ASSIGNMENT-13.3

NAME:ANUSHA PEDDAPELLI

BATCH NO.:06

ROLL NO.:2403A51103

TASK-1: Remove Repetition

PROMPT:

```
It uses a series of if/elif statements to calculate the area of different shapes.
I would like you to improve it by removing this conditional logic.
Please propose a solution that is more modular, scalable, and follows good object-oriented design principles.
```

CODE:

```python
import math

class Shape:
    """A base class for all shapes."""
    def area(self):
        raise NotImplementedError("Subclasses must implement this method")

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

# --- How to use it ---

# Create instances of shapes
my_rectangle = Rectangle(10, 5)
my_square = Square(7)
my_circle = Circle(3)

# Calculate areas without needing to know the shape type
print(f"Rectangle area: {my_rectangle.area()}") # Output: Rectangle area: 50
print(f"Square area: {my_square.area()}")       # Output: Square area: 49
print(f"Circle area: {my_circle.area()}")       # Output: Circle area: 28.274333882308138

# To add a new shape, just create a new class:
class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height

my_triangle = Triangle(10, 5)
print(f"Triangle area: {my_triangle.area()}")   # Output: Triangle area: 25.0
```

OUTPUT:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\Praneeth Cheekati\OneDrive\Desktop\ai> & "C:/Users/Praneeth Cheekati/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Praneeth Cheekati/OneDrive/Desktop/ai/13.3.py"
Rectangle area: 50
Square area: 49
Circle area: 28.274333882308138
Triangle area: 25.0
PS C:\Users\Praneeth Cheekati\OneDrive\Desktop\ai> & "C:/Users/Praneeth Cheekati/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Praneeth Cheekati/OneDrive/Desktop/ai/13.3.py"
```

powershell
Python: ...
Python: LAB...
Python: lab_...
Python: 12.1
Python: task...
Python: task...

OBSERVATION:

Please propose a solution that is more modular, scalable, and follows good object-oriented design principles.

The original calculate_area function uses a long if/elif chain to determine which area calculation to perform.
Violates the Open/Closed Principle:
To add a new shape, you must modify the existing function.
A well-designed system should be open for extension but closed for modification.
Poor Scalability:
As more shapes are added, the function becomes increasingly long and complex, making it harder to read and maintain.
Mixing Concerns:
The function mixes the logic for dispatching (figuring out the shape) with the calculation logic for all shapes.

TASK-2: Error Handling in Legacy Code

PROMPT:

```
 I have the following legacy Python function.
It's supposed to read a file, but it has a bug (.close() is called on a string) and no error handling. '
'Please refactor this code to be robust and safe. '
'It should use modern Python practices to handle potential '
'fileFileNotFoundError exceptions and guarantee that the file is always closed properly after being opened."
```

CODE:

13.3.py 9+ ● 13.3-1.py ×

13.3-1.py > ...

```python
def read_file_safely(filename: str) -> str | None:
    """
    Reads data from a file with robust error handling.

    This function uses a 'with' statement to ensure the file is automatically
    closed and a 'try...except' block to gracefully handle exceptions.

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

    Returns:
        str | None: The content of the file as a string, or None if an error occurs.
    """
    try:
        # The 'with' statement ensures f.close() is called automatically
        # after the block is exited, even if errors occur.
        with open(filename, "r", encoding="utf-8") as f:
            data = f.read()
        return data
    except FileNotFoundError:
        # Gracefully handle the case where the file does not exist.
        print(f"Error: The file '{filename}' was not found.")
        return None
    except IOError as e:
        # Handle other potential input/output errors (e.g., permissions).
        print(f"Error reading file '{filename}': {e}")
        return None

# --- How to use it ---

# 1. Create a dummy file to test the success case
with open("my_data.txt", "w") as f:
    f.write("Hello, world!")

# 2. Test with a file that exists
content = read_file_safely("my_data.txt")
if content:
```

Ln 50, Col 1 Spaces: 4 UTF-8 CRLF {} Python

```python
with open("my_data.txt", "w") as f:
    f.write("Hello, world!")

# 2. Test with a file that exists
content = read_file_safely("my_data.txt")
if content:
    print(f"Successfully read content: '{content}'")
# Expected Output: Successfully read content: 'Hello, world!'

print("-" * 20)

# 3. Test with a file that does NOT exist
non_existent_content = read_file_safely("file_that_does_not_exist.txt")
if non_existent_content is None:
    print("Function correctly handled the missing file.")
# Expected Output:
# Error: The file 'file_that_does_not_exist.txt' was not found.
# Function correctly handled the missing file.
```

OUTPUT:

PROBLEMS 143 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai> & "C:/Users/Praneeeth Cheekati/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Praneeeth Cheekati/OneDrive/Desktop/ai/13.3-1.py"
Successfully read content: 'Hello, world!'
--------------------
Error: The file 'file_that_does_not_exist.txt' was not found.
Function correctly handled the missing file.
PS C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai>
```

Ln 50, Col 1 Spaces: 4 UTF-8 CRLF {} Python Chat quota reached 3.11.9 (Microsoft Store) Go Live

OBSERVATION:

Observation

The refactored code is a significant improvement over the legacy version, making it robust, safe, and easier to understand.

