Project Report: Python CLI To-Do List Manager

Vinayak Vikram Singh(25BAI10262)

November 23, 2025

Contents

Chapter 1

Introduction

The Simple Python CLI To-Do List is a lightweight, command-line interface application designed to solve the need for quick, accessible, and persistent task management. This project serves as an ideal demonstration of core Python capabilities, including procedural programming, functions, loops, and file input/output (I/O) using the standard library. The core functionality is based on CRUD (Create, Read, Update, Delete) operations, ensuring task data is reliably stored in a plain text file between sessions.

1.1 Problem Statement

Individuals need a lightweight, accessible, and persistent tool to track personal or professional tasks without relying on complex, resource-heavy software or continuous internet connectivity. A simple, file-based command-line interface application is needed to offer core task management functionality quickly and efficiently using standard, portable file types.

1.2 Scope of the Project

The project is strictly limited to a single-user, local, Command Line Interface (CLI) application. Key functionalities include task addition, viewing, marking tasks as done, task deletion, and local persistence via a plain text file (`todo_list.txt`). Complex features such as networking, graphical interfaces, scheduling, or encryption are explicitly outside the scope.

Chapter 2

Requirements Analysis

2.1 Functional Requirements (FR)

The following table outlines the specific behaviors the system must exhibit.

Table 2.1: Functional Requirements

ID	Requirement	Description
FR-01	Task Addition	The user must be able to input a new task description via the CLI, which is then appended to the list with a `[ ]` (To Do) status.
FR-02	Task Viewing	The system must display the entire list of tasks, including their 1-based index number and current status (To Do or Done).
FR-03	Mark Task Done	The user must be able to select a task by its index number and change its status from `[ ]` to `[x]` (Done).
FR-04	Task Deletion	The user must be able to select a task by its index number and permanently remove it from the list.
FR-05	File Persistence	The system must automatically save the current list state to a local file (`todo_list.txt`) after any modification.
FR-06	File Loading	The system must load tasks from `todo_list.txt` upon startup. If the file does not exist, it must start with an empty list.
FR-07	Exit	The system must provide an explicit menu option for exiting the application gracefully.

2.2 Non-functional Requirements (NFR)

These requirements define the quality attributes of the system.

Table 2.2: Non-functional Requirements

ID	Requirement	Description
NFR-01	Performance	Task loading and saving operations must complete within 1 second for lists up to 1000 tasks.
NFR-02	Reliability	The system must gracefully handle file not found errors (`FileNotFoundError`) without crashing.
NFR-03	Security	Low requirement; tasks are stored in a non-encrypted plain text file, suitable for non-sensitive data.
NFR-04	Maintainability	The code must be modular, divided into separate functions for file operations and core task logic.
NFR-05	Compatibility	Must run on any operating system (Windows, macOS, Linux) with Python 3.x installed.

