

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:
 - Identify global variables used across functions.
 - Refactor the code to pass values using function parameters.
 - Improve modularity and testability.
- Sample Legacy Code:

rate = 0.1

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

- Expected Output:
 - Refactored version passing rate as a parameter or using a configuration structure.

Code :

```
❖ A_13.5.py > ...
1  '''Task Description #1
2  (Refactoring □ Removing Global Variables)
3  • Task: Use AI to eliminate unnecessary global variables from the
4  code.
5  • Instructions:
6  o Identify global variables used across functions.
7  o Refactor the code to pass values using function parameters.
8  o Improve modularity and testability.
9  • Sample Legacy Code:
10 rate = 0.1
11 def calculate_interest(amount):
12     return amount * rate
13 print(calculate_interest(1000))
14 • Expected Output:
15 o Refactored version passing rate as a parameter or using a
16 configuration structure.'''
17 # Refactored Code without Global Variables
18 def calculate_interest(amount, rate):
19     return amount * rate
20 # Example usage
21 interest_rate = 0.1
22 print(calculate_interest(1000, interest_rate))
23
```

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:
 - Readability
 - Logical simplification
 - 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")
```

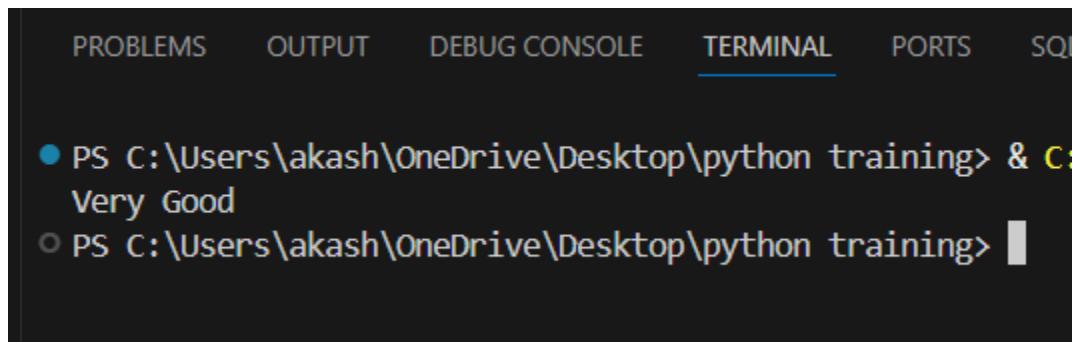
Expected Outcome:

- o Flattened logic using guard clauses or a mapping-based approach.

Code :

```
🐍 A_13.5.py > ...  
1  '''Task Description #2  
2  (Refactoring Deeply Nested Conditionals)  
3  • Task: Use AI to refactor deeply nested if[] elif[] else logic into a  
4  cleaner structure.  
5  • Focus Areas:  
6  o Readability  
7  o Logical simplification  
8  o Maintainability  
9  Legacy Code:  
10 score = 78  
11 if score >= 90:  
12 print("Excellent")  
13 else:  
14 if score >= 75:  
15 print("Very Good")  
16 else:  
17 if score >= 60:  
18 print("Good")  
19 else:  
20 print("Needs Improvement")  
21 Expected Outcome:  
22 o Flattened logic using guard clauses or a mapping-based  
23 approach.''  
24 score = 78  
25 if score >= 90:  
26 |   print("Excellent")  
27 elif score >= 75:  
28 |   print("Very Good")  
29 elif score >= 60:  
30 |   print("Good")  
31 else:  
32 |   print("Needs Improvement")  
33
```

Output :



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL  
● PS C:\Users\akash\OneDrive\Desktop\python training> & C:  
Very Good  
○ PS C:\Users\akash\OneDrive\Desktop\python training> █
```

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:
 - DRY principle
 - Context managers
 - 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 :

```
⌚ A_13.5.py > ...
1  '''Task 3
2  (Refactoring Repeated File Handling Code)
3  • Task: Use AI to refactor repeated file open/read/close logic.
4  • Focus Areas:
5  o DRY principle
6  o Context managers
7  o Function reuse
8  Legacy Code:
9  f = open("data1.txt")
10 print(f.read())
11 f.close()
12 f = open("data2.txt")
13 print(f.read())
14 f.close()
15 Expected Outcome:
16 o Reusable function using with open() and parameters.''''
17 # read the task carefully and understand the requirements
18 # outcome should be a reusable function that uses context managers to handle file operations
19 def read_file(file_name):
20     """Reads the content of a file and prints it."""
21     with open(file_name, 'r') as f:
22         print(f.read())
23 # Using the reusable function to read multiple files
24 read_file("data1.txt")
25 read_file("data2.txt")
26
```

Output :

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS S
● 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:

- Time complexity
 - 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")
```

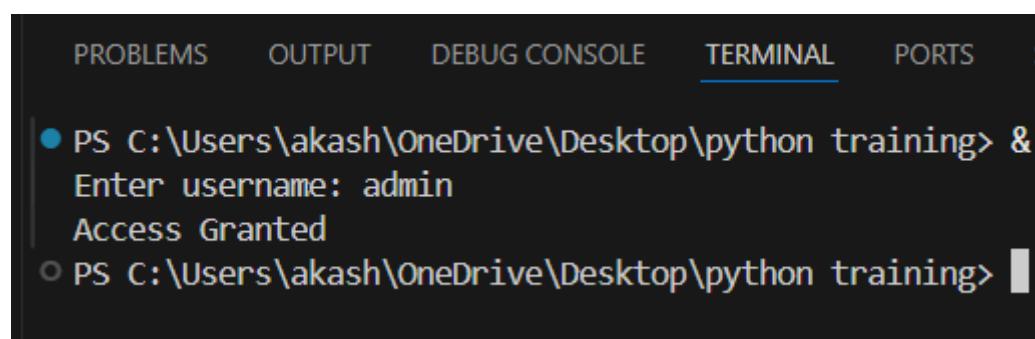
Expected Outcome:

- Use of sets or dictionaries with complexity justification.

Code :

```
❖ A_13.5.py > ...
1  '''Task 4
2  (Optimizing Search Logic)
3  • Task: Refactor inefficient linear searches using appropriate
4  data structures.
5  • Focus Areas:
6  o Time complexity
7  o Data structure choice
8  Legacy Code:
9  users = ["admin", "guest", "editor", "viewer"]
10 name = input("Enter username: ")
11 found = False
12 for u in users:
13     if u == name:
14         found = True
15     print("Access Granted" if found else "Access Denied")
16 Expected Outcome:
17 o Use of sets or dictionaries with complexity justification.''
18 # Refactored code using a set for efficient search
19 users = {"admin", "guest", "editor", "viewer"} # Using a set
20 name = input("Enter username: ")
21 if name in users: # O(1) average time complexity for set lookup
22     print("Access Granted")
23 else:
24     print("Access Denied")
25
26
```

Output :



The screenshot shows a terminal window with the following content:

- 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:

- Object-Oriented principles
 - Encapsulation

Legacy Code:

```
salary = 50000
```

```
tax = salary * 0.2
```

```
net = salary - tax
```

```
print(net)
```

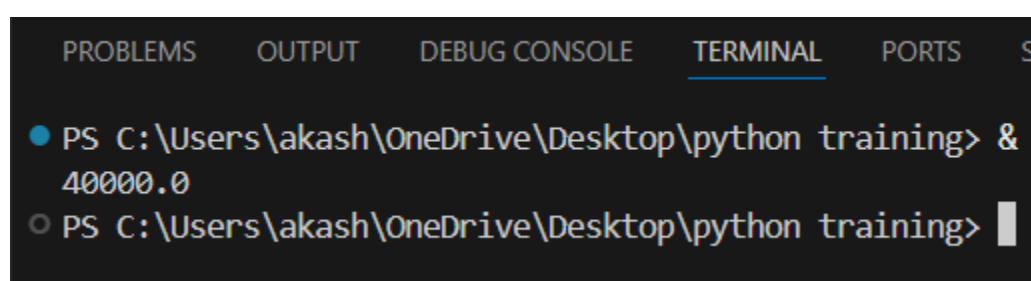
Expected Outcome:

- A class like EmployeeSalaryCalculator with methods and attributes.

Code :

```
❖ A_13.5.py > ...
1  '''Task 5
2  (Refactoring Procedural Code into OOP Design)
3  • Task: Use AI to refactor procedural code into a class-based
4  design.
5  • Focus Areas:
6  o Object-Oriented principles
7  o Encapsulation
8  Legacy Code:
9  salary = 50000
10 tax = salary * 0.2
11 net = salary - tax
12 print(net)
13 Expected Outcome:
14 o A class like EmployeeSalaryCalculator with methods and
15 attributes to calculate net salary.
16 o Improved code organization and maintainability.'''
17 class EmployeeSalaryCalculator:
18     def __init__(self, salary):
19         self.salary = salary
20
21     def calculate_tax(self):
22         return self.salary * 0.2
23
24     def calculate_net_salary(self):
25         tax = self.calculate_tax()
26         net_salary = self.salary - tax
27         return net_salary
28 # Example usage
29 employee = EmployeeSalaryCalculator(50000)
30 net_salary = employee.calculate_net_salary()
31 print(net_salary)
32
```

Output :



The screenshot shows a terminal window with the following output:

```
● 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:

- Algorithmic optimization
 - 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:

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

Code :

```
❖ A_13.5.py > ...
1  '''Task 6 (Refactoring for Performance Optimization)
2  • Task: Use AI to refactor a performance-heavy loop handling
3  large data.
4  • Focus Areas:
5  o Algorithmic optimization
6  o Use of built-in functions
7  Legacy Code:
8  total = 0
9  for i in range(1, 1000000):
10 if i % 2 == 0:
11     total += i
12 print(total)
13 Expected Outcome:
14 o Optimized logic using mathematical formulas or
15 comprehensions, with time comparison.'''
16 # Original Code
17 total = 0
18 for i in range(1, 1000000):
19     if i % 2 == 0:
20         total += i
21 print(total)
22 # Optimized Code using Mathematical Formula
23 n = 999998 # Largest even number less than 1,000,000
24 total_optimized = n * (n + 2) // 4 # Sum of
25 # even numbers from 1 to n
26 print(total_optimized)
27 # Time Comparison
28 import time
29 # Timing Original Code
30 start_time = time.time()
31 total = 0
32 for i in range(1, 1000000):
33     if i % 2 == 0:
34         total += i
35 end_time = time.time()
36 print(f"Original Code Time: {end_time - start_time:.6f} seconds")
37 # Timing Optimized Code
38 start_time = time.time()
39 n = 999998
40 total_optimized = n * (n + 2) // 4
41 end_time = time.time()
42 print(f"Optimized Code Time: {end_time - start_time:.6f} seconds")
43 |
```

Output :

The screenshot shows a terminal window with the following content:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS S
● PS C:\Users\akash\OneDrive\Desktop\python training> &
249999500000
249999500000
Original Code Time: 0.130870 seconds
Optimized Code Time: 0.000004 seconds
○ PS C:\Users\akash\OneDrive\Desktop\python training>
```

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:
 - Functional-style refactoring
 - Predictability

Legacy Code:

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

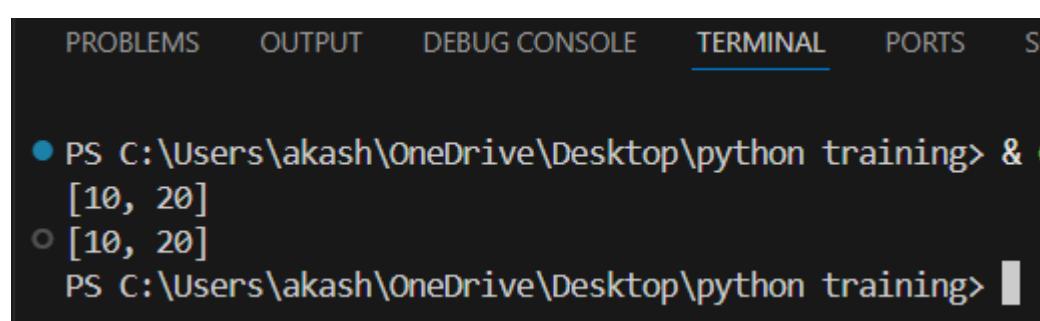
Expected Outcome:

- o Refactored function returning values instead of mutating globals.

Code :

```
❖ A_13.5.py > ...
1  '''Task 7 (Removing Hidden Side Effects)
2  • Task: Refactor code that modifies shared mutable state.
3  • Focus Areas:
4  o Functional-style refactoring
5  o Predictability
6  Legacy Code:
7  data = []
8  def add_item(x):
9    data.append(x)
10 add_item(10)
11 add_item(20)
12 print(data)
13 Expected outcome:
14 o Refactored function returning values instead of mutating
15 globals.
16 o Improved code predictability and testability.
17 Refactored Code:''
18 def add_item(x, data):
19   return data + [x]
20 data = []
21 data = add_item(10, data)
22 data = add_item(20, data)
23 print(data)
24
```

Output :



The screenshot shows a terminal window with the following output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS S
● PS C:\Users\akash\OneDrive\Desktop\python training> &
[10, 20]
○ [10, 20]
PS C:\Users\akash\OneDrive\Desktop\python training>
```

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:
 - Readability
 - 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")
```

```
print("Password too short")
```

Expected Outcome:

- o Separate validation functions with clear responsibility

code :

```
❖ A_13.5.py > ...
1  '''Task 8
2  |(Refactoring Complex Input Validation Logic)
3  • Task: Use AI to simplify and modularize complex validation
4  rules.
5  • Focus Areas:
6  o Readability
7  o Testability
8  Legacy Code:
9  password = input("Enter password: ")
10 if len(password) >= 8:
11     if any(c.isdigit() for c in password):
12         if any(c.isupper() for c in password):
13             print("Valid Password")
14         else:
15             print("Must contain uppercase")
16         else:
17             print("Must contain digit")
18         else:
19             print("Password too short")
20 Expected Outcome:
21 o Separate validation functions with clear responsibility
22 o Improved readability and maintainability
23 o Easier to test individual validation rules'''
```

```

24     def is_valid_length(password):
25         return len(password) >= 8
26
27     def has_digit(password):
28         return any(c.isdigit() for c in password)
29
30     def has_uppercase(password):
31         return any(c.isupper() for c in password)
32
33     def is_valid_password(password):
34         if not is_valid_length(password):
35             print("Password too short")
36             return False
37         if not has_digit(password):
38             print("Must contain digit")
39             return False
40         if not has_uppercase(password):
41             print("Must contain uppercase")
42             return False
43         print("Valid Password")
44         return True
45
46 password = input("Enter password: ")
47 is_valid_password(password)

```

output :

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    S

PS C:\Users\akash\OneDrive\Desktop\python training> &
○ Enter password: vikram
Password too short
PS C:\Users\akash\OneDrive\Desktop\python training> []

```

Observation :

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

