**Section 1.1**  
  
  
What I Asked:  
  
Write pseudocode for a simple CLI-based task manager using basic Python techniques (lists, functions, loops, input). The user should be able to add, view, and remove tasks. Do not use any advanced concepts only what's taught in an introductory Python course.  
  
Ai Response:  
  
1. Initialize an empty list called tasks

2. Define a function display\_menu():

- Print numbered options:

1. Add task

2. List tasks

3. Remove task

4. Quit

3. Define function add\_task():

- Prompt user for task input

- Append task to tasks list

4. Define function list\_tasks():

- If list is empty: print "No tasks"

- Else: enumerate and print each task

5. Define function remove\_task():

- Prompt user for index to remove

- Convert to integer

- Use try/except to remove by index

6. Define function main():

- While True:

- Call display\_menu()

- Get user input

- Use if-elif to run appropriate function

- If choice is 4: break

7. Call main()

Course Reference:  
  
I used the planning method from Week 4 ,Planning Before Coding, which emphasized listing features, identifying inputs/outputs, and outlining structure before implementation.

**Section 1.2**

What I asked:  
  
Refine the pseudocode to include input validation using simple if statements and try/except blocks. Make sure it handles invalid input gracefully.  
  
Ai Response:

- In remove\_task():

- Before converting input to int, check if input is digit

- Add try/except to catch out-of-range errors

- In main():

- Validate menu choice is between 1–4

- Else: print "Invalid option"

Course Reference:  
  
This refinement applied input validation and defensive programming from Week 6, ensuring the program does not crash from unexpected input.  
  
- **Second Prompt Reference**

What I asked:

Refine the pseudocode again to make each action modular and reusable. Use separate functions for user input and task list display logic.

Ai Response:

- Define get\_user\_choice() to handle and validate input

- Move input() out of main() into its own function

- Use main() only as a control flow handler (calling display\_menu, get\_user\_choice, etc.)

Couse Reference:  
  
This refinement applied function decomposition from Week 5, Function Design Principles separating logic into small reusable parts.

**Section 1.3**

I used ideas from Week 6, Defensive Programming and Week 5, Function Design in my refinements. At the start, the pseudocode focused on how the program was to be structured, with each action listed one after another. Still, it did not have formal acceptance and did not use modules as language constructs.

To make the first refinement, I added input checks and exception blocks, so only safe inputs were used for task deletion by index. The change shows that week 6 emphasizes action taken to prevent invalid user activity. I rearranged the program in the second edition by isolating repeated prompts for input into dedicated functions which aligns with Week 5’s advice to simplify the main loop and ensure each function has a clear purpose.

Both changes focus on Week 4’s message to plan before writing code and the simple control flow ideas we developed with labs. Evolving the pseudocode using Weeks 4–6’s lessons, I was left with a streamlined plan for coding, effective input handling and modular programming.

**Section 2.1**

Here is some buggy Python code for a task manager. Can you identify the errors using only techniques from a beginner course (like print debugging)? Do not use any advanced tools.

# broken\_task\_manager.py

tasks = []

def add\_task(task):

tasks.append(task)

print(f"Added: {task}")

def remove\_task(index):

if index < len(tas): # Error 1

removed = tasks[index]

del tasks[index]

print(f"Removed: {removed}")

else:

print("Invalid index!")

def list\_tasks():

if not tasks:

print("No tasks available.")

else:

for i, t in enumerate(task): # Error 2

print(f"{i+1}: {t}")

def main():

add\_task("Buy milk")

add\_task("Pay bills")

add\_task("Walk dog")

list\_tasks()

remove\_task(1)

list\_task() # Error 3

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

main()

**Ai Response:**

1. Line: `if index < len(tas):` → `tas` is undefined (should be `tasks`)

2. Line: `for i, t in enumerate(task):` → `task` is undefined (should be `tasks`)

3. Line: `list\_task()` → Typo, function is called `list\_tasks()`

**Course Reference:**  
  
I applied the debugging approach from Week 6, Debugging Practice, using print-based testing and visual inspection to trace common syntax, logic, and naming errors.

Section 2.2  
  
What I wrote:   
  
Please give me a corrected version of the above code using only basic functions, lists, loops, and input/output. Include doctest examples for each function using beginner-level format.  
  
**Ai Response:**  
  
Cleaned-up version with proper naming and doctest support.

Final Corrected Code (Manual Rewrite)

tasks = [] # list to store tasks

def add\_task(task):

"""

Add a task to the task list.

>>> tasks.clear()

>>> add\_task("Test task")

Added: Test task

"""

tasks.append(task)

print(f"Added: {task}")

def remove\_task(index):

"""

Remove a task by its index in the list.

>>> tasks[:] = ['A', 'B', 'C']

>>> remove\_task(1)

Removed: B

>>> remove\_task(5)

Invalid index!

"""

if 0 <= index < len(tasks):

removed = tasks[index]

del tasks[index]

print(f"Removed: {removed}")

else:

print("Invalid index!")

def list\_tasks():

"""

List all tasks currently in memory.

