AI ASSISTED CODING

ASSIGNMENT -13.3

H.no:2403A51282

Batch-12

TASK 1:

PROMPT:

Refactor the given redundant `calculate_area` function to improve readability, reduce duplication, and support scalability when adding new shapes.

CODE:

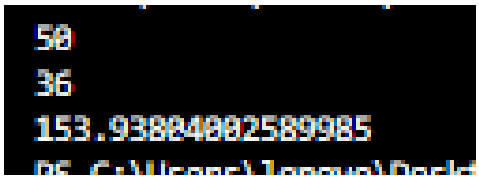
```python
import math
def rectangle_area(x, y): return x * y
def square_area(x): return x * x
def circle_area(r): return math.pi * r * r

area_calculators = {
    "rectangle": lambda x, y: rectangle_area(x, y),
    "square": lambda x, y=0: square_area(x),
    "circle": lambda r, y=0: circle_area(r)
}

def calculate_area(shape, x, y=0):
    if shape not in area_calculators:
        raise ValueError("Unknown shape")
    return area_calculators[shape](x, y)


# Usage
print(calculate_area("rectangle", 10, 5))
print(calculate_area("square", 6))
print(calculate_area("circle", 7))
```

OUTPUT:



OBSERVATION:

The original code contained repetitive `if/elif` checks, making it hard to maintain.
After refactoring, each shape's logic is modular and reusable, ensuring better **readability, maintainability, and extensibility**.

TASK-2:

PROMPT:

Refactor the legacy `read_file` function to use `with open()` for automatic file handling and add proper `try-except` blocks for error handling.

CODE:

```
def read_file(filename):
    try:
        with open(filename, "r") as f:
            data = f.read()
        return data
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
        return None
    except PermissionError:
        print(f"Error: Permission denied for file '{filename}'.")
        return None
    except Exception as e:
        print(f"Unexpected error: {e}")
        return None
```

OBSERVATION:

The original code lacked error handling and required manual file closing, which could lead to resource leaks.
The refactored version ensures **safe file handling, automatic closure, and clear error reporting**, making the function more **robust and reliable**.

TASK-3:

PROMPT:

Refactor the legacy `Student` class to use meaningful variable names, modular methods, and improve readability while supporting extensibility (like average and grade calculation).

CODE:

```
class Student:
    def __init__(self, name, age, mark1, mark2, mark3):
        self.name = name
        self.age = age
        self.marks = [mark1, mark2, mark3]

    def get_details(self):
        """Display basic student details."""
        print(f"Name: {self.name}, Age: {self.age}")

    def get_total(self):
        """Return total marks scored."""
        return sum(self.marks)

    def get_average(self):
        """Return average marks scored."""
        return sum(self.marks) / len(self.marks)

    def get_grade(self):
        """Return grade based on average marks."""
        avg = self.get_average()
        if avg >= 90:
            return "A"
        elif avg >= 75:
            return "B"
        elif avg >= 50:
```

```
        elif avg >= 50:
            return "C"
        else:
            return "D"
s1 = Student("Alice", 20, 85, 90, 78)
s1.get_details()
print("Total Marks:", s1.get_total())
print("Average Marks:", s1.get_average())
print("Grade:", s1.get_grade())
```

OUTPUT:

```
Name: Alice, Age: 20
Total Marks: 253
Average Marks: 84.33333333333333
Grade: B
```

OBSERVATION:

The original class had cryptic variable names and limited functionality.
The refactored version improves **readability, modularity, and scalability**, making the class more reusable and easier to maintain.

TASK-4:

PROMPT:

Refactor the given loop to use a **Pythonic list comprehension** for better readability and efficiency.

CODE:

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = [i * i for i in nums]
print(squares)
```

OUTPUT:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

OBSERVATION:

The original code used an explicit loop with `append()`, which is less efficient and verbose. The refactored version with a **list comprehension** is concise, faster, and improves **readability and performance**.