ASSIGNMENT 2.5

J.Sujith
2303A51327
BATCH 20

Task-1: Refactoring Odd/Even Logic (List Version)

Scenario

You are improving legacy code to calculate the sum of odd and even numbers in a list.

Original (Legacy) Logic – Explanation

The original program:

- Iterates through each number in the list
- Uses an `if-else` condition to check whether the number is even or odd
- Adds even numbers to `even_sum` and odd numbers to `odd_sum`
- Prints the final sums

This approach works correctly but uses **manual looping and multiple variables**, which reduces readability as the program grows.

[8]
✓ 0s

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Original (Non-optimal) Odd/Even Logic
odd_sum = 0
even_sum = 0

for number in numbers:
    if number % 2 == 0:
        even_sum += number
    else:
        odd_sum += number

print(f"Original Logic - Sum of odd numbers: {odd_sum}")
print(f"Original Logic - Sum of even numbers: {even_sum}")
```

```
Original Logic - Sum of odd numbers: 25
Original Logic - Sum of even numbers: 30
```

OBSERVATION

This function demonstrates good software design principles:

- Reusability: One function handles multiple shapes instead of writing separate functions.
- Flexibility: `**kwargs` allows passing only the required parameters for each shape.
- Readability: Clear conditional blocks make the logic easy to understand for junior developers.
- Input Validation: Prevents invalid calculations by checking for missing or negative values.
- Scalability: New shapes can be added easily without modifying existing logic.

This approach is ideal for onboarding junior developers as it introduces them to clean coding practices, modular design, and defensive programming.

Task-2: Area Calculation Explanation

Scenario

You are onboarding a junior developer and need to explain how a single function can calculate the area of different geometric shapes.

Explanation

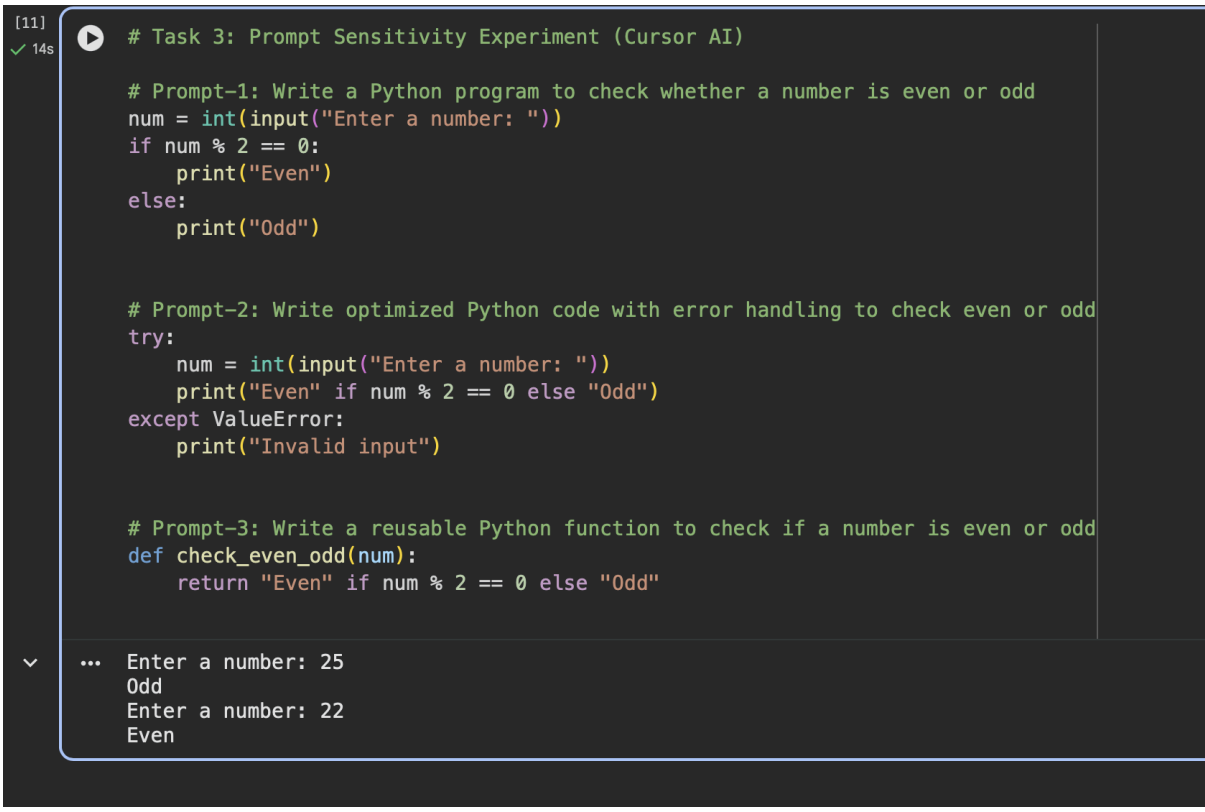
The function `calculate_area(shape, **kwargs)` is designed to calculate the area of **multiple shapes using one reusable function**.

