Assignment-13.3

Goli Harini Reddy

2303A51544

Batch:08

Task 1: Refactoring – Removing Code Duplication

Objective

To eliminate repeated logic by extracting reusable functions.

Task Description

Use AI assistance to refactor a legacy Python script that contains repeated

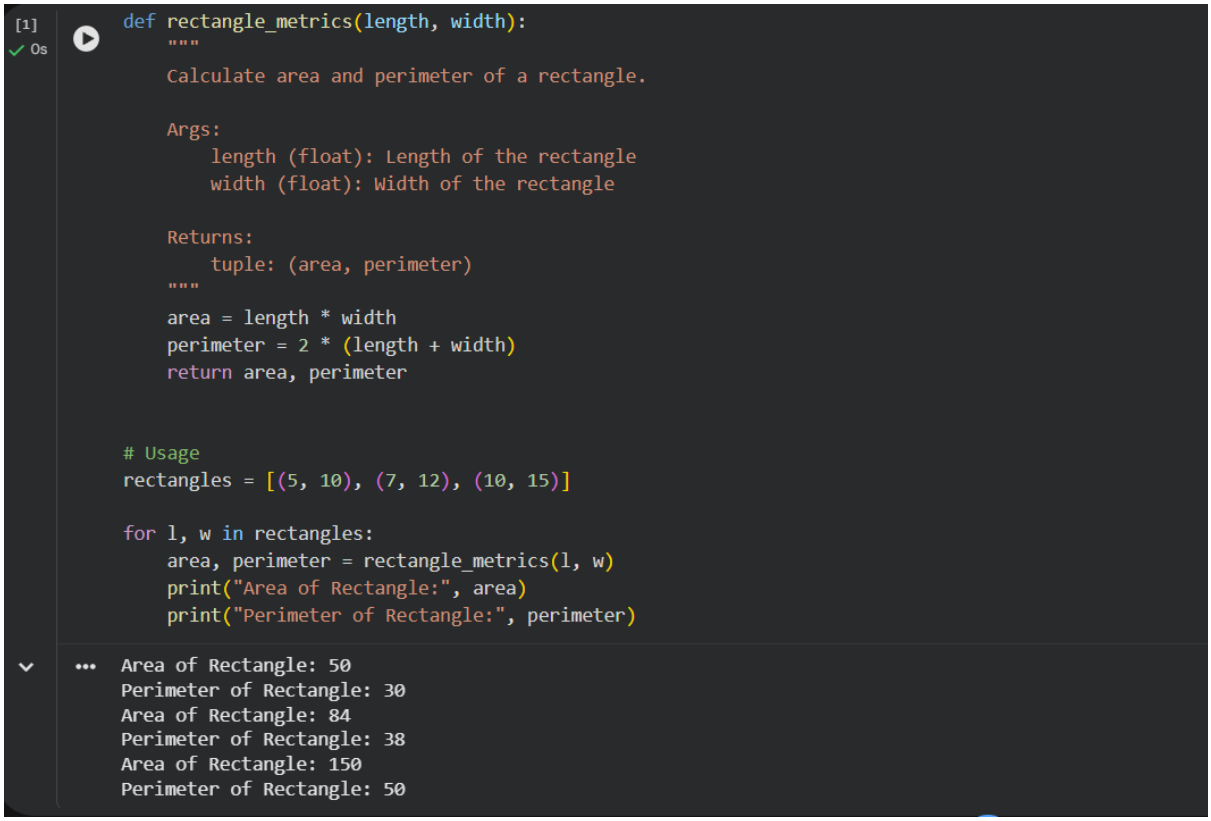
blocks of code calculating the area and perimeter of rectangles.

