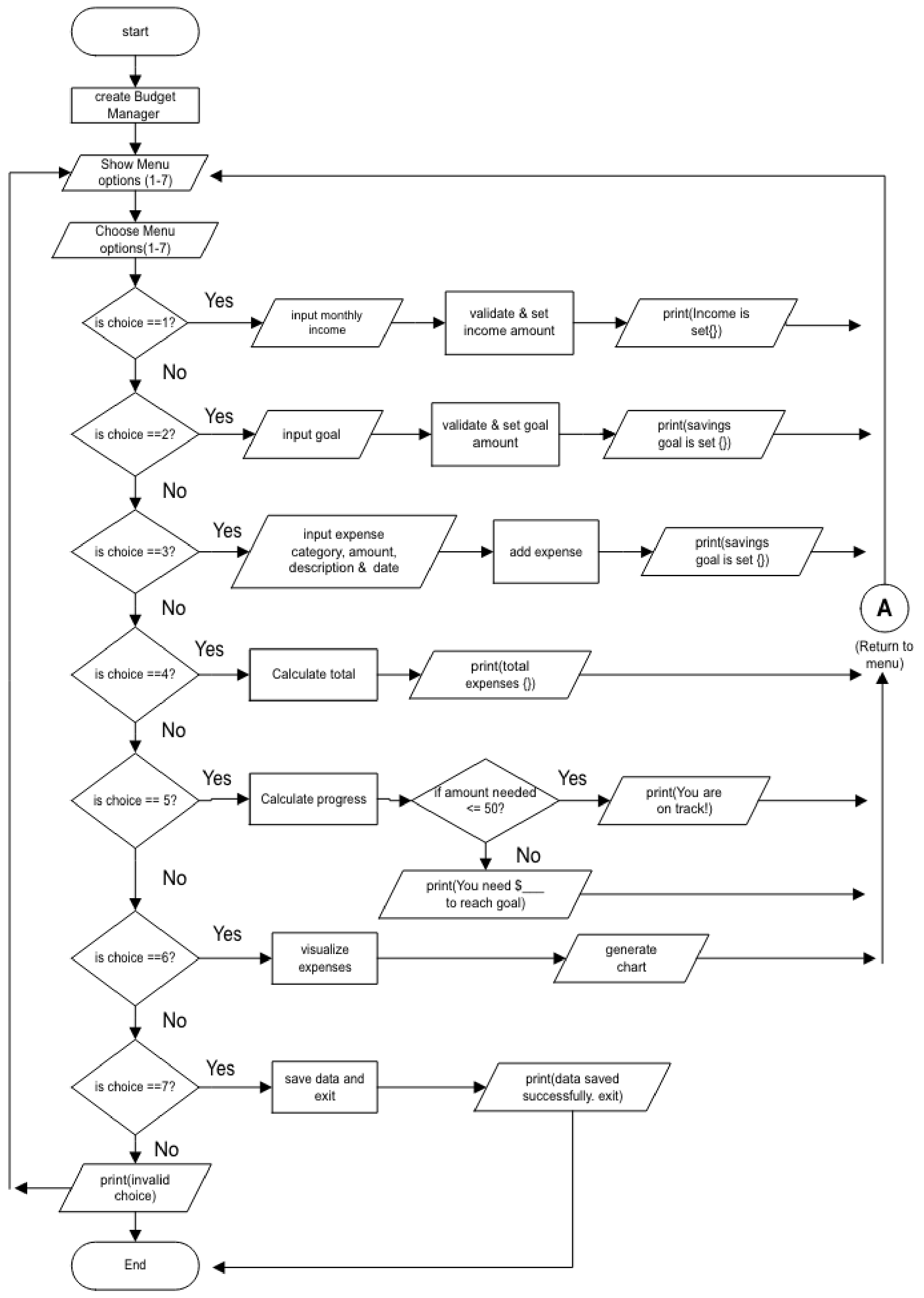
CS-500-10018-M01 Introduction to Programming

Dr. Shadha Tabatabai

January 18, 2026

Process Summary - Financial Planner

To apply my core programming skills, I chose to develop a financial application, which began with starter code in a program called Sense, which later morphed into an IDE to improve readability, functionality, and testing. The project focuses on building a user-friendly program to help individuals manage their income, track expenses, and work toward a savings goal. Along with the program, I developed a flowchart (diagram shown) as a pre-coding tool to visualize the logic and



structure of the Financial Planner program. This serves as a blueprint for the program. By developing a fully functional financial planning application using OOP principles, secure input handling, modular design, and automated testing, I created a professional-grade program that mirrors real-world software development practices and deepened my understanding of building reliable, scalable systems.

