ASSIGNMENT-13.5

H.no-2303A51176

Batch – 29

Task 1 – Refactoring – Removing Global Variables

Prompt :

Eliminate unnecessary global variables from the

CODE:

```python
Assignment_13.5.py > ...
# rate = 0.1
# def calculate_interest(amount):
# return amount * rate
# print(calculate_interest(1000))
#eliminate unnecessary global variables from the code and ensure that the function is self-contained and does not rely on external state.
def calculate_interest(amount, rate=0.1):
    """Calculates the interest based on the given amount and rate."""
    return amount * rate  # Use the rate parameter instead of a global variable
interest = calculate_interest(1000)  # Call the function with the amount and use the default rate
print(interest)  # Print the calculated interest
```

OUTPUT:

```
PROBLEMS  DEBUG CONSOLE  OUTPUT  TERMINAL  PORTS

PS C:\Users\hp\OneDrive\Desktop\AI_Assited_Coding>  & 'C:\Users\hp\AppData\Local\Programs\Pyt
debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '55478' '--' 'c:\Users\hp\OneDrive
100.0
Very Good
PS C:\Users\hp\OneDrive\Desktop\AI_Assited_Coding>
```

JUSTIFICATION:

Global variables create tight coupling and make testing difficult. Passing values through function parameters improves modularity, reduces hidden dependencies, and makes functions reusable and predictable.

Task 2 – Refactoring Deeply Nested Conditionals

Prompt :

Refactor deeply nested if–elif–else logic into a cleaner structure.

CODE:

```python
# score = 78
# if score >= 90:
# print("Excellent")
# else:
# if score >= 75:
# print("Very Good")
# else:
# if score >= 60:
# print("Good")
# else:
# print("Needs Improvement")
#refactor deeply nested if-elif-else logic into a clean structure
score = 78
if score >= 90:
    print("Excellent")
elif score >= 75:  # Use elif to avoid deep nesting
    print("Very Good")
elif score >= 60:  # Use elif to avoid deep nesting score = 78
    print("Good")
else:
    print("Needs Improvement")  # This will be executed if none of the above conditions are met
```

OUTPUT:

```
PS C:\Users\hp\OneDrive\Desktop\AI_Assited_Coding>  c:; cd 'c:\Users\hp\On
14\python.exe' 'c:\Users\hp\.vscode\extensions\ms-python.debugpy-2025.18.0
_Assited_Coding\Assignment_13.5.py'
Very Good
PS C:\Users\hp\OneDrive\Desktop\AI_Assited_Coding>
```

JUSTIFICATION:

Deep nesting decreases readability and increases cognitive complexity. Flattening the logic using guard clauses or mapping improves clarity, simplifies debugging, and enhances maintainability.

Task 3 - Refactoring Repeated File Handling Code

Prompt :

Refactor repeated file open/read/close logic.

CODE:

```python
# f = open("data1.txt")
# print(f.read())
# f.close()
# f = open("data2.txt")
# print(f.read())
# f.close()
#refactor repeated file open/read/close logic.
def read_file(file_name):
    """Reads the content of a file and returns it."""
    with open(file_name, 'r') as f:  # Use a context manager to handle file opening and closing
        return f.read()  # Return the content of the file
print(read_file("data1.txt"))  # Read and print the content of data1.txt
print(read_file("data2.txt"))  # Read and print the content of data2.txt
```

JUSTIFICATION:

Repeated file open–read–close operations violate the DRY principle and increase error risk. Using a reusable function with context managers (with open) ensures automatic resource management, reduces duplication, and improves reliability.

Task 4 - Optimizing Search Logic

Prompt :

Refactor inefficient linear searches using appropriate data structures.

CODE:

```
48
49
50  # users = ["admin", "guest", "editor", "viewer"]
51  # name = input("Enter username: ")
52  # found = False
53  # for u in users:
54  # if u == name:
55  # found = True
56  # print("Access Granted" if found else "Access Denied")
57  #Refactor inefficient linear searches using appropriate data structurees
58  users = {"admin", "guest", "editor", "viewer"}  # Use a set for O(1) average time complexity lookups
59  name = input("Enter username: ")
60  if name in users:  # Check for membership in the set
61      print("Access Granted")
62  else:
63      print("Access Denied")
64
```

OUTPUT:

```
PS C:\Users\hp\OneDrive\Desktop\AI_Assited_Coding>  c:; cd 'c:\Users\hp\Or
14\python.exe' 'c:\Users\hp\.vscode\extensions\ms-python.debugpy-2025.18.
_Assited_Coding\Assignment_13.5.py'
Enter username: guest
Access Granted
```

JUSTIFICATION:

Linear search requires O(n) time, which becomes inefficient for large datasets. Using sets or dictionaries provides average O(1) lookup time, improving performance and demonstrating appropriate data structure usage.

