Assignment : 13.5

Name : D .Akash

Ht.No : 2303A51422

Batch : 21

Task Description #1

(Refactoring – Removing Global Variables)

• Task: Use AI to eliminate unnecessary global variables from the code.

• Instructions:

o Identify global variables used across functions.

o Refactor the code to pass values using function parameters.

o Improve modularity and testability.

• Sample Legacy Code:

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

• Expected Output:

o Refactored version passing rate as a parameter or using a configuration structure.

Code :

```
A_13.5.py > ...
'''Task Description #1
(Refactoring – Removing Global Variables)
• Task: Use AI to eliminate unnecessary global variables from the
code.
• Instructions:
o Identify global variables used across functions.
o Refactor the code to pass values using function parameters.
o Improve modularity and testability.
• Sample Legacy Code:
rate = 0.1
def calculate_interest(amount):
return amount * rate
print(calculate_interest(1000))
• Expected Output:
o Refactored version passing rate as a parameter or using a
configuration structure.'''
# Refactored Code without Global Variables
def calculate_interest(amount, rate):
    return amount * rate
# Example usage
interest_rate = 0.1
print(calculate_interest(1000, interest_rate))
```

Output :

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL HISTORY  TASK MONITOR  QUERY RESULTS
[Running] python -u "c:\Users\akash\OneDrive\Desktop\python training\A_13.5.py"
100.0

[Done] exited with code=0 in 0.599 seconds
```

Observation : removing the global variable improves modularity and testability. By passing the interest rate as a parameter instead of relying on a global variable, the function becomes independent and

reusable. This reduces hidden dependencies and makes unit testing easier because different rate values can be injected without modifying global state. Overall, the design becomes more flexible and maintainable.

Task Description #2 :

(Refactoring Deeply Nested Conditionals)

• Task: Use AI to refactor deeply nested if–elif–else logic into a

cleaner structure.

• Focus Areas:

o Readability

o Logical simplification

o Maintainability

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")

Expected Outcome:

o Flattened logic using guard clauses or a mapping-based

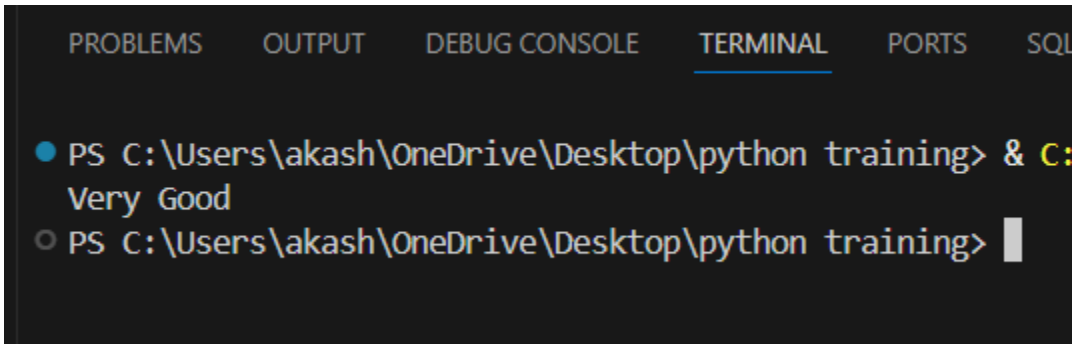
approach.

Code :

A_13.5.py > ...

```
'''Task Description #2
(Refactoring Deeply Nested Conditionals)
• Task: Use AI to refactor deeply nested if-elif-else logic into a
cleaner structure.
• Focus Areas:
o Readability
o Logical simplification
o Maintainability
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")
Expected Outcome:
o Flattened logic using guard clauses or a mapping-based
approach.'''
score = 78
if score >= 90:
    print("Excellent")
elif score >= 75:
    print("Very Good")
elif score >= 60:
    print("Good")
else:
    print("Needs Improvement")

```

Output :



Observation : Flattening nested conditionals improves readability and logical clarity. The reduced indentation makes the code cleaner and easier to modify.It enhances maintainability and reduces complexity.

Task Description 3 :

(Refactoring Repeated File Handling Code)

• Task: Use AI to refactor repeated file open/read/close logic.

• Focus Areas:

o DRY principle

o Context managers

o Function reuse

Legacy Code:

```
f = open("data1.txt")
print(f.read())
f.close()
f = open("data2.txt")
print(f.read())
f.close()
```

Expected Outcome:

o Reusable function using with open() and parameters.

Code :

```python
'''Task 3
(Refactoring Repeated File Handling Code)
• Task: Use AI to refactor repeated file open/read/close logic.
• Focus Areas:
o DRY principle
o Context managers
o Function reuse
Legacy Code:
f = open("data1.txt")
print(f.read())
f.close()
f = open("data2.txt")
print(f.read())
f.close()
Expected Outcome:
o Reusable function using with open() and parameters.'''
# read the task carefully and understand the requirements
# outcome should be a reusable function that uses context managers to handle file operations
def read_file(file_name):
    """Reads the content of a file and prints it."""
    with open(file_name, 'r') as f:
        print(f.read())
# Using the reusable function to read multiple files
read_file("data1.txt")
read_file("data2.txt")
```

Output :

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\akash\OneDrive\Desktop\python training> &
read_file("data1.txt")

read_file("data2.txt")
PS C:\Users\akash\OneDrive\Desktop\python training>
```

Observation : Using a reusable function with a context manager follows the DRY principle.It ensures automatic file closing and safer resource handling.The code becomes cleaner, reusable, and more reliable.

Task 4 :

(Optimizing Search Logic)

• Task: Refactor inefficient linear searches using appropriate data structures.

• Focus Areas:

o Time complexity

o Data structure choice

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")
```

Expected Outcome:

o Use of sets or dictionaries with complexity justification.

Code :

A_13.5.py > ...

```
'''Task 4
(Optimizing Search Logic)
• Task: Refactor inefficient linear searches using appropriate
data structures.
• Focus Areas:
o Time complexity
o Data structure choice
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")
Expected Outcome:
o Use of sets or dictionaries with complexity justification.'''
# Refactored code using a set for efficient search
users = {"admin", "guest", "editor", "viewer"}  # Using a set
name = input("Enter username: ")
if name in users:  # O(1) average time complexity for set lookup
    print("Access Granted")
else:
    print("Access Denied")
```

Output :

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\akash\OneDrive\Desktop\python training> &
Enter username: admin
Access Granted
PS C:\Users\akash\OneDrive\Desktop\python training>
```

Observation : Replacing a list with a set improves search time from O(n) to O(1).The code becomes shorter and more efficient.
This improves scalability for larger datasets.

Task 5 :

(Refactoring Procedural Code into OOP Design)

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

• Focus Areas:

o Object-Oriented principles

o Encapsulation

Legacy Code:

```
salary = 50000
tax = salary * 0.2
net = salary - tax
print(net)
```

Expected Outcome:

o A class like EmployeeSalaryCalculator with methods and attributes.

Code :

A_13.5.py > ...

```
'''Task 5
(Refactoring Procedural Code into OOP Design)
• Task: Use AI to refactor procedural code into a class-based
design.
• Focus Areas:
o Object-Oriented principles
o Encapsulation
Legacy Code:
salary = 50000
tax = salary * 0.2
net = salary - tax
print(net)
Expected Outcome:
o A class like EmployeeSalaryCalculator with methods and
attributes to calculate net salary.
o Improved code organization and maintainability.'''
class EmployeeSalaryCalculator:
    def __init__(self, salary):
        self.salary = salary

    def calculate_tax(self):
        return self.salary * 0.2

    def calculate_net_salary(self):
        tax = self.calculate_tax()
        net_salary = self.salary - tax
        return net_salary
# Example usage
employee = EmployeeSalaryCalculator(50000)
net_salary = employee.calculate_net_salary()
print(net_salary)

```

Output :