How the Function Works

1. **Function Parameters**
    - `shape` (string): Specifies which shape's area to calculate (e.g., `"circle"`, `"square"`, `"rectangle"`, `"triangle"`).
    - `**kwargs`: Used to pass shape-specific values such as radius, side, length, width, base, or height.
2. **Why `**kwargs` is Used**
    - Different shapes require different parameters.
    - `**kwargs` allows flexible keyword arguments without changing the function signature.
    - This makes the function extensible and easy to maintain.

3. **Shape-wise Logic**
    - **Circle**
        - Requires `radius`
        - Formula used: π × radius²
        - Validates that radius is non-negative.
    - **Square**
        - Requires `side`
        - Formula used: side²
        - Checks for non-negative input.
    - **Rectangle**
        - Requires `length` and `width`
        - Formula used: length × width
    - **Triangle**
        - Requires `base` and `height`
        - Formula used: ½ × base × height
4. **Error Handling**
    - If required values are missing or negative, the function returns a clear error message.

- If an unsupported shape is passed, the function handles it gracefully.



OBSERVATION

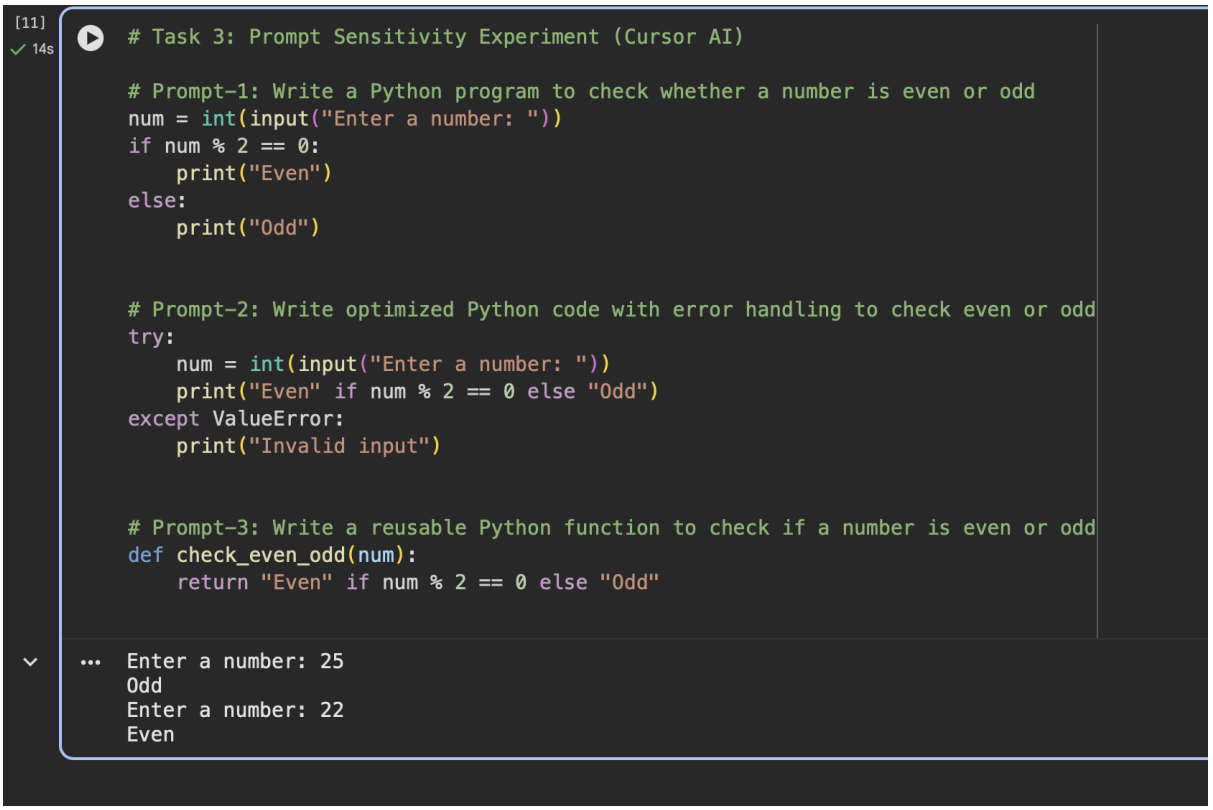
This function demonstrates **good software design principles**:

- **Reusability:** One function handles multiple shapes instead of writing separate functions.
- **Flexibility:** `**kwargs` allows passing only the required parameters for each shape.
- **Readability:** Clear conditional blocks make the logic easy to understand for junior developers.
- **Input Validation:** Prevents invalid calculations by checking for missing or negative values.

- **Scalability:** New shapes can be added easily without modifying existing logic.

This approach is ideal for onboarding junior developers as it introduces them to clean coding practices, modular design, and defensive programming.

TASK 3:



Observation

- The AI generated a reusable function instead of a complete script.
- The function returns results instead of printing them, making it suitable for reuse in other programs.