The application of Object-Oriented Programming (OOP) Principles played a central role in the structure and functionality of my application, and it is widely used to develop budget planning applications because it creates a modular, scalable, and maintainable codebase. For the application to begin, three *Classes* had to be created to set the blueprint for creating objects: `Expense,` `BudgetCategory,` and `BudgetManager.` They determine the characteristics and behaviors of the objects that communicate by calling each other's method (Ipayeislamiat, 2024). *Encapsulation* is implemented in the Expense class through input validation inside the `__init__` method, which prevents invalid objects from being created by rejecting negative amounts or empty descriptions. Storing data in attributes such as `self.amount`, `self.date,` and `self.description`, each object protects its own internal state and ensures only valid, safe data enters the system. This strengthens encapsulation by requiring *Objects* to protect their internal data through their own methods. The methods `add_expense()` or `total_expenses()` serve as a contract; as long as the user follows the interface, they are insulated from changes to the underlying logic. This defines *Abstractions* focusing on what an object does rather than how it does it. To complement these principles, *Inheritance* was used to extend the system's behavior without modifying the original classes. I created a `TestableBudgetManager` subclass that overrides `load_data()`, allowing the test suite to run independently of external files and demonstrating how inheritance supports extensibility for testability. *Polymorphism* enables the system to treat a `TestableBudgetManager` as if it were a standard `BudgetManager`. Even though the code calls the same method, `load_data()`, the system dynamically selects which version to execute based on the object's actual type at runtime.

I organized the project into separate modules using Code Structure for Reusability and Maintainability. I applied the Single Responsibility Principle to each module, ensuring each module had a clear, focused responsibility (Rodriguez, 2021). For example, *financial.py* contains the domain logic (Expense, BudgetCategory, BudgetManager, and the main program), while *storage.py* handles JSON persistence and file handling. Then *test/test_financial.py* handles the automated unit tests. The structure promotes reusability across the Classes for future features such as a GUI, an API, or a reporting system, and *loose coupling*, which allows the modules to have minimal dependencies on each other's internal working. Maintainability will help if data storage requirements change; only *storage.py* needs to be updated. As for the Separation of concerns, each file handles a single major responsibility, reducing complexity and improving readability. This modular approach was guided by the tutor on how to mirror professional software engineering practices and ensure that an application can grow without becoming difficult to manage.

Throughout the project, I followed Python's recommended Coding Standards and Readability Principles, including PEP 8 naming conventions for function names (`add_expense`) and class names (`BudgetManager`). 2. To improve readability, I used clear, descriptive variable names. 3. Comments where clarification was needed. 4. Consistent indentations by using four spaces with no tabs and spacing. Avoided extraneous whitespace and maximum line length, such as 79 characters. 5. When I import, I make sure the order is in the standard library, third-party, and local application format. 6. Whether I use single or double string quotes, it remains consistent throughout the project. 7. Logical grouping of methods within classes, such as clear grouping: created Expense to store the calculator totals. 8. Short focus functions to avoid unnecessary complexity by having one line and one responsibility, such as on one line: `def total_expense(self) -> float:`, then next line with four indentations: `return sum(e.amount for e in self.expense)`, for easy reading. 9. Refactoring the JSON persistence logic into storage.py improved readability and maintainability by removing file-handling clutter from the main business logic.