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  S

PS C:\Users\akash\OneDrive\Desktop\python training> &
40000.0
PS C:\Users\akash\OneDrive\Desktop\python training>
```

Observation : Converting procedural code to a class introduces encapsulation. It organizes related logic into a structured design. This makes the system easier to extend and maintain.

Task 6 :

(Refactoring for Performance Optimization)

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

• Focus Areas:

o Algorithmic optimization

o Use of built-in functions

Legacy Code:

```
total = 0
for i in range(1, 1000000):
if i % 2 == 0:
total += i
print(total)
```

Expected Outcome:

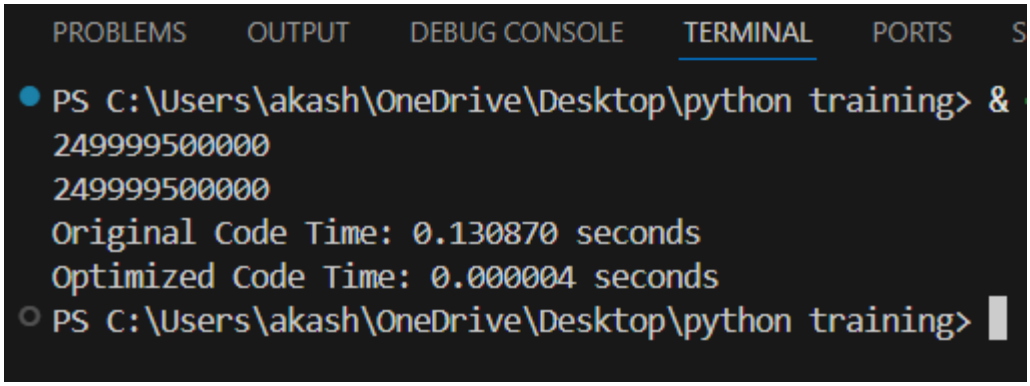
o Optimized logic using mathematical formulas or comprehensions, with time comparison.

Code :

A_13.5.py > ...

```
'''Task 6 (Refactoring for Performance Optimization)
• Task: Use AI to refactor a performance-heavy loop handling
large data.
• Focus Areas:
o Algorithmic optimization
o Use of built-in functions
Legacy Code:
total = 0
for i in range(1, 1000000):
if i % 2 == 0:
total += i
print(total)
Expected Outcome:
o Optimized logic using mathematical formulas or
comprehensions, with time comparison.'''
# Original Code
total = 0
for i in range(1, 1000000):
    if i % 2 == 0:
        total += i
print(total)
# Optimized Code using Mathematical Formula
n = 999998  # Largest even number less than 1,000,000
total_optimized = n * (n + 2) // 4  # Sum of
# even numbers from 1 to n
print(total_optimized)
# Time Comparison
import time
# Timing Original Code
start_time = time.time()
total = 0
for i in range(1, 1000000):
    if i % 2 == 0:
        total += i
end_time = time.time()
print(f"Original Code Time: {end_time - start_time:.6f} seconds")
# Timing Optimized Code
start_time = time.time()
n = 999998
total_optimized = n * (n + 2) // 4
end_time = time.time()
print(f"Optimized Code Time: {end_time - start_time:.6f} seconds")
```

Output :



Observation :

Using a mathematical formula removes unnecessary looping.
Time complexity improves significantly from O(n) to O(1).
This greatly enhances performance for large ranges.

Task 7:

(Removing Hidden Side Effects)

• Task: Refactor code that modifies shared mutable state.

• Focus Areas:

o Functional-style refactoring

o Predictability

Legacy Code:

```
data = []

def add_item(x):

data.append(x)

add_item(10)

add_item(20)

print(data)
```

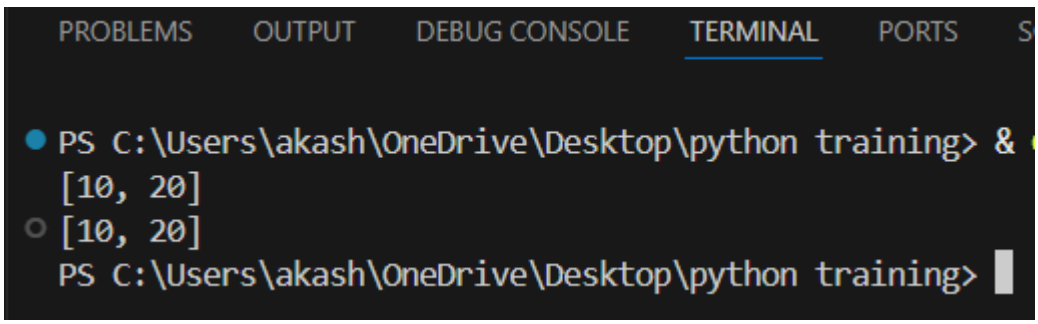
Expected Outcome:

o Refactored function returning values instead of mutating globals.

Code :

```
A_13.5.py > ...
'''Task 7 (Removing Hidden Side Effects)
• Task: Refactor code that modifies shared mutable state.
• Focus Areas:
o Functional-style refactoring
o Predictability
Legacy Code:
data = []
def add_item(x):
data.append(x)
add_item(10)
add_item(20)
print(data)
Expected Outcome:
o Refactored function returning values instead of mutating
globals.
o Improved code predictability and testability.
Refactored Code:'''
def add_item(x, data):
    return data + [x]
data = []
data = add_item(10, data)
data = add_item(20, data)
print(data)
```

Output :



Observation :

Avoiding mutation of global state removes hidden side effects.
Functions become predictable and easier to test.
This supports a cleaner, functional programming style.

Task 8 :

(Refactoring Complex Input Validation Logic)

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

• Focus Areas:

o Readability

o Testability

Legacy Code:

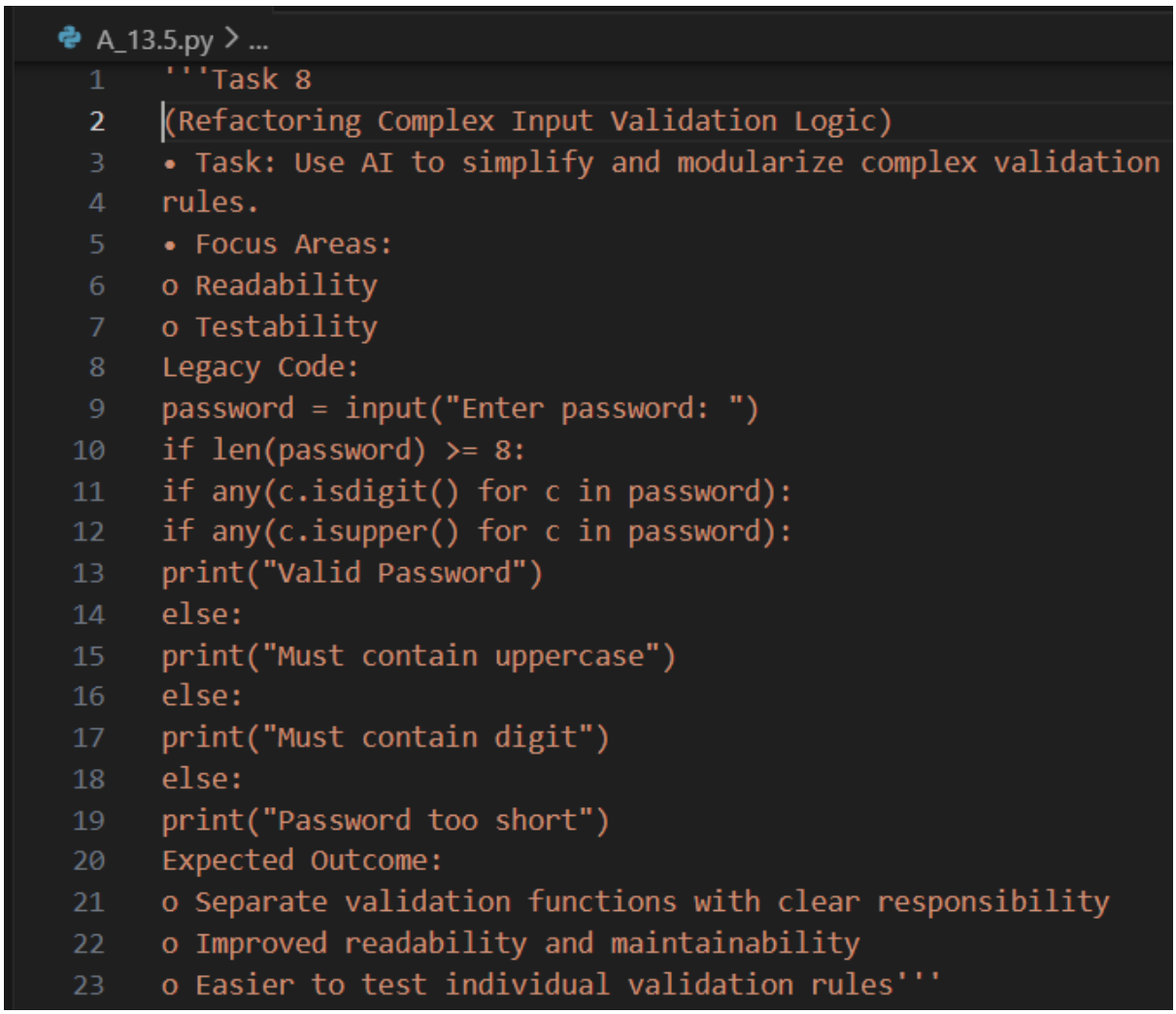
```
password = input("Enter password: ")
if len(password) >= 8:
if any(c.isdigit() for c in password):
if any(c.isupper() for c in password):
print("Valid Password")
