LAB ASSIGNMENT 13.2

< AI ASSISTED CODING >

Program : B. Tech (CSE)

Specialization : AIML

Course Title : AI Assisted coding

Semester : III

Academic Session : 2025-2026

Name of Student : Bommineni Spandana

Enrollment No : 2403a520005

Batch No. : 01

Date : 28-10-2025

- **Task Description #1** – Remove Repetition
  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:**

Refactor the following Python function to eliminate repetitive conditional logic:

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
```

Use a cleaner, modular design—such as dictionary-based dispatch or separate functions for each shape—to improve readability and maintainability.

Code :

```python
def calculate_area_refactored(shape, x, y=0):
    area_calculators = {
        "rectangle": lambda x, y: x * y,
        "square": lambda x, y: x * x,
        "circle": lambda x, y: 3.14 * x * x,
    }
    if shape in area_calculators:
        return area_calculators[shape](x, y)
    else:
        return "Invalid shape"

# Get user input
shape_input = input("Enter the shape (rectangle, square, or circle): ").lower()
x_input = float(input("Enter the first dimension (e.g., side length or radius): "))
y_input = float(input("Enter the second dimension (only for rectangle, enter 0 otherwise): "))

# Calculate and print the area
print(f"The area of the {shape_input} is: {calculate_area_refactored(shape_input, x_input, y_input)}")
```

Output :

```
Enter the shape (rectangle, square, or circle): square
Enter the first dimension (e.g., side length or radius): 4
Enter the second dimension (only for rectangle, enter 0 otherwise): 4
The area of the square is: 16.0
```

Explanation : calculate_area_refactored(shape, x, y=0) function:

Takes shape (a string representing the shape), x (the first dimension), and y (the second dimension, with a default value of 0) as input.

area_calculators is a dictionary that maps shape names to anonymous functions (lambdas) that calculate the area for each shape.

It checks if the provided shape exists as a key in the area_calculators dictionary.

If the shape is found, it calls the corresponding lambda function with x and y to calculate the area and returns the result.

If the shape is not found, it returns "Invalid shape".

Observation : Refactored Logic: The code successfully refactors the area calculation logic by using a dictionary (area_calculators) to map shape names to their respective area calculation functions. This eliminates the need for multiple elif statements, making the code more readable and easier to maintain.

Modular Design: The use of lambda functions within the dictionary promotes a more modular design, where each shape's area calculation is self-contained.

User Input: The code includes user input prompts to make the area calculation interactive. It takes the shape, the first dimension, and an optional second dimension from the user.

Task Description #2 – Error Handling in Legacy Code

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:

Improve the following legacy Python function by adding proper error handling and using modern best practices:

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

return data

Refactor the code to use a with open() context manager and wrap the file operations in a try-except block to handle potential exceptions gracefully.

Code :

```
def read_file_improved(filename):
    try:
        with open(filename, "r") as f:
            data = f.read()
            return data
    except FileNotFoundError:
        return f"Error: The file '{filename}' was not found."
    except Exception as e:
        return f"An error occurred while reading the file: {e}"

# Get user input for the filename
filename_input = input("Enter the name of the file to read: ")

# Call the improved function and print the result
print(read_file_improved(filename_input))
```

Output :

```
Enter the name of the file to read: AI Assisted coding
Error: The file 'AI Assisted coding' was not found.
```

Explanation : read_file_improved(filename) function:

Takes filename (a string representing the name of the file to read) as input.

try...except block: This block is used for error handling. Code within the try block is executed, and if an error occurs, the code within the corresponding except block is executed.

with open(filename, "r") as f:: This is a with statement using a context manager. It opens the specified filename in read mode ("r"). The with statement ensures that the file is automatically closed even if errors occur, preventing resource leaks. The opened file object is assigned to the variable f.

data = f.read(): Reads the entire content of the file into the data variable.

return data: If the file is read successfully, the function returns the data.

Observation : Safe File Handling: It uses with open(...) to make sure the file is closed automatically, even if there's an error.

Handles Missing Files: It specifically checks if the file doesn't exist and tells you if it's not found.

Catches Other Errors: It also catches any other unexpected problems that might happen while reading the file.

- **Task Description #3** – Complex Refactoring

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:

Refactor the following legacy Python class to enhance clarity and modularity:

```
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
```

Update the code with meaningful variable names, add docstrings, improve the output formatting in the details method, and consider storing marks in a list to simplify total calculation using sum().

Code :

```python
class Student:
    """Represents a student and stores their details like name, age, and marks."""
    def __init__(self, name, age, *marks):
        """
        Initializes a Student object with name, age, and a list of marks.

