

ASSIGNMENT 13.1

RAVITEJA VADLURI || HT : 2303A51942 || BATCH 12

Task: Use AI to refactor a given Python script that contains multiple repeated code blocks.

Original code with repeated blocks

```
1 # Task: Use AI to refactor a given Python script that contains multiple repeated code blocks.
2 # Original code with repeated blocks
3 def calculate_area(radius):
4     area = 3.14 * radius * radius
5     return area
6
7 def calculate_circumference(radius):
8     circumference = 2 * 3.14 * radius
9     return circumference
10
11 def calculate_diameter(radius):
12     diameter = 2 * radius
13     return diameter
14
15 # Refactored code using a single function to handle all calculations
16 def calculate_circle_properties(radius):
17     area = 3.14 * radius * radius
18     circumference = 2 * 3.14 * radius
19     diameter = 2 * radius
20     return area, circumference, diameter
21
22 # Example usage
23 radius = 5
24 area, circumference, diameter = calculate_circle_properties(radius)
25 print(f"Area: {area}")
26 print(f"Circumference: {circumference}")
27 print(f"Diameter: {diameter}")
```

OUTPUT

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\ADMIN\Desktop\AI_AC LAB> & C:\Users\ADMIN\AppData\Local\Python\pythoncore-3.14-64\python.exe "c:/Users/ADMIN/Desktop/AI_AC LAB/.py"
Area: 78.5
Circumference: 31.400000000000002
Diameter: 10
PS C:\Users\ADMIN\Desktop\AI_AC LAB>
```

OBSERVATIONS

Task 1 – Refactoring repeated circle calculation functions

Observation:

- The original code had separate functions for area, circumference, and diameter.
- The refactored version combines all calculations into one function (calculate_circle_properties).
- This reduces repetition, improves maintainability, and keeps related logic together.

Result: Cleaner code with fewer functions and easier reuse.

#Task 2: Use AI to analyze a Python script with nested loops and complex conditionals.

Original code with nested loops and complex conditiona

```
#Task: Use AI to analyze a Python script with nested loops and complex conditionals.
# Original code with nested loops and complex conditionals
def process_data(data):
    result = []
    for item in data:
        if isinstance(item, list):
            for sub_item in item:
                if sub_item > 10:
                    result.append(sub_item * 2)
                else:
                    result.append(sub_item)
        elif isinstance(item, dict):
            for key, value in item.items():
                if value > 5:
                    result.append(value * 3)
                else:
                    result.append(value)
        else:
            if item > 0:
                result.append(item * 4)
            else:
                result.append(item)
    return result

# Refactored code using a single function to handle all cases
def process_data(data):
    result = []
    for item in data:
        if isinstance(item, list):
            result.extend([sub_item * 2 if sub_item > 10 else sub_item for sub_item in item])
        elif isinstance(item, dict):
            result.extend([value * 3 if value > 5 else value for value in item.values()])
```

OUTPUT

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell + v
PS C:\Users\ADMIN\Desktop\AI_AC LAB> & C:\Users\ADMIN\AppData\Local\Python\pythoncore-3.14-64\python.exe "c:/Users/ADMIN/Desktop/AI_AC LAB/.py"
Area: 78.5
Circumference: 31.400000000000002
Diameter: 10
PS C:\Users\ADMIN\Desktop\AI_AC LAB> & C:\Users\ADMIN\AppData\Local\Python\pythoncore-3.14-64\python.exe "c:/Users/ADMIN/Desktop/AI_AC LAB/.py"
[4, -2, 5, 30, 3, 21]
PS C:\Users\ADMIN\Desktop\AI_AC LAB>
```

Task 2 – Analyzing nested loops and complex conditionals

Observation:

- Original code used multiple nested loops and if-else blocks, making it lengthy.
- Refactored version uses **list comprehensions and extend()**, simplifying logic.
- Improves readability and performance while preserving functionality.

Result: Reduced complexity and more Pythonic style.

