VIT®

BHOPAL

NAME: Ananya Kumar

REGISTRATION NO.: 25BOE10055

PROJECT NAME: To Do List Application

To-Do List Application Project Report

1. Introduction

The **To-Do List Application** is a foundational project designed to address the basic need for task organization and management. In today's fast-paced environment, having a simple, reliable system to track pending actions is crucial for personal productivity and goal attainment. This project focuses on developing a highly minimalist and efficient solution using the **Command Line Interface (CLI)** paradigm.

The application serves as a demonstration of core programming concepts, including **data persistence using file I/O**, fundamental data structure manipulation (lists), and basic user interaction loops. It provides a simple, dependency-free utility built entirely in Python, emphasizing functionality over graphical complexity.

---

2. Project Goals and Objectives

The primary goal of this project is to create a functional, persistent, single-user To-Do List application.

Key Objectives:

- **Implement Task Persistence:** To successfully save and retrieve task data from a local file (`tasks.txt`) to ensure that data is retained between application sessions.
- **Develop Core CRUD Operations:** To enable users to create (Add), read (View), and **D**elete tasks reliably.
- **Ensure Usability:** To provide a clear, numbered, and easily navigable text-based menu for a smooth command-line experience.
- **Maintain Simplicity:** To adhere to a low-complexity design using only standard Python libraries, prioritizing rapid development and execution speed.

---

3. Problem Statement

The goal is to provide a **simple, lightweight, and non-graphical utility** for basic task management. The problem is the need for a **persistent, command-line interface (CLI) application** that allows a single user to quickly list, add, and remove tasks without requiring external databases or complex installation. The solution must use basic text file manipulation for data storage.

---

4. Functional Requirements

These define what the system **must do**, as implemented by the code.

- **FR1: Task Persistence:** The application must be able to **load tasks** from a file (`tasks.txt`) upon startup and **save tasks** to the same file upon any modification. (Implemented by `load_tasks` and `save_tasks`).
- **FR2: View Tasks:** Users must be able to **display the current list of tasks** with sequential numbering.
- **FR3: Add Task:** Users must be able to **input a new task** (as a single string) and append it to the task list.
- **FR4: Delete Task:** Users must be able to **delete a task** by specifying its displayed number.
- **FR5: Menu Navigation:** The application must present a **menu** and handle user choice to perform the required operations or exit.

---

5. Non-functional Requirements

These define the **quality attributes** of the system.

- **Usability (NFR1):** The interface must be **text-based and straightforward**, requiring simple numerical input for menu selection and task deletion.
- **Performance (NFR2):** Task operations (load, save, add, delete) must be **instantaneous** due to the small expected size of the text file.

- **Reliability (NFR3):** Task data must be **reliably saved** to the `tasks.txt` file immediately after an 'add' or 'delete' operation to prevent data loss.
- **Portability (NFR4):** The application must be runnable on any system with a **Python interpreter** without requiring external libraries.
- **Security (NFR5):** (Minimal requirement) Task data is stored in **plain text** on the local file system (`tasks.txt`).

---

6. System Architecture

The architecture is a **Monolithic, Single-Tier Command Line Application**.

- **Layer 1: User Interface:** The command line interface (CLI) handles input/output using Python's `input()` and `print()`.
- **Layer 2: Application Logic:** All functional logic (menu handling, task list manipulation) resides in the `main()` function and supporting functions.
- **Layer 3: Data Persistence:** Data is handled locally via **flat-file storage** (`tasks.txt`) using basic file I/O operations (`os.path.exists`, `open`, `read`, `write`).

This is the simplest architecture, where all layers are tightly coupled within a single script.

