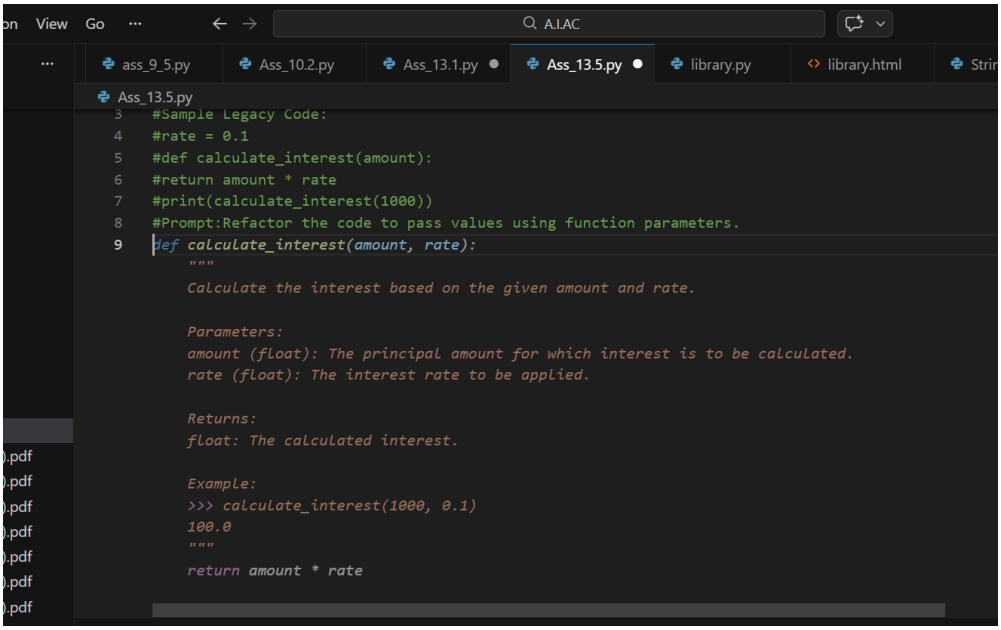