Starter (Legacy) Code

```
# Legacy script with repeated logic
print("Area of Rectangle:", 5 * 10)
print("Perimeter of Rectangle:", 2 * (5 + 10))
print("Area of Rectangle:", 7 * 12)
print("Perimeter of Rectangle:", 2 * (7 + 12))
print("Area of Rectangle:", 10 * 15)
print("Perimeter of Rectangle:", 2 * (10 + 15))
```

Expected Outcome

- A reusable function to calculate area and perimeter
- No duplicated code blocks
- Proper docstrings for all functions



```
[1]
✓ 0s
```

```python
def rectangle_metrics(length, width):
    """
    Calculate area and perimeter of a rectangle.

    Args:
        length (float): Length of the rectangle
        width (float): Width of the rectangle

    Returns:
        tuple: (area, perimeter)
    """
    area = length * width
    perimeter = 2 * (length + width)
    return area, perimeter


# Usage
rectangles = [(5, 10), (7, 12), (10, 15)]

for l, w in rectangles:
    area, perimeter = rectangle_metrics(l, w)
    print("Area of Rectangle:", area)
    print("Perimeter of Rectangle:", perimeter)
```

```
Area of Rectangle: 50
Perimeter of Rectangle: 30
Area of Rectangle: 84
Perimeter of Rectangle: 38
Area of Rectangle: 150
Perimeter of Rectangle: 50
```

Observation:

Before Refactoring

- Same area and perimeter logic repeated multiple times.
- Difficult to maintain — any formula change must be updated in many places.
- Code length was unnecessarily large.
- Higher risk of human error.

After Refactoring

- Logic extracted into a single reusable function.
- Code became shorter and cleaner.
- Easy to maintain and extend.
- Improved modularity and scalability.

Task 2: Refactoring – Optimizing Loops and Conditionals

Objective

To improve performance by replacing inefficient nested loops with optimized structures.

Task Description

Use AI to analyze and refactor a script that checks the presence of elements using nested loops.

Starter (Legacy) Code

```
names = ["Alice", "Bob", "Charlie", "David"]
search_names = ["Charlie", "Eve", "Bob"]
for s in search_names:
found = False
for n in names:
if s == n:
found = True
if found:
print(f"{s} is in the list")
else:
print(f"{s} is not in the list")
```

Expected Outcome

- Optimized solution using set lookups or comprehensions
- Performance comparison before and after refactoring

Task 2: Refactoring – Optimizing Loops and Conditionals

[2] ✓ 0s

```
names = ["Alice", "Bob", "Charlie", "David"]
search_names = ["Charlie", "Eve", "Bob"]

# Convert list to set for O(1) lookup
name_set = set(names)

for s in search_names:
    if s in name_set:
        print(f"{s} is in the list")
    else:
        print(f"{s} is not in the list")
```

```
Charlie is in the list
Eve is not in the list
Bob is in the list
```

Comparison:

Version	Time Complexity	Reason
Legacy nested loops	O(n × m)	checks every element
Refactored (set)	O(n + m)	constant time lookup

Observation:

Before Refactoring

- Used nested loops.
- Time complexity was **O(n × m)**.
- Inefficient for large datasets.
- More comparisons performed than necessary.

After Refactoring

- Used a **set** for membership lookup.
- Time complexity improved to **O(n + m)**.
- Code became more Pythonic and readable.
- Performance improved significantly for large inputs.

Task 3: Refactoring – Extracting Reusable Functions

Objective

To modularize code by extracting calculations into reusable functions.

Task Description

Refactor a legacy script where price and tax calculations are written inline.

Starter (Legacy) Code

```
price = 250
tax = price * 0.18
total = price + tax
print("Total Price:", total)
price = 500
tax = price * 0.18
total = price + tax
print("Total Price:", total)
```

Expected Outcome

- A function calculate_total(price)
- Cleaner and reusable code structure
- Proper documentation

Task 3: Refactoring – Extracting Reusable Functions

```
[3]
✓ 0s
def calculate_total(price):
    """
    Calculate total price including tax.

    Args:
        price (float): Base price

    Returns:
        float: Total price including 18% tax
    """
    TAX_RATE = 0.18
    tax = price * TAX_RATE
    total = price + tax
    return total


# Usage
prices = [250, 500]

for p in prices:
    print("Total Price:", calculate_total(p))
```

```
Total Price: 295.0
Total Price: 590.0
```

Observation:

Before Refactoring

- Tax calculation logic duplicated.
- Hard to update tax rate in multiple places.
- Poor modularity.
- Violates DRY principle.

After Refactoring

- Created reusable function calculate_total(price).
- Centralized tax logic.
- Easier updates and testing.
- Cleaner program structure.

Task 4: Refactoring – Replacing Hardcoded Values with Constants

Objective

To improve maintainability by replacing magic numbers with named constants.

Task Description

Use AI to identify hardcoded values and replace them with constants.

