AI ASSISTED CODING

Program :B.tech(CSE)

Name of Student : ANANTHA MANIDEEP

Enrollment No. :2403A52078

Batch No. :02

Date :**28**/10/2025

LAB ASSIGNMENT-13.3:

➢ TASK DESCRIPTION-1:

Remove Repetition-

Task: Provide AI with the following redundant code and ask it to refactor.
Python Code.

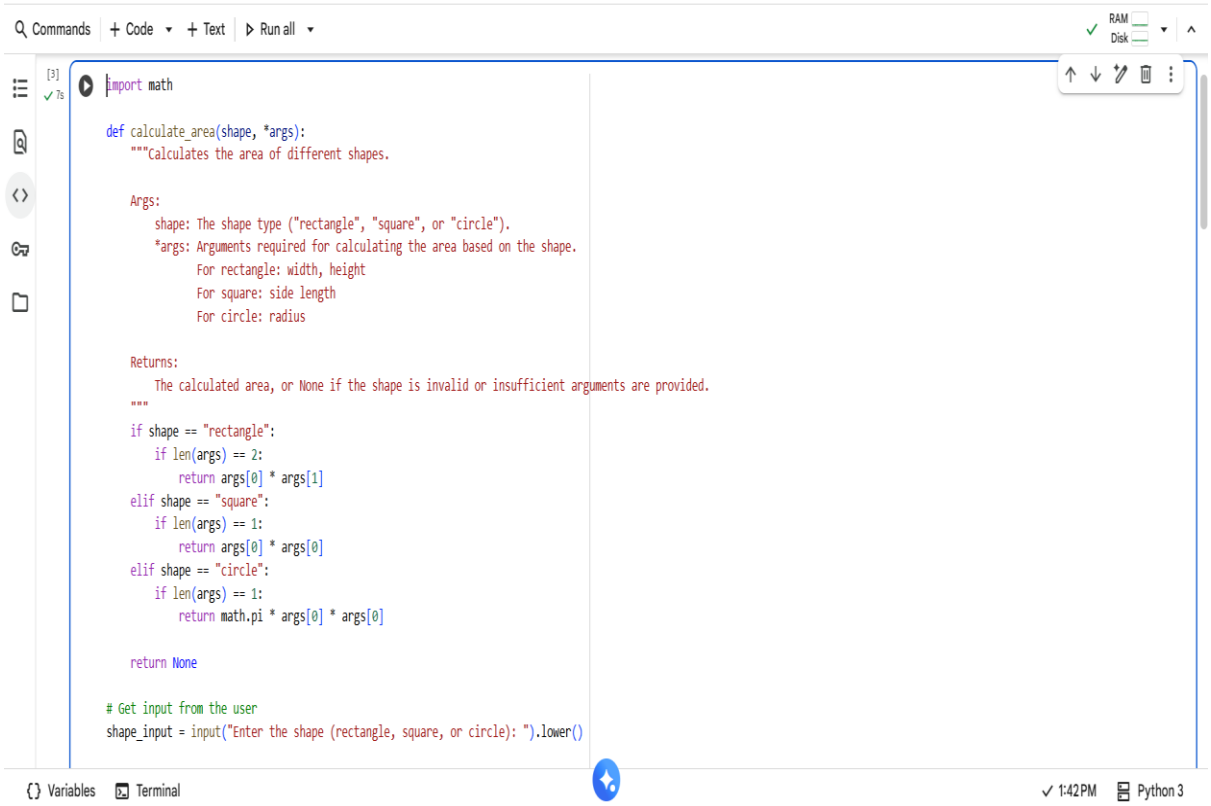
```
def calculate_area(shape, x, y=0):
if shape == "rectangle":
return x * y
elif shape == "square":
return x * x
elif shape == "circle":
return 3.14 * x * x
```

PROMPT:

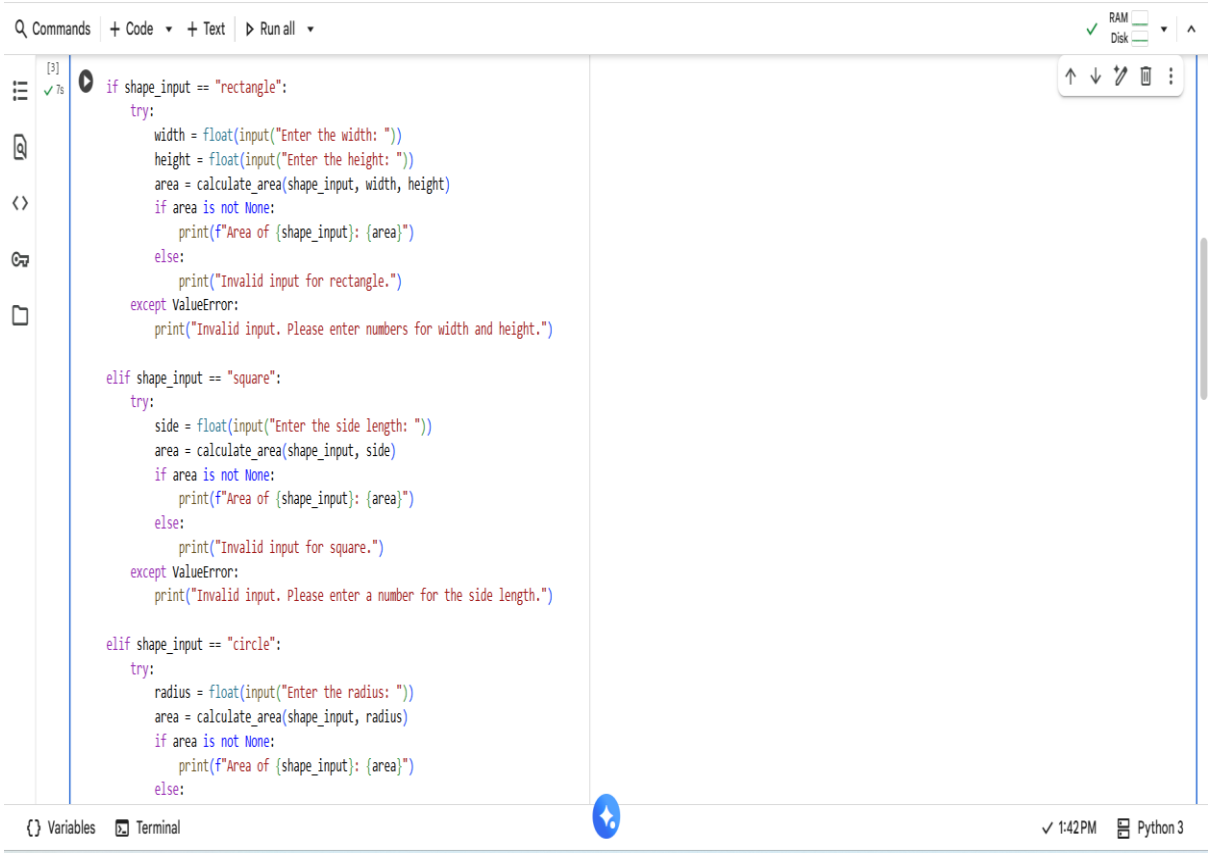
Refactor the following Python code to remove repetition and make it cleaner and more efficient:

```
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
```

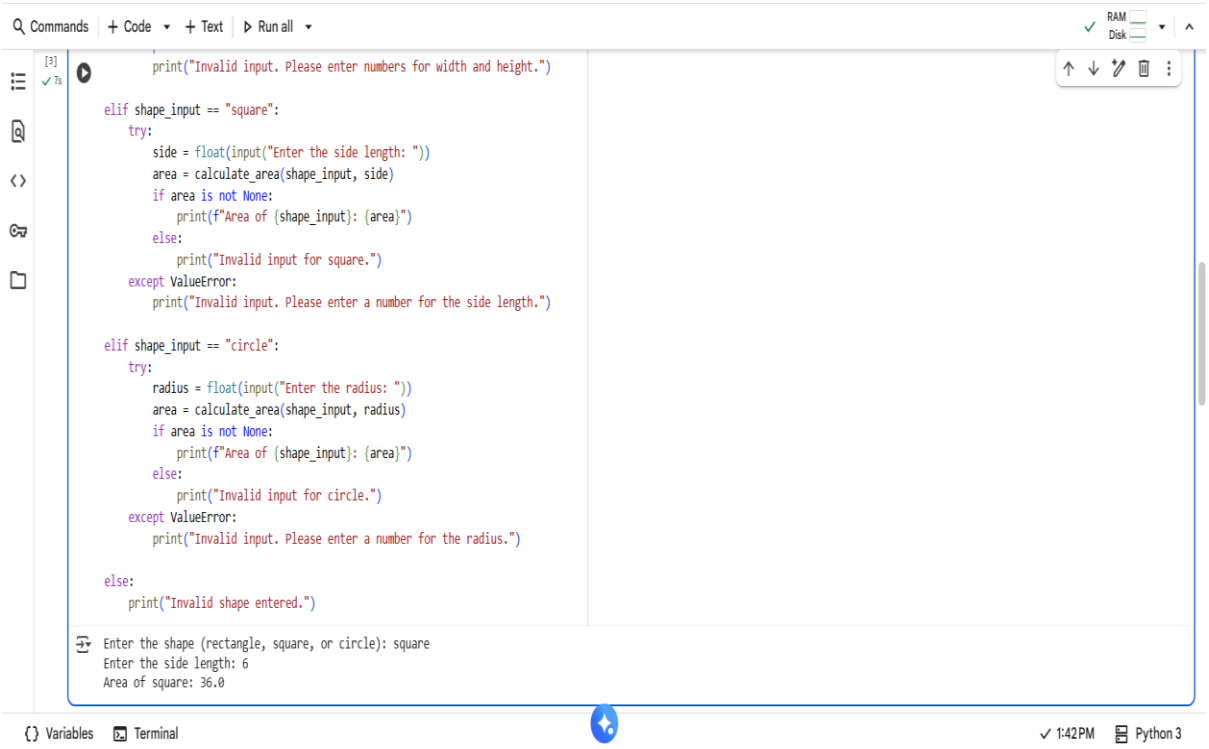
CODE With OUTPUT:



```
import math

def calculate_area(shape, *args):
    """Calculates the area of different shapes.

    Args:
        shape: The shape type ("rectangle", "square", or "circle").
        *args: Arguments required for calculating the area based on the shape.
               For rectangle: width, height
               For square: side length
               For circle: radius

    Returns:
        The calculated area, or None if the shape is invalid or insufficient arguments are provided.
    """
    if shape == "rectangle":
        if len(args) == 2:
            return args[0] * args[1]
    elif shape == "square":
        if len(args) == 1:
            return args[0] * args[0]
    elif shape == "circle":
        if len(args) == 1:
            return math.pi * args[0] * args[0]

    return None

# Get input from the user
shape_input = input("Enter the shape (rectangle, square, or circle): ").lower()
```



```
if shape_input == "rectangle":
    try:
        width = float(input("Enter the width: "))
        height = float(input("Enter the height: "))
        area = calculate_area(shape_input, width, height)
        if area is not None:
            print(f"Area of {shape_input}: {area}")
        else:
            print("Invalid input for rectangle.")
    except ValueError:
        print("Invalid input. Please enter numbers for width and height.")

elif shape_input == "square":
    try:
        side = float(input("Enter the side length: "))
        area = calculate_area(shape_input, side)
        if area is not None:
            print(f"Area of {shape_input}: {area}")
        else:
            print("Invalid input for square.")
    except ValueError:
        print("Invalid input. Please enter a number for the side length.")

elif shape_input == "circle":
    try:
        radius = float(input("Enter the radius: "))
        area = calculate_area(shape_input, radius)
        if area is not None:
            print(f"Area of {shape_input}: {area}")
        else:
```