1. **Bug Fix:** The original code would have crashed with an `AttributeError` because it incorrectly tried to call `.close()` on a string. The refactored code correctly reads the data into the `data` variable and lets the `with` statement handle closing the file.
2. **Resource Safety with `with open()`:** The `with` statement is the standard for working with resources like files in Python. It guarantees that the file is properly closed the moment the code exits the `with` block, regardless of whether it completes successfully or an error is raised. This prevents resource leaks.
3. **Error Handling with `try...except`:** The `try...except` block prevents the program from crashing if the file doesn't exist. Instead of an unhandled `FileNotFoundError`, it now catches the exception, prints a user-friendly error message, and returns `None`. This allows the code that calls the function to handle the failure gracefully.
4. **Modern Practices:** The use of type hints (`filename: str`, `-> str | None`) and specifying the file encoding (`encoding="utf-8"`) makes the code clearer, more predictable, and less prone to bugs when used by others.

TASK-3: Complex Refactoring PROMPT:

```
"""
I have this legacy Python Student class that needs refactoring.

 The variable names are cryptic (n, a, m1), it lacks documentation, and the way it handles marks is not scalable.
 Please refactor this class to improve its readability and modularity.
  I'd like to see clearer variable names, proper docstrings,'
  ' and a more flexible way to store and calculate the total marks, perhaps using a list."
```

CODE:

13.3.py 9+ | 13.3-1.py | from typing import List Untitled-1

```python
from typing import List

class Student:
    """
    Represents a student with their name, age, and a list of marks.
    """

    def __init__(self, name: str, age: int, marks: List[float]):
        """
        Initializes a Student object.

        Args:
            name (str): The full name of the student.
            age (int): The age of the student.
            marks (List[float]): A list of the student's marks.
        """
        self.name = name
        self.age = age
        self.marks = marks

    def __str__(self) -> str:
        """
        Returns a user-friendly string representation of the student's details.
        This method is automatically called by print().
        """
        return f"Name: {self.name}, Age: {self.age}"

    def calculate_total_marks(self) -> float:
        """
        Calculates the sum of all marks for the student.

        Returns:
            float: The total of the student's marks.
        """
        return sum(self.marks)

    def calculate_average_mark(self) -> float:
```

```python
class Student:
    def calculate_total_marks(self) -> float:
        Returns:
        """
        return sum(self.marks)

    def calculate_average_mark(self) -> float:
        """
        Calculates the average of all marks.

        Returns:
            float: The average mark, or 0.0 if there are no marks.
        """
        if not self.marks:
            return 0.0
        return sum(self.marks) / len(self.marks)

# --- How to use the refactored class ---

# Create a student instance with a list of marks
student1 = Student(name="John Doe", age=21, marks=[85, 90, 78, 92])

# Print details using the __str__ method
print(student1)
# Expected Output: Name: John Doe, Age: 21

# Calculate and print the total and average marks
total = student1.calculate_total_marks()
average = student1.calculate_average_mark()

print(f"Total Marks: {total}")      # Expected Output: Total Marks: 345
print(f"Average Mark: {average:.2f}") # Expected Output: Average Mark: 86.25
```

OUTPUT:

PROBLEMS 143 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai> & "C:/Users/Praneeeth Cheekati/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Praneeeth Cheekati/OneDrive/Desktop/ai/13.3-2.PY"
Name: John Doe, Age: 21
Total Marks: 345
Average Mark: 86.25
PS C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai>
```

powershell
Python: ...
Python: LAB...
Python: lab_...
Python: 12.1
Python: task...
Python: task...

OBSERVATION:

The original `Student` class is functional but suffers from several issues common in older or hastily written code:

- **Poor Naming:** Single-letter variable names like `n`, `a`, and `m1` are cryptic and make the code difficult to understand without context.
- **Lack of Documentation:** Without docstrings, a new developer has to read the implementation to understand what the class and its methods do.
- **Inflexible Design:** The marks (`m1`, `m2`, `m3`) are hardcoded as separate attributes. If a student had two marks or five marks, the class structure would need to be changed. This violates the Open/Closed principle.
- **Rigid Methods:** The `total` method is not scalable and would need to be updated every time the number of marks changes. The `details` method prints directly to the console, which mixes logic with presentation and makes the method less reusable.

The goal of the refactoring is to address these issues by applying modern Python best practices.

TASK-4: Inefficient Loop Refactoring

PROMPT:

nums = [1,2,3,4,5,6,7,8,9,10] squares = []
for i in nums: squares.append(i * i).give
me the refacted code

CODE:

13-3-4.py > ...

```python
# List of numbers from 1 to 10
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

squares = [i * i for i in nums]

# Printing the original list of numbers
print("Original numbers:", nums)

# Printing the list of squares
print("Squares of the numbers:", squares)

```

OUTPUT:

```
> & C:/Users/DELL/AppData/Local/anaconda3/python.exe "c:/Users/DELL/Desktop/vs code/.vscode/13-3-4.py"
Original numbers: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Squares of the numbers: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
PS C:\Users\DELL\Desktop\vs code\.vscode>
```

OBSERVATION:

The refactored code replaces the multi-line `for` loop with a single, highly expressive **list comprehension**. This is the preferred method in modern Python for several reasons:

1. **Conciseness and Readability:** The list comprehension `[i * i for i in nums]` is a single line that clearly and succinctly describes its purpose: "create a new list containing `i * i` for each `i` in `nums`". For Python developers, this is more readable than the original three-line loop.

2. **Performance:** List comprehensions are generally faster than explicit `for` loops that use `.append()`. This is because the looping and appending logic is handled by highly optimized C code at the interpreter level, avoiding the overhead of repeated `.append()` method calls in Python.

3. **Declarative Style:** The list comprehension is more *declarative* (it describes *what* you want) rather than *imperative* (it describes *how* to do it step-by-step). This often leads to code that is easier to reason about and less prone to bugs.

In summary, the AI-suggested refactoring to a list comprehension is a best-practice improvement that results in cleaner, faster, and more Pythonic code.