#Task 3: Use AI to refactor a legacy script where multiple calculations are embedded directly inside the main code block

Original code with calculations embedded in the main code block

```
63
64 #Task: Use AI to refactor a legacy script where multiple calculations are embedded directly inside the main code block
65 # Original code with calculations embedded in the main code block
66 def main():
67     radius = 5
68     area = 3.14 * radius * radius
69     circumference = 2 * 3.14 * radius
70     diameter = 2 * radius
71     print(f"Area: {area}")
72     print(f"Circumference: {circumference}")
73     print(f"Diameter: {diameter}")
74 # Refactored code with calculations moved to a separate function
75 def calculate_circle_properties(radius):
76     area = 3.14 * radius * radius
77     circumference = 2 * 3.14 * radius
78     diameter = 2 * radius
79     return area, circumference, diameter
80 def main():
81     radius = 5
82     area, circumference, diameter = calculate_circle_properties(radius)
83     print(f"Area: {area}")
84     print(f"Circumference: {circumference}")
85     print(f"Diameter: {diameter}")
86 if __name__ == "__main__":
87     main()
```

OUTPUT

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell + v
PS C:\Users\ADMIN\Desktop\AI_AC LAB> & C:\Users\ADMIN\AppData\Local\Python\pythoncore-3.14-64\python.exe "c:/Users/ADMIN/Desktop/AI_AC LAB/.py"
Area: 78.5
Circumference: 31.400000000000002
Diameter: 10
PS C:\Users\ADMIN\Desktop\AI_AC LAB> █
```

Task 3 – Moving calculations out of the main block

Observation:

- Original script mixed calculation logic inside main().
- Refactored version separates logic into a function (calculate_circle_properties).
- Follows **modular programming** principles.

Result: Code becomes reusable, testable, and easier to maintain

Task4 : Use AI to identify and replace all hardcoded “magic numbers” in the code with named constants.

```
89 # Task: Use AI to identify and replace all hardcoded “magic numbers” in the code with named constants.
90 # Original code with hardcoded magic numbers
91 def calculate_area(radius):
92     area = 3.14 * radius * radius
93     return area
94 def calculate_circumference(radius):
95     circumference = 2 * 3.14 * radius
96     return circumference
97 def calculate_diameter(radius):
98     diameter = 2 * radius
99     return diameter
100 #Refactored code with named constants
101 PI = 3.14
102 def calculate_area(radius):
103     area = PI * radius * radius
104     return area
105 def calculate_circumference(radius):
106     circumference = 2 * PI * radius
107     return circumference
108 def calculate_diameter(radius):
109     diameter = 2 * radius
110     return diameter
111 # Example usage
112 radius = 5
113 area = calculate_area(radius)
114 circumference = calculate_circumference(radius)
115 diameter = calculate_diameter(radius)
116 print(f"Area: {area}")
117 print(f"Circumference: {circumference}")
118 print(f"Diameter: {diameter}")
```

Task 4 – Replacing magic numbers with named constants

Observation:

Hardcoded values like 3.14 were replaced with a constant PI.

Improves readability and avoids errors if the value needs updating.

Result: Better code clarity and maintainability.

```
120
121 # Task 5: Use AI to improve readability by renaming unclear variables and adding inline comments.
122 # Original code with unclear variable names and no comments
123 def calc(r):
124     a = 3.14 * r * r
125     c = 2 * 3.14 * r
126     d = 2 * r
127     return a, c, d
128 # Refactored code with improved variable names and inline comments
129 PI = 3.14
130 def calculate_circle_properties(radius):
131     # Calculate the area of the circle
132     area = PI * radius * radius
133     # Calculate the circumference of the circle
134     circumference = 2 * PI * radius
135     # Calculate the diameter of the circle
136     diameter = 2 * radius
137     return area, circumference, diameter
138 # Example usage
139 radius = 5
140 area, circumference, diameter = calculate_circle_properties(radius)
141 print(f"Area: {area}")
142 print(f"Circumference: {circumference}")
143 print(f"Diameter: {diameter}")
144
```

Task 5 – Improving readability (variable names + comments)

Observation:

Unclear variables (a, c, d) were replaced with meaningful names (area, circumference, diameter).

Inline comments explain each calculation.

Result: Code becomes beginner-friendly and self-documenting.

Task 6: Use AI to refactor a Python script that contains repeated if–else logic for grading students.