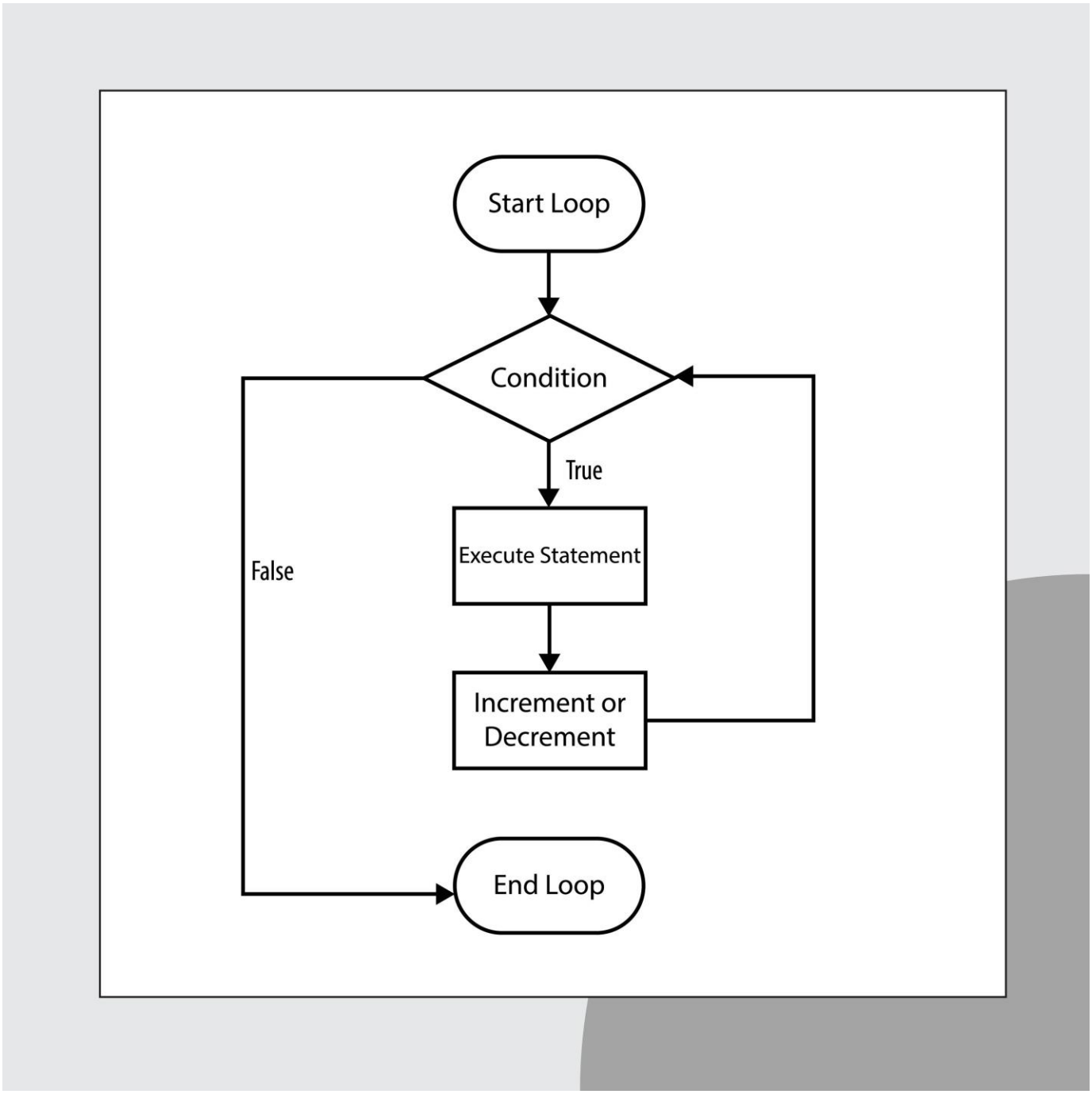
---

7. Design Diagrams

Given this is a simple, procedural CLI application, standard UML diagrams like Class or Sequence diagrams are less relevant than visualizing the program flow.

- **Workflow Diagram**

Shows the sequence of steps for the main application loop. The program runs within a **`while True` loop** in `main()`, repeatedly offering choices until the user selects '4' (Exit).