else:
print("Must contain uppercase")
else:
print("Must contain digit")
else:
```

print("Password too short")

Expected Outcome:

o Separate validation functions with clear responsibility

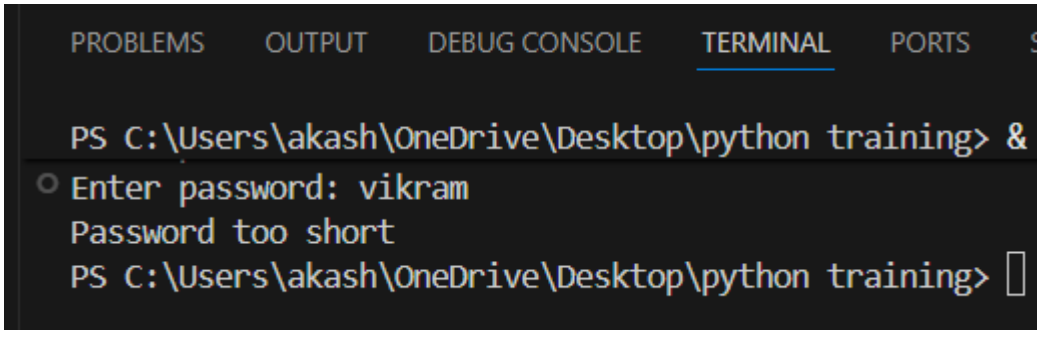
code :



```
A_13.5.py > ...
'''Task 8
(Refactoring Complex Input Validation Logic)
• Task: Use AI to simplify and modularize complex validation
rules.
• Focus Areas:
o Readability
o Testability
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")
Expected Outcome:
o Separate validation functions with clear responsibility
o Improved readability and maintainability
o Easier to test individual validation rules'''
```

```python
def is_valid_length(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 is_valid_password(password):
    if not is_valid_length(password):
        print("Password too short")
        return False
    if not has_digit(password):
        print("Must contain digit")
        return False
    if not has_uppercase(password):
        print("Must contain uppercase")
        return False
    print("Valid Password")
    return True
password = input("Enter password: ")
is_valid_password(password)

```

output :



Observation :

Separating validation rules into functions improves readability.
Each rule becomes independently testable and reusable.
The code is easier to extend and maintain.