AI ASSISTED CODING

Name:Saniya

HT.NO:2403A510E7

Batch:05

LAB ASSIGNMENT :13.3

TASK1:

PROMPT:

Refactor the following Python function to remove repetition and improve modularity. Use either a dictionarybased dispatch or separate functions to make the code cleaner and more maintainable.

Python Code(BEFORE)

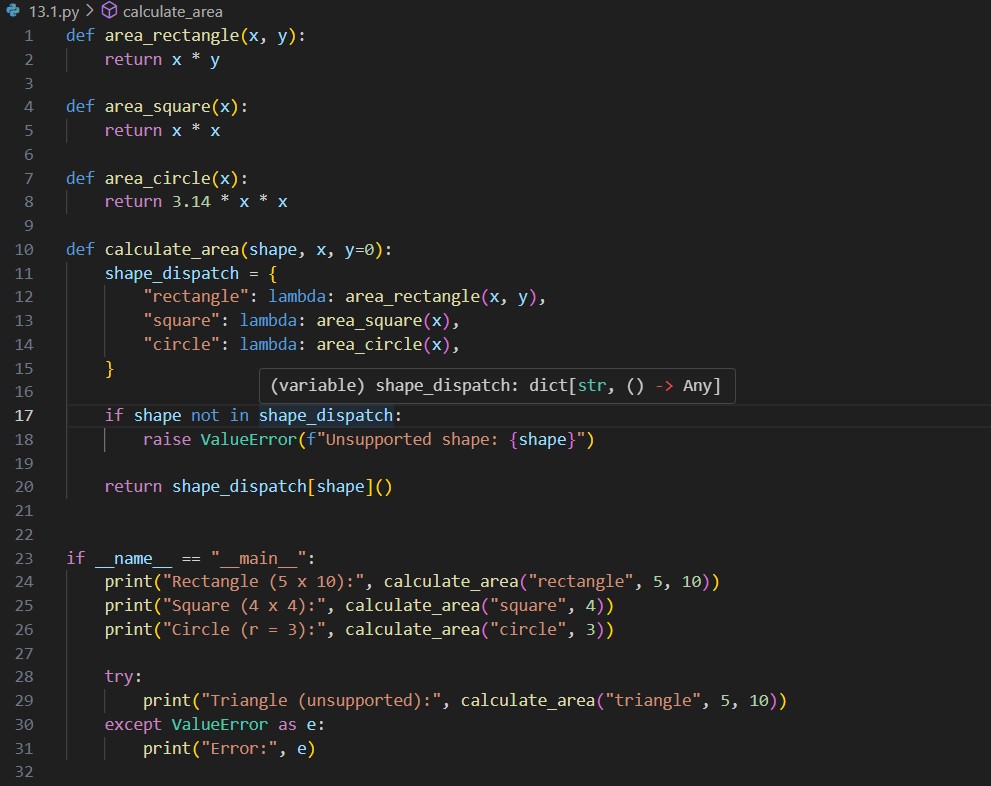
def calculate\_area(shape, x, y=0):

if shape == "rectangle":

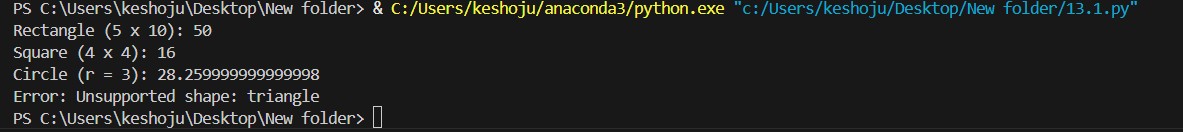
return x \* y elif shape == "square":

return x \* x elif shape == "circle": return 3.14 \* x \* x

CODE GENERATED:



OUTPUT:



EXPLANATION:

* The refactored Python code for calculating the area of different shapes—rectangle, square, and circle—uses a cleaner and more modular approach compared to the original version. Instead of handling all logic within a single function using repetitive if-elif statements, the code defines separate functions for each shape

(area\_rectangle, area\_square, and area\_circle).