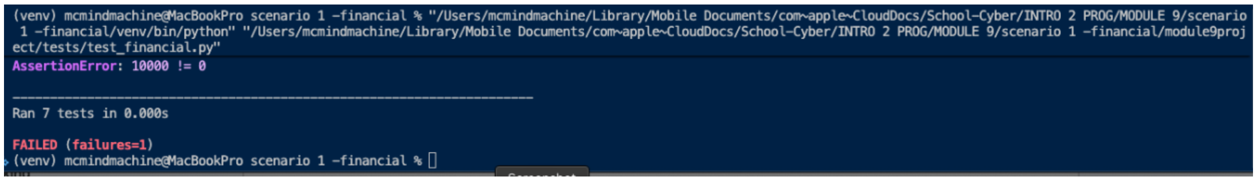
Several Python built-in Libraries and External Packages were used to enhance functionality and reduce complexity. *JSON* (built-in) is used to save and load financial record data because it is readable and does not require a complex database. *Unit tests* (built-in) act as a regression-prevention safety net, allowing me to refactor or add new features without unintentionally breaking existing behavior. The tutor mentioned that it ensures future enhancements, such as adding a tax calculator, can integrate cleanly with the current system. *Datetime* (built-in) for validating date formats, which sorts and tracks by time, and *matplotlib* (external) were used in a virtual environment (venv). The tutor suggested it would isolate dependencies and prevent version conflicts with other projects. This setup ensures a controlled, reproducible environment for reliable testing with the `TestableBudgetManager` and keeps the global Python installation clean, making the program more efficient and professional.

Secure Coding Practices are best practices to help prevent common vulnerabilities and ensure the application behaves safely even when unexpected input is entered. I implemented input validation and type checking to address buffer and number overflows (Hughes, 2025). For example, in my Expense class, it checks `if amount < 0` before saving the data, then it validates that the input provided to the class is safe to use. This prevents "logical overflows" where a negative expense could accidentally increase a user's balance. To improve modularity and reduce redundancy, JSON persistence was moved into a dedicated `JSONStorage` helper class. The `BudgetManager` now delegates saving through simplified wrapper methods (`save_data()`), which call `JSONStorage.save()`. This will make my code easier to maintain, test, and extend.

Error-handling mechanisms and Program Reliability features were implemented using *try/except* blocks for file operations. *ValueError* exceptions for invalid expenses. Graceful fallback behavior when no data file exists, and user-friendly error messages during input validation, such as "Invalid input. Please enter a positive number or an Invalid date format. Please enter date in YYYY-MM-DD." These constructs ensure the program does not crash due to invalid input or missing files. Instead, it guides the user to resolve the issue to improve overall stability and user experience, rather than terminating unexpectedly. Giving my program a controlled flow is part of its robustness strategy.

My testing methodology involved creating a comprehensive suite of unit tests using Python's unittest library to validate the behavior of the *Core class* (e.g., BudgetCategory). I focused on *Edge cases*: Rejecting invalid negative expenses. *Integration tests*: Verify that multiple parts of my system work together correctly, e.g., income tracking and saving goal calculation. *Automatic category creation*: Verifies through testing, ensuring that adding an expense to a nonexistent category correctly generates a new `BudgetCategory` object. *Validation of invalid data:* Tests confirm the logic; Expense validation inside the `Expense` class, and a `TestableBudgetManager` class to isolate tests from file I/O, avoiding loading JSON files during tests (Silveira, 2025). These rigorous tests confirmed that all programs' functionality met the specified requirements. Screenshots of some of the tests that were conducted:

1) *Integration testing* triggered an `AssertionError` when expected and actual values didn't match. It revealed a flaw in how components interacted, helping catch logic issues before deployment.

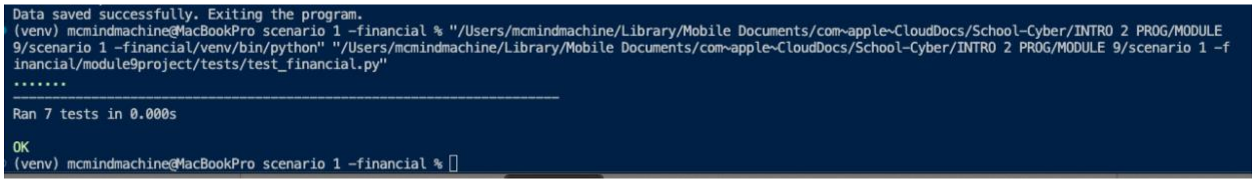


```
(venv) mcmindmachine@MacBookPro scenario 1 -financial % "/Users/mcmindmachine/Library/Mobile Documents/com~apple~CloudDocs/School-Cyber/INTRO 2 PROG/MODULE 9/scenario
 1 -financial/venv/bin/python" "/Users/mcmindmachine/Library/Mobile Documents/com~apple~CloudDocs/School-Cyber/INTRO 2 PROG/MODULE 9/scenario 1 -financial/module9proj
ect/tests/test_financial.py"
AssertionError: 10000 != 0

----------------------------------------------------------------------
Ran 7 tests in 0.000s

FAILED (failures=1)
(venv) mcmindmachine@MacBookPro scenario 1 -financial %
```

