# Assignment: 13.2

NAME: DARIPELLI SRIYA

BATCH: 01

HALL NO:2403A51350

COURSE: ai assistant coding

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

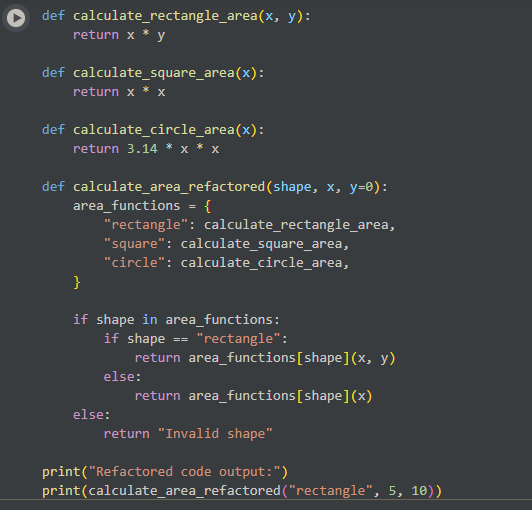
Expected Output  
• Refactored version with dictionary-based dispatch or separate  
functions.  
• Cleaner and modular design.

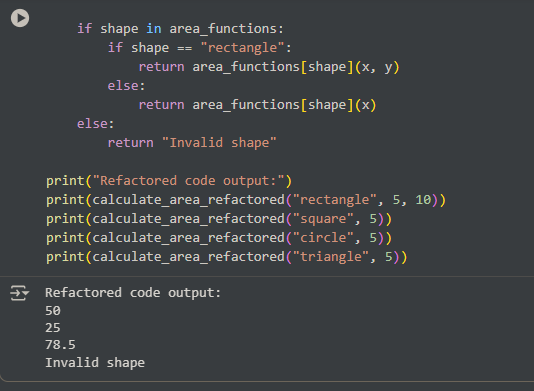
# Prompt:

Generate refactored code for

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

# SCREENSHOTS:





# OBSERVATION:

It is observed that the code is written in more understandable and with more functions than before also without changing the originality of the code

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:

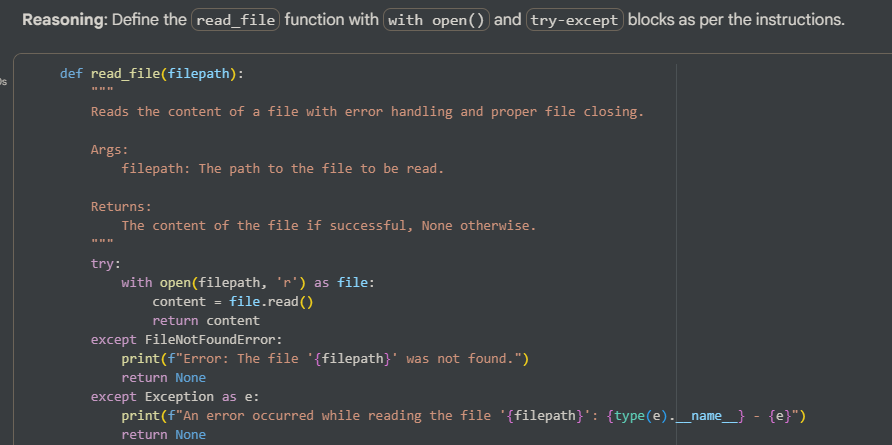
# prompt:

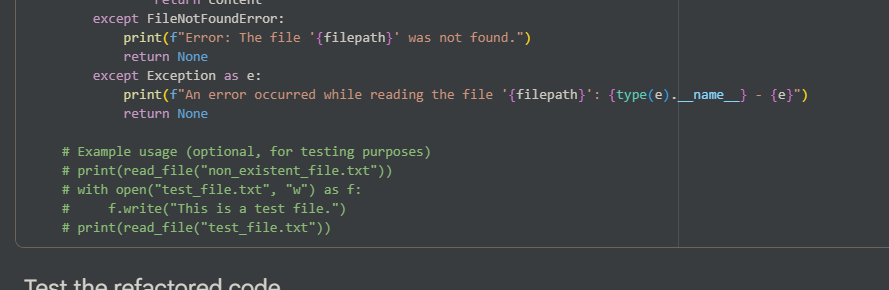
generate code with legacy function without proper error handling

