School of Computer Science and Artificial Intelligence

Lab Assignment - 13.5

Program : B. Tech (CSE)

Specialization :AIML

Course Title :AI Assisted Coding

Course Code :23CS002PC304

Semester : VI

Academic Session : 2025-2026

Name of Student : Shiva Charan

Enrollment No. : 2303A52160

Batch No. : 34

Date :27/2/26

Lab 13: Code Refactoring – Improving Legacy Code with AI Suggestions:

Task Description #1 (Refactoring – Removing Global Variables)

* Sample Legacy Code:

```
rate = 0.1
def calculate_interest(amount):
    return amount * rate
print(calculate_interest(1000))
```

Prompt

Refactor the following Python code to eliminate unnecessary global variables. Pass required values using function parameters to improve modularity, testability, and maintainability. Follow Python best practices.

*Refactored Code (Passing Parameter):

```
def calculate_interest(amount, rate):
    return amount * rate

def main():
    rate = 0.1
    print(calculate_interest(1000, rate))

if __name__ == "__main__":
    main()
```

```
100.0
```

Improvement:

Removed global dependency

Increased modularity

Easier unit testing

Task Description #2 : (Refactoring Deeply Nested Conditionals):

* Sample Legacy Code:

```
score = 78
if score >= 90:
    print("Excellent")
else:
    if score >= 75:
print("Very Good")
    else:
        if score >= 60:
            print("Good")
        else:
print("Needs Improvement")
```

Prompt :

```
Refactor the following deeply nested if-else structure into a cleaner and more readable format. Improve logical flow, maintainability, and follow Python best practices.
```

Refactored Code

```python
def evaluate_score(score):
    # Define the grading scale as a list of (threshold, grade) tuples
    # Order matters: higher thresholds should come first.
    grades = [
        (90, "Excellent"),
        (75, "Very Good"),
        (60, "Good")
    ]

    for threshold, grade_message in grades:
        if score >= threshold:
            return grade_message

    # Default case if no threshold is met
    return "Needs Improvement"

print(evaluate_score(78))
```

Output:

```
Very Good
```

Improvement:

Easily extendable

Logical simplification

Cleaner structure

Centralized rule definition

Task 3 (Refactoring Repeated File Handling Code):

* Sample Legacy Code:

```
f = open("data1.txt")

print(f.read())

f.close()

f = open("data2.txt")

print(f.read())

f.close()
```

Prompt :

```
Refactor the repeated file open/read/close logic using Python context managers. Apply the DRY principle and create a reusable function to improve maintainability and safety.
```

Refactored Code

```
# Create data1.txt
with open("data1.txt", "w") as f:
    f.write("This is the content of data1.txt.\n")

# Create data2.txt
with open("data2.txt", "w") as f:
    f.write("This is the content of data2.txt.\n")

def read_file(filename):
    with open(filename, "r") as f:
        print(f.read())

filenames = ["data1.txt", "data2.txt"]
for filename in filenames:
    read_file(filename)
```

Output:

```
This is the content of data1.txt.

This is the content of data2.txt.
```

Improvement:

Better testability

Separation of logic and output

More flexible for future use

Task 4 (Optimizing Search Logic)

- **Task: Refactor inefficient linear searches using appropriate data structures**.

*** Sample Legacy Code:**

```
users = ["admin", "guest", "editor", "viewer"]

name = input("Enter username: ")
found = False

for u in users:
    if u == name:
        found = True

print("Access Granted" if found else "Access Denied")
```

prompt:

```
Refactor the inefficient linear search logic using a more appropriate data
structure. Improve time complexity and justify the choice of data structure.
```

Refactored Code:

```
users = {"admin", "guest", "editor", "viewer"}  # Set for fast lookup

name = input("Enter username: ")

if name in users:
    print("Access Granted")
else:
    print("Access Denied")
```

Output:

```
Enter username: guest
Access Granted
```

Improvement:

Faster for large user databases

Cleaner and more readable

Better data structure choice

Task 5 (Refactoring Procedural Code into OOP Design)

• Task: Use AI to refactor procedural code into a class-based design.

* Sample Legacy Code:

```
salary = 50000

tax = salary * 0.2

net = salary - tax

print(net)
```

prompt:

```
Refactor the following procedural salary calculation code into an object-oriented design. Apply encapsulation and create a class with appropriate attributes and methods.
```

Refactored Code:

```
class EmployeeSalaryCalculator:
    def __init__(self, salary, tax_rate=0.2):
        self.salary = salary
        self.tax_rate = tax_rate

    def calculate_tax(self):
        return self.salary * self.tax_rate

    def calculate_net_salary(self):
        return self.salary - self.calculate_tax()


employee = EmployeeSalaryCalculator(50000)
print(employee.calculate_net_salary())
```

Output:

```
40000.0
```

Improvement:

Encapsulation

Salary and tax rate are stored inside the object.

