**Task Description #1 – 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

**Expected Output**

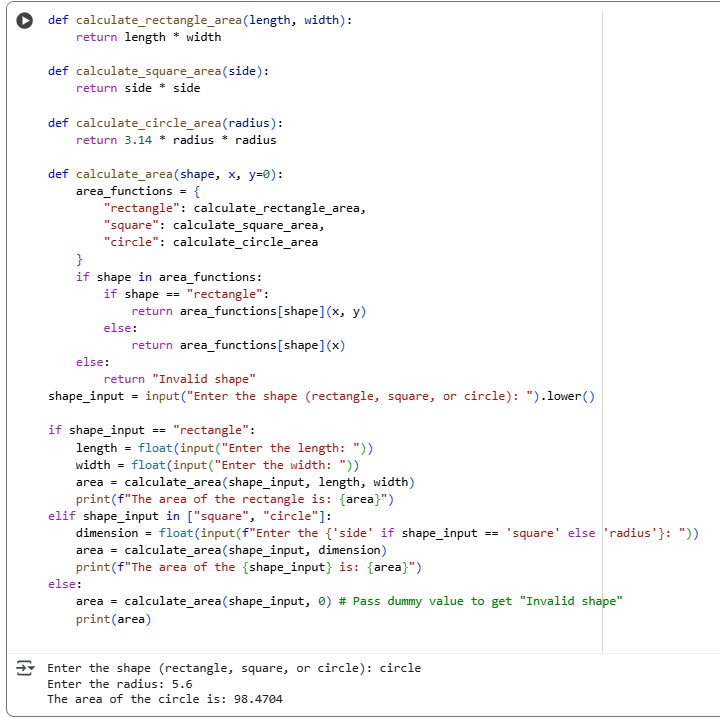
* Refactored version with dictionary-based dispatch or separate functions.
* Cleaner and modular design.

PROMPT:

Generate Refactored version with dictionary-based dispatch or separate functions.

MY OBSERVATION:

This code defines separate functions to calculate the area of a rectangle, square, and circle, organizing the logic cleanly. It uses a dictionary to map shape names to their corresponding area functions, which makes the calculate\_area function concise and extensible. However, the calculate\_area function still contains a conditional to handle the difference in required arguments between a rectangle (which needs length and width) and the other shapes (which only need one dimension). The user input section effectively prompts for the necessary dimensions based on the chosen shape and then calls calculate\_area accordingly, printing the result. The code also handles invalid shape inputs gracefully by returning an "Invalid shape" message.



**Task Description #2 – 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

**Expected Output:**

AI refactors with with open() and try-except:

PROMPT:

Refactor the code with open() and try-except



MY OBSERVATION:

The read\_file function is well-designed with proper error handling using a try-except block. It attempts to open and read the content of the specified file using a context manager (with open), which ensures the file is properly closed after reading. If the file is not found, it catches the FileNotFoundError and returns a clear, user-friendly error message indicating the missing file. Additionally, it catches any other unexpected exceptions, returning a generic error message with the exception details. This makes the function robust and safe to use, as it won’t crash the program if an error occurs while reading the file.

**Task Description #3 – 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

**Expected Output:**

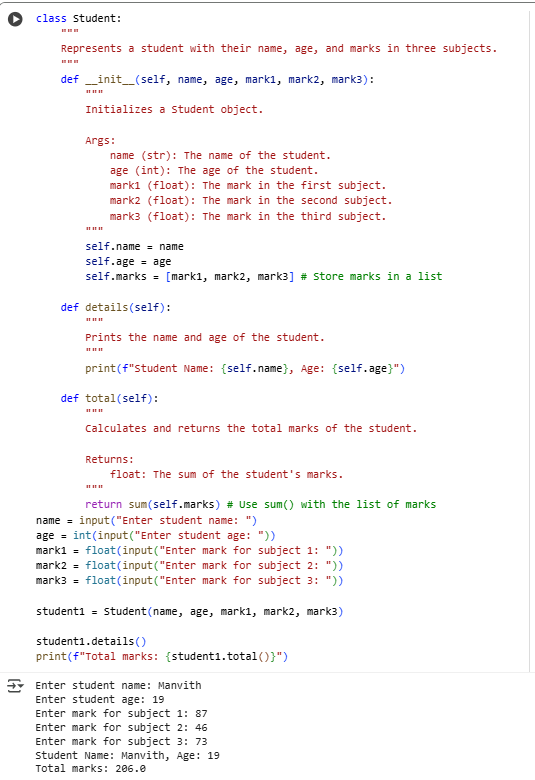
* AI improves naming (name, age, marks).
* Adds docstrings.
* Improves print readability.
* Possibly uses sum(self.marks) if marks stored in a list.

PROMPT:

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

MY OBSERVATION:

The Student class is a straightforward and well-structured representation of a student’s basic information, including their name, age, and marks in three subjects. The class uses an initializer method to set these attributes, storing the three marks conveniently in a list for easy manipulation. It provides a details method to print the student's name and age, and a total method that calculates and returns the sum of the marks using Python’s built-in sum() function. The code also includes user input prompts to create an instance of the Student class dynamically, demonstrating practical use. Overall, the design is clear, modular, and easy to extend or reuse for managing student data.



**Task Description #4 – 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)

**Expected Output:** AI suggested a **list comprehension**

**MY OBSERVATION:**

This code uses a **list comprehension** to create a new list called squares by squaring each number in the original list nums. It’s a concise and efficient way to transform each element, producing [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]. List comprehensions like this are a Pythonic approach to generating lists without the need for explicit loops, making the code clean and easy to read.

