**AI ASSISTED CODING**

**Lab assignment 13.3**

**Name:** Gattu Varshith

**Roll Number:** 2403A510E3

**Batch:** 24BTCAICSB05

**Task Description 1:**

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 code into a cleaner and modular design.

Use either a dictionary-based dispatch or separate functions for each shape.

Code to refactor:

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

Requirements:

- Remove the if-elif chain.

- Use dictionary dispatch OR separate functions for each shape.

- Ensure code is modular, readable, and maintainable.

**Code Generated:**

import math

def area\_rectangle(x, y):

    return x \* y

def area\_square(x):

    return x \* x

def area\_circle(r):

    return math.pi \* r \* r

# Dictionary-based dispatch

AREA\_FUNCTIONS = {

    "rectangle": lambda x, y: area\_rectangle(x, y),

    "square": lambda x, \_: area\_square(x),

    "circle": lambda r, \_: area\_circle(r)

}

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

    try:

        return AREA\_FUNCTIONS[shape](x, y)

    except KeyError:

        raise ValueError(f"Unsupported shape: {shape}")

# ---- Test cases ----

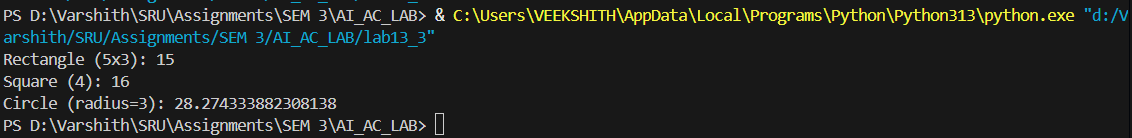
if \_\_name\_\_ == "\_\_main\_\_":

    print("Rectangle (5x3):", calculate\_area("rectangle", 5, 3))  # 15

    print("Square (4):", calculate\_area("square", 4))             # 16

    print("Circle (radius=3):", calculate\_area("circle", 3))      # 28.27...

**Output:**

****

**Observation:**

The refactored version eliminates the long if-elif chain by using dictionary-based dispatch, making the code cleaner and more modular. Each shape has its own dedicated function, which improves readability and maintainability. This design also makes it easier to extend the program with new shapes in the future.

**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  
Expected Output:  
AI refactors with with open() and try-except:

**Prompt:**

Refactor the following legacy Python code that lacks proper error handling.

Use `with open()` for better resource management and wrap it in a try-except block.

Code to refactor:

def read\_file(filename):

f = open(filename, "r")

data = f.read()

f.close()

return data

Requirements:

Use with open() instead of manually opening/closing.

Add error handling for missing files or read errors.

Keep the code clean and Pythonic.

Include at least one test case.

**Code Generated:**

def read\_file(filename):

    try:

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

            return f.read()

    except FileNotFoundError:

        print(f"Error: The file '{filename}' was not found.")

        return None

    except IOError as e:

        print(f"Error reading file '{filename}': {e}")

        return None

# ---- Test cases ----

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

    # Case 1: Existing file

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

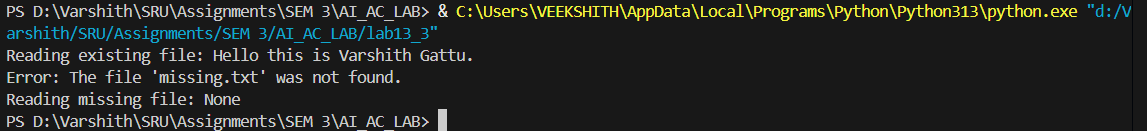
        f.write("Hello, World!")

    print("Reading existing file:", read\_file("sample1.txt")) # Should print file content

    # Case 2: Missing file

    print("Reading missing file:", read\_file("missing.txt"))  # Should print error and return None

**Output:**



**Observation:**

The refactored code uses with open() to ensure files are properly closed even if errors occur. Adding try-except improves robustness by gracefully handling missing or unreadable files. This makes the function safer for real-world applications where file access might fail.

**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:**

Refactor the following legacy Python class to improve readability and modularity.

Code to refactor:

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

Requirements:

Use meaningful attribute names (e.g., name, age, marks).

Add docstrings for the class and its methods.

Store marks in a list instead of individual variables.

Use sum(self.marks) for total calculation.

Improve print readability in details().

Add at least one test case.

**Code Generated:**

class Student:

    """

    A class to represent a student with basic details and marks.

    """

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

        """

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

        """

        self.name = name

        self.age = age

        self.marks = marks

    def details(self) -> None:

        """

        Print student details in a readable format.

        """

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

    def total\_marks(self) -> int:

        """

        Calculate the total marks obtained by the student.

        """

        return sum(self.marks)

# ---- Test case ----

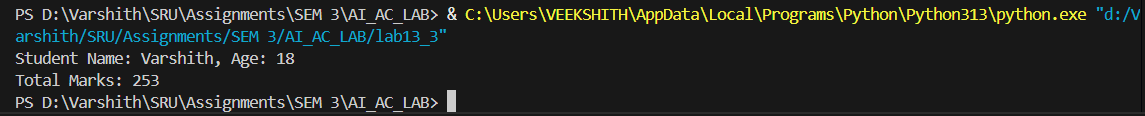
if \_\_name\_\_ == "\_\_main\_\_":

    student1 = Student("Alice", 20, [85, 90, 78])

    student1.details()

    print("Total Marks:", student1.total\_marks())  # 253

**Output:**



**Observation:**

The refactored class improves readability by using descriptive attribute names and structured methods. Storing marks in a list enables flexibility and simplifies calculations using sum(). Adding docstrings improves maintainability, while formatted string output enhances clarity. This design is modular and easily extendable.

**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 the following inefficient loop in Python to make it more concise and Pythonic.

Code to refactor:

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

squares = []

for i in nums:

squares.append(i \* i)

Requirements:

Replace the loop with a list comprehension.

Keep the code clean and readable.

Add at least one test case to verify output.

**Code Generated:**

# Refactored using list comprehension

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

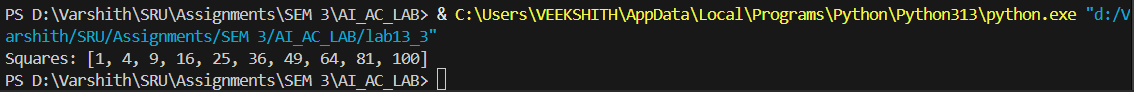
squares = [i \* i for i in nums]

# ---- Test case ----

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

    print("Squares:", squares)

**Output:**

****

**Observation:**

The refactored version replaces the explicit loop with a list comprehension, making the code more concise and Pythonic. It improves readability while maintaining the same functionality. This approach is both efficient and easier to extend for transformations.