Logic is wrapped inside methods.

✔ Reusability

Can create multiple employee objects.

Different tax rates can be passed.

✔ Maintainability

Easy to modify tax logic.

Code is structured and organized.

✔ Scalability

Can extend class with:

Bonus calculation

Allowances

Deductions

Annual salary reports

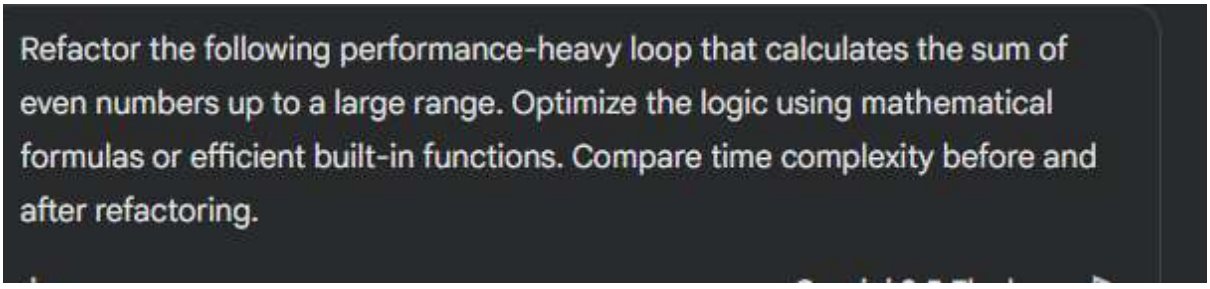
Task 6 (Refactoring for Performance Optimization)

Task: Use AI to refactor a performance-heavy loop handling large data.

* Sample Legacy Code:

```
total = 0

for i in range(1, 1000000):

    if i % 2 == 0:

        total += i


print(total)
```

prompt:



Refactored Code:

```python
def sum_even_naive(limit):
    """Calculates the sum of even numbers up to 'limit' using a naive for loop."""
    total_sum = 0
    for num in range(0, limit + 1, 2):
        total_sum += num
    return total_sum

# Demonstrate the function with an example
limit_value = 100
result = sum_even_naive(limit_value)
print(f"The sum of even numbers up to {limit_value} is: {result}")

limit_value = 10
result = sum_even_naive(limit_value)
print(f"The sum of even numbers up to {limit_value} is: {result}")
```

```
The sum of even numbers up to 100 is: 2550
The sum of even numbers up to 10 is: 30
```

Implement Optimized Built-in Method

Subtask:

Implement an optimized method to calculate the sum of even numbers up to a specified limit using the built-in `sum(range())` function.

Reasoning: I will define a function to calculate the sum of even numbers using the built-in `sum()` and `range()` functions, which directly addresses the subtask requirement for an optimized method.

```python
def sum_even_builtin(limit):
    """Calculates the sum of even numbers up to 'limit' using sum(range())."""
    # range(start, stop, step) - stop is exclusive, so limit + 1 is used to include the limit if it's even
    return sum(range(0, limit + 1, 2))

# Demonstrate the function with an example
limit_value = 100
result_builtin = sum_even_builtin(limit_value)
print(f"The sum of even numbers up to {limit_value} using built-in functions is: {result_builtin}")

limit_value = 10
result_builtin = sum_even_builtin(limit_value)
print(f"The sum of even numbers up to {limit_value} using built-in functions is: {result_builtin}")
```

```
The sum of even numbers up to 100 using built-in functions is: 2550
The sum of even numbers up to 10 using built-in functions is: 30
```

Implement Mathematical Formula

Subtask:

Calculate the sum of even numbers using the mathematical formula for an arithmetic progression (n * (n + 1), where n is upper limit).