Chapter 3

System Design

3.1 System Architecture

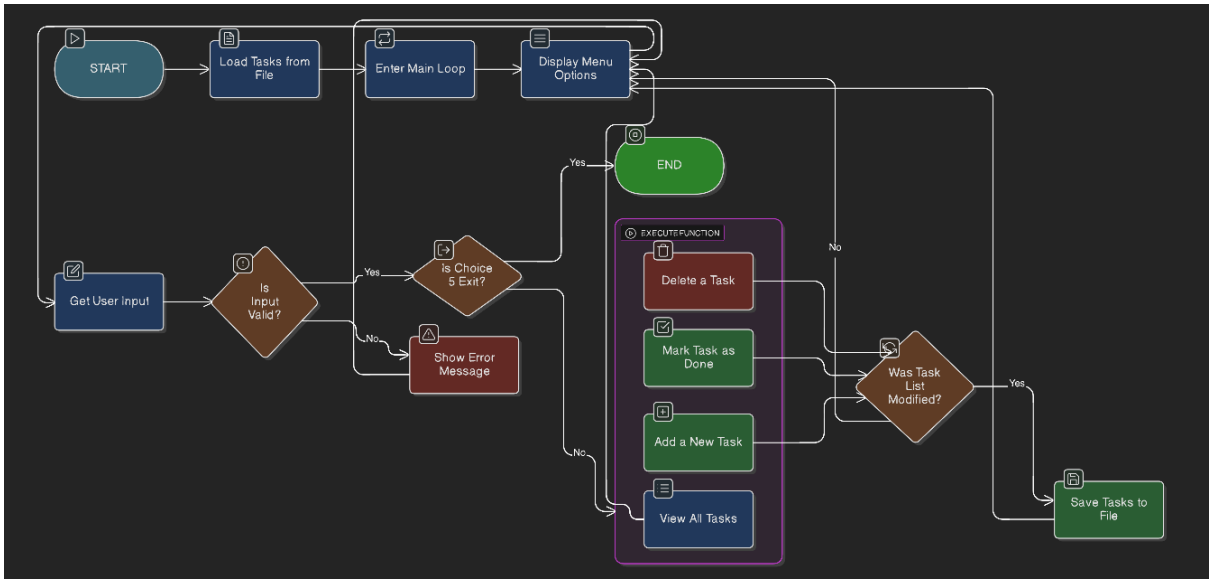
The system architecture is a two-tier structure centered on the Python script.

Architecture Description

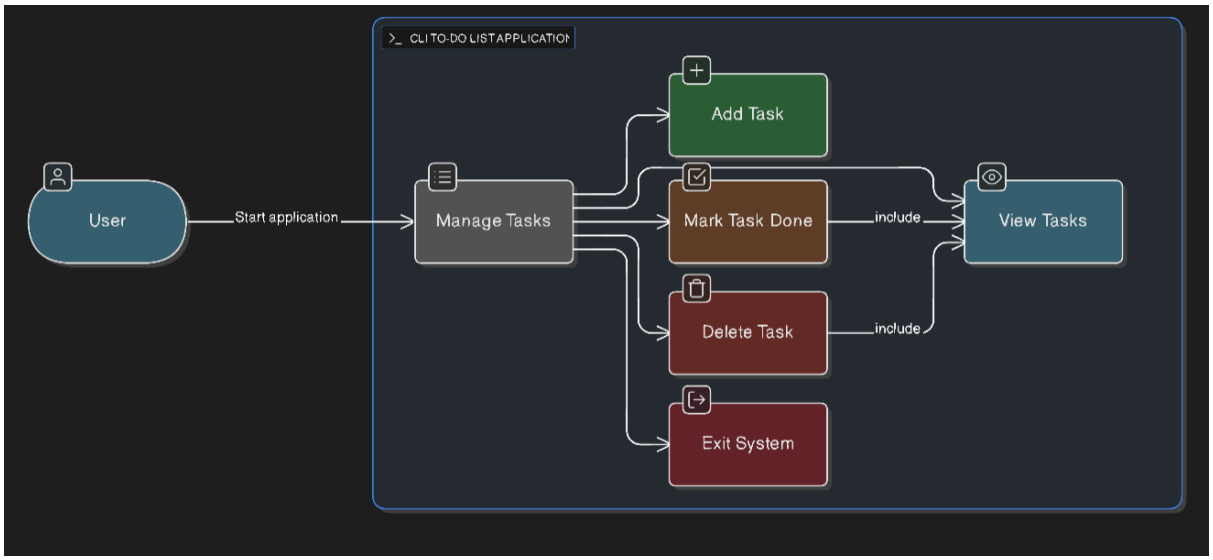
The user interacts with the **Python Script** via the **Terminal** (CLI). The script uses the `tasks` list in **Memory** for runtime operations and relies on Python's native file I/O to read and write data to the **Plain Text File** (`todo_list.txt`) for persistence. There are no external services or network dependencies.

