

A PROJECT REPORT ON

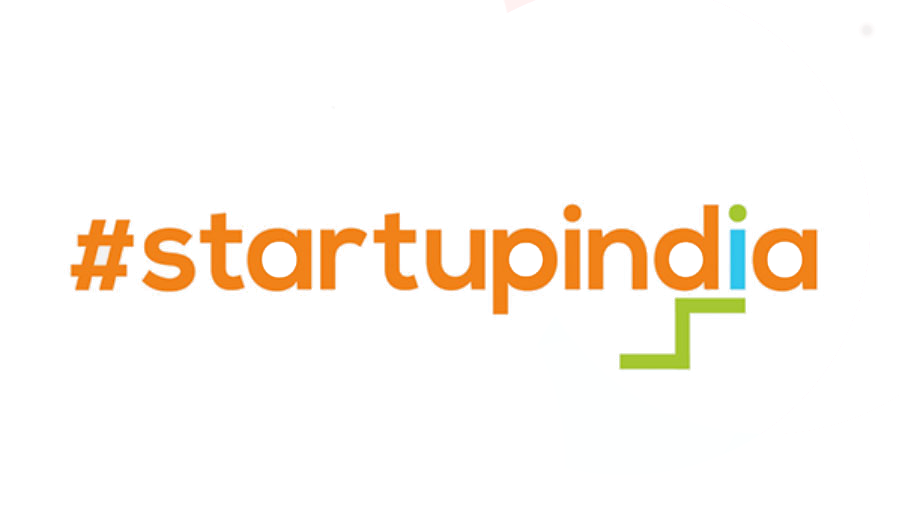
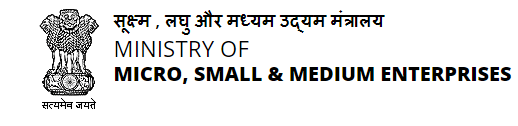
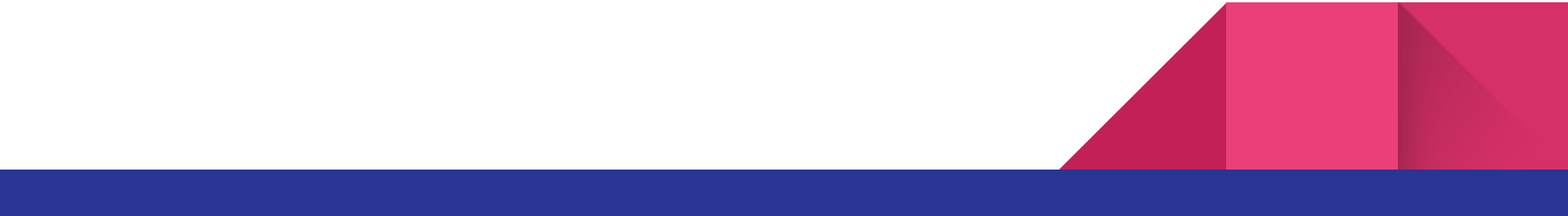
To Do list (Python)

### Submitted To: Submitted By:

P Abhishek B. Tharun kumar reddy.







**Shri Buddha Chandrasekhar**

Chief Coordinating Officer(CCO)

AICTE

**P Abishek**

Human Resources (HR)

Slash Mark IT Startup

ABSTRACT

The main objective of the provided project is to implement a simple, interactive to-do list application that allows users to manage their tasks efficiently. The key functionalities provided by the code include:

1. **Displaying Tasks**: Showing the current list of tasks along with their completion status (done or not done).
2. **Adding Tasks**: Allowing users to add new tasks to the to-do list.
3. **Marking Tasks as Completed**: Enabling users to mark specific tasks as completed.
4. **Removing Tasks**: Giving users the ability to remove tasks from the list.
5. **User Interaction**: Providing a user-friendly menu-driven interface that guides users through various options and operations.

In essence, the Project aims to offer a straightforward tool for task management, helping users keep track of their to-do items and maintain an organized list of tasks.

# TABLE OF CONTENTS

#### Declaration 1 Abstract 2 Table of Contents 3



Problem Statement 4 o Purpose 4 o Goals and Objectives 4 o Description \_ 5



#### Scope 6 Introduction 7 o



#### Software Details \_ 8-10



#### Reference Code 11-12

#### Implementation Work Details(Reference code) 14-18-Key Functions 19-21

Error modifications 22-23



Updated code 24-25



Input and Output 26-31



Conclusion 32



PROBLEM STATEMENT

You are tasked with managing a simple command-line based To-Do List application. The application should allow users to perform the following actions:

1. Display the current list of tasks.
2. Add new tasks to the list.
3. Mark tasks as completed.
4. Remove tasks from the list.
5. Exit the application.

PURPOSE

The primary purpose of this project is to help users keep track of their tasks and

manage their to-do list efficiently. The application provides basic functionality for

organizing tasks, marking them as completed, and removing them when no longer

needed. This helps users maintain a clear and organized list of tasks, ensuring that

nothing important is forgotten or overlooked.

GOALS AND OBJECTIVES

The goal of the above program is to provide a simple, command-line based To-Do List application that allows users to efficiently manage their tasks. It aims to help users keep track of their tasks, mark tasks as completed, and remove tasks from their list in an easy and intuitive manner.

1. **Task Management**: Enable users to manage their tasks by providing functionalities to add, display, mark as completed, and remove tasks from their to-do list.
2. **Usability**: Ensure the application is user-friendly and straightforward, allowing users to interact with it easily through a command-line interface.
3. **Task Tracking**: Allow users to keep track of their tasks and their completion status, helping them stay organized and focused on their priorities.
4. **Flexibility**: Provide a simple yet flexible solution that can be easily modified or extended to include additional features as needed.

### DESCRIPTION

The Command-Line To-Do List Application is a straightforward Python program designed to assist users in managing their daily tasks efficiently. This minimalist tool offers a text-based interface where users can add new tasks, view their current task list, mark tasks as completed, and remove tasks from the list. The application displays tasks along with their statuses—either "Not Done" or "Done"—allowing users to track their progress easily. Tasks are stored in a list of dictionaries, each containing the task description and its completion status. Key functions include displaying the task list, adding new tasks, marking tasks as completed, and removing tasks, all of which are accessed through a main menu presented in a loop until the user opts to quit. This design ensures that the program is both user-friendly and efficient. The primary purpose of the application is to help users maintain an organized and up-to-date list of tasks, improving productivity and ensuring important tasks are not forgotten. Its simplicity makes it ideal for users who prefer a no-frills approach to task management, while its flexibility allows for easy customization or extension with additional features if desired.