```
#Task6: Use AI to refactor a Python script that contains repeated if-else logic for grading students.  
# Original code with repeated if-else logic
```

```
def grade_student(score):  
    if score >= 90:  
        return 'A'  
    elif score >= 80:  
        return 'B'  
    elif score >= 70:  
        return 'C'  
    elif score >= 60:  
        return 'D'  
    else:  
        return 'F'
```

```
# # Refactored code using a single function to handle grading logic
```

```
def grade_student(score):  
    if score >= 90:  
        return 'A'  
    elif score >= 80:  
        return 'B'  
    elif score >= 70:  
        return 'C'  
    elif score >= 60:  
        return 'D'  
    else:  
        return 'F'
```

OUTPUT

```
170 # Example usage  
171 scores = [95, 85, 75, 65, 55]  
172 grades = [grade_student(score) for score in scores]  
173 print(grades)  
174
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

powershell + v

```
.py"  
PS C:\Users\ADMIN\Desktop\AI_AC LAB> & C:\Users\ADMIN\AppData\Local\Python\pythoncore-3.14-64\python.exe "c:/Users/ADMIN/Desktop/AI_AC LAB/  
.py"  
PS C:\Users\ADMIN\Desktop\AI_AC LAB> & C:\Users\ADMIN\AppData\Local\Python\pythoncore-3.14-64\python.exe "c:/Users/ADMIN/Desktop/AI_AC LAB/  
.py"  
['A', 'B', 'C', 'D', 'F']  
PS C:\Users\ADMIN\Desktop\AI_AC LAB>
```

Task 6 – Refactoring repeated grading logic

Observation:

- Grading logic was already centralized in one function.
- Example usage with list comprehension improves usage pattern.
- Demonstrates reusable business logic.

Result: Consistent grading logic and scalable for multiple students

- **Task:7 Use AI to refactor procedural input–processing logic into functions.**

```
# #Task: Use AI to refactor procedural input-processing logic into functions.
# Original code with procedural input-processing logic
# name = input("Enter your name: ")
# age = int(input("Enter your age: "))
# if age >= 18:
#     print(f"Hello {name}, you are an adult.")
# else:
#     print(f"Hello {name}, you are a minor.")
# Refactored code with input-processing logic moved to a function
def process_user_input():
    name = input("Enter your name: ")
    age = int(input("Enter your age: "))
    if age >= 18:
        print(f"Hello {name}, you are an adult.")
    else:
        print(f"Hello {name}, you are a minor.")
if __name__ == "__main__":
    process_user_input()
```

OUTPUT

```
184 def process_user_input():
185     name = input("Enter your name: ")
186     age = int(input("Enter your age: "))
187     if age >= 18:
188         print(f"Hello {name}, you are an adult.")
189     else:
190         print(f"Hello {name}, you are a minor.")
191 if __name__ == "__main__":
192     process_user_input()
193
194
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell + v

```
PS C:\Users\ADMIN\Desktop\AI_AC LAB> & C:\Users\ADMIN\AppData\Local\Python\pythoncore-3.14-64\python.exe "c:/Users/ADMIN/Desktop/AI_AC LAB/.py"
Enter your name: alexa
Enter your age: 19
Hello alexa, you are an adult.
PS C:\Users\ADMIN\Desktop\AI_AC LAB>
```

Task 7 – Converting procedural input logic into a function

Observation:

Input handling moved into process_user_input() function.

Allows reuse and easier testing.

Use of if __name__ == "__main__" follows best practice.

Result: Better program structure and modularity.

Task 8: Use AI to refactor inefficient list processing logic.