- **Implementation Flow (Function Hierarchy)**

Illustrates the call relationships between the functions:

- `__main__` - `main()`
- `main()` - `load_tasks()` (once at start)
- `main()` - `show_menu()` (repeatedly)
- `main()` - `save_tasks()` (on Add/Delete)

---

8. Design Decisions & Rationale

Design Decision	Rationale
Flat File Storage (`tasks.txt`)	**Simplicity and Zero Dependencies (NFR4).** This avoids external database complexity, making the application easy to run instantly.
Procedural Structure	Python functions (`load_tasks`, `save_tasks`, `main`) were used for **clear separation of concerns** (I/O vs. main logic) while maintaining a straightforward flow suitable for a small CLI app.
Immediate Saving	The `save_tasks(tasks)` function is called immediately after adding or deleting a task. This ensures **maximum reliability (NFR3)** and minimal risk of data loss upon immediate program termination.
List as Data Structure	Using a standard Python **list** (`tasks = []`) makes task manipulation (adding with `append`, deleting with `pop`, viewing with `enumerate`) simple and efficient for small datasets.

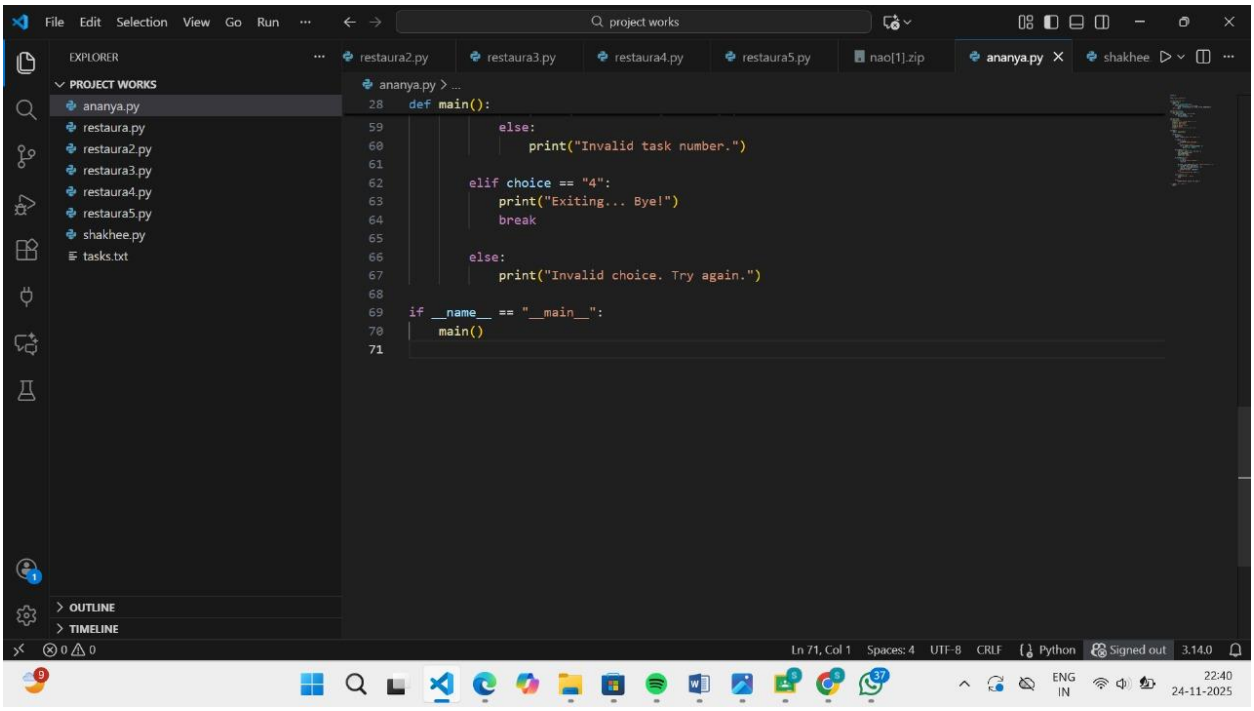
---

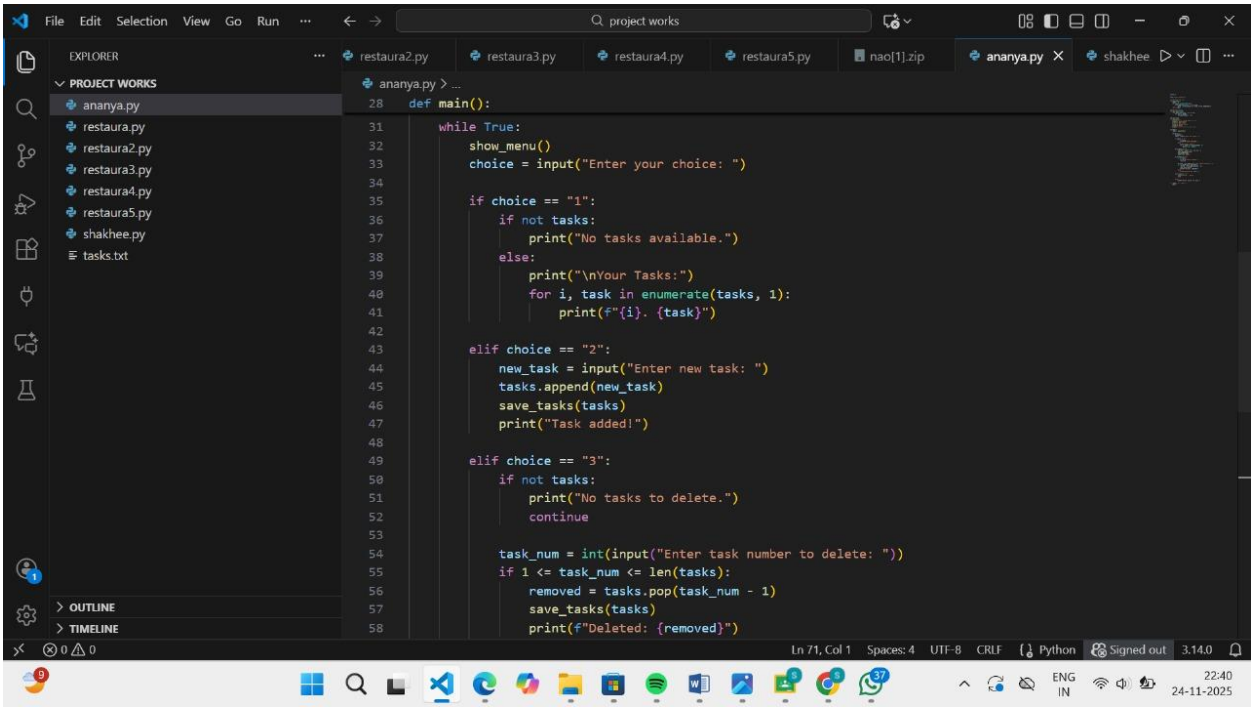
