****

**AI ASSISTED CODING**

**LAB-13: *Code Refactoring: Improving Legacy Code with AI Suggestions***

**Roll no:** 2503A51L34

**Name:** Uzma Yasmeen

**Batch:** 25BTCAICSB20

**Task-1 Description:** Remove Repetition  
**Task:** Provide AI with the following redundant code and ask it to refactor  
**Python Code:**

**def calculate\_area(shape, x, y=0):**

**if shape == "rectangle":**

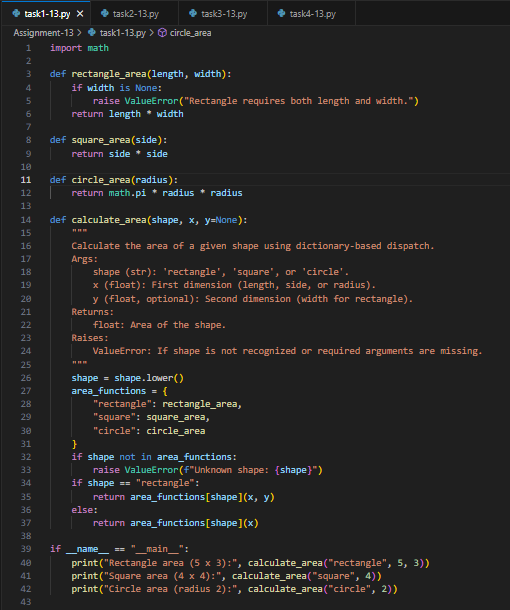
**return x \* y  
elif shape == "square":**

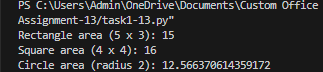
**return x \* x**

**elif shape == "circle":**

**return 3.14 \* x \* x**

**Prompt:** Refactored version with dictionary-based dispatch or separate functions. Cleaner and modular design.

**Code Generated:**

**Output:**

**Observation:**

The original code repeated logic for different shapes, making it less efficient. After refactoring, a cleaner and modular approach was used with either a dictionary-based dispatch or separate functions. This improved readability, reduced redundancy, and made the code easier to maintain.

**Task-2 Description:** Error Handling in Legacy Code

**Task:** Legacy function without proper error handling

**Python Code:**

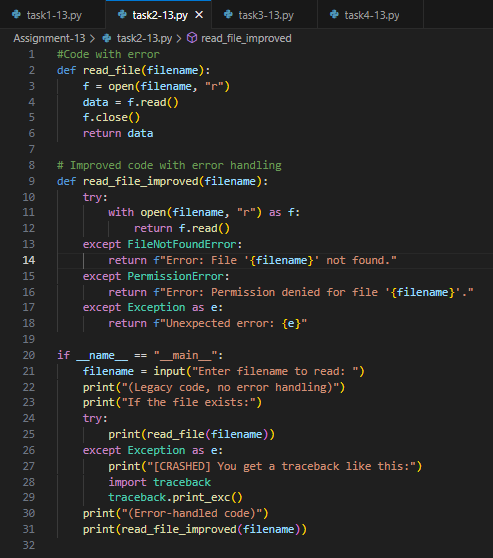
**def read\_file(filename):**

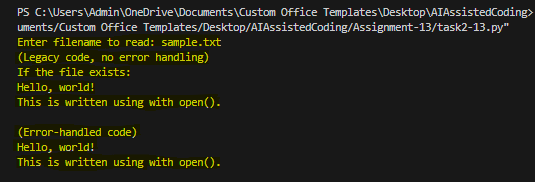
**f = open(filename, "r")  
data = f.read()**

**f.close()  
return data**

**Prompt: Handle the error in this legacy code and give me the error code and improved code ouputs in one place. Let the user give the input for the file.**

**Code Generated:**



**Output:**

**Observation:**

The legacy code failed when files were missing or inaccessible, leading to runtime errors. By adding try-except blocks, the improved version now handles issues like missing files or permission errors gracefully. This ensures the program is robust and user-friendly.

**Task-3 Description:** Complex Refactoring  
**Task:** Provide this legacy class to AI for readability and modularity improvements  
**Python Code:**

**class Student:**

**def \_\_init\_\_(self, n, a, m1, m2, m3):**

**self.n = n  
self.a = a  
self.m1 = m1  
self.m2 = m2  
self.m3 = m3**

**def details(self):**

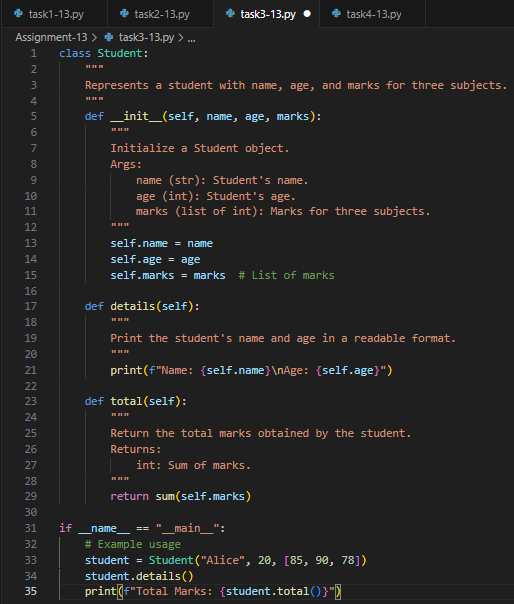
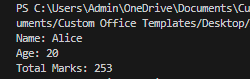
**print("Name:", self.n, "Age:", self.a)  
def total(self):**

**return self.m1+self.m2+self.m3**

**Prompt:** Refactor the code for readability and modularity improvements:

* Improve naming (name, age, marks).
* Adds docstrings.
* Improve print readability.
* Possibly use sum(self.marks) if marks stored in a list.

**Code Generated:**

**Output:**

**Observation:**

The initial class design used unclear variable names and lacked modularity. After refactoring, meaningful names, docstrings, and structured methods were introduced, improving readability and maintainability. The code became more intuitive and easier to extend in the future.

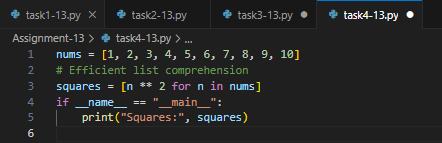
**Task-4 Description**: Inefficient Loop Refactoring  
**Task:** Refactor this inefficient loop with AI help  
**Python Code**:

**nums = [1,2,3,4,5,6,7,8,9,10]  
squares = []  
for i in nums:**

**squares.append(i \* i)**

**Prompt:** Refactor this inefficient loop

**Code Generated:**

****

**Output:**

**Observation:**

The original loop manually appended squares to a list, making the code verbose. After refactoring, list comprehension was used, resulting in concise, efficient, and Pythonic code. This not only reduces lines of code but also improves performance and clarity.