3.2 Design Diagrams

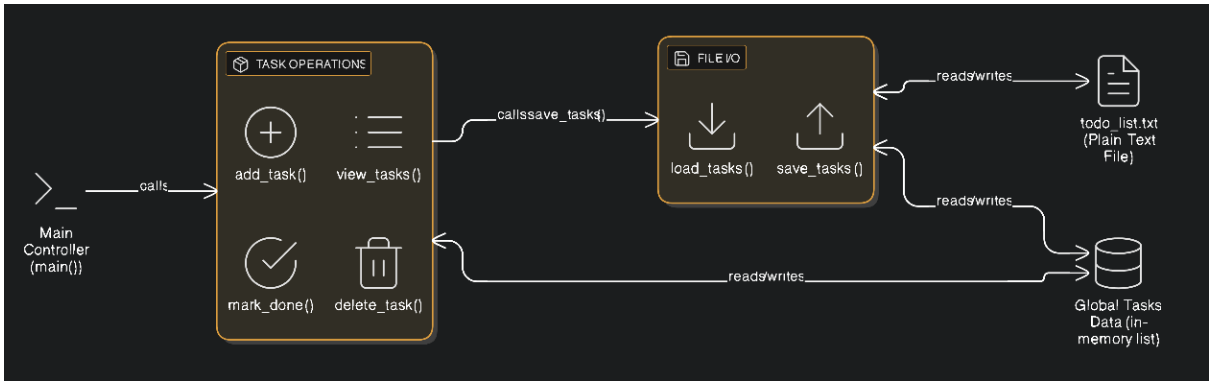
Process Flow (Workflow) Diagram



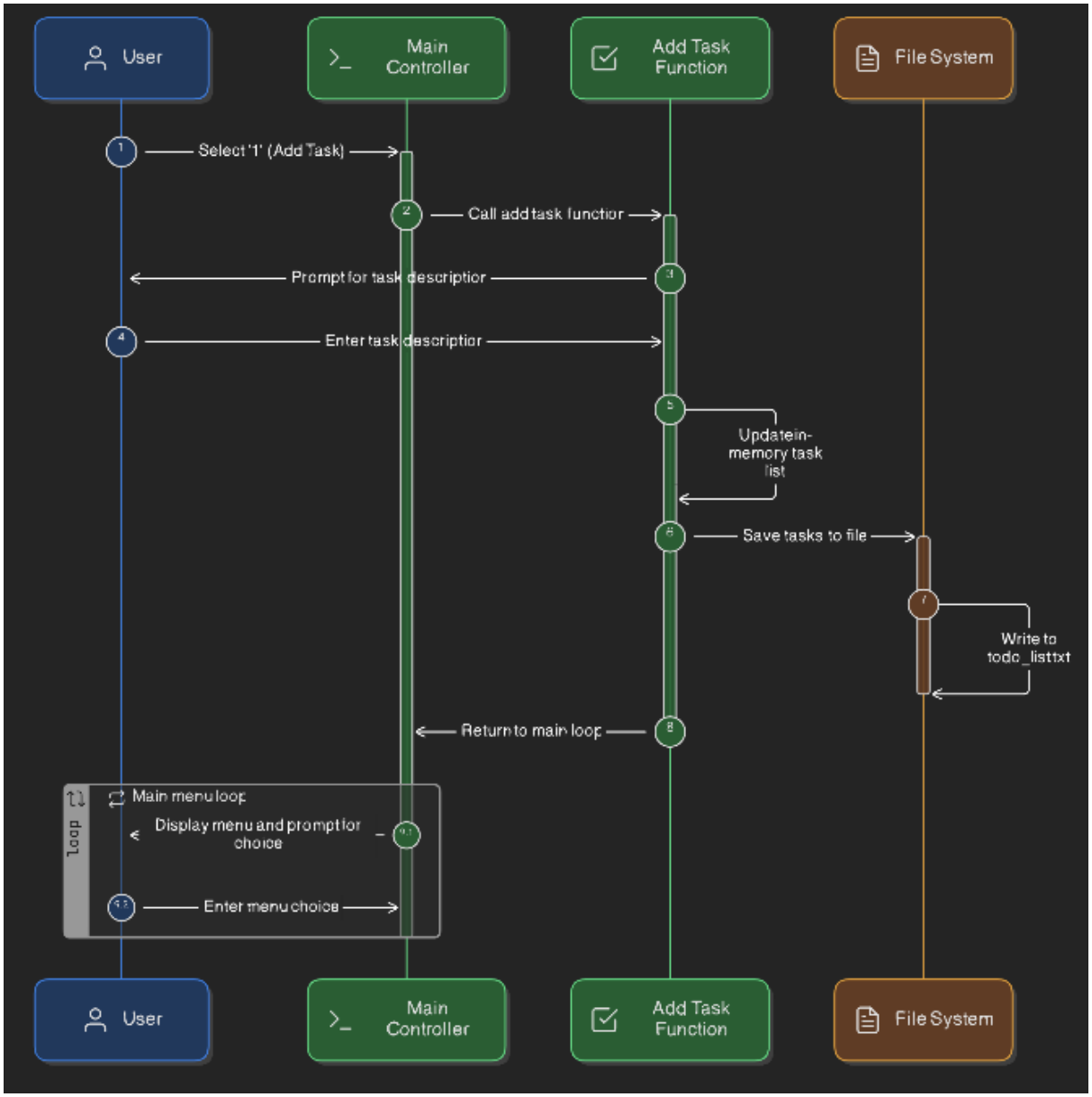
Use Case Diagram



Component Diagram



Sequence Diagram Prompt (Adding a New Task)



Chapter 4

Design Decisions & Implementation

4.1 Design Decisions & Rationale

- **Data Format (Plain Text):** A plain text file ('.txt') was chosen for simplicity, zero dependency, and human readability. This fulfills the NFR-03 (Security) as no sensitive data requires encryption, and NFR-04 (Maintainability).
- **Status Embedded in String:** The task status (`[ ]` or `[x]`) is stored directly within the task string (e.g., `[ ] Buy milk`). This avoids the complexity of parsing structured data (like JSON or CSV) and maintains the simplicity of the single-string-per-task model.
- **Input Handling (Inside Function):** The logic to prompt for user input (e.g., task description or index) is handled *inside* the relevant function (`add_task`, `mark_done`). This keeps the `main()` loop clean, focused only on menu routing, and prevents argument mismatch errors.
- **File Mode (`'w'`):** The `save_tasks` function uses write mode ("w") to overwrite the entire file with the current state of the `tasks` list. This is simpler and less error-prone than appending or finding and replacing specific lines, which would require more complex file manipulation logic.