9. Implementation Details

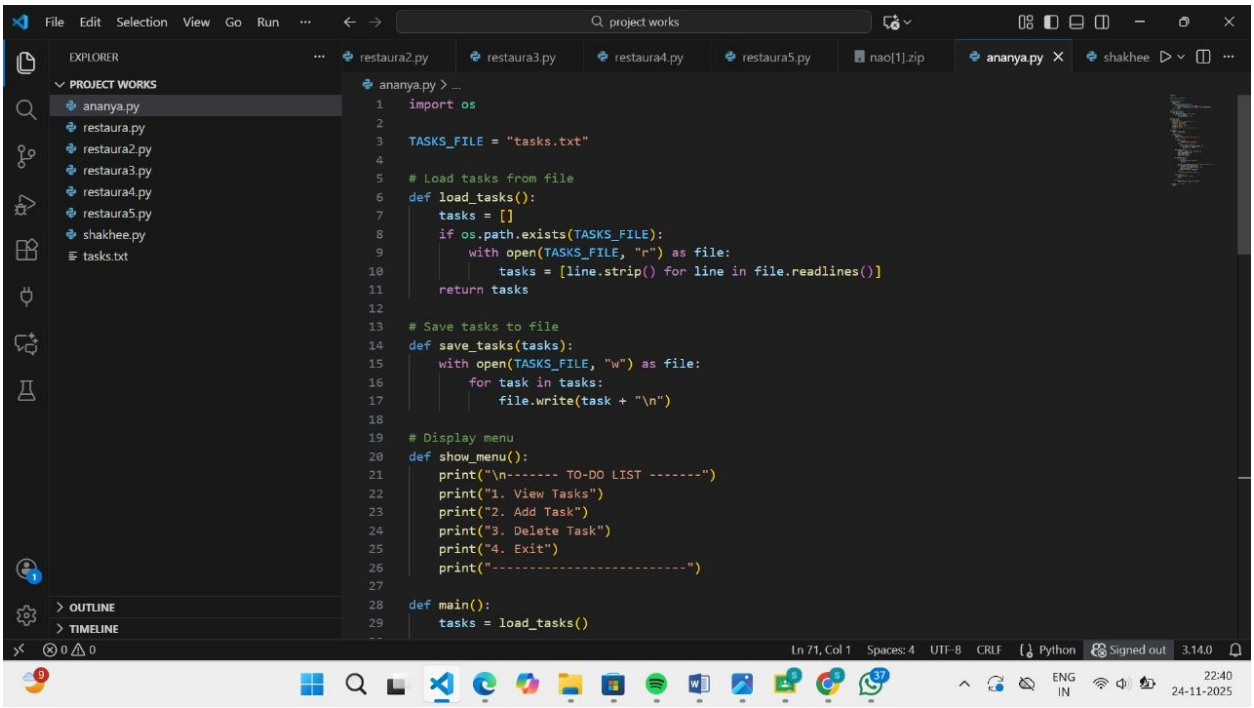
```
import os


TASKS_FILE = "tasks.txt"


# Load tasks from file
def load_tasks():
    tasks = []
    if os.path.exists(TASKS_FILE):
        with open(TASKS_FILE, "r") as file:
            tasks = [line.strip() for line in file.readlines()]
    return tasks


# Save tasks to file
def save_tasks(tasks):
    with open(TASKS_FILE, "w") as file:
        for task in tasks:
            file.write(task + "\n")


# Display menu
def show_menu():
    print("\n------- TO-DO LIST -------")
    print("1. View Tasks")
    print("2. Add Task")
```