### SCOPE

#### Functional Scope:

1. **Task Management**:
   * Adding new tasks.
   * Viewing the current list of tasks.
   * Marking tasks as completed.
   * Removing tasks from the list.
2. **User Interaction**:
   * Command-line interface with a main menu for options.
   * Handling user inputs for various actions.

#### Technical Scope:

1. **Programming Language**: Implemented in Python.
2. **Data Storage**: In-memory list of dictionaries for tasks.
3. **Modularity**: Distinct functions for each operation.
4. **Main Loop**: Continuous loop for user interaction until quitting.

#### Limitations:

1. **Single User**: Designed for a single user per session.
2. **No Persistent Storage**: Tasks are not saved between sessions.
3. **Basic Features**: Lacks advanced features like prioritization, deadlines, and categories.
4. **Basic Error Handling**: Limited error handling for invalid inputs.

#### Potential Extensions:

1. **Persistent Storage**: Saving tasks to a file or database.
2. **Advanced Features**: Adding task prioritization, deadlines, and categories.
3. **User Interface**: Developing a graphical user interface (GUI).
4. **Multi-User Support**: Supporting multiple users and concurrent sessions.
5. **Integration**: Integrating with other productivity tools.

### INTRODUCTION

The Command-Line To-Do List Application is a streamlined Python program designed to help users manage their daily tasks effectively. The application offers a minimalist, text-based interface, making it accessible and easy to use for individuals who prefer straightforward solutions over complex task management tools. By providing essential functionalities such as adding tasks, displaying the current task list, marking tasks as completed, and removing tasks, this application aims to improve users' productivity and ensure they stay organized without unnecessary frills.

At the heart of the application is a simple yet powerful structure that stores tasks in a list of dictionaries. Each dictionary entry contains a task description and its completion status, allowing users to keep track of what needs to be done and what has already been accomplished. The user interacts with the application through a command-line interface that continuously presents a menu of options. This interface guides users through various actions, from adding new tasks to marking existing ones as completed or removing them from the list. The use of Python ensures the program is both easy to read and modify, making it suitable for quick implementation and future enhancements.

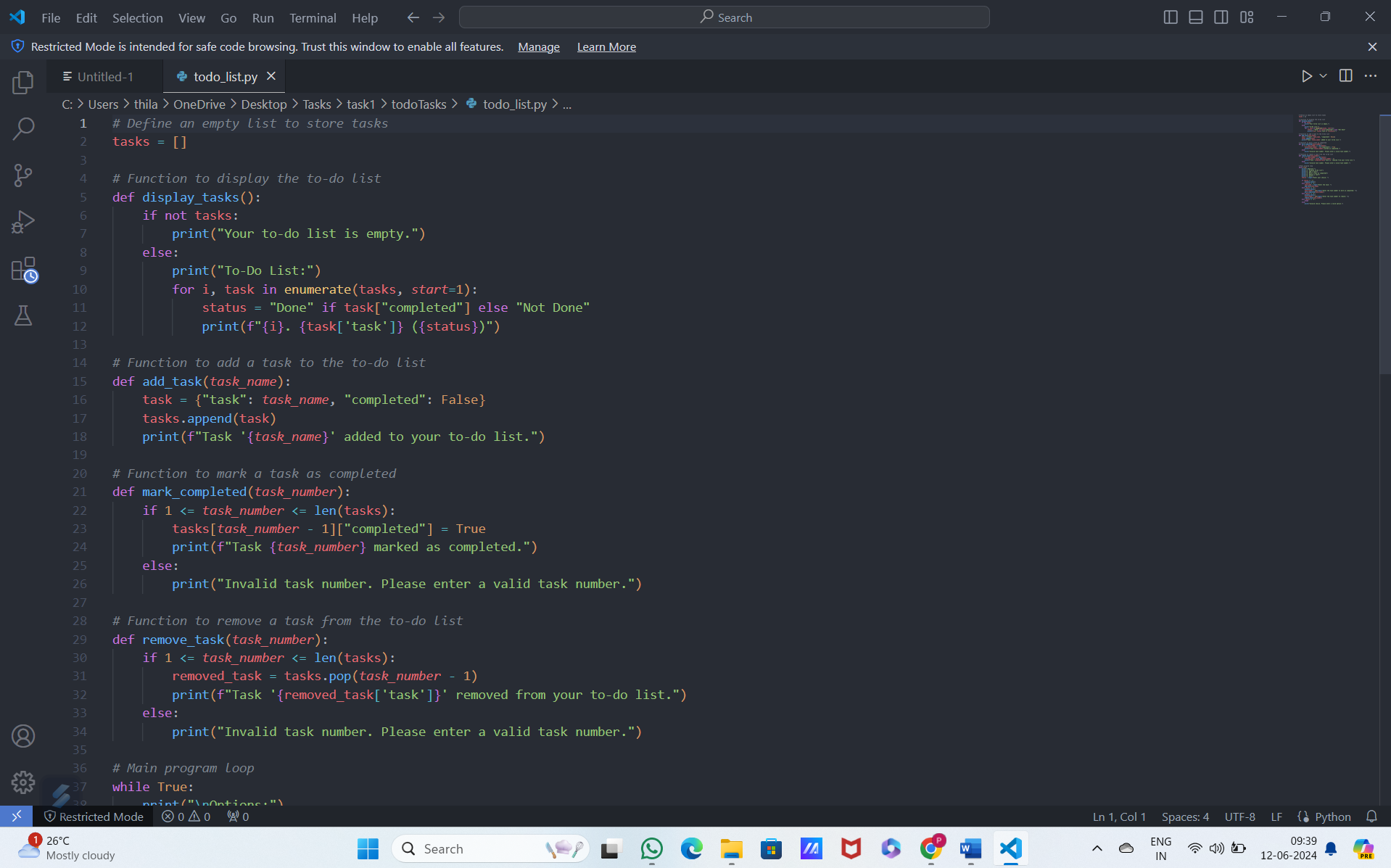
Despite its simplicity, the Command-Line To-Do List Application is designed with potential for growth. While it currently lacks persistent storage and advanced features like task prioritization, deadlines, and categorization, these can be added to extend its functionality. Future enhancements could include saving tasks to a file or database, developing a graphical user interface for a more user-friendly experience, and supporting multiple users. This foundational tool is ideal for those looking for a basic, efficient way to manage tasks while also offering a base for further development and customization.