* These functions encapsulate the formula for each shape’s area, making the code more organized and easier to maintain. A dictionary called shape\_dispatch is then used to map each shape name to its corresponding function, using lambda expressions to pass arguments.
* This technique, known as dictionary-based dispatch, replaces conditional statements and simplifies the control flow. The calculate\_area function checks if the input shape is valid and raises a ValueError if it isn't, providing clear error handling. If the shape is supported, it calls the appropriate function via the dictionary. Additionally, the script includes test cases using print() statements to demonstrate the function's usage and outputs, along with a try-except block to handle unsupported shapes.
* Overall, this refactored version is more scalable, readable, and better structured for future modifications or extensions.

TASK2:

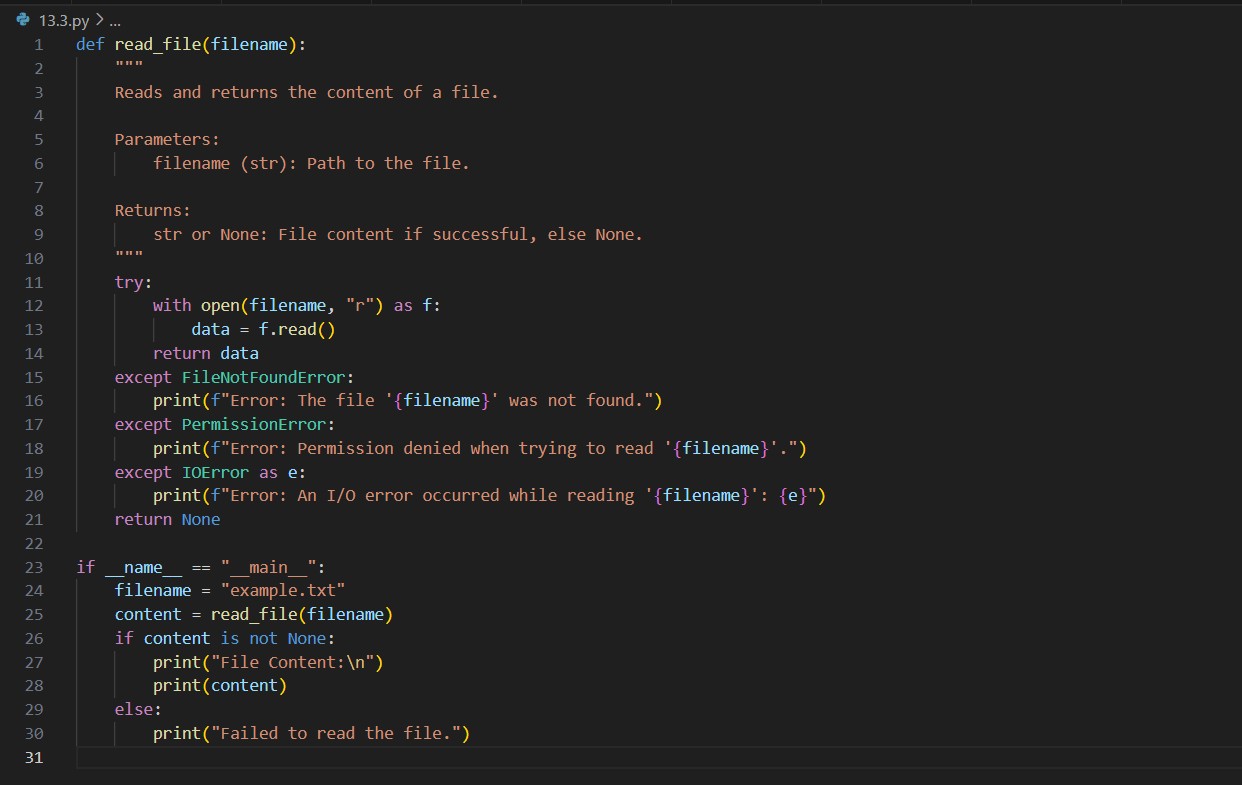
PROMPT:

Refactor the following legacy Python function to include proper error handling and use the with open() context manager instead of manually opening and closing the file.

