AI-ASSISTED CODING

ASSIGNMENT-13.3

Code Refactoring: Improving Legacy Code with AI  
Suggestions

**NAME: CHIPPA BHUVAN PRAKASH**

**HALL-TICKET: 2403A510F3**

**BATCH NO: 06**

**TASK-1**:

**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 code into a cleaner, modular version using dictionary-based dispatch or separate functions:**

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:**

# Refactored Code

def rectangle(x, y):

    return x \* y

def square(x):

    return x \* x

def circle(x):

    return 3.14 \* x \* x

area\_dispatch = {

    "rectangle": lambda x, y: rectangle(x, y),

    "square": lambda x, y=0: square(x),

    "circle": lambda x, y=0: circle(x)

}

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

    return area\_dispatch[shape](x, y)

"""

Program to calculate the area of different shapes

using dictionary-based function dispatch.

"""

# Shape-specific functions

def rectangle(x, y):

    """Return area of a rectangle."""

    return x \* y

def square(x):

    """Return area of a square."""

    return x \* x

def circle(x):

    """Return area of a circle."""

    return 3.14 \* x \* x

# Dictionary dispatch for cleaner design

area\_dispatch = {

    "rectangle": lambda x, y: rectangle(x, y),

    "square": lambda x, y=0: square(x),

    "circle": lambda x, y=0: circle(x)

}

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

    """

    Calculate area based on shape type.

    Uses dictionary-based dispatch to call appropriate function.

    """

    if shape not in area\_dispatch:

        raise ValueError(f"Shape '{shape}' is not supported")

    return area\_dispatch[shape](x, y)

# ---- Test Cases ----

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

    print("Rectangle (10 x 5):", calculate\_area("rectangle", 10, 5))

    print("Square (side 7):", calculate\_area("square", 7))

    print("Circle (radius 3):", calculate\_area("circle", 3))

    # Additional tests

    print("Rectangle (4 x 6):", calculate\_area("rectangle", 4, 6))

    print("Square (side 12):", calculate\_area("square", 12))

    print("Circle (radius 10):", calculate\_area("circle", 10))

**OUTPUT:**

**A black screen with white numbers and symbols

AI-generated content may be incorrect.**

**OBSERVATION :**

The given code was refactored by removing repetitive if-elif checks and replacing them with a dictionary-based dispatch. Separate functions were created for each shape, which makes the code modular, cleaner, and easier to extend when adding new shapes in the future.

TASK -2:

**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 given legacy function to use with open() for safer file handling and add try-except for error handling:**

def read\_file(filename):

f = open(filename, "r")

data = f.read()

f.close()

return data

CODE GENERATED:

"""

Refactored file reader function with proper error handling.

"""

# Refactored Code

def read\_file(filename):

    """

    Safely reads the content of a file.

    Uses 'with open()' for automatic file closing

    and try-except for error handling.

    """

    try:

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

            return f.read()

    except FileNotFoundError:

        return f"Error: The file '{filename}' was not found."

    except Exception as e:

        return f"An error occurred: {e}"

# ---- Test Cases ----

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

    # Case 1: Reading an existing file

    with open("sample.txt", "w") as f:

        f.write("Hello, this is a test file.")

    print("Test 1 (existing file):")

    print(read\_file("sample.txt"))  # Should print the file content

    # Case 2: Reading a non-existing file

    print("\nTest 2 (missing file):")

    print(read\_file("nofile.txt"))  # Should return error message

OUTPUT:  
A black screen with white text

AI-generated content may be incorrect.

OBSERVATION:

The code was refactored by using with open() to automatically close files and by adding try-except blocks to handle errors gracefully. This makes the function safer, more reliable, and easier to maintain.

TASK – 3:

**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 given legacy Student class for better readability and modularity by improving naming, adding docstrings, and simplifying calculations.**

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:

# Refactored Student Class with Test Cases

class Student:

    """

    Represents a student with basic details and marks.

    """

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

        """

        Initialize student with name, age, and list of marks.

        """

        self.name = name

        self.age = age

        self.marks = marks

    def show\_details(self):

        """Prints student details in a readable format."""

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

    def total\_marks(self):

        """Returns the total marks of the student."""

        return sum(self.marks)

# ---- Test Cases ----

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

    # Test Case 1

    student1 = Student("Alice", 16, [85, 90, 78])

    student1.show\_details()

    print("Total Marks:", student1.total\_marks())

    print()  # Blank line for readability

    # Test Case 2

    student2 = Student("Bob", 17, [72, 88, 95])

    student2.show\_details()

    print("Total Marks:", student2.total\_marks())

OUTPUT:

A screenshot of a computer screen

AI-generated content may be incorrect.

OBSERVATION:

The refactored code improves variable names for clarity and stores marks in a list for flexibility. It adds docstrings to explain the purpose of methods.Printing is made more readable using f-strings.

TASK – 4:

**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 to make it more efficient and Pythonic using a list comprehension:**

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

squares = []

for i in nums:

squares.append(i \* i)

CODE GENERATED:

# Refactored Code using List Comprehension

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

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

print(squares)

OUTPUT:



OBSERVATION:

The loop was replaced with a **list comprehension**, making the code shorter and more readable.