### SOFTWARE DETAILS

Visual Studio Code (VS Code) is a highly popular, open-source code editor developed by Microsoft. Released in 2015, it has quickly become a favorite among developers due to its lightweight nature and powerful features. VS Code supports a wide range of programming languages and frameworks, making it a versatile tool for various types of development, from web applications to machine learning projects. Its user interface is highly customizable, allowing developers to tweak themes, keyboard shortcuts, and layout to fit their workflow. The built-in Git integration is a significant advantage, enabling seamless version control and collaboration directly within the editor.

One of the standout features of VS Code is its extensive marketplace of extensions. These extensions can enhance functionality with features like advanced debugging, code snippets, linting, and integrations with various development tools and services. The editor also includes IntelliSense, which provides intelligent code completion, parameter info, and quick info on hover, significantly boosting productivity. Moreover, VS Code's live share feature allows real-time collaborative coding, making it easier for teams to work together remotely. Regular updates and a strong community support make Visual Studio Code an evolving tool that adapts to the ever-changing landscape of software development.

VS Code



### COMPLETE TO DO LIST (REFERENCE)CODE:

*# Define an empty list to store tasks*

tasks = []

*# Function to display the to-do list*

def display\_tasks():

    if not tasks:

        print("Your to-do list is empty.")

    else:

        print("To-Do List:")

        for i, task in enumerate(tasks, *start*=1):

            status = "Done" if task["completed"] else "Not Done"

            print(f"{i}. {task['task']} ({status})")

*# Function to add a task to the to-do list*

def add\_task(*task\_name*):

    task = {"task": *task\_name*, "completed": False}

    tasks.append(task)

    print(f"Task '{*task\_name*}' added to your to-do list.")

*# Function to mark a task as completed*

def mark\_completed(*task\_number*):

    if 1 <= *task\_number* <= len(tasks):

        tasks[*task\_number* - 1]["completed"] = True

        print(f"Task {*task\_number*} marked as completed.")

    else:

        print("Invalid task number. Please enter a valid task number.")

*# Function to remove a task from the to-do list*

def remove\_task(*task\_number*):

    if 1 <= *task\_number* <= len(tasks):

        removed\_task = tasks.pop(*task\_number* - 1)

        print(f"Task '{removed\_task['task']}' removed from your to-do list.")

    else:

        print("Invalid task number. Please enter a valid task number.")

*# Main program loop*

while True:

    print("\nOptions:")

    print("1. Display to-do list")

    print("2. Add a task")

    print("3. Mark a task as completed")

    print("4. Remove a task")

    print("5. Quit")

 choice = input("Enter your choice: ")

    if choice == '1':

        display\_tasks()

    elif choice == '2':

        task\_name = input("Enter the task: ")

        add\_task(task\_name)

    elif choice == '3':

        display\_tasks()

        task\_number = int(input("Enter the task number to mark as completed: "))

        mark\_completed(task\_number)

    elif choice == '4':

        display\_tasks()

        task\_number = int(input("Enter the task number to remove: "))

        remove\_task(task\_number)

    elif choice == '5':

        break

    else:

        print("Invalid choice. Please enter a valid option.")

### **IMPLEMENTATION WORK DETAILS** INPUT/OUTPUT SCREENSHOTS

**Example Interaction**

**Main Menu Display**

**Output:**

Options:

1. Display to-do list

2. Add a task

3. Mark a task as completed

4. Remove a task

5. Quit

Enter your choice:

Scenario 1: Displaying an Empty To-Do List

Input:

1

Output:

Your to-do list is empty.

Scenario 2: Adding Tasks

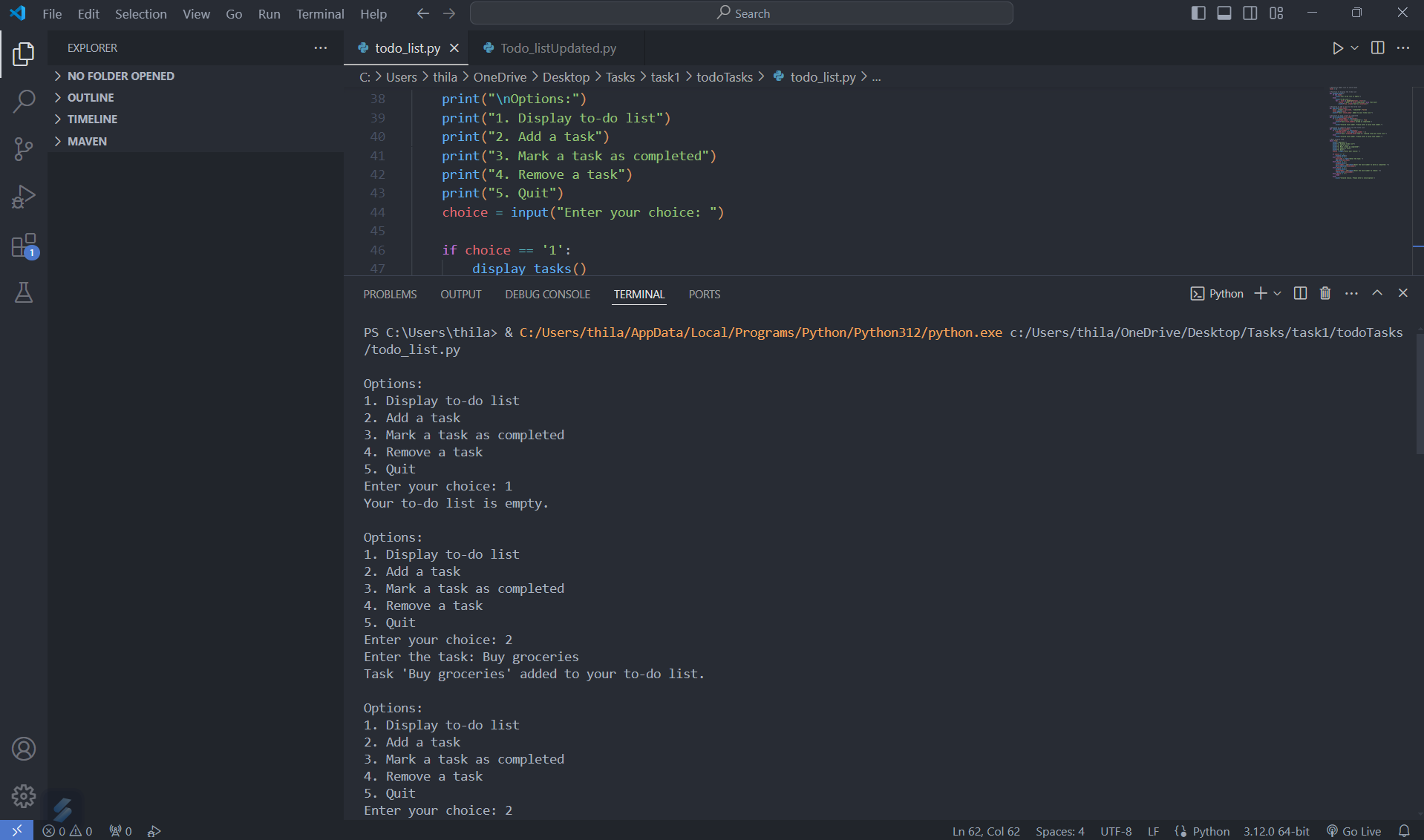
Input:

2

Enter the task: Buy groceries

Output:

Task 'Buy groceries' added to your to-do list.

Input:

2

Enter the task: Call the doctor

Output:

Task 'Call the doctor' added to your to-do list.

Scenario 3: Displaying the To-Do List with Tasks

Input:

1

Output:

To-Do List:

1. Buy groceries (Not Done)

2. Call the doctor (Not Done)

Scenario 4: Marking a Task as Completed

Input:

3

To-Do List:

1. Buy groceries (Not Done)

2. Call the doctor (Not Done)

Enter the task number to mark as completed: 1

Output:

Task 1 marked as completed.

Input:

1

Output:

To-Do List:

1. Buy groceries (Done)

2. Call the doctor (Not Done)

Scenario 5: Removing a Task

Input:

4

To-Do List:

1. Buy groceries (Done)

2. Call the doctor (Not Done)

Enter the task number to remove: 2

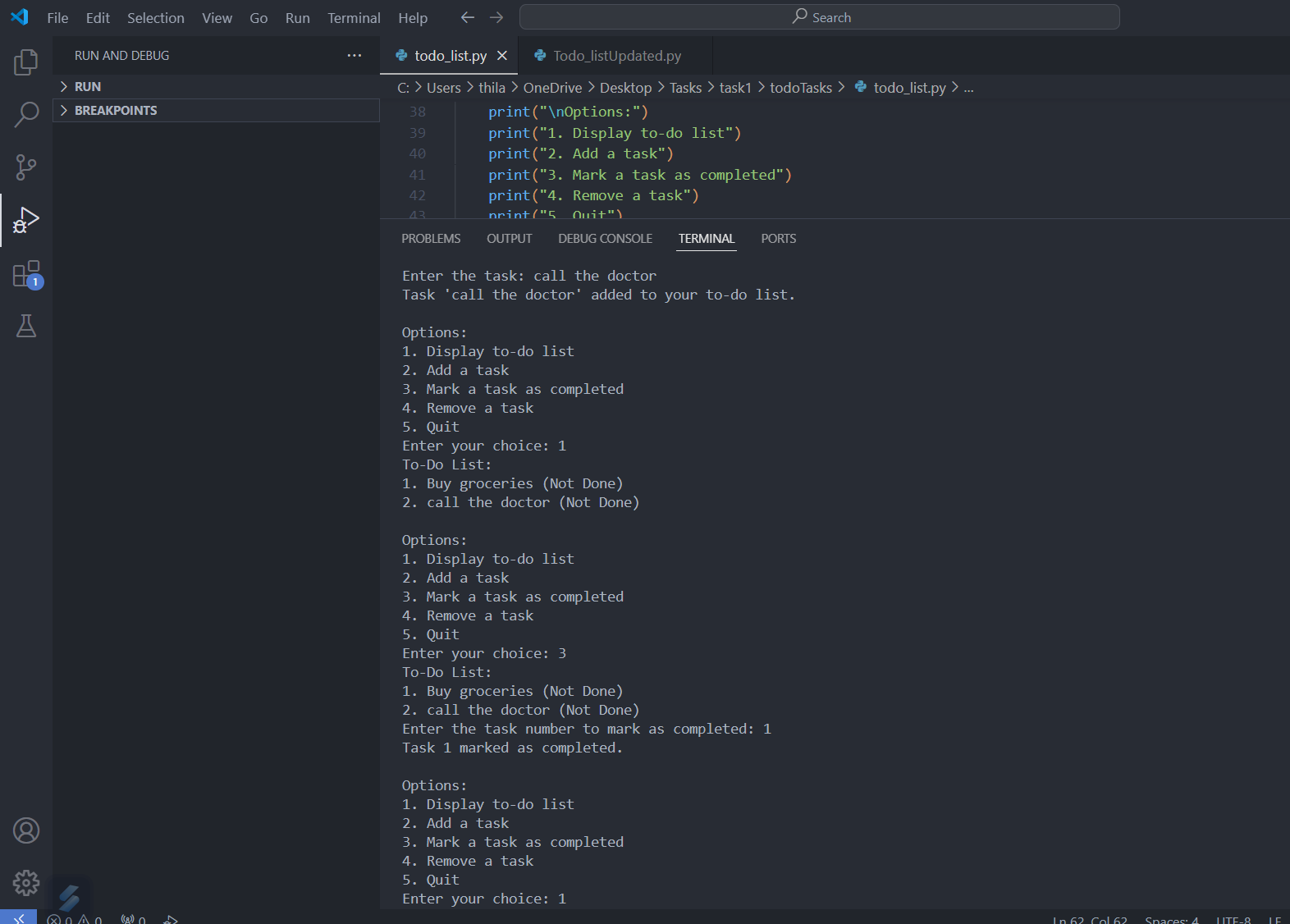
Output:

Task 'Call the doctor' removed from your to-do list.