EXPLANATION:

1. **import math**: This line imports the math module, which provides mathematical functions and constants like math.pi for a more accurate value of pi.

1. **def calculate_area(shape, *args):**: This defines a function called calculate_area that takes the shape type as the first argument and then accepts a variable number of additional arguments (*args) which are the dimensions needed for the calculation.

2. **Docstring**: The text within the triple quotes is a docstring, explaining what the function does, its arguments, and what it returns.

3. **Conditional Logic (if, elif, else)**: The code uses if, elif (else if), and else statements to check the value of the shape variable and perform the corresponding area calculation:

    - If shape is "rectangle", it checks if two arguments (width and height) are provided and calculates the area as width * height.
    - If shape is "square", it checks if one argument (side length) is provided and calculates the area as side * side.
    - If shape is "circle", it checks if one argument (radius) is provided and calculates the area as math.pi * radius * radius.
    - If the shape is none of the recognized types or the number of arguments is incorrect, the function returns None.

4. **Getting User Input**:

    - shape_input = input("Enter the shape (rectangle, square, or circle): ").lower(): This line prompts the user to enter the shape they want to calculate the area for and

converts their input to lowercase using .lower() to make the comparison case-insensitive.

5. **Handling User Input and Calculating Area**: The code then uses another set of if, elif, and else statements to handle the user's shape_input:
    - For each shape, it prompts the user to enter the required dimensions (width and height for rectangle, side length for square, radius for circle).
    - **try...except ValueError**: This block is used for error handling. It attempts to convert the user's input into a floating-point number using float(). If the user enters something that cannot be converted to a number, a ValueError occurs, and the except block is executed, printing an error message.
    - **Calling calculate_area**: If the input is valid, the calculate_area function is called with the shape and the entered dimensions.
    - **Printing the Result**: If the calculate_area function returns a valid area (not None), the code prints the calculated area. Otherwise, it prints an "Invalid input" message specific to the shape.
6. **Invalid Shape Input**: If the user enters a shape that is not "rectangle", "square", or "circle", the final else block is executed, printing "Invalid shape entered."