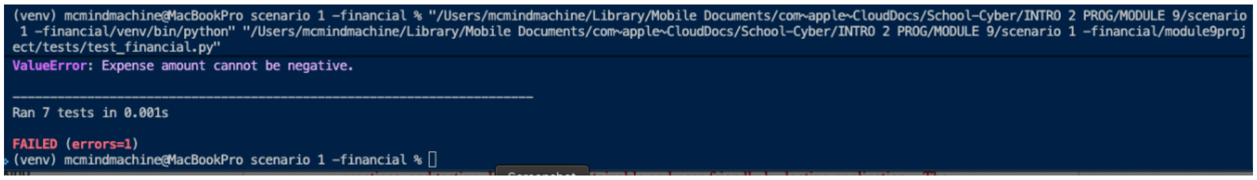
2) *Unit testing and Validation of requirements*. Confirms all tests passed, meets the functional and technical requirements, and that the `TestableBudgetManager` behaves correctly, ensuring the integration logic worked as intended.



```
Data saved successfully. Exiting the program.
(venv) mcmindmachine@MacBookPro scenario 1 -financial % "/Users/mcmindmachine/Library/Mobile Documents/com~apple~CloudDocs/School-Cyber/INTRO 2 PROG/MODULE
9/scenario 1 -financial/venv/bin/python" "/Users/mcmindmachine/Library/Mobile Documents/com~apple~CloudDocs/School-Cyber/INTRO 2 PROG/MODULE 9/scenario 1 -f
inancial/module9project/tests/test_financial.py"
.......
----------------------------------------------------------------------
Ran 7 tests in 0.000s

OK
(venv) mcmindmachine@MacBookPro scenario 1 -financial %
```

3) *Edge cases and invalid data testing* ensure the system correctly raises `ValueError` for negative expenses. This validated my secure coding practices and resolved any logic issues before deployment.



```
(venv) mcmindmachine@MacBookPro scenario 1 -financial % "/Users/mcmindmachine/Library/Mobile Documents/com~apple~CloudDocs/School-Cyber/INTRO 2 PROG/MODULE 9/scenario
 1 -financial/venv/bin/python" "/Users/mcmindmachine/Library/Mobile Documents/com~apple~CloudDocs/School-Cyber/INTRO 2 PROG/MODULE 9/scenario 1 -financial/module9proj
ect/tests/test_financial.py"
ValueError: Expense amount cannot be negative.

----------------------------------------------------------------------
Ran 7 tests in 0.001s

FAILED (errors=1)
(venv) mcmindmachine@MacBookPro scenario 1 -financial %
```

In conclusion, the project allowed me to apply core programming skills in a realistic, professional context. By using object-oriented design, modular architecture, secure coding practices, and testing, I created a maintainable and user-friendly budgeting application. The experience deepened my understanding of core software engineering principles and helped build a strong foundation for working in a professional environment.

References

Hughes, C. (2025, January 3). Secure by design vs by default-which software development concept is better? Csonline. https://www.csoonline.com/article/3631188/secure-by-design-vs-by-default-which-software-development-concept-is-better.html

Ipayeislamiat. (2024, January 78). *Budget Tracker Program-(OOP)*. Medium. https://medium.com/@ipayeislamiat/budget-tracker-program-oop-ffd9223a620b

Rodriguez, J. (2021, April 13). *Single Responsibility Principle*. Dev. https://dev.to/josuerodriguez98/single-responsibility-principle-5h09

Silveira, O. S. (2025, April 8). *Unit tests in Python: A beginner's guide*. Dataquest. https://www.dataquest.io/blog/unit-tests-python/

Zybooks. (n.d.). *Participation Activity- Modules 1-8*. https://learn.snhu.edu/d2l/le/content/2108542/viewContent/45814708/View