📝 Command Line To-Do List Manager Documentation

1. Project Overview

This project is a functional **Command Line Task Manager** (a To-Do List application). It allows users to add, view, mark as complete, and delete tasks. The most significant feature is **data persistence**, meaning the tasks are saved to a file and loaded every time the program runs, ensuring no data is lost between sessions.

1.1 Key Learning Objectives

- **File I/O (`open()`, `read()`, `write()`):** Reading from and writing to a text file to save data.
- **Data Structures (Dictionaries):** Using a **dictionary** (`tasks`) to store structured task data (ID, title, status).
- **Module Usage (`os`):** Utilizing the `os` module for checking file existence.
- **Menu-Driven Interface:** Implementing a continuous, interactive user experience.
- **Program Entry Point (`if __name__ == "__main__":`):** Understanding and utilizing the standard convention for running the main function.

---

2. Program Structure and Data Model

2.1 Global Configuration

The program uses one global constant and imports one module:

- `import os`: Provides functions for interacting with the operating system, specifically used here to check if the data file exists.
- `FILE_NAME = "tasks.txt"`: Defines the name of the file used to store the persistent task data.

2.2 Task Data Structure

All tasks are stored in a single Python dictionary named `tasks`.

Data Structure	Key (Task ID)	Value (Task Details)	Example Entry

Dictionary (tasks)	Integer (e.g., 1, 2, 3)	Nested Dictionary	1: {"title": "Code Documentation", "status": "complete"}

---

3. Function Analysis (Data Persistence)

These two functions are responsible for ensuring the tasks survive when the program closes.

3.1 load_tasks()

This function is called at the beginning of the main() routine to retrieve saved data.

1. It checks if tasks.txt exists using os.path.exists().
2. If the file exists, it opens the file in **read mode ("r")**.
3. It iterates through each line of the file. Each line is stripped of whitespace and split by the delimiter " | ".
4. The split parts (ID, title, status) are then used to reconstruct the tasks dictionary.

> **Data Format in tasks.txt:** The file stores tasks in a simple, pipe-separated format: [ID] | [TITLE] | [STATUS].

3.2 save_tasks(tasks)

This function is called only when the user selects the **Exit (5)** option.

1. It opens tasks.txt in **write mode ("w")**. This *overwrites* the file with the current state of the tasks dictionary.
2. It iterates through the tasks dictionary and writes each task to the file, using an **f-string** to format the data back into the pipe-separated string format, followed by a newline character (\n).

---

4. Function Analysis (Core Logic)

4.1 add_task(tasks)

- Prompts the user for the task title.
- **Task ID Generation:** It finds the maximum existing Task ID using max(tasks.keys(), default=0) and adds **1** to it, ensuring each new task has a unique, sequential ID.
- Initializes the new task with the status set to "incomplete".

4.2 view_tasks(tasks)

- Checks if the `tasks` dictionary is empty.
- If tasks exist, it loops through the dictionary and prints the ID, title, and current status in a user-friendly format: `[ID] Title - status`.

4.3 `mark_task_complete(tasks)`

- Prompts the user for a Task ID.
- Checks if the entered ID exists in the `tasks` dictionary.
- If found, it accesses the task's inner dictionary and changes the `"status"` key's value to `"complete"`.

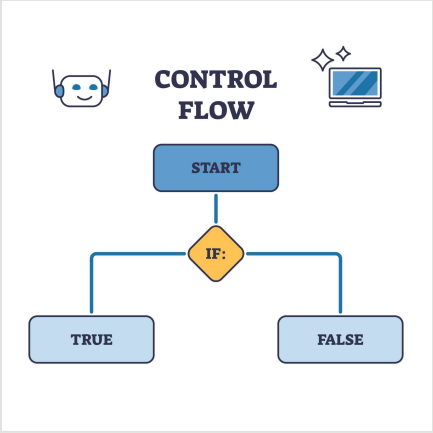
4.4 `delete_task(tasks)`

- Prompts the user for a Task ID.
- Checks if the entered ID exists.
- If found, it uses the **`.pop(task_id)`** dictionary method to remove the entry completely and prints the title of the deleted task for confirmation.

---

5. Main Execution (`main()` function)

The `main()` function serves as the program's control center.



Getty Images

1. **Initialization:** `tasks = load_tasks()` is called to fetch any saved data.
2. **Menu Loop:** A `while True` loop repeatedly displays the menu and accepts a `choice` from the user.
3. **Operation Execution:** `if-elif` statements map the choice to the corresponding function (e.g., `"1"` calls `add_task(tasks)`).
4. **Exit:** If `"5"` is chosen, `save_tasks(tasks)` is called to write the data to the file, and the loop is terminated with `break`.