ASSIGNMENT-13.3

NAME:DONTHI MEGHANA

BATCH NO.:05

ROLL NO.:2403A510D9

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:

```
**Slapy X*

**Ilapy Table

**Class Angel

**Class A
```

OUTPUT:

```
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> & "C:\Users\Praneeeth Cheekati\AppData\Local\Microsoft\windowsApps\python3.11.exe" "C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai> & "C:\Users\
```

OBSERVATION:

```
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.

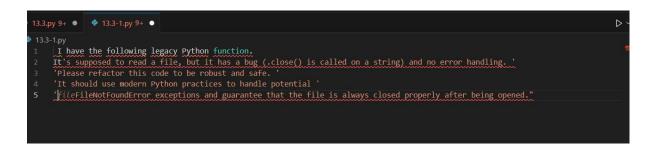
Poor Scalability:

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:



CODE:

```
₱ 13.3-1.py > ...
      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:
              with open(filename, "r", encoding="utf-8") as f:
                  data = f.read()
              return data
          except FileNotFoundError:
              print(f"Error: The file '{filename}' was not found.")
          except IOError as e:
              print(f"Error reading file '{filename}': {e}")
      # --- How to use it ---
      with open("my_data.txt", "w") as f:
          f.write("Hello, world!")
      content = read file safely("my data.txt")
      if content:
```

```
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!'

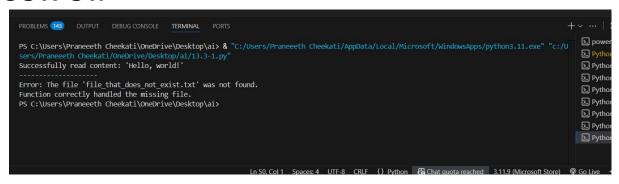
print("-" * 20)

# 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:



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+ ● ② 13.3-1.py
                             from typing import List Untitled-1
     from typing import List
         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:
```

```
class Student:
    def calculate total marks(self) -> float:
       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 ---
student1 = Student(name="John Doe", age=21, marks=[85, 90, 78, 92])
print(student1)
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 13 OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai> & "C:/Users/Praneeeth Cheekati/AppData/Local/Microsoft/WindowsApps/python3.11.ex

PS C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai> & "C:/Users/Praneeeth Cheekati/AppData/Local/Microsoft/WindowsApps/python3.11.ex

Description:

Python: LAB...

Python: Lab...

Python: lab...

Python: 12.1

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 > ...
1  # List of numbers from 1 to 10
2  nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
3
4  squares = [i * i for i in nums]
5
6  # Printing the original list of numbers
7  print("Original numbers:", nums)
8
9  # Printing the list of squares
10  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.