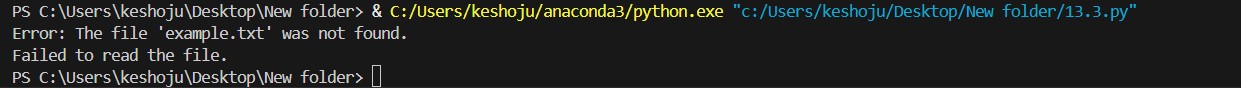
Python Code(BEFORE) def read\_file(filename): f = open(filename, "r") data = f.read()

f.close() return data

CODE GENERATED:



OUTPUT:



EXPLANATION:

The read\_file function is designed to safely read the contents of a file specified by its filename. It uses the with open() context manager to ensure the file is properly closed after reading, even if an error occurs. The function includes try-except blocks to handle common file-related errors gracefully, such as when the file is not found

(FileNotFoundError), permission is denied

(PermissionError), or other I/O errors occur (IOError). If the file is successfully read, the function returns its contents as a string. Otherwise, it prints a clear error message and returns None. The script also contains an example usage section that attempts to read a file called "example.txt", printing its contents if successful or informing the user if reading the file failed. This approach improves reliability and user feedback compared to the original version that lacked error handling.

TASK 3:

PROMPT:

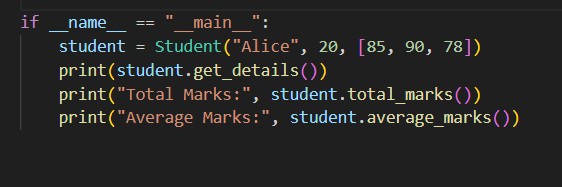
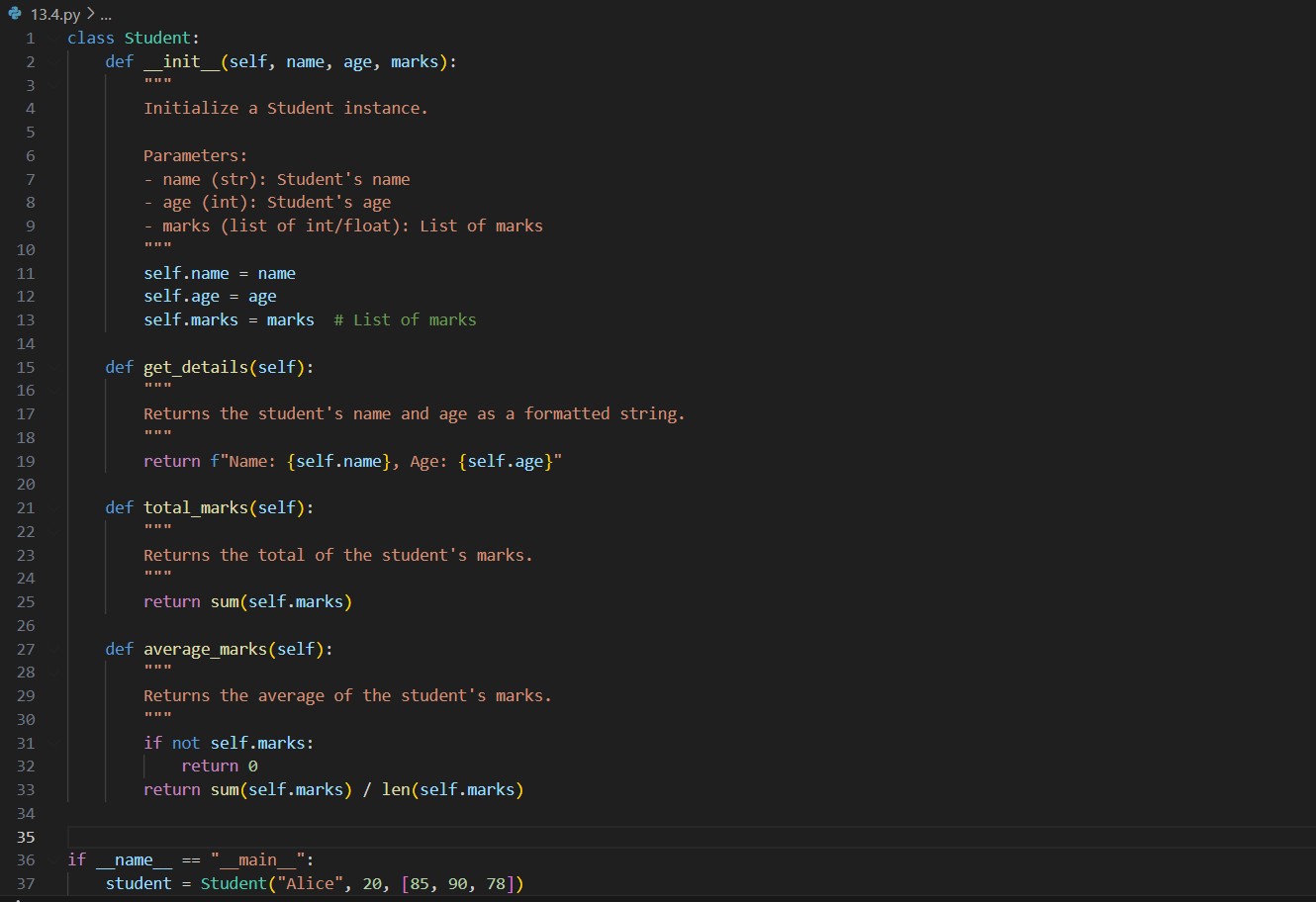
Refactor the following legacy Python class to improve readability, modularity, and maintainability. The class stores a student’s name, age, and three marks, and calculates the total of the marks.