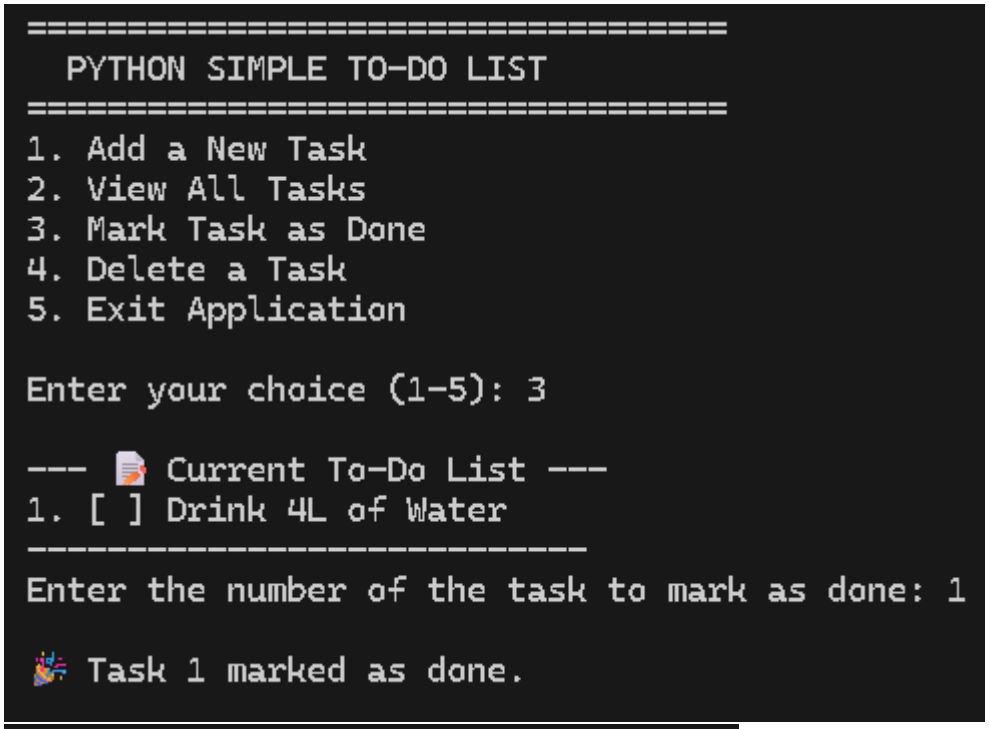
4.2 Implementation Details

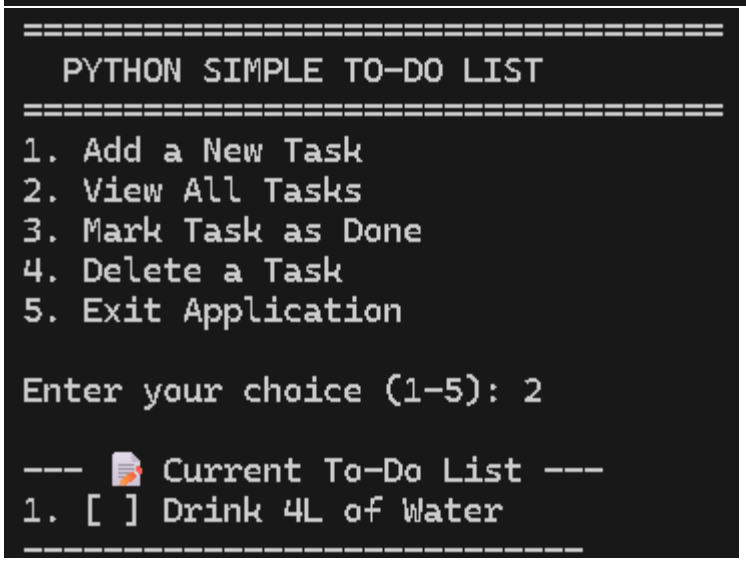
The application is implemented in a single Python file, `todo_cli.py`.

- **Global List:** A global list named `tasks` is used to hold the current in-memory state. Functions that modify this list must declare `global tasks`.
- **1-Based Indexing:** All user-facing interaction (displaying and requesting indices) uses 1-based numbering (`i + 1` in `view_tasks`). Internally, this is converted to Python's 0-based indexing (`int(index) - 1`) for list operations.
- **Error Handling:** `try-except ValueError` blocks are used to validate that the user input for the task index is indeed a number. Boundary checks (`0 <= task_index < len(tasks)`) are used to prevent `IndexError` if the user enters a number outside the list range.
- **Persistence Logic:** The `load_tasks` function handles the initial `FileNotFoundError`, ensuring the program never crashes on its first run.

4.3 Screenshots / Results

1. **Startup and Task Addition:** Terminal view showing the main menu, user selection of option '1', input of the task description, and the confirmation message indicating successful task addition.
2. **Task Viewing:** Terminal view showing the complete list of tasks, including their index number and status marker (`[ ]`).
3. **Marking Done:** Terminal view showing the list, user selection of option '3' and the task index, followed by a confirmation message and the list update (task status changed to `[x]`).
4. **Task Deletion:** Terminal view showing the list, user selection of option '4' and the task index, followed by a confirmation message that the task was permanently removed.