Reasoning: I will define a Python function to calculate the sum of even numbers up to a specified limit using the mathe formula for an arithmetic progression, as outlined in the instructions.

```python
def sum_even_math(limit):
    """Calculates the sum of even numbers up to 'limit' using a mathematical formula."""
    # The sum of the first k even numbers (2 + 4 + ... + 2k) is k * (k + 1).
    # Here, k is the count of even numbers up to 'limit'.
    # If limit is 10, even numbers are 2, 4, 6, 8, 10. k = 5.
    # We can get k by limit // 2.
    n = limit // 2
    return n * (n + 1)

# Demonstrate the function with an example
limit_value = 100
result_math = sum_even_math(limit_value)
print(f"The sum of even numbers up to {limit_value} using the mathematical formula is: {result_math}")

limit_value = 10
result_math = sum_even_math(limit_value)
print(f"The sum of even numbers up to {limit_value} using the mathematical formula is: {result_math}")
```

```
The sum of even numbers up to 100 using the mathematical formula is: 2550
The sum of even numbers up to 10 using the mathematical formula is: 30
```

```
limit_value = 10_000_000

print(f"Comparing performance for summing even numbers up to {limit_value}:\n")

# --- Naive Loop Method ---
start_time_naive = time.time()
result_naive = sum_even_naive(limit_value)
end_time_naive = time.time()
elapsed_time_naive = end_time_naive - start_time_naive
print(f"Naive Loop Method:\n  Sum: {result_naive}\n  Time taken: {elapsed_time_naive:.6f} seconds\n")

# --- Optimized Built-in Method ---
start_time_builtin = time.time()
result_builtin = sum_even_builtin(limit_value)
end_time_builtin = time.time()
elapsed_time_builtin = end_time_builtin - start_time_builtin
print(f"Built-in Method (sum(range())):\n  Sum: {result_builtin}\n  Time taken: {elapsed_time_builtin:.6f} seconds

# --- Mathematical Formula Method ---
start_time_math = time.time()
result_math = sum_even_math(limit_value)
end_time_math = time.time()
elapsed_time_math = end_time_math - start_time_math
print(f"Mathematical Formula Method:\n  Sum: {result_math}\n  Time taken: {elapsed_time_math:.6f} seconds\n")
```

Output:

```
Comparing performance for summing even numbers up to 10000000:

Naive Loop Method:
  Sum: 25000005000000
  Time taken: 0.739287 seconds

Built-in Method (sum(range())):
  Sum: 25000005000000
  Time taken: 0.108327 seconds

Mathematical Formula Method:
  Sum: 25000005000000
  Time taken: 0.000076 seconds
```

Improvement

Removes conditional check

Cleaner and more Pythonic

Still O(n/2) ≈ O(n) but faster in practice

Task 7 (Removing Hidden Side Effects)

• Task: Refactor code that modifies shared mutable state:

* Sample Legacy Code:

Legacy Code:

```
data = []
def add_item(x):
data.append(x)
add_item(10)
add_item(20)
print(data)
```

prompt:

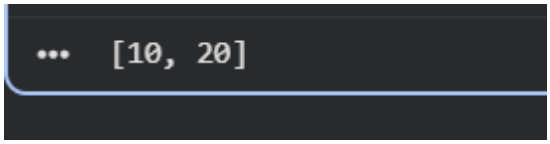
Refactor the following code to eliminate hidden side effects caused by modifying shared mutable global state. Use a functional programming approach to improve predictability and testability.

Refactored Code:

```
def add_item(data, x):
    return data + [x]

data = []
data = add_item(data, 10)
data = add_item(data, 20)

print(data)
```

```
[10, 20]
```

Output:



Improvement

No global variable modification

No hidden side effects

Function is pure (same input → same output)

Easier to test

More predictable behaviour

Task 8 (Refactoring Complex Input Validation Logic)

• Task: Use AI to simplify and modularize complex validation rules.

* Sample Legacy Code:

```
password = input("Enter password: ")
if len(password) >= 8:
    if any(c.isdigit() for c in password):
        if any(c.isupper() for c in password):
print("Valid Password")
        else:
print("Must contain uppercase")
    else:
print("Must contain digit")
```

else:

print("Password too short")

prompt:

Modularize complex validation rules into separate functions.

Refactored Code:

```
def is_long_enough(password):
    return len(password) >= 8

def has_digit(password):
    return any(c.isdigit() for c in password)

def has_uppercase(password):
    return any(c.isupper() for c in password)

def validate_password(password):
    if not is_long_enough(password):
        return "Password too short"
    if not has_digit(password):
        return "Must contain digit"
    if not has_uppercase(password):
        return "Must contain uppercase"
    return "Valid Password"

password = input("Enter password: ")
print(validate_password(password))
```

Output:

```
Enter password: sdfg234jkl
Must contain uppercase
```

Improvement:

Modular design

Easy testing

Clear responsibility per function