

AI assistant lab:13.3

Name:balaji

Rollno:2403A51102

Batch:6

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 redundant Python function calculate_area to improve readability, remove repetition, and use a more modular design without changing its functionality.

Code:

```
13.3.py >  area_circle
1 def area_rectangle(length, width):
2     | return length * width
3
4 def area_square(side):
5     | return side * side
6
7 def area_circle(radius):
8     | return 3.14 * radius * radius
9
10 # Dictionary mapping shape names to functions
11 area_functions = {
12     | "rectangle": lambda x, y: area_rectangle(x, y),
13     | "square": lambda x, _: area_square(x),
14     | "circle": lambda x, _: area_circle(x)
15 }
16
17 def calculate_area(shape, x, y=0):
18     | shape = shape.lower()
19     | if shape in area_functions:
20     |     | return area_functions[shape](x, y)
21     | else:
22     |     | raise ValueError(f"Unsupported shape: {shape}")
23
24 #  Print area calculations
25 print("Rectangle area (5 x 10):", calculate_area("rectangle", 5, 10))
26 print("Square area (4 x 4):", calculate_area("square", 4))
27 print("Circle area (r = 3):", calculate_area("circle", 3))
```

Output:

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\P. BALAJI\OneDrive\Desktop\AI asistant lab> & "C:/Users/P. BALAJI/AppData/Local/Programs/Python/Python39-64/Python.exe" -i -c "
Rectangle area (5 x 10): 50
Square area (4 x 4): 16
Circle area (r = 3): 28.259999999999998
PS C:\Users\P. BALAJI\OneDrive\Desktop\AI asistant lab>

```

Observation: The refactored code improves readability by replacing multiple if-elif checks with a dictionary mapping shapes to dedicated functions, enhancing modularity and

making the code easier to maintain and extend. It removes repetition, allows clearer error handling for unsupported shapes, and preserves the original behavior, resulting in cleaner, more scalable, and more efficient 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
```

prompt: Refactor the legacy `read_file` function to include proper error handling using try-except blocks and ensure the file is safely closed using a context manager (with statement), while preserving the original functionality.

code:

```
1 def read_file(filename):
2     try:
3         with open(filename, "r") as f:
4             data = f.read()
5             print(f"File '{filename}' read successfully.")
6             return data
7     except FileNotFoundError:
8         print(f"Error: The file '{filename}' was not found.")
9     except IOError:
10        print(f"Error: Could not read the file '{filename}'.")
11    except Exception as e:
12        print(f"An unexpected error occurred: {e}")
13    return None
14
15 # Example usage:
16 content = read_file("example.txt")
17
18 if content is not None:
19     print("File Content:")
20     print(content)
21 else:
22     print("Failed to read file.")
```

output:

```
PS C:\Users\P. BALAJI\OneDrive\Desktop\AI asistant lab> & "C:/User
/AI asistant lab/t2.py"
Error: The file 'example.txt' was not found.
Failed to read file.
PS C:\Users\P. BALAJI\OneDrive\Desktop\AI asistant lab>
```

observation: The refactored function improves safety and reliability by automatically managing file closing with `with open()`, and gracefully handles errors such as missing files or read failures through specific exceptions. This prevents program crashes and provides informative error messages, making the code

more robust and user-friendly without changing its intended behavior.

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 legacy Student class to improve readability and modularity by using clearer variable names, encapsulating marks in a list instead of separate variables, adding meaningful methods like average(), and enhancing documentation, while preserving existing functionality.

Code:

```
t3.py > Student > details
1  class Student:
2      def __init__(self, name, age, marks):
3          """
4              Initialize a Student object.
5
6              Args:
7                  name (str): Student's name
8                  age (int): Student's age
9                  marks (list of int): List of marks in subjects
10             """
11             self.name = name
12             self.age = age
13             self.marks = marks
14
15     def details(self):
16         """Print student details."""
17         print(f"Name: {self.name}, Age: {self.age}")
18
19     def total(self):
20         """Calculate and return the total marks."""
21         return sum(self.marks)
22
23     def average(self):
24         """Calculate and return the average marks."""
25         return sum(self.marks) / len(self.marks)
26
27     # Example usage:
28     student1 = Student("Alice", 20, [85, 90, 88])
29     student1.details()
30     print("Total Marks:", student1.total())
31     print("Average Marks:", student1.average())
```

Output:

```
>  
/AI asistant lab/t3.py"  
Name: Alice, Age: 20  
Total Marks: 263  
Average Marks: 87.66666666666667  
PS C:\Users\P. BALAJI\OneDrive\Desktop\AI asistant lab>
```

Observation: The refactored class improves clarity by replacing ambiguous attribute names (n, a, m1, m2, m3) with descriptive names (name, age, marks list). It adds modularity by encapsulating marks and introducing new methods such as average() to extend functionality without code duplication. Clear method docstrings and consistent formatting make the code more maintainable and easier to extend.

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)
```

Prompt: Refactor the given loop that appends the squares of numbers to a list into a more efficient and

Pythonic form using list comprehension, preserving the original output.

Code:

```
t4.py / ...  
1  # Define a list of numbers  
2  nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
3  print("Original list of numbers:", nums)  
4  
5  # Use list comprehension to create a list of their squares  
6  squares = [i * i for i in nums]  
7  print("Calculated squares of the numbers.")  
8  
9  # Print the resulting list of squares  
10 print("Squares of the given numbers are:", squares)
```

Output:

```
/AI asistant lab/t4.py"  
Original list of numbers: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
Calculated squares of the numbers.  
Squares of the given numbers are: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]  
PS C:\Users\P. BALAJI\OneDrive\Desktop\AI asistant lab>
```

Observation: The refactored code replaces the explicit loop and `append()` calls with a clean, concise list comprehension that achieves the same result in fewer lines. This improves readability and performance by leveraging Python's optimized syntax, making the code more elegant and maintainable without changing its functionality.