Task 5 - Refactoring Procedural Code into OOP Design

Prompt :

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

CODE:

```
66
67  # salary = 50000
68  # tax = salary * 0.2
69  # net = salary - tax
70  # print("Net Salary:", net)
71  #efactor procedural code into a class-based design
72  class Employee:
73      def __init__(self, salary):
74          self.salary = salary  # Initialize the salary attribute
75
76      def calculate_net_salary(self):
77          tax = self.salary * 0.2  # Calculate tax based on the salary
78          net = self.salary - tax  # Calculate net salary by subtracting tax from salary
79          return net  # Return the net salary
80  employee = Employee(50000)  # Create an instance of the Employee class with a salary of 50000
81  print("Net Salary:", employee.calculate_net_salary())  # Print the net salary by calling the method
82
```

OUTPUT:

```
PS C:\Users\hp\OneDrive\Desktop\AI_Assited_Coding>  c:; cd 'c:\Users\hp\OneDrive\D
14\python.exe' 'c:\Users\hp\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x
_Assited_Coding\Assignment_13.5.py'
Net Salary: 40000.0
```

JUSTIFICATION:

Procedural salary computation lacks structure and scalability. Implementing a class encapsulates related data and behavior, promotes reusability, and aligns with object-oriented design principles.

Task 6 - Refactoring for Performance Optimization)

Prompt :

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

CODE:

```
83
84  # total = 0
85  # for i in range(1, 1000000):
86  # if i % 2 == 0:
87  # total += i
88  # print(total)
89  #refactor a performance-heavy loop handling large data
90  total = sum(i for i in range(1, 1000000) if i % 2 == 0)  # Use a generator expression with sum() for better performance
91  print(total)  # Print the total of even numbers from 1 to 999999
92
```

OUTPUT:

```
PS C:\Users\hp\OneDrive\Desktop\AI_Assited_Coding>  c:; cd 'c:\Users\hp\O
14\python.exe' 'c:\Users\hp\.vscode\extensions\ms-python.debugpy-2025.18.
_Assited_Coding\Assignment_13.5.py'
249999500000
```

JUSTIFICATION:

Iterative loops over large ranges can be computationally expensive. Using mathematical formulas or Python built-in constructs reduces time complexity, improves execution speed, and optimizes performance.

Task 7 - Removing Hidden Side Effects

Prompt :

Refactor code that modifies shared mutable state.

CODE:

```
93
94  # data = []
95  # def add_item(x):
96  # data.append(x)
97  # add_item(10)
98  # add_item(20)
99  # print(data)
100 #Refactor code that modifies shared mutable state.
101 class DataStore:
102     def __init__(self):
103         self.data = []  # Initialize an empty list to store data
104
105     def add_item(self, x):
106         self.data.append(x)  # Add an item to the data list
107 data_store = DataStore()  # Create an instance of the DataStore class
108 data_store.add_item(10)  # Add an item to the data store
109 data_store.add_item(20)  # Add another item to the data store
110 print(data_store.data)  # Print the data stored in the data store
111
```

OUTPUT:

```
PS C:\Users\hp\OneDrive\Desktop\AI_Assited_Coding>  c:; cd 'c:\Users\hp\O
14\python.exe' 'c:\Users\hp\.vscode\extensions\ms-python.debugpy-2025.18.
_Assited_Coding\Assignment_13.5.py'
[10, 20]
```

JUSTIFICATION:

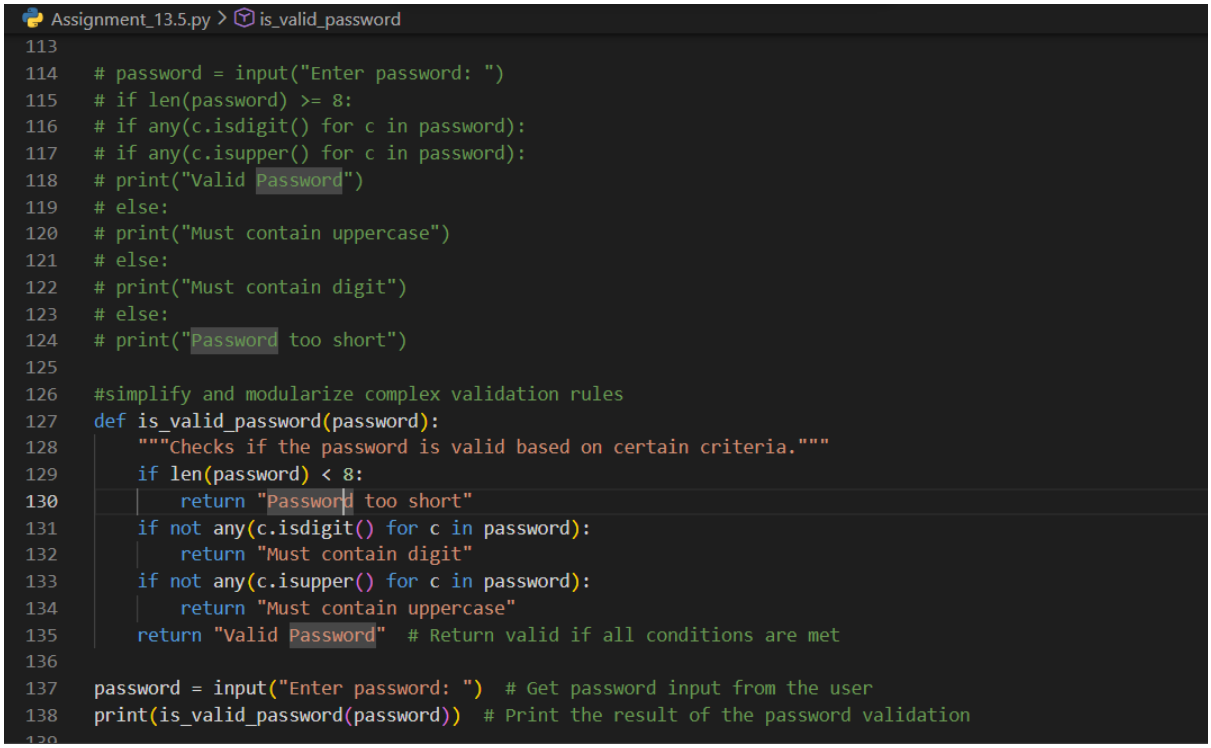
Functions that modify shared mutable state introduce side effects and unpredictable behavior. Returning values instead of altering globals improves functional purity, testability, and program reliability

Task 8 - Refactoring Complex Input Validation Logic

Prompt :

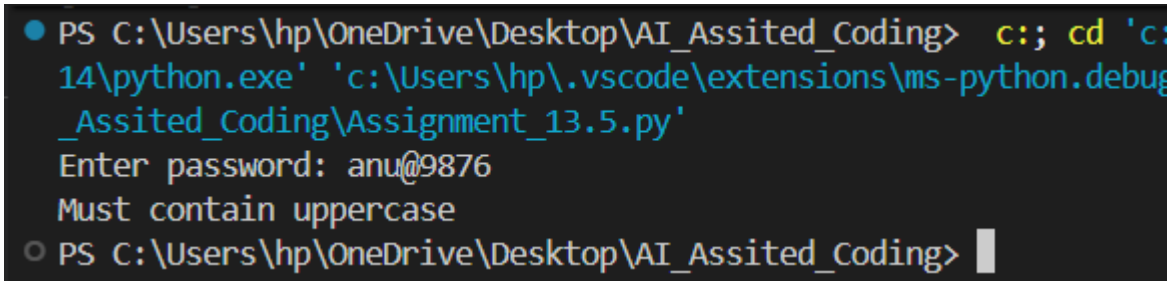
Use AI to simplify and modularize complex validation rules.

CODE:



```
Assignment_13.5.py > is_valid_password
113
114  # password = input("Enter password: ")
115  # if len(password) >= 8:
116  # if any(c.isdigit() for c in password):
117  # if any(c.isupper() for c in password):
118  # print("Valid Password")
119  # else:
120  # print("Must contain uppercase")
121  # else:
122  # print("Must contain digit")
123  # else:
124  # print("Password too short")
125
126  #simplify and modularize complex validation rules
127  def is_valid_password(password):
128      """Checks if the password is valid based on certain criteria."""
129      if len(password) < 8:
130          return "Password too short"
131      if not any(c.isdigit() for c in password):
132          return "Must contain digit"
133      if not any(c.isupper() for c in password):
134          return "Must contain uppercase"
135      return "Valid Password"  # Return valid if all conditions are met
136
137  password = input("Enter password: ")  # Get password input from the user
138  print(is_valid_password(password))  # Print the result of the password validation
```

OUTPUT:



```
PS C:\Users\hp\OneDrive\Desktop\AI_Assited_Coding>  c:; cd 'c:
14\python.exe' 'c:\Users\hp\.vscode\extensions\ms-python.debug
_Assited_Coding\Assignment_13.5.py'
Enter password: anu@9876
Must contain uppercase
PS C:\Users\hp\OneDrive\Desktop\AI_Assited_Coding>
```

JUSTIFICATION:

Nested validation conditions reduce readability and hinder testing. Separating validation rules into individual functions improves clarity, enables independent testing, and supports scalable validation logic.