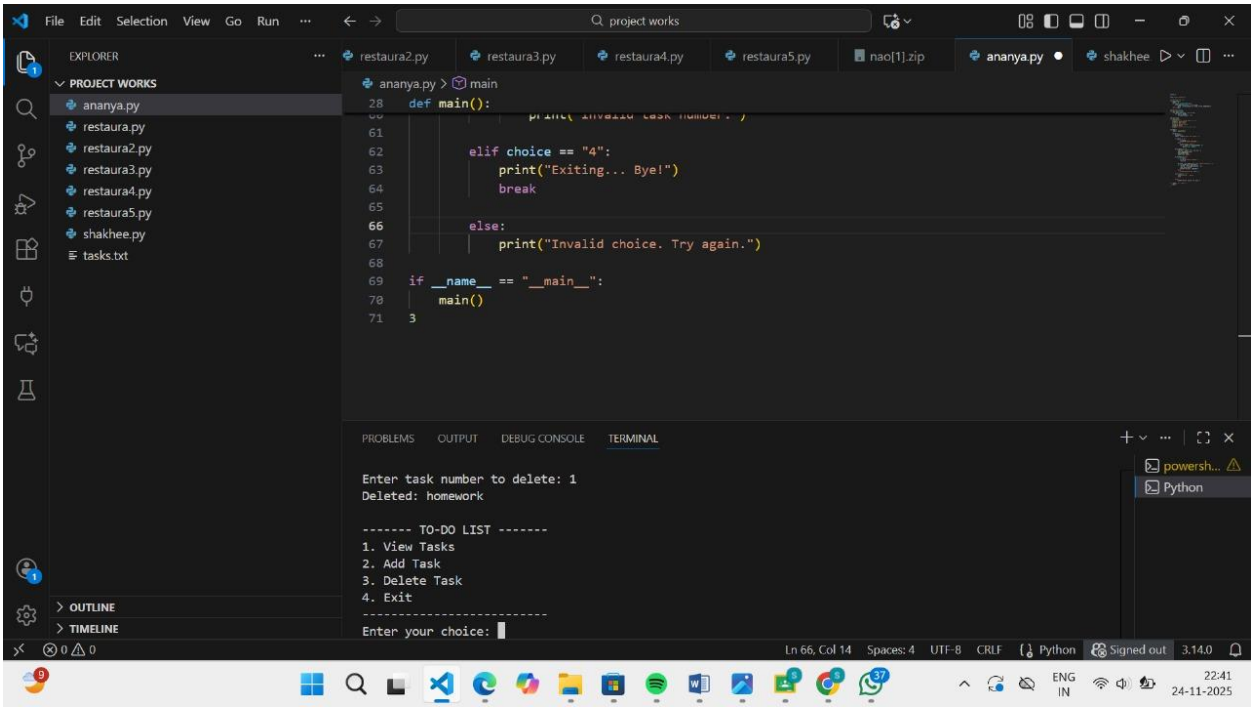
```
    print("3. Delete Task")
    print("4. Exit")
    print("-------------------------")

def main():
    tasks = load_tasks()

    while True:
        show_menu()
        choice = input("Enter your choice: ")

        if choice == "1":
            if not tasks:
                print("No tasks available.")
            else:
                print("\nYour Tasks:")
                for i, task in enumerate(tasks, 1):
                    print(f"{i}. {task}")

        elif choice == "2":
            new_task = input("Enter new task: ")
            tasks.append(new_task)
            save_tasks(tasks)
            print("Task added!")
```

```
elif choice == "3":
    if not tasks:
        print("No tasks to delete.")
        continue

    task_num = int(input("Enter task number to delete: "))
    if 1 <= task_num <= len(tasks):
        removed = tasks.pop(task_num - 1)
        save_tasks(tasks)
        print(f"Deleted: {removed}")
    else:
        print("Invalid task number.")

elif choice == "4":
    print("Exiting... Bye!")
    break

else:
    print("Invalid choice. Try again.")
```