Starter (Legacy) Code

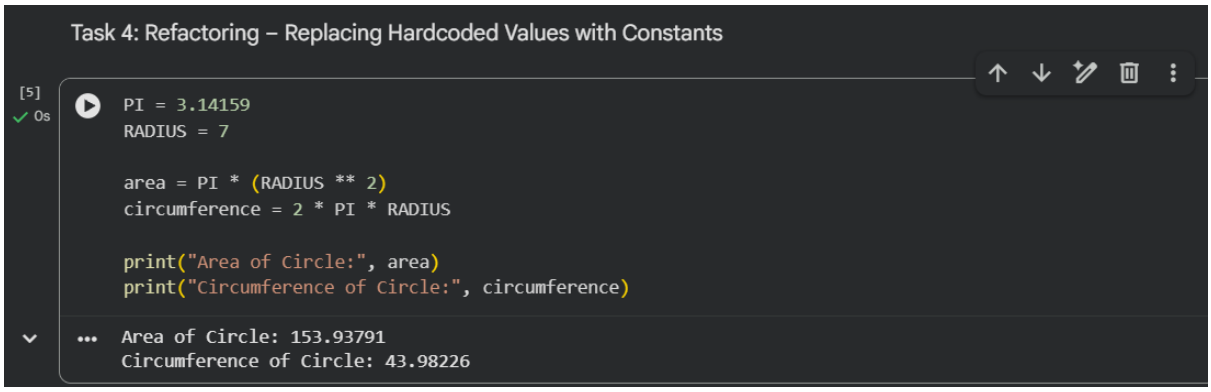
print("Area of Circle:", 3.14159 * (7 ** 2))

print("Circumference of Circle:", 2 * 3.14159 * 7)

Expected Outcome

• Constants such as PI and RADIUS

• Cleaner, maintainable code



Observation:

Before Refactoring

- Magic numbers (3.14159 and 7) used directly.
- Poor readability.
- Difficult to update values.
- Risk of inconsistency.

After Refactoring

- Introduced named constants (PI, RADIUS).
- Code became self-explanatory.
- Easier future modifications.
- Improved maintainability.

Task 5: Refactoring – Improving Variable Naming and Readability

Objective

To enhance readability using descriptive variable names and comments.

Task Description

Refactor a script with unclear variable names.

Starter (Legacy) Code

a = 10

b = 20

c = a * b / 2

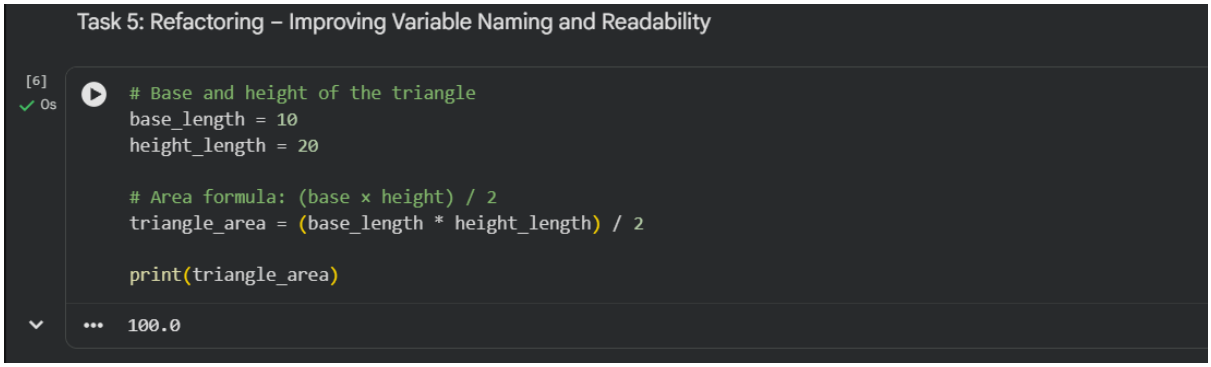
print(c)

Expected Outcome

• Descriptive variable names

• Inline comments explaining logic

• Identical output



Observation:

Before Refactoring

- Variables a, b, c were unclear.
- Hard to understand program purpose.
- Poor code readability.
- Not suitable for team development.

After Refactoring

- Used descriptive names (base_length, height_length, triangle_area).

- Added inline comments.
- Logic became self-documenting.
- Output remained identical.