➢ 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
```

PROMPT:

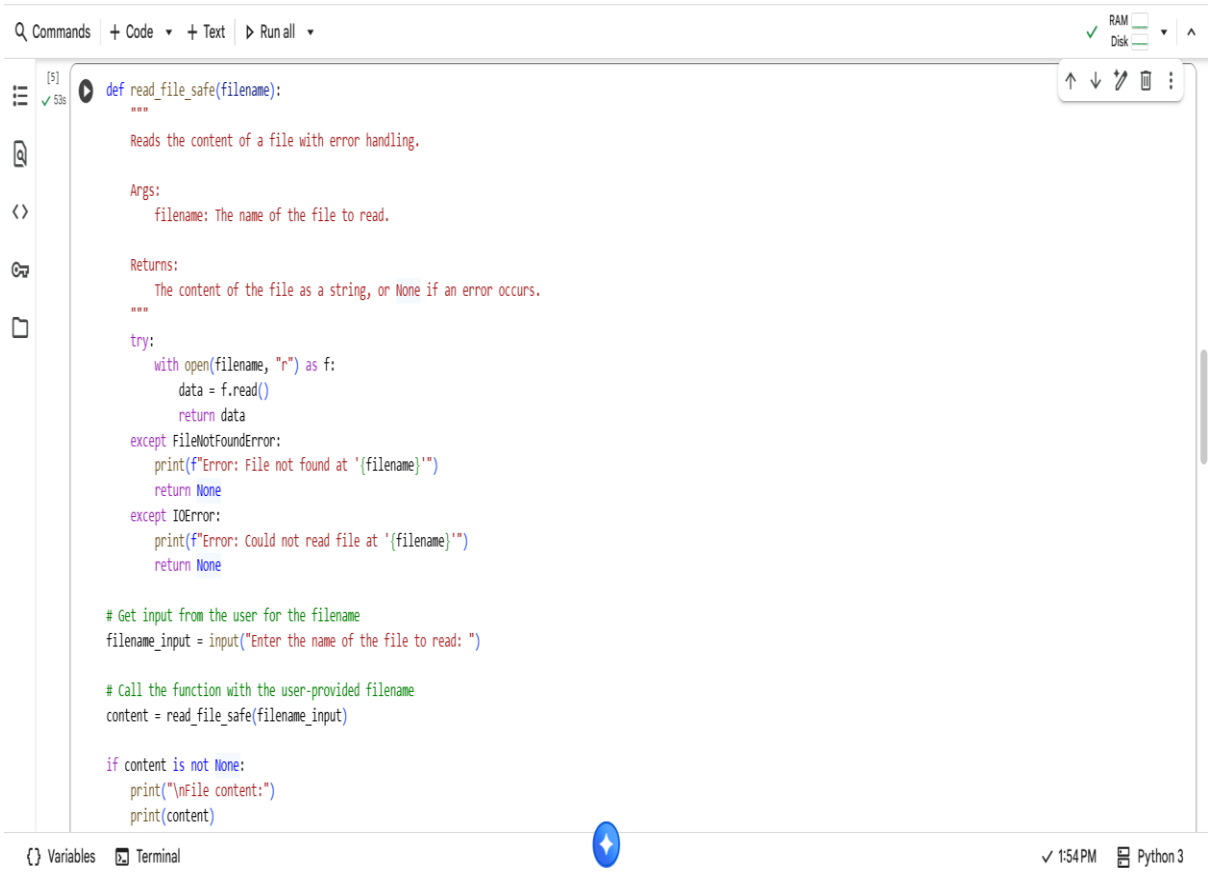
Add proper error handling to the following legacy Python function so it can safely handle missing files or read errors:

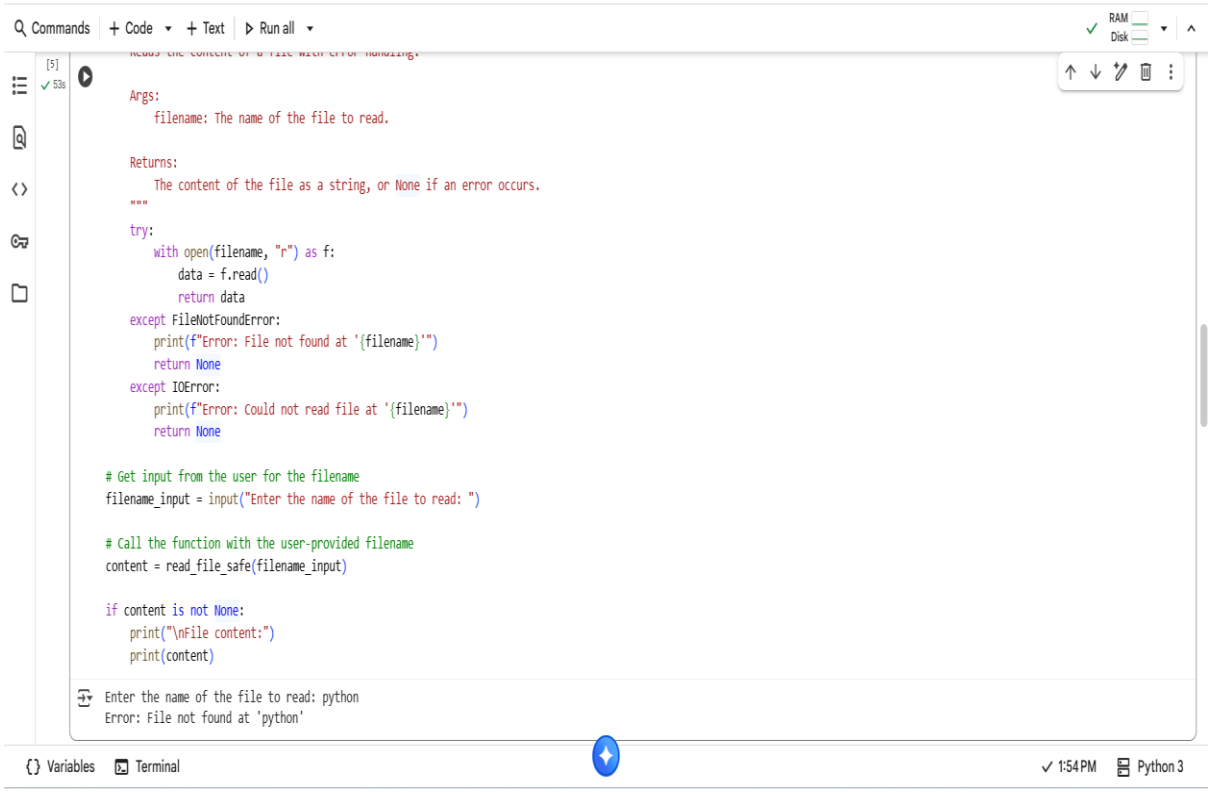
```
def read_file(filename):
  f = open(filename, "r")
  data = f.read()
```

f.close()

return data

CODE With OUTPUT:





EXPLANATION:

1. **def read_file_safe(filename):**: This defines a function named read_file_safe that takes one argument: filename, which is the name of the file you want to read.
2. **Docstring**: The text within the triple quotes explains what the function does, its arguments, and what it returns. It highlights that the function includes error handling.
3. **try...except block**: This is the core of the error handling.
    - **try:**: The code inside the try block is what the program will attempt to execute.
    - **with open(filename, "r") as f:**: This is a safe way to open a file. open(filename, "r") opens the file specified by filename in read mode ("r"). The with statement ensures that the file is automatically closed even if errors occur. The opened file object is assigned to the variable f.
    - **data = f.read()**: This line reads the entire content of the file and stores it as a string in the data variable.