```
---
194
195 # task: Use AI to refactor inefficient list processing logic.
196 # Original code with inefficient list processing logic
197 # def process_list(input_list):
198 #     output_list = []
199 #     for item in input_list:
200 #         if item % 2 == 0:
201 #             output_list.append(item * 2)
202 #         else:
203 #             output_list.append(item * 3)
204 #     return output_list
205 # Refactored code using list comprehension for efficient list processing
206 def process_list(input_list):
207     return [item * 2 if item % 2 == 0 else item * 3 for item in input_list]
208 # Example usage
209 input_list = [1, 2, 3, 4, 5]
210 output_list = process_list(input_list)
211 print(output_list)
212
213
214
```

OUTPUT

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
powershell + v
PS C:\Users\ADMIN\Desktop\AI_AC LAB> & C:\Users\ADMIN\AppData\Local\Python\pythoncore-3.14-64\python.exe "c:/Users/ADMIN/Desktop/AI_AC LAB/.py"
Enter your name: alexa
Enter your age: 19
Hello alexa, you are an adult.
PS C:\Users\ADMIN\Desktop\AI_AC LAB> & C:\Users\ADMIN\AppData\Local\Python\pythoncore-3.14-64\python.exe "c:/Users/ADMIN/Desktop/AI_AC LAB/.py"
[3, 4, 9, 8, 15]
PS C:\Users\ADMIN\Desktop\AI_AC LAB> █
```

Task 8 – Refactoring inefficient list processing

Observation:

- Original loop replaced with **list comprehension**.
- Code becomes shorter and faster.
- Functional behavior remains the same.

Result: Efficient and Pythonic list processing.

All OBSERVATIONS

Task 1 – Refactoring repeated circle calculation functions

Observation:

- The original code had separate functions for area, circumference, and diameter.
- The refactored version combines all calculations into one function (`calculate_circle_properties`).
- This reduces repetition, improves maintainability, and keeps related logic together.

Result: Cleaner code with fewer functions and easier reuse.

Task 2 – Analyzing nested loops and complex conditionals

Observation:

- Original code used multiple nested loops and if-else blocks, making it lengthy.
- Refactored version uses list comprehensions and `extend()`, simplifying logic.
- Improves readability and performance while preserving functionality.
- **Result:** Reduced complexity and more Pythonic style.

Task 3 – Moving calculations out of the main block

Observation:

- Original script mixed calculation logic inside `main()`.
- Refactored version separates logic into a function (`calculate_circle_properties`).
- Follows modular programming principles.

Result: Code becomes reusable, testable, and easier to maintain.

Task 4 – Replacing magic numbers with named constants

Observation:

- Hardcoded values like 3.14 were replaced with a constant `PI`.
- Improves readability and avoids errors if the value needs updating.

Result: Better code clarity and maintainability.

Task 5 – Improving readability (variable names + comments)

Observation:

- Unclear variables (`a`, `c`, `d`) were replaced with meaningful names (`area`, `circumference`, `diameter`).
- Inline comments explain each calculation.

Result: Code becomes beginner-friendly and self-documenting.

Task 6 – Refactoring repeated grading logic

Observation:

- Grading logic was already centralized in one function.
- Example usage with list comprehension improves usage pattern.
- Demonstrates reusable business logic.

Result: Consistent grading logic and scalable for multiple students.

Task 7 – Converting procedural input logic into a function

Observation:

- Input handling moved into `process_user_input()` function.
- Allows reuse and easier testing.
- Use of `if __name__ == "__main__"` follows best practice.

Result: Better program structure and modularity.

Task 8 – Refactoring inefficient list processing

Observation:

- Original loop replaced with list comprehension.
- Code becomes shorter and faster.
- Functional behavior remains the same.

Result: Efficient and Pythonic list processing.