>>> tasks[:] = ['X', 'Y']

>>> list\_tasks() # doctest: +SKIP

1: X

2: Y

"""

if not tasks:

print("No tasks available.")

else:

for i, t in enumerate(tasks):

print(f"{i+1}: {t}")

def main():

add\_task("Buy milk")

add\_task("Pay bills")

add\_task("Walk dog")

list\_tasks()

remove\_task(1)

list\_tasks()

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

import doctest

doctest.testmod(verbose=True)

main()

Fixes and Issues the Code Had

| Issue | Fix | Concept |
| --- | --- | --- |
| tas typo | changed to tasks | Week 6 variable tracing |
| enumerate(task) | changed to enumerate(tasks) | Week 4 loops |
| list\_task() | corrected to list\_tasks() | Week 5 function naming |
| No error handling | added if index in range | Week 6 defensive programming |
| No tests | added doctest examples | Week 9 testing practices |

**Course Reference:**My error handling approach comes from Week 6Defensive Programming.

My testing approach follows Week 9, Doctest Examples for Validation.

**Section 3.1**

**What I Asked AI:**  
  
Review this function using only error handling techniques taught in an introductory programming course (basic try/except only). What problems do you see in this version?  
  
**Problems Identified:**  
  
Problems Identified in safe\_weather\_data\_fetch(city):

No input validation:

If city is an empty string, the request still runs and may return unclear errors.

Solution from Week 6: Use if not city: print(...)

Bare except block:

Hides all exceptions and gives generic "Error occurred" message.

Week 8 allows basic try/except, but even a message like "Could not connect" is better.

No handling for missing keys:

Assumes data['current\_condition'][0] is always present.

Could throw a KeyError or IndexError.

Week 8 solution: nest another try/except for safe data extraction.

**Course Reference:**  
  
The approach uses Week 6 and Week 8 techniques.

- Validating the user’s input (in Week 6)

- Learning how to deal with exceptions (Week 8)

- Defensive programming principles that we worked on in Week 6

**Section 3.3**  
  
 **Similarities:**

- Both versions use basic try/except blocks for error handling

- Both validate the city input and return dictionaries or None

**Differences:**

- My version prints clearer error messages instead of returning generic strings

- The ideal version has slightly more structured logic (e.g., separated inner try), mine keeps it minimal for Week 8 clarity

**Course Connection:**

My version best demonstrates the \*\*basic exception handling pattern from Week 8\*\*, avoiding advanced structures or exception types.

**Improvement Opportunity:**

After reviewing Chapter 6, I’d improve error feedback using more specific messages or maybe a status code dictionary for user-facing output.

**Section 4**

Details on What Is Part of This Implementation:

* In Basic Lists (Week 4), tasks are temporarily stored in a global tasks list inside the code.
* Each primary action in the program, for example adding, listing and removing tasks, is handled separately as a function (Week 5 with Modular Function Design).
* While Loops (Week 5): The main loop of the program displays the menu and follows the user’s commands until “Quit” is selected.
* Using input() and print() let users and the computer communicate simply in a beginner-level Command Line Interface (CLI).
* I checked for correct input and valid keys in the program so crashes could be avoided, by using both if statements and try/except blocks (Week 6).
* Doctest-Based Testing (Week 9): Embedded doctests are in each function, showing the correct usage and covering testing wherever possible.
* Following the rules of the class: Students cannot use advanced libraries, must write in a non-object-oriented manner and all logic covered within introductory programming.

**Section 4.2**

Proper Balance Between Daily Workflow and Course Learning:

While working on this project, I first used the approach I learned in Week 4 by creating pseudocode and then made it better with input from AI. When necessary, I updated the AI answers to make sure they went along with the course requirements. I applied Colab to carry out checks step by step while our class approach of making small functions like `display\_menu()` and `get\_user\_choice()` made the loop simpler and easier to troubleshoot.

Use of Course Concept in Practice:

I used what I learned in Week 6 about error handling when writing `remove\_task()`. Defensive Programming helped me add conditions that prevent the program from giving errors when the task list is empty. Most of these cast minor problems aside during the actual run. I used `doctest` from Week 9, inserting fast-forward (`+SKIP`) with functions that rely on user input and verifying changes to the task with assertions.

Using Skills Elsewhere:

In the end, I will rely on Iterative Refinement to create testable segments from the complex logic. The feedback I got from Assignment 2 stated that I wasn’t checking the input carefully enough. Using lessons from Week 6, I will now write checks so that the next actions are done safely and responsibly. Adding tests as soon as I start a feature will become a priority.

**Section 4 Summary:**  
  
In Section 4, I made the CLI Task Manager on my own with tools from Weeks 4 to 6 such as basic lists, while loops, if-else logic and small user input/output operations. Each created function—display\_menu(), get\_user\_choice() and main()—follows the rules shown by the course and the examples for doctest in Week 9 are included. Input functions are not tested using # doctest since they need to be evaluated differently. It includes defensive programming to confirm any user input is correct and not remove tasks that do not exist. In addition, I reflected on my coursework using course concepts such as “Planning Before Coding” (Week 4), “Function Design” (Week 5), “Error Handling Mindset” (Week 6) and tested in practice some of what was learned in Week 9. I integrate these concepts with the way I manage my work, set up projects and look forward to enhancements. All in all, this part of the report (section) shows I can put together course concepts, use core programming skills and maintain clear records that meet all testing, debugging and transparency requirements set by Section 4.