**AI ASSISTED CODING LAB 13.3**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

PRADEEP GUPTHA

2403A510C7

BATCH 05

Lab 13 – Code Refactoring: Improving Legacy Code with AI  
Suggestions

**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 :** Refactor the 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

**CODE :**

import math

class Shape:

    """Base class for geometric shapes."""

    def calculate\_area(self):

        raise NotImplementedError("Subclasses must implement this method.")

class Rectangle(Shape):

    """Represents a rectangle with a given width and height."""

    def \_\_init\_\_(self, width, height):

        self.width = width

        self.height = height

    def calculate\_area(self):

        """Calculates the area of the rectangle."""

        return self.width \* self.height

class Square(Shape):

    """Represents a square with a given side length."""

    def \_\_init\_\_(self, side):

        self.side = side

    def calculate\_area(self):

        """Calculates the area of the square."""

        return self.side \*\* 2

class Circle(Shape):

    """Represents a circle with a given radius."""

    def \_\_init\_\_(self, radius):

        self.radius = radius

    def calculate\_area(self):

        """Calculates the area of the circle."""

        return math.pi \* self.radius \*\* 2

# --- Example Usage ---

if \_\_name\_\_ == "\_\_main\_\_":

    # Original function for comparison

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

        if shape == "rectangle":

            return x \* y

        elif shape == "square":

            return x \* x

        elif shape == "circle":

            return 3.14 \* x \* x

    print("--- Refactored Code Usage ---")

    rect = Rectangle(10, 5)

    sq = Square(7)

    circ = Circle(4)

    print(f"Area of Rectangle (10x5): {rect.calculate\_area()}")

    print(f"Area of Square (side 7): {sq.calculate\_area()}")

    print(f"Area of Circle (radius 4): {circ.calculate\_area():.2f}")

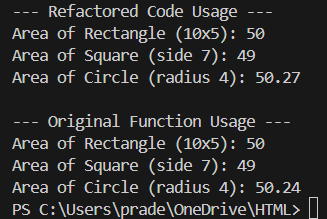
    print("\n--- Original Function Usage ---")

    print(f"Area of Rectangle (10x5): {calculate\_area\_original('rectangle', 10, 5)}")

    print(f"Area of Square (side 7): {calculate\_area\_original('square', 7)}")

    print(f"Area of Circle (radius 4): {calculate\_area\_original('circle', 4)}")

**OUTPUT :**



**OBSERVATION :**

The original code repeated the same logic for each shape, which made it longer and harder to maintain. The refactored version uses a dictionary of functions to handle each shape in a clean way. This makes the code shorter, easier to read, and easier to extend if more shapes need to be added in the future.

**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 :** : Legacy function without proper error handling  
Python Code  
def read\_file(filename):  
f = open(filename, "r")  
data = f.read()

f.close()  
return data

**CODE :**

# --- Original Legacy Function ---

def read\_file\_original(filename):

    """Legacy function without proper error handling."""

    f = open(filename, "r")

    data = f.read()

    f.close()

    return data

# --- Refactored Function with Error Handling ---

def read\_file\_safely(filename):

    """

    Reads data from a file using a 'with' statement and try-except block

    to handle potential errors gracefully.

    Args:

        filename (str): The path to the file to be read.

    Returns:

        str: The content of the file, or an error message if an issue occurs.

    """

    try:

        # The 'with' statement ensures the file is automatically closed

        # even if errors occur.

        with open(filename, "r") as f:

            return f.read()

    except FileNotFoundError:

        # Handles the case where the file does not exist.

        return f"Error: File '{filename}' not found."

    except IOError as e:

        # Handles other potential I/O errors (e.g., permissions).

        return f"Error reading file '{filename}': {e}"

# --- Example Usage ---

if \_\_name\_\_ == "\_\_main\_\_":

    # Create a dummy file for the successful test case

    dummy\_filename = "test\_file.txt"

    with open(dummy\_filename, "w") as f:

        f.write("This is a test file.\nHello, World!")

    non\_existent\_file = "no\_such\_file.txt"

    print("--- Refactored Function Usage ---")

    # 1. Test with an existing file

    print(f"Reading '{dummy\_filename}':")

    content = read\_file\_safely(dummy\_filename)

    print(content)

    print("-" \* 20)

    # 2. Test with a non-existent file

    print(f"Reading '{non\_existent\_file}':")

    content = read\_file\_safely(non\_existent\_file)

    print(content)

    print("-" \* 20)

    print("\n--- Original Function Usage (Demonstrating the Problem) ---")