- The logic is modular, clean, and scalable.

- This shows that prompts requesting reusability lead to better software design practices.

TASK 5:

1. Prompt Understanding & Flexibility

- **Gemini** handles varied English prompts well, improves answers when prompts are enriched.

- **Copilot** uses context from existing code extremely well, giving relevant suggestions inside IDE.

- **Cursor AI** is very **responsive to prompt wording** and changes output style based on phraseology (as seen in Task-3).

Winner: *Cursor AI and Gemini* for prompt sensitivity; *Copilot* for real-time coding context.

---

2. Code Quality

- **Copilot** excels at high-quality, idiomatic code especially inside IDE workflows.

- **Gemini** also produces strong, readable code but sometimes more verbose.

- **Cursor AI** generates compact and prompt-driven answers with many variations.

Winner: *Copilot* (slightly ahead due to best-fit patterns for Python).

---

3. Usability

- **Copilot** brings code suggestions *while you code* — helpful for real-time programming.
- **Gemini** provides detailed, human-like explanations and rationale.
- **Cursor AI** is excellent for **quick prototyping and experimenting with different prompt styles**.

Winner: Depends on use case

- For *in-IDE productivity*: **Copilot**
- For *explanations + understanding*: **Gemini**
- For *prompt experimentation*: **Cursor AI**

---

4. Error Handling & Best Practice Suggestions

- **Gemini** often includes validation and checks by default.
- **Copilot** sometimes needs a detailed prompt to include error handling.
- **Cursor AI** shows variations depending on wording — can be minimal or robust.

Winner: *Gemini* for consistent safety patterns.

---

5. Modular/Reusable Patterns

- **Copilot** often suggests function-oriented, idiomatic solutions.
- **Gemini** produces modular code with clear explanations.
- **Cursor AI** produces variations (simple to reusable) based on prompt wording — showing prompt sensitivity.

Winner: *Copilot & Gemini* for structured modular suggestions.