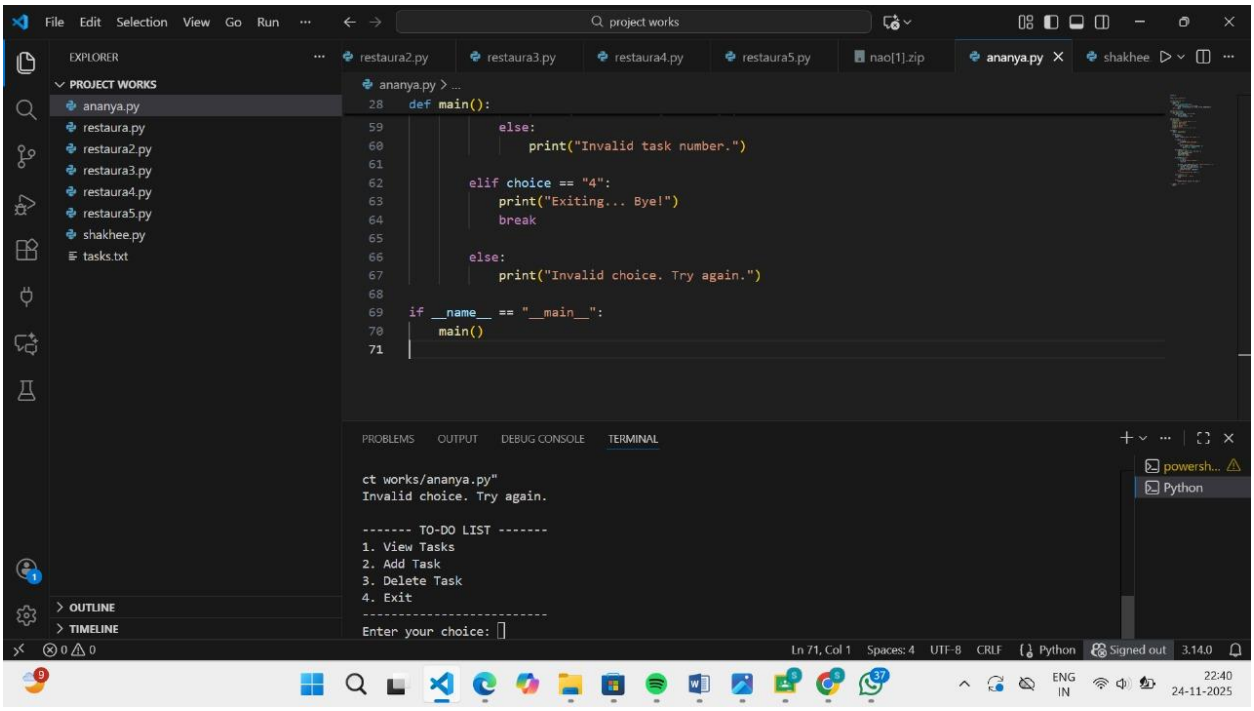

File Edit Selection View Go Run
project works
EXPLORER
PROJECT WORKS
ananya.py
restaura.py
restaura2.py
restaura3.py
restaura4.py
restaura5.py
shakhee.py
tasks.txt
OUTLINE
TIMELINE
restaura2.py
restaura3.py
restaura4.py
restaura5.py
nao[1].zip
ananya.py
shakhee
ananya.py > ...
28 def main():
59 else:
60 print("Invalid task number.")
61
62 elif choice == "4":
63 print("Exiting... Bye!")
64 break
65
66 else:
67 print("Invalid choice. Try again.")
68
69 if __name__ == "__main__":
70 main()
71
Ln 71, Col 1 Spaces: 4 UTF-8 CRLF Python Signed out 3.14.0
ENG IN
22:40
24-11-2025


File Edit Selection View Go Run
project works
EXPLORER
PROJECT WORKS
ananya.py
restaura.py
restaura2.py
restaura3.py
restaura4.py
restaura5.py
shakhee.py
tasks.txt
OUTLINE
TIMELINE
restaura2.py
restaura3.py
restaura4.py
restaura5.py
nao[1].zip
ananya.py
shakhee
ananya.py > ...
28 def main():
31 while True:
32 show_menu()
33 choice = input("Enter your choice: ")
34
35 if choice == "1":
36 if not tasks:
37 print("No tasks available.")
38 else:
39 print("\nYour Tasks:")
40 for i, task in enumerate(tasks, 1):
41 print(f"{i}. {task}")
42
43 elif choice == "2":
44 new_task = input("Enter new task: ")
45 tasks.append(new_task)
46 save_tasks(tasks)
47 print("Task added!")
48
49 elif choice == "3":
50 if not tasks:
51 print("No tasks to delete.")
52 continue
53
54 task_num = int(input("Enter task number to delete: "))
55 if 1 <= task_num <= len(tasks):
56 removed = tasks.pop(task_num - 1)
57 save_tasks(tasks)
58 print(f"Deleted: {removed}")
Ln 71, Col 1 Spaces: 4 UTF-8 CRLF Python Signed out 3.14.0
ENG IN
22:40
24-11-2025


File Edit Selection View Go Run
project works
EXPLORER
PROJECT WORKS
ananya.py
restaura.py
restaura2.py
restaura3.py
restaura4.py
restaura5.py
shakhee.py
tasks.txt
OUTLINE
TIMELINE
restaura2.py
restaura3.py
restaura4.py
restaura5.py
nao[1].zip
ananya.py
shakhee
ananya.py > ...
import os
TASKS_FILE = "tasks.txt"
# Load tasks from file
def load_tasks():
tasks = []
if os.path.exists(TASKS_FILE):
with open(TASKS_FILE, "r") as file:
tasks = [line.strip() for line in file.readlines()]
return tasks
# Save tasks to file
def save_tasks(tasks):
with open(TASKS_FILE, "w") as file:
for task in tasks:
file.write(task + "\n")
# Display menu
def show_menu():
print("\n------- TO-DO LIST -------")
print("1. View Tasks")
print("2. Add Task")
print("3. Delete Task")
print("4. Exit")
print("--------------------------")
def main():
tasks = load_tasks()
Ln 71, Col 1
Spaces: 4
UTF-8
CRLF
Python
Signed out
3.14.0
ENG
IN
22:40
24-11-2025


