**Code Metrics Report**

### **Overview**

This report provides an analysis of the provided Python code, evaluating its structure, functionality, and maintainability.

### **Metrics Summary**

| **Metric** | **Count** | **Description** |
| --- | --- | --- |
| **LOC** | 68 | Total number of lines, including blank lines and comments. |
| **PLOC** | 60 | Physical lines of code: lines containing actual code, excluding comments and blanks. |
| **Comment Lines** | 5 | Number of lines containing comments. |

### **Detailed Observations**

1. **Lines of Code (LOC):**
   * The provided script contains **68 total lines**.
   * This includes:
     + **5 lines of comments**, providing a brief description of the script's functionality and purpose.
     + **3 blank lines** to enhance readability.
2. **Physical Lines of Code (PLOC):**
   * Out of the total lines, **60 lines** contain actual executable code.
   * The code is distributed across several functions and the main menu-based interface.
3. **Comments:**
   * **Count:** 5
   * Observations:
     + The script includes docstrings for all functions, which describe their purpose and expected behavior.
     + However, inline comments explaining complex parts of the logic are absent.

### **Code Structure Analysis**

1. **Key Strengths:**
   * Functions are well-defined and modular, focusing on single responsibilities:
     + add\_task: Adds a task to the task list.
     + list\_tasks: Displays all tasks.
     + complete\_task: Marks a specified task as completed.
     + main: Implements the interactive menu interface.
   * Clear separation of concerns ensures readability and maintainability.
   * Validation is implemented for user input (e.g., checking task priority range and input types).
2. **Potential Weaknesses:**
   * Tasks are stored in a global list, which may not scale well in larger applications. A more structured approach, such as a class-based implementation, would improve scalability.
   * Error handling could be more descriptive in cases of invalid input.
   * The task list does not persist between sessions. Incorporating a database or file-based storage would enhance usability.
3. **Dependencies:**
   * The script depends only on Python's standard library (typing module), ensuring portability.

### **Recommendations**

1. **Enhance Documentation:**
   * Expand the docstrings to include parameter types and return values.
   * Add inline comments where logic may not be immediately apparent (e.g., input validation and task marking logic).
2. **Error Handling:**
   * Provide more descriptive error messages to guide users during input errors.
   * Consider adding exceptions for edge cases like an empty task list during completion attempts.
3. **Refactor for Scalability:**
   * Replace the global tasks list with a class or data structure encapsulating task management.
   * Implement file-based or database storage to persist tasks between application runs.
4. **Extend Functionality:**
   * Add features like task deletion, task editing, and sorting tasks by priority or completion status.
   * Include a confirmation prompt before exiting the application to prevent accidental closure.
5. **Testing:**
   * Create unit tests for each function to validate functionality and prevent future regressions.

### **Conclusion**

The provided Python script demonstrates a functional and modular implementation of a task manager application with a clear menu-based interface. While the code is readable and effective for its purpose, incorporating better documentation, scalability improvements, and persistent storage would enhance its robustness and usability.