    - **return data**: If the file is opened and read successfully, the function returns the data (the file content).
4. **except FileNotFoundError:**: This block is executed if a FileNotFoundError occurs within the try block. This happens when the file specified by filename does not exist at the given path. It prints an error message indicating that the file was not found and then the function returns None.

5. **except IOError:**: This block is executed if an IOError occurs within the try block. This can happen for various reasons related to input/output operations, such as permission issues or hardware problems that prevent the file from being read. It prints an error message indicating that the file could not be read and then the function returns None.
6. **Getting User Input**:
    - **filename_input = input("Enter the name of the file to read: ")**: This line prompts the user to enter the name of the file they want to read and stores their input in the filename_input variable.
7. **Calling the Function and Displaying Output**:
    - **content = read_file_safe(filename_input)**: This line calls the read_file_safe function with the filename provided by the user and stores the returned value (either the file content or None) in the content variable.
    - **if content is not None:**: This checks if the content variable is not None. If it's not None, it means the file was read successfully.
    - **print("\nFile content:")**: This prints a header before displaying the file content.
    - **print(content)**: This prints the actual content of the file that was read.

➢ TASK DESCRIPTION-3:

Complex Refactoring-
Task: Provide this legacy class to AI for readability and modularity improvements:

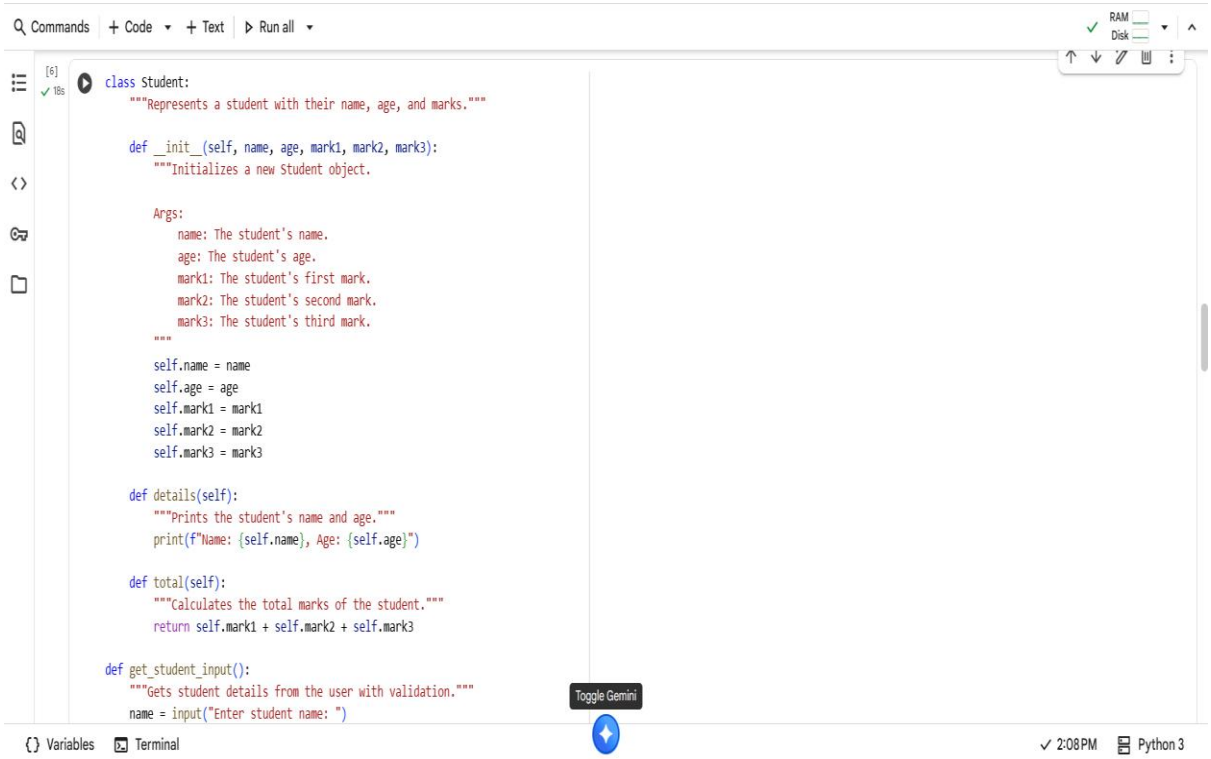
Python Code
class Student:
def __init__(self, n, a, m1, m2, m3):
self.n = n
self.a = a
self.m1 = m1
self.m2 = m2
self.m3 = m3
def details(self):
print("Name:", self.n, "Age:", self.a)
def total(self):
return self.m1+self.m2+self.m3

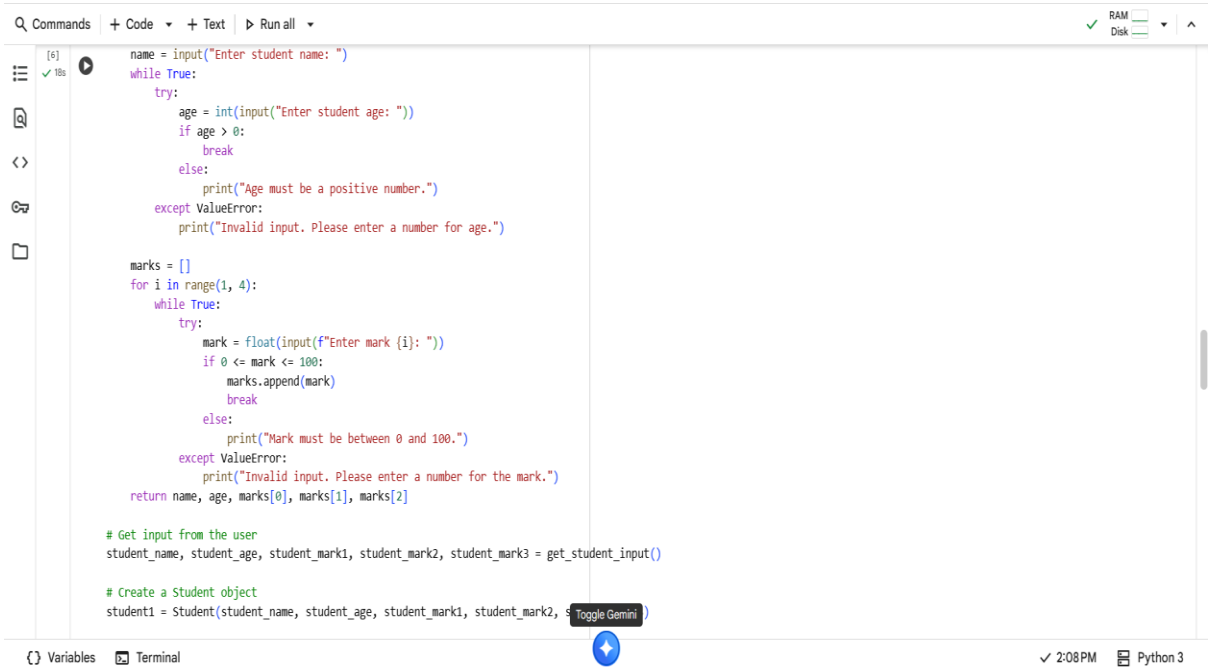
PROMPT:

Refactor the following legacy Python class to improve readability and make the code more modular and well-structured:

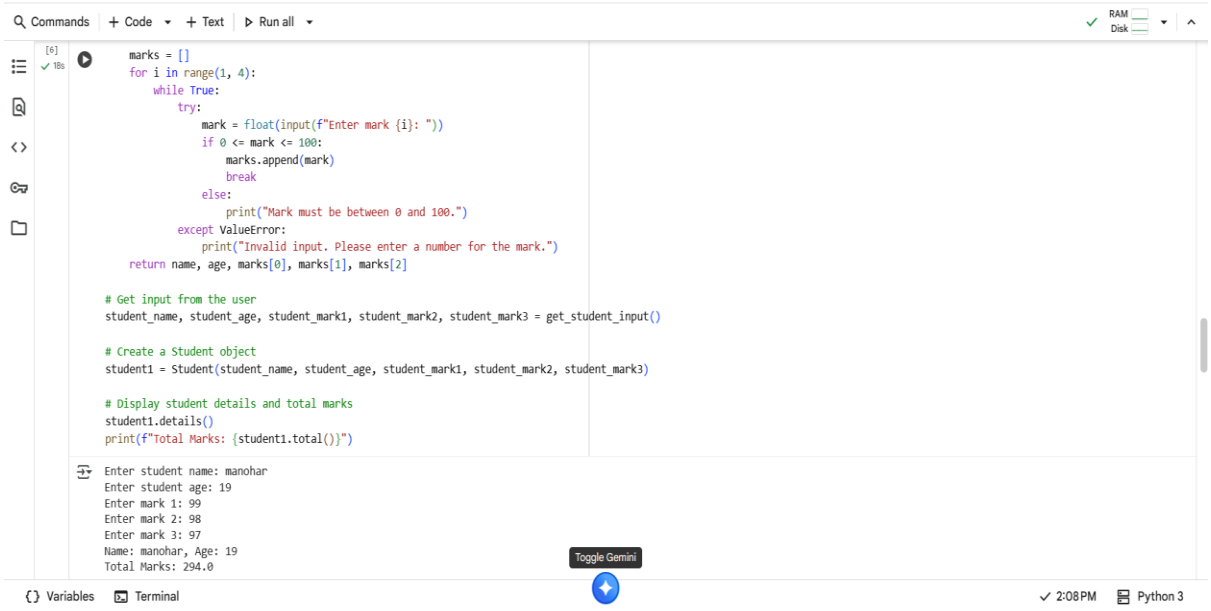
```
class Student:
    def _init_(self, n, a, m1, m2, m3):
        self.n = n
        self.a = a
        self.m1 = m1
        self.m2 = m2
        self.m3 = m3
    def details(self):
        print("Name:", self.n, "Age:", self.a)
    def total(self):
        return self.m1 + self.m2 + self.m3
```

CODE With OUTPUT:





```
    name = input("Enter student name: ")
    while True:
        try:
            age = int(input("Enter student age: "))
            if age > 0:
                break
            else:
                print("Age must be a positive number.")
        except ValueError:
            print("Invalid input. Please enter a number for age.")

    marks = []
    for i in range(1, 4):
        while True:
            try:
                mark = float(input(f"Enter mark {i}: "))
                if 0 <= mark <= 100:
                    marks.append(mark)
                    break
                else:
                    print("Mark must be between 0 and 100.")
            except ValueError:
                print("Invalid input. Please enter a number for the mark.")
    return name, age, marks[0], marks[1], marks[2]

# Get input from the user
student_name, student_age, student_mark1, student_mark2, student_mark3 = get_student_input()

# Create a Student object
student1 = Student(student_name, student_age, student_mark1, student_mark2, s
```

```
    marks = []
    for i in range(1, 4):
        while True:
            try:
                mark = float(input(f"Enter mark {i}: "))
                if 0 <= mark <= 100:
                    marks.append(mark)
                    break
                else:
                    print("Mark must be between 0 and 100.")
            except ValueError:
                print("Invalid input. Please enter a number for the mark.")
    return name, age, marks[0], marks[1], marks[2]

# Get input from the user
student_name, student_age, student_mark1, student_mark2, student_mark3 = get_student_input()

# Create a Student object
student1 = Student(student_name, student_age, student_mark1, student_mark2, student_mark3)

# Display student details and total marks
student1.details()
print(f"Total Marks: {student1.total()}")
```

```
Enter student name: manohar
Enter student age: 19
Enter mark 1: 99
Enter mark 2: 98
Enter mark 3: 97
Name: manohar, Age: 19
Total Marks: 294.0
```


EXPLANANTION:

The Student Class:

- **class Student:**: This line defines a blueprint for creating Student objects.
- **"""Represents a student with their name, age, and marks."""**: This is a docstring that explains the purpose of the class.
- **def __init__(self, name, age, mark1, mark2, mark3):**: This is the constructor method. It's called when you create a new Student object.
    - self: Refers to the instance of the class being created.

    - name, age, mark1, mark2, mark3: These are the parameters that you pass when creating a Student object.
    - self.name = name, self.age = age, etc.: These lines assign the values passed as arguments to the corresponding attributes (variables) of the Student object.
- **def details(self):**: This method is defined within the Student class.
    - self: Refers to the instance of the class.
    - print(f"Name: {self.name}, Age: {self.age}"): This line prints the name and age of the student using the self.name and self.age attributes. The f"" is a formatted string literal that allows you to embed variables directly in the string.
- **def total(self):**: This method calculates the total marks of the student.
    - return self.mark1 + self.mark2 + self.mark3: It returns the sum of the three mark attributes of the student object.

2. The get_student_input() Function:

- **def get_student_input():**: This function is designed to get student information from the user with built-in validation.
- **name = input("Enter student name: ")**: Prompts the user to enter the student's name and stores it in the name variable.
- **while True:**: This starts an infinite loop that will continue until explicitly broken.
- **try...except ValueError:**: This is used for error handling when getting the age and marks.
    - try:: The code inside the try block is attempted.
    - age = int(input("Enter student age: ")): Prompts for age and tries to convert the input to an integer.
    - if age > 0:: Checks if the entered age is positive. If it is, the loop breaks.
    - else: print("Age must be a positive number."): If the age is not positive, an error message is printed.
    - except ValueError: print("Invalid input. Please enter a number for age."): If the input cannot be converted to an integer, a ValueError occurs, and this block prints an error message.
- The code for getting marks is similar, using a for loop to get three marks and validating that they are numbers between 0 and 100.
- **return name, age, marks[0], marks[1], marks[2]**: The function returns the collected name, age, and the three marks as a tuple.

3. Using the Class and Function:

- **student_name, student_age, student_mark1, student_mark2, student_mark3 = get_student_input()**: This line calls the get_student_input() function and unpacks the returned tuple into five individual variables.

- **student1 = Student(student_name, student_age, student_mark1, student_mark2, student_mark3)**: This creates a new Student object named student1, passing the values obtained from user input to the constructor.
- **student1.details()**: This calls the details() method on the student1 object, which prints the student's name and age.
- **print(f"Total Marks: {student1.total()}")**: This calls the total() method on the student1 object to get the total marks and then prints the result.

TASK DISCRIPTION-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)

PROMPT:

Refactor the following inefficient loop to make it more concise and Pythonic:

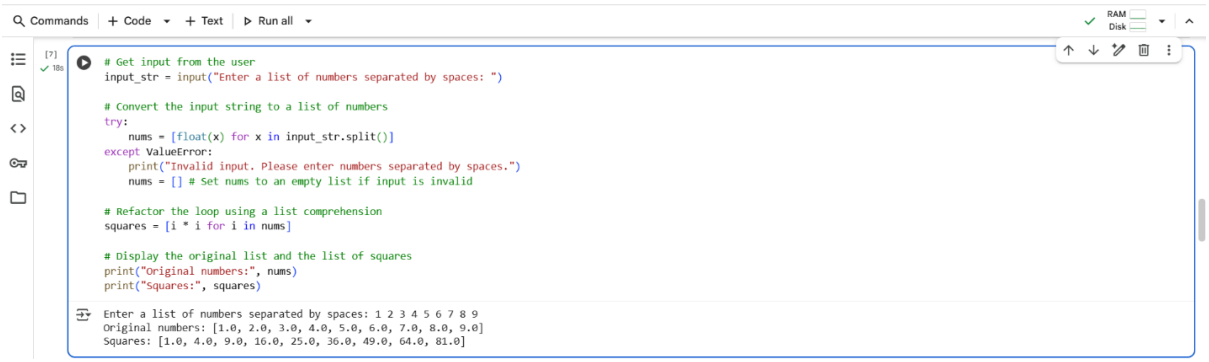
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

squares = []

for i in nums:

    squares.append(i * i)

CODE With OUTPUT:



EXPLANATION:

1. **# Get input from the user**: This is a comment indicating the purpose of the following line.
    - **input_str = input("Enter a list of numbers separated by spaces: ")**: This line prompts the user to enter a sequence of numbers separated by spaces and stores the entered string in the variable input_str.
2. **# Convert the input string to a list of numbers**: This comment explains the next section of code.
    - **try...except ValueError:**: This block handles potential errors that might occur when converting the input string to numbers.
        - **try:**: The code within the try block is attempted.
        - **nums = [float(x) for x in input_str.split()]**: This is a **list comprehension**, a concise way to create lists in Python.
            - input_str.split(): This splits the input_str string into a list of substrings using spaces as the delimiter. For example, if input_str is "1 2 3", this will produce ['1', '2', '3'].
            - for x in ...: This iterates through each substring (x) in the list created by split().
            - float(x): This attempts to convert each substring x into a floating-point number.
            - [...]: The square brackets indicate that the result of the operation (float(x)) for each item in the loop will be collected into a new list called nums.
        - **except ValueError:**: If any of the substrings cannot be converted to a float (e.g., the user enters text instead of numbers), a ValueError is raised, and the code inside this block is executed.
            - print("Invalid input. Please enter numbers separated by spaces."): An error message is printed to the user.
            - nums = []: The nums list is set to an empty list to avoid errors later in the code if the input was invalid.
3. **# Refactor the loop using a list comprehension**: This comment highlights the refactoring of a traditional for loop into a list comprehension.
    - **squares = [i * i for i in nums]**: This is another list comprehension.
        - for i in nums: It iterates through each number (i) in the nums list.
        - i * i: For each number, it calculates its square.

        - [...]: The results of the squaring operation are collected into a new list called squares. This achieves the same result as the original for loop but in a more compact way.

4. **# Display the original list and the list of squares**: This comment explains the final output section.
    - **print("Original numbers:", nums)**: This line prints the original list of numbers entered by the user.
    - **print("Squares:", squares)**: This line prints the list containing the squares of the numbers.

---END---