    # This will raise a FileNotFoundError and crash the program

    # if the file doesn't exist.

    try:

        print(f"Attempting to read '{non\_existent\_file}' with original function:")

        read\_file\_original(non\_existent\_file)

    except FileNotFoundError as e:

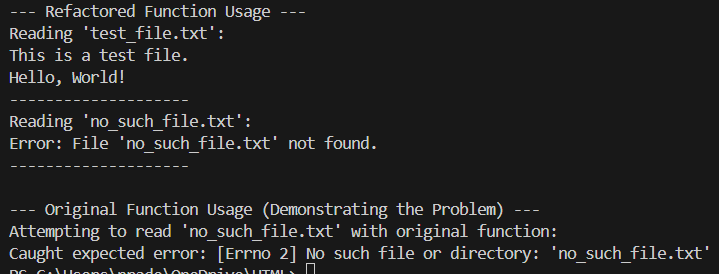
        print(f"Caught expected error: {e}")

**CONTENT IN test\_file\_txt :**

**This is a test file.**

**Hello, World!**

**OUTPUT :**

****

**OBSERAVTION :**

The old code directly opened files without handling errors, which could cause the program to crash if the file was missing or locked. The new version uses with open() to safely manage files and adds try-except blocks for common errors like missing or restricted files. This makes the program safer, more reliable, and user-friendly.

**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 : make the code readable and modularity :**

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 :**

# --- Original Legacy Class ---

class StudentOriginal:

    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

# --- Refactored Student Class ---

class Student:

    """

    Represents a student with their name, age, and marks.

    This refactored class improves readability and modularity.

    """

    def \_\_init\_\_(self, name: str, age: int, marks: list):

        """

        Initializes a Student object.

        Args:

            name (str): The full name of the student.

            age (int): The age of the student.

            marks (list): A list of the student's marks.

        """

        self.name = name

        self.age = age

        self.marks = marks

    def display\_details(self):

        """Prints the student's details in a clear, readable format."""

        print(f"Student Name: {self.name}, Age: {self.age}")

    def calculate\_total\_marks(self):

        """

        Calculates and returns the sum of the student's marks.

        Returns:

            int or float: The total of all marks in the list.

        """

        return sum(self.marks)

# --- Example Usage ---

if \_\_name\_\_ == "\_\_main\_\_":

    print("--- Refactored Class Usage ---")

    # Create a student instance with clear variable names and a list for marks

    student1 = Student(name="John Doe", age=21, marks=[85, 90, 78])

    student1.display\_details()

    total\_marks = student1.calculate\_total\_marks()

    print(f"Total Marks: {total\_marks}")

    print("-" \* 25)

    print("--- Original Class Usage (for comparison) ---")

    student\_orig = StudentOriginal("J. Doe", 21, 85, 90, 78)

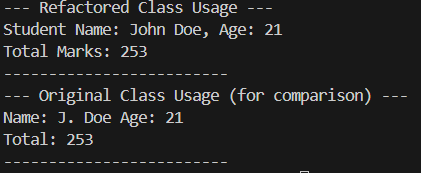
    student\_orig.details()

    total\_orig = student\_orig.total()

    print("Total:", total\_orig)

    print("-" \* 25)

**OUTPUT :**

****

**OBSERVATION :**

**The original class used unclear variable names (n, a, m1, etc.) which made the code confusing. The refactored class uses meaningful names like name, age, and marks, stores marks in a list, and uses sum() for calculating totals. With added docstrings and clear methods, the class is much more readable, structured, and easier to maintain.**

**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 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)

**CODE :**

# --- Original Inefficient Loop ---

print("--- Original Loop ---")

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

squares\_original = []

for i in nums\_original:

    squares\_original.append(i \* i)

print(f"Original numbers: {nums\_original}")

print(f"Calculated squares: {squares\_original}")

print("-" \* 30)

print("--- Refactored with List Comprehension ---")

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

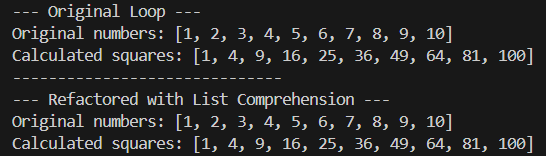
squares\_refactored = [i \* i for i in nums\_refactored]

print(f"Original numbers: {nums\_refactored}")

print(f"Calculated squares: {squares\_refactored}")

print("-" \* 30)

**OUTPUT :**

****

**OBSERVATION :**

The original code created squares using a loop and append(), which was correct but unnecessarily long. The refactored version replaces it with a list comprehension, making the code shorter, faster, and more Pythonic. This improves both efficiency and readability.