Input:

1

Output:



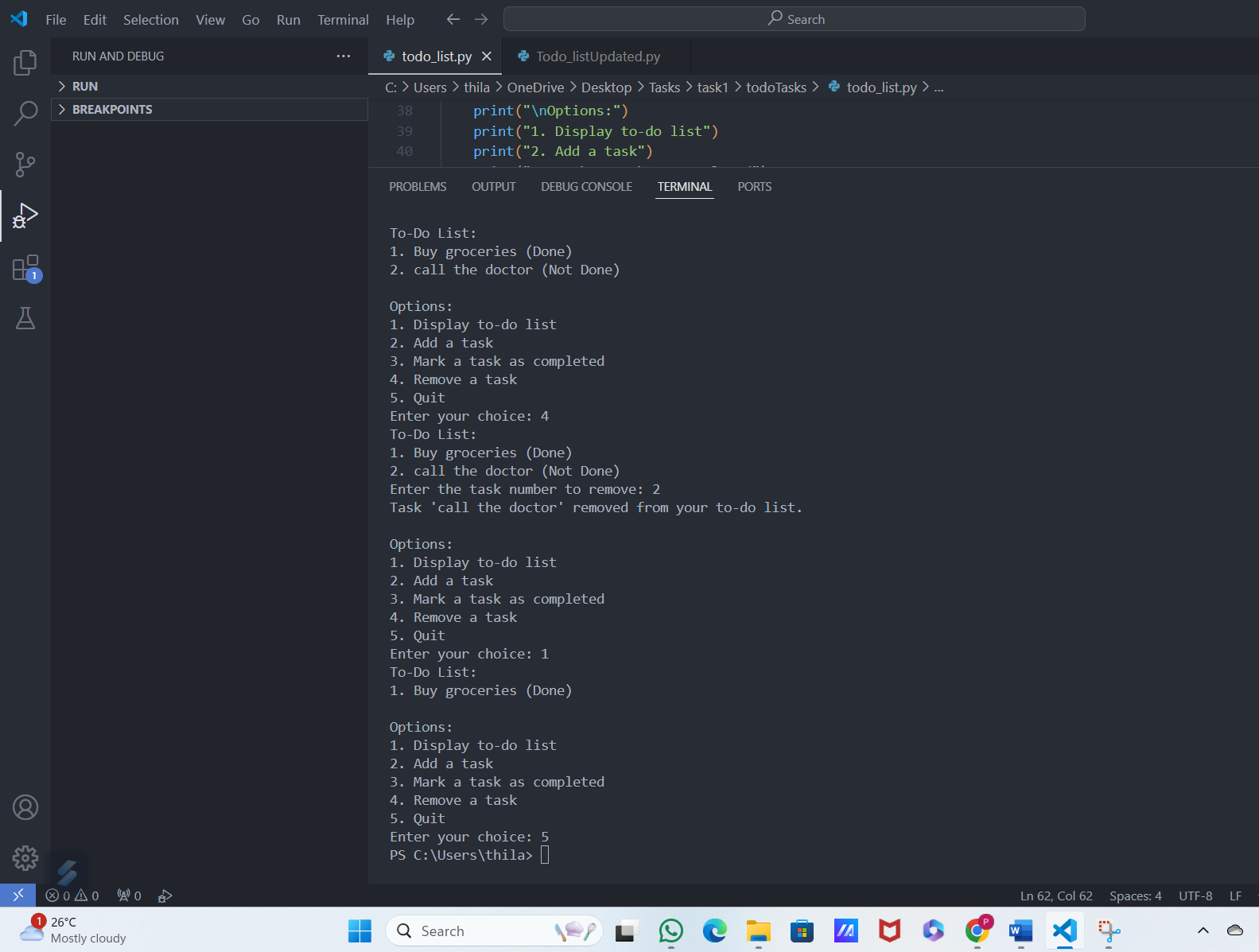
To-Do List:

1. Buy groceries (Done)

Scenario 6: Exiting the Program

Input:

5



This example demonstrates the main interactions between the user and the to-do list

program, illustrating how inputs are processed and how outputs are presented at each step.

### KEY FUNCTIONS OF CODE:

display\_tasks():

Purpose: Display the current list of tasks.

Behavior:

If the tasks list is empty, it prints "Your to-do list is empty."

Otherwise, it prints each task with its status ("Done" or "Not Done").

Parameters: None

Returns: None

add\_task(task\_name):

Purpose: Add a new task to the to-do list.

Behavior:

Creates a dictionary with the task name and a completed status set to False.

Appends this dictionary to the tasks list.

Prints a confirmation message with the task name.

Parameters:

task\_name (str): The name of the task to be added.

Returns: None

mark\_completed(task\_number):

Purpose: Mark a specific task as completed.

Behavior:

Checks if the task number is valid (within the range of the tasks list).

Sets the completed status of the specified task to True.

Prints a confirmation message with the task number.

If the task number is invalid, it prints an error message.

Parameters:

task\_number (int): The number of the task to be marked as completed.

Returns: None

remove\_task(task\_number):

Purpose: Remove a specific task from the to-do list.

Behavior:

Checks if the task number is valid (within the range of the tasks list).

Removes the specified task from the tasks list.

Prints a confirmation message with the task name.

If the task number is invalid, it prints an error message.

Parameters:

task\_number (int): The number of the task to be removed.

Returns: None

Main Program Loop

Purpose: Provide a user interface to interact with the to-do list.

Behavior:

Displays a menu with options: Display to-do list, Add a task, Mark a task as completed, Remove a task, and Quit.

Prompts the user for a choice and calls the appropriate function based on the input.

Repeats until the user chooses to quit.

Example Usage

Display the to-do list:

User selects option 1.

Calls display\_tasks().

Add a task:

User selects option 2.

Prompts for the task name.

Calls add\_task(task\_name) with the provided task name.

Mark a task as completed:

User selects option 3.

Calls display\_tasks() to show the current tasks.

Prompts for the task number.

Calls mark\_completed(task\_number) with the provided task number.

Remove a task:

User selects option 4.

Calls display\_tasks() to show the current tasks.

Prompts for the task number.

Calls remove\_task(task\_number) with the provided task number.

Quit:

User selects option 5.

Exits the loop and ends the program.