        Args:
            name (str): The name of the student.
            age (int): The age of the student.
            *marks (float): Variable number of marks for the student.
        """
        self.student_name = name
        self.student_age = age
        self.student_marks = list(marks)

    def details(self):
        """Displays the details of the student."""
        print(f"Name: {self.student_name}")
        print(f"Age: {self.student_age}")
        print(f"Marks: {self.student_marks}")

    def total(self):
        """Calculates and returns the sum of the student's marks."""
        return sum(self.student_marks)
```

```python
# Get user input
student_name_input = input("Enter student's name: ")
student_age_input = int(input("Enter student's age: "))
marks_input_str = input("Enter student's marks separated by spaces: ")
student_marks_input = [float(mark) for mark in marks_input_str.split()]

# Create a Student object
student_object = Student(student_name_input, student_age_input, *student_marks_input)

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

Output :

```
Enter student's name: akhila
Enter student's age: 19
Enter student's marks separated by spaces: 91 90 88 87 92
Name: akhila
Age: 19
Marks: [91.0, 90.0, 88.0, 87.0, 92.0]
Total Marks: 448.0
```

Explanation : Student Class Definition:

class Student:: This line defines a class named Student.

"""Represents a student and stores their details like name, age, and marks.""": This is a class docstring that explains the purpose of the Student class.

init(self, name, age, *marks): This is the constructor method.

It takes self (a reference to the object itself), name, age, and *marks as arguments.

*marks allows you to pass a variable number of marks to the constructor, which are then collected into a tuple.

The docstring explains that this method calculates and returns the sum of the student's marks.

Observation : Improved Readability: The variable names like student_name, student_age, and student_marks are much clearer than the original n, a, m1, etc.

Better Organization: The use of docstrings for the class and methods makes the code's purpose and usage easy to understand.Flexible Mark Handling: Storing

marks in a list (student_marks) and using *marks in the constructor allows for a variable number of marks, making the class more flexible.Simplified Total Calculation: Using sum() on the student_marks list is a clean and efficient way to calculate the total marks.Clear Output: The details() method uses f-strings for well-formatted and easy-to-read output.User Interaction: The code includes clear prompts for the user to enter student details.

- **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:

Optimize the following Python loop for better performance and readability:

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

squares = []

for i in nums:

  squares.append(i * i)

Refactor the code using a more concise approach, such as a list comprehension, to generate the list of squared values.

Code :

```
# Get user input
student_name_input = input("Enter student's name: ")
student_age_input = int(input("Enter student's age: "))
marks_input_str = input("Enter student's marks separated by spaces: ")
student_marks_input = [float(mark) for mark in marks_input_str.split()]

# Create a Student object
student_object = Student(student_name_input, student_age_input, *student_marks_input)

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

Output :

```
Enter student's name: akhila
Enter student's age: 19
Enter student's marks separated by spaces: 91 90 88 87 92
Name: akhila
Age: 19
Marks: [91.0, 90.0, 88.0, 87.0, 92.0]
Total Marks: 448.0
```

Explanation : nums_input_str = input("Enter numbers separated by spaces: "): This line prompts the user to enter a series of numbers, separated by spaces, and stores the input as a single string in the variable nums_input_str.nums = [int(num) for num in nums_input_str.split()]: This is a list comprehension.

nums_input_str.split(): This splits the input string into a list of individual strings based on spaces.

for num in ...: This iterates through each string in the list created by split().

int(num): This converts each string num into an integer.

[...]: This creates a new list called nums containing the integer values.

squares = [i * i for i in nums]: This is another list comprehension, which is a more concise way to create a list of squared values.

for i in nums: This iterates through each number i in the nums list.

i * i: This calculates the square of the current number i.

[...]: This creates a new list called squares containing the squared values.

Observation : Takes Numbers from User: It asks you to type in numbers, separated by spaces.

Smartly Squares Numbers: It quickly figures out the square of each number you enter using a neat Python trick called a "list comprehension".

Clean and Short: The way it squares the numbers is very short and easy to read compared to a longer loop.

Shows Original and Squared: It prints both the numbers you entered and their squared versions.