Python Code(BEFORE) class Student: def \_\_init\_\_(self, n, a, m1, m2, m3):

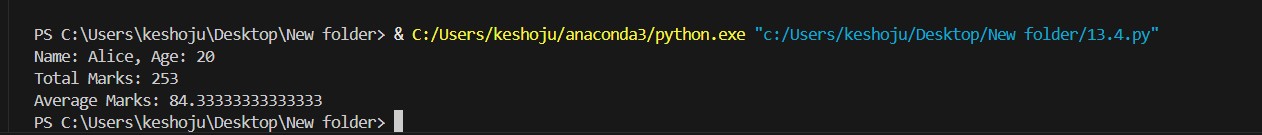
self.n = n self.a = a self.m1 = m1 self.m2 = m2 self.m3 = m3 def details(self):

print("Name:", self.n, "Age:", self.a) def total(self): return self.m1+self.m2+self.m3

CODE GENERATED:



OUTPUT:



EXPLANATION:

* The refactored Student class is designed to be more readable, modular, and maintainable than the original version. It uses clear and descriptive variable names such as name, age, and marks, replacing the vague n, a, m1, m2, and m3. Instead of storing marks in separate variables, it uses a **list of marks**, making the class more flexible and scalable — you can now store any number of marks, not just three.
* The constructor (\_\_init\_\_) takes three parameters: the student's name, age, and a list of marks. The get\_details() method returns a formatted string with the student's name and age instead of printing it directly, which makes the method more reusable in different contexts (e.g., logging, web apps, etc.).
* The total\_marks() method calculates the total by summing the list of marks, and the average\_marks() method returns the average. It also includes a check to avoid division by zero if the list of marks is empty.
* This version separates data storage, computation, and presentation, making the class **easier to extend**, **test**, and **integrate** with other parts of a program. It's now better aligned with object-oriented best practices.

TASK 4:

PROMPT:

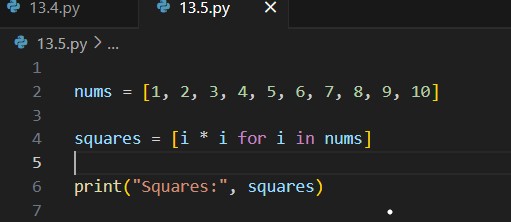
Refactor the following Python code to improve its efficiency and readability. The current version uses a for loop to compute the squares of numbers in a list.

Python Code(BEFORE)

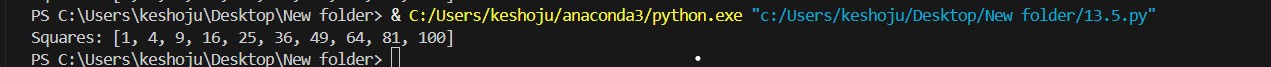
nums = [1,2,3,4,5,6,7,8,9,10] squares = [] for i in nums:

squares.append(i \* i)

CODE GENERATED:



OUTPUT:



EXPLANATION:

The refactored code uses a list comprehension to generate a list of square numbers from an existing list of integers. Instead of using a traditional for loop with append(), the list comprehension [i \* i for i in nums] creates the new list in a single, readable line. This approach is more concise, improves performance, and is considered more Pythonic. It clearly expresses the intent of the code—squaring each number in the list— while reducing boilerplate and enhancing maintainability.