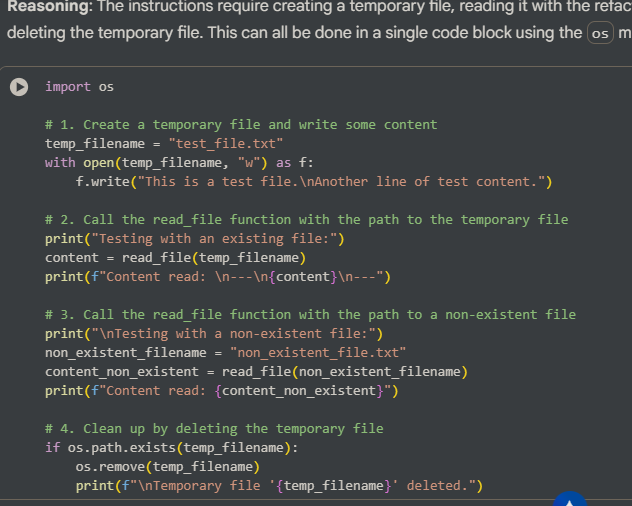
def read\_file(filename):  
f = open(filename, "r")  
data = f.read()

f.close()  
return data

# screenshots:







# Observation:

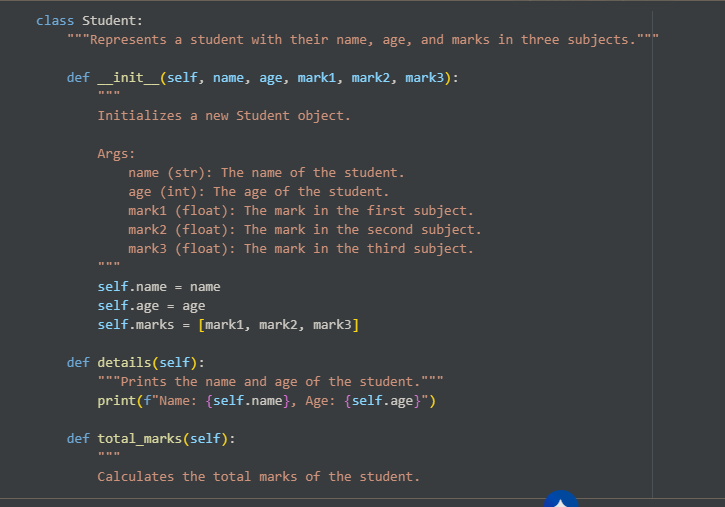
* The refactored read\_file function, incorporating with open() and try-except, was successfully implemented.
* Testing with an existing file confirmed that the function correctly read the file content.
* Testing with a non-existent file showed that the function successfully caught the FileNotFoundError, printed an informative message, and returned None, demonstrating robust error handling.

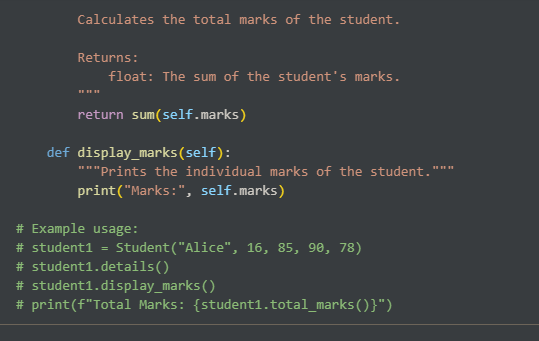
Task Description #3 – Complex Refactoring  
Task: Provide this legacy class to AI for readability and modularity  
improvements:  
Python Code  
class Student:  
def \_\_init\_\_(self, n, a, m1, m2, m3):  
self.n = n  
self.a = a  
self.m1 = m1  
self.m2 = m2  
self.m3 = m3  
def details(self):  
print("Name:", 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:

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

# Screenshots:





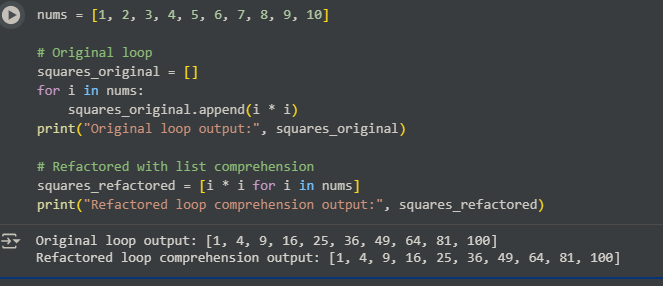
# Observation:

* **Improved Naming:** The variable names are now more descriptive (name, age, marks) instead of single letters (n, a, m1, m2, m3).
* **Docstrings:** Docstrings have been added to the class and its methods, explaining their purpose, arguments, and return values. This significantly improves code documentation.
* **Improved Print Readability:** The details method now uses an f-string for cleaner and more readable output.
* **Marks in a List:** The marks are now stored in a list (self.marks), which makes it easier to manage and perform operations on them (like calculating the total using sum()). A new method display\_marks was added to show the individual marks.
* **Clearer Total Calculation:** The total method was renamed to total\_marks and uses the built-in sum() function on the marks list, which is more concise and readable.

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

Prompt: Refactor this inefficient loop   
nums = [1,2,3,4,5,6,7,8,9,10]  
squares = []  
for i in nums:  
squares.append(i \* i)

# Screenshots:



# Observation:

The key observation here is the difference in the code structure. The original code uses a multi-line for loop with an explicit append call, while the refactored code achieves the same result in a single line using a list comprehension [i \* i for i in nums]. This demonstrates the conciseness and readability that list comprehensions offer for creating lists based on existing iterables.