==========
PYTHON SIMPLE TO-DO LIST
==========
1. Add a New Task
2. View All Tasks
3. Mark Task as Done
4. Delete a Task
5. Exit Application
Enter your choice (1-5): 3
--- Current To-Do List ---
1. [ ] Drink 4L of Water
Enter the number of the task to mark as done: 1
Task 1 marked as done.


==========
PYTHON SIMPLE TO-DO LIST
==========
1. Add a New Task
2. View All Tasks
3. Mark Task as Done
4. Delete a Task
5. Exit Application
Enter your choice (1-5): 2
--- Current To-Do List ---
1. [ ] Drink 4L of Water

```
==================================
  PYTHON SIMPLE TO-DO LIST
==================================
1. Add a New Task
2. View All Tasks
3. Mark Task as Done
4. Delete a Task
5. Exit Application

Enter your choice (1-5): 1
Enter the task description: Drink 4L of Water

✅ Task added: 'Drink 4L of Water'
```

```
==================================
  PYTHON SIMPLE TO-DO LIST
==================================
1. Add a New Task
2. View All Tasks
3. Mark Task as Done
4. Delete a Task
5. Exit Application

Enter your choice (1-5):
```

```
==================================
  PYTHON SIMPLE TO-DO LIST
==================================
1. Add a New Task
2. View All Tasks
3. Mark Task as Done
4. Delete a Task
5. Exit Application

Enter your choice (1-5): 4

--- 📝 Current To-Do List ---
1. [x] Drink 4L of Water
----------------------------
Enter the number of the task to delete: 1

🗑️Task deleted: [x] Drink 4L of Water
```

Chapter 5

Quality Assurance

5.1 Testing Approach

The testing was conducted manually via the command line, focusing on edge cases and core functional requirements.

- **Unit Testing (Manual):** Each function (`add_task`, `mark_done`, `delete_task`, `load_tasks`, `save_tasks`) was tested in isolation by directly calling it from the interpreter to verify its effect on the global `tasks` list.
- **Integration Testing:** The main application loop (`main()`) was tested to ensure the menu routing correctly calls the corresponding functions.
- **Edge Case Testing:**
  - Testing for an empty list (verifying `view_tasks` handles this gracefully).
  - Inputting non-numeric characters for index selection (`ValueError` handling).
  - Inputting index numbers outside the list bounds (`IndexError` prevention).
  - Exiting and restarting the program to confirm persistence (loading and saving).

5.2 Challenges Faced

The primary challenges encountered involved managing the flow of data between the function calls and the global state.

- **Argument Mismatch:** Initially, the `main()` loop called functions without arguments, while the functions were defined to expect them (e.g., `add_task()`). This was resolved by modifying the function definitions to handle input internally and removing the required arguments.
- **Global State Management:** Ensuring that every modifying function correctly declared `global tasks` and called `save_tasks()` was crucial to maintaining data integrity and persistence.

5.3 Learnings & Key Takeaways

- **Procedural Design:** Gained experience designing a program using a clear procedural hierarchy, where the `main()` function acts as the central router.

- **File I/O Robustness:** Learned best practices for file handling in Python, particularly using `try...except FileNotFoundError` for graceful startup and the simplicity of the `'w'` mode for complete state saving.
- **User Experience in CLI:** Understood the importance of providing clear, 1-based indexing for the user while managing the complexity of 0-based indexing internally.

Chapter 6

Conclusion

6.1 Future Enhancements

While the core objectives were met, the following features could be added in future iterations:

- **Task Editing:** A new menu option to allow users to modify the text of an existing task without deleting and recreating it.
- **Priority Levels:** Implementing a system to assign and sort tasks by priority (e.g., High, Medium, Low).
- **Filtering:** Options to view only ”Done” tasks or only ”To Do” tasks.
- **Date/Time Tracking:** Adding timestamps for task creation and completion.

6.2 References

- Book: Computer Science with Python by Sumita Arora
- vityarthi.com
- Source code available at: https://github.com/mauve-nayak/ToDoList-Manager