### ERRORS AND DRAWBACKS OF CODE:

**Errors and Drawbacks of todo\_list.py Addressed in Todo\_listUpdated.py**

1. **Global State Management**:
   * **Error/Drawback**: In todo\_list.py, the tasks list is defined in the global scope. This can lead to unintended side effects if other parts of the program modify the global state.
   * **Modification**: In Todo\_listUpdated.py, the tasks list is encapsulated within the TodoList class, avoiding potential conflicts and making the state management more robust.
2. **Lack of Encapsulation**:
   * **Error/Drawback**: Functions and data are not encapsulated, leading to less modular code that is harder to maintain and extend.
   * **Modification**: By using a class (TodoList), the updated version encapsulates both data (tasks) and behavior (methods for task operations), enhancing modularity and maintainability.
3. **Scalability and Reusability**:
   * **Error/Drawback**: The procedural approach in todo\_list.py is less scalable and reusable. For example, creating multiple to-do lists would require significant changes.
   * **Modification**: The class-based structure in Todo\_listUpdated.py allows for creating multiple instances of TodoList, each with its own state, improving scalability and reusability.
4. **Code Organization and Readability**:
   * **Error/Drawback**: Mixing global variables with functions can make the code harder to follow and understand, especially as the codebase grows.
   * **Modification**: Organizing the code into a class improves readability by logically grouping related functionality together.
5. **Error Handling and Validation**:
   * **Error/Drawback**: The procedural code in todo\_list.py does not handle invalid input robustly within a cohesive structure, making error handling less centralized.
   * **Modification**: By using methods within a class, Todo\_listUpdated.py centralizes error handling and validation, making the code easier to manage and extend.

**Detailed Analysis of todo\_list.py**

**Global State Management**

* **Drawback**: The tasks list is globally defined, meaning it can be accessed and modified from anywhere in the code, leading to potential side effects and bugs.

**Lack of Encapsulation**

* **Drawback**: Functions like display\_tasks, add\_task, mark\_completed, and remove\_task operate on a global tasks list, making it difficult to isolate and test individual components.

**Scalability and Reusability**

* **Drawback**: The procedural approach is less flexible. For example, if a user wants to manage multiple to-do lists, the code would need substantial modifications.

**Code Organization and Readability**

* **Drawback**: Mixing global variables with functions makes the codebase less organized, which can become problematic as the program grows in size and complexity.

The procedural approach in todo\_list.py introduces several drawbacks, including potential issues with global state management, lack of encapsulation, reduced scalability and reusability, and less organized code. These issues are effectively addressed in Todo\_listUpdated.py by refactoring the code into an object-oriented structure. This improves modularity, maintainability, scalability, and overall code organization.

### COMPLETE TO DO LIST (UPDATED)CODE:

class TodoList:

    def \_\_init\_\_(*self*):

*self*.tasks = []

    def display\_tasks(*self*):

        if not *self*.tasks:

            print("Your to-do list is empty.")

        else:

            print("To-Do List:")

            for i, task in enumerate(*self*.tasks, *start*=1):

                status = "Done" if task["completed"] else "Not Done"

                print(f"{i}. {task['task']} ({status})")

    def add\_task(*self*, *task\_name*):

        task = {"task": *task\_name*, "completed": False}

*self*.tasks.append(task)

        print(f"Task '{*task\_name*}' added to your to-do list.")

    def mark\_completed(*self*, *task\_number*):

        if 1 <= *task\_number* <= len(*self*.tasks):

*self*.tasks[*task\_number* - 1]["completed"] = True

            print(f"Task {*task\_number*} marked as completed.")

        else:

            print("Invalid task number. Please enter a valid task number.")

    def remove\_task(*self*, *task\_number*):

        if 1 <= *task\_number* <= len(*self*.tasks):

            removed\_task = *self*.tasks.pop(*task\_number* - 1)

            print(f"Task '{removed\_task['task']}' removed from your to-do list.")

        else:

            print("Invalid task number. Please enter a valid task number.")

    def get\_valid\_task(*self*):

        task\_name = input("Enter the task: ").strip()

        while not task\_name:

            task\_name = input("Task cannot be empty or just spaces. Enter a valid task: ").strip()

        return task\_name

def main\_menu(*self*):

        while True:

            print("\nOptions:")

            print("1. Display to-do list")

            print("2. Add a task")

            print("3. Mark a task as completed")

            print("4. Remove a task")

            print("5. Quit")

            choice = input("Enter your choice: ")

            if choice == '1':

*self*.display\_tasks()

            elif choice == '2':

                task\_name = *self*.get\_valid\_task()

*self*.add\_task(task\_name)

            elif choice == '3':

*self*.display\_tasks()

                try:

                    task\_number = int(input("Enter the task number to mark as completed: "))

*self*.mark\_completed(task\_number)

                except ValueError:

                    print("Invalid input. Please enter a number.")

            elif choice == '4':

*self*.display\_tasks()

                try:

                    task\_number = int(input("Enter the task number to remove: "))

*self*.remove\_task(task\_number)

                except ValueError:

                    print("Invalid input. Please enter a number.")

            elif choice == '5':

                print("Goodbye!")

                break

            else:

                print("Invalid choice. Please enter a valid option.")

if \_\_name\_\_ == "\_\_main\_\_":

    todo\_list = TodoList()

    todo\_list.main\_menu()

### INPUT/OUTPUT SCREENSHOTS

### (UPDATED CODE)

Example Interaction

Main Menu Display

Output:

Options:

1. Display to-do list

2. Add a task

3. Mark a task as completed

4. Remove a task

5. Quit

Enter your choice:

Scenario 1: Displaying an Empty To-Do List

Input:

1

Output:

Your to-do list is empty.

Scenario 2: Adding Tasks

Input:

2

Enter the task: Buy groceries

Output:

Task 'Buy groceries' added to your to-do list.

Input:

2

Enter the task: Call the doctor

Output:

Task 'Call the doctor' added to your to-do list.

Scenario 3: Displaying the To-Do List with Tasks

Input:

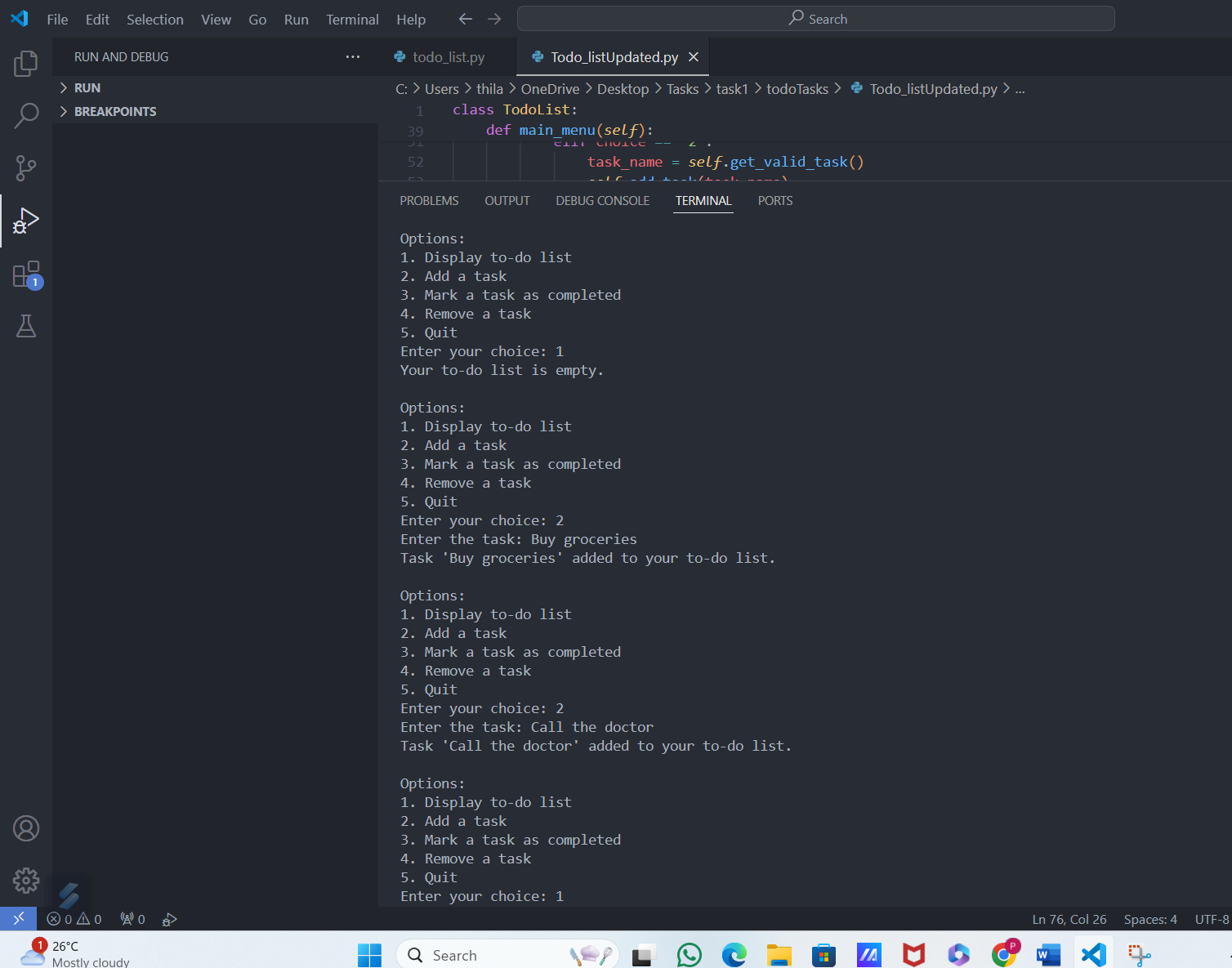
1

Output:

To-Do List:

1. Buy groceries (Not Done)

2. Call the doctor (Not Done)



Scenario 4: Marking a Task as Completed

Input:

3

Output:

To-Do List:

1. Buy groceries (Not Done)

2. Call the doctor (Not Done)

Enter the task number to mark as completed:

Input:

1

Output:

Task 1 marked as completed.

Input:

1

Output:

To-Do List:

1. Buy groceries (Done)

2. Call the doctor (Not Done)

Scenario 5: Removing a Task

Input:

4

Output:

To-Do List:

1. Buy groceries (Done)

2. Call the doctor (Not Done)

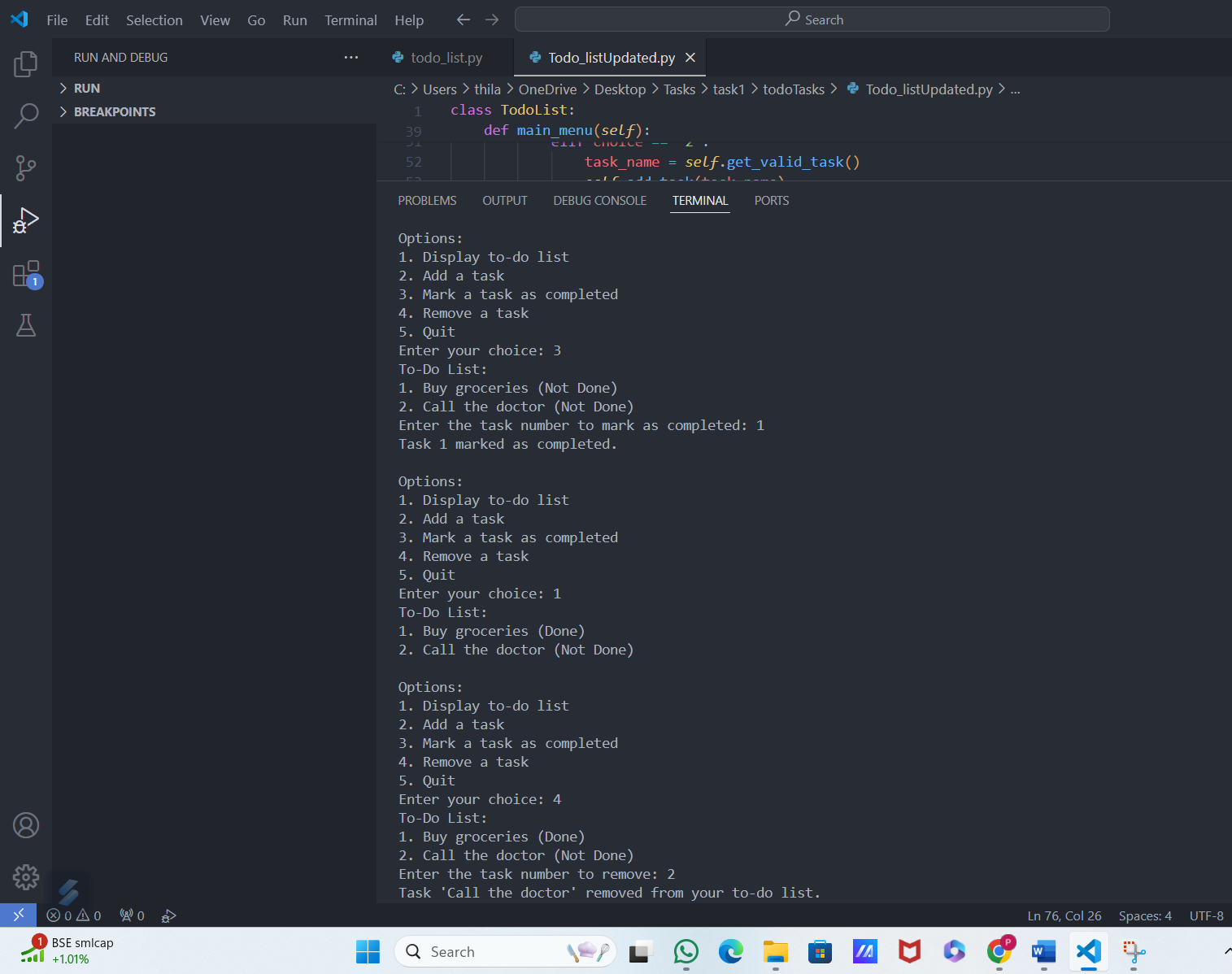
Enter the task number to remove:

Input:

2

Output:

Task 'Call the doctor' removed from your to-do list.



Input:

1

Output:

To-Do List:

1. Buy groceries (Done)

Scenario 6: Handling Invalid Input (Non-numeric)

Input:

3

Output:

To-Do List:

1. Buy groceries (Done)

Enter the task number to mark as completed:

Input:

abc

Output:

Invalid input. Please enter a number.

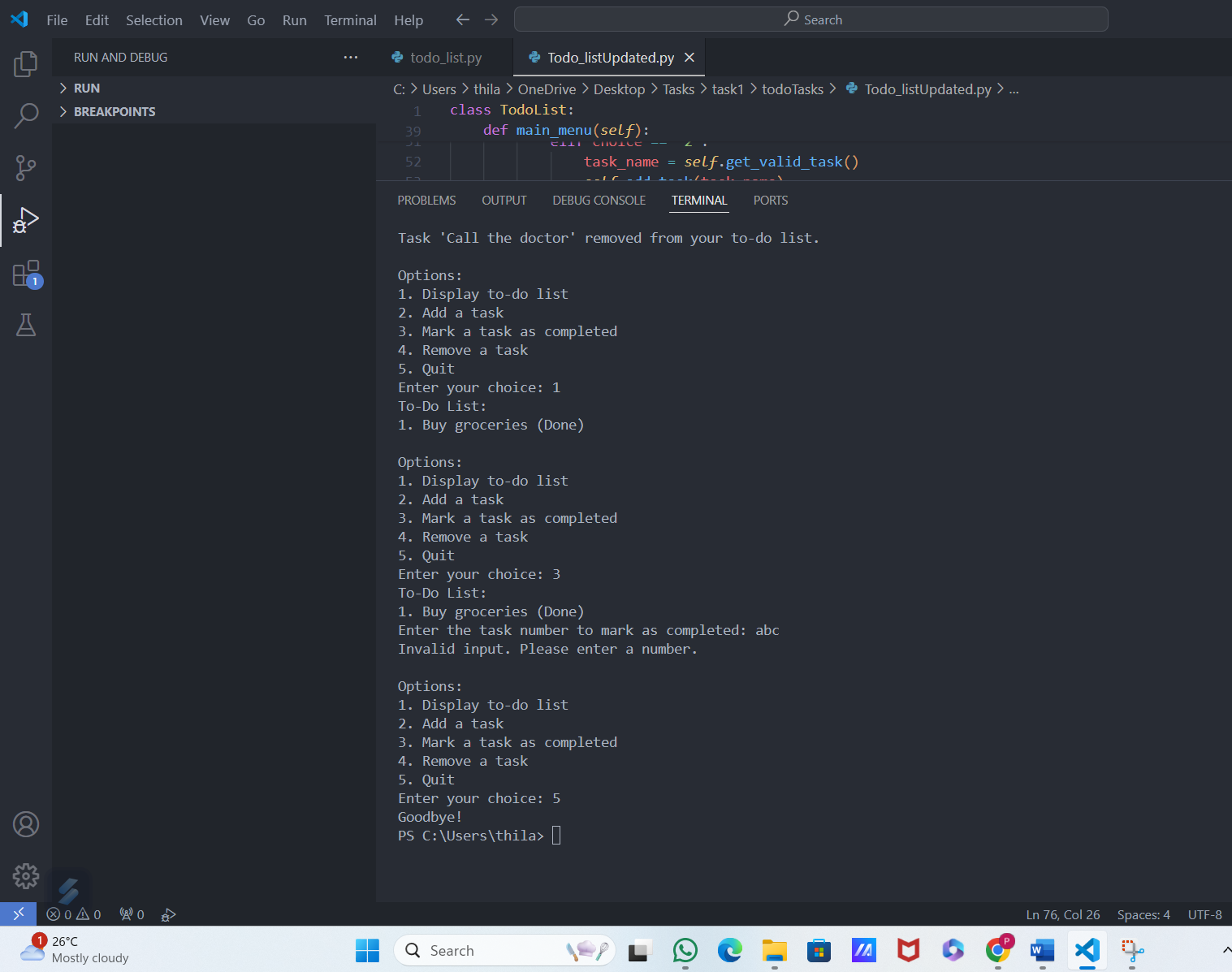
Scenario 7: Exiting the Program

Input:

5

Output:

Goodbye!



This example demonstrates the main interactions between the user and the to-do list program, illustrating how inputs are processed and how outputs are presented at each step.

### CONCLUSION

The new code is a more structured and organized version of the original to-do list program, refactored into an object-oriented format by defining a TodoList class. This encapsulation improves readability and maintainability. The class contains methods for displaying tasks, adding tasks, marking tasks as completed, and removing tasks, similar to the original functions. However, these methods now operate on self.tasks, making it clear that they belong to a specific instance of TodoList. This approach promotes better code organization and easier debugging by keeping related functionalities together within a single class.

Additionally, the new code introduces input validation and error handling improvements. The get\_valid\_task method ensures that tasks are not empty or consist solely of whitespace, enhancing user experience by preventing invalid task entries. Error handling has also been improved in the main\_menu method with try-except blocks for user input, ensuring that the program can handle invalid inputs gracefully without crashing. This added robustness makes the new code more user-friendly and resilient to incorrect user inputs. Overall, the refactoring into a class-based design and the inclusion of validation and error handling represent significant improvements over the original implementation.