File Edit Selection View Go Run
project works
EXPLORER
PROJECT WORKS
ananya.py
restaura.py
restaura2.py
restaura3.py
restaura4.py
restaura5.py
shakhee.py
tasks.txt
OUTLINE
TIMELINE
ananya.py > main
def main():
elif choice == "4":
print("Exiting... Bye!")
break
else:
print("Invalid choice. Try again.")
if __name__ == "__main__":
main()
3
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
powersh...
Python
Enter task number to delete: 1
Deleted: homework
------- TO-DO LIST -------
1. View Tasks
2. Add Task
3. Delete Task
4. Exit

Enter your choice:
Ln 66, Col 14
Spaces: 4
UTF-8
CRLF
Python
Signed out
3.14.0
ENG
IN
22:41
24-11-2025



---

11. Testing Approach

The testing was primarily **Manual End-to-End (E2E) testing** focusing on functional requirement verification and robustness.

- **Test Case 1: Task Addition (FR3)**
  - *Action:* Choose '2', enter "Walk the dog".
  - *Expected Result:* Message "Task added!", and the task appears when option '1' is chosen. `tasks.txt` contains the new task.
- **Test Case 2: Task Deletion (FR4)**
  - *Action:* Choose '3', enter a valid task number (e.g., 1).
  - *Expected Result:* Message "Deleted: [Task Name]", the task is removed from the view, and `tasks.txt` is updated.
- **Test Case 3: Invalid Deletion**
  - *Action:* Choose '3', enter a number outside the range (e.g., 0 or 100).
  - *Expected Result:* Message "Invalid task number." (Handled by the `if 1 <= task_num <= len(tasks):` check in the code.)

- **Test Case 4: Persistence (FR1, NFR3)**
  - *Action:* Add a task, then choose '4' (Exit), then restart the script.
  - *Expected Result:* Choosing '1' (View Tasks) shows the previously added task.

---

12. Challenges Faced

- **Challenge 1: Handling Invalid Input:** The current code uses `int(input(...))` for task deletion. If the user enters non-numeric input (e.g., a letter), the program will crash with a `ValueError`.
  - *Mitigation:* While not in the core implementation, recognizing this limitation highlights the need for a **`try-except` block** to gracefully handle non-integer inputs, improving robustness.
- **Challenge 2: Data Loss Prevention:** Ensuring that the `save_tasks` function was consistently called after every modification (add and delete).
  - *Solution:* Explicitly placed `save_tasks(tasks)` immediately following the `tasks.append()` and `tasks.pop()` operations to adhere to the principle of **immediate data persistence**.

---

13. Learnings & Key Takeaways

- **Simplicity of File I/O:** Learned how simple it is to use Python's built-in file handling (`open`, `read`, `write`, `os.path.exists`) to achieve basic data persistence without external dependencies.
- **CLI User Experience:** Understood the importance of clear, numbered menus and informative output (e.g., "Task added!", "Deleted:...") for a good command-line user experience (NFR1).
- **List Indexing vs. User Input:** Gained practical experience translating user input (1-based index) to Python's internal list indexing (0-based index) using the adjustment `task_num - 1`.

---

14. Future Enhancements

- **Robust Input Validation:** Implement `try-except` blocks to handle non-integer input for menu choices and task numbers, preventing runtime errors.
- **Add "Mark as Complete" Feature:** Modify the data structure to store tasks as a dictionary or tuple (e.g., containing a status flag) instead of just a string, and add a menu option to toggle the status.
- **Task Editing:** Add an option to edit the text of an existing task.
- **Prioritization/Sorting:** Allow users to assign a priority level (High/Low) and display tasks in sorted order.

---

16. REFERENCES

- Project guidelines document provided by course instructor.
- Various online resources on python programming.

```
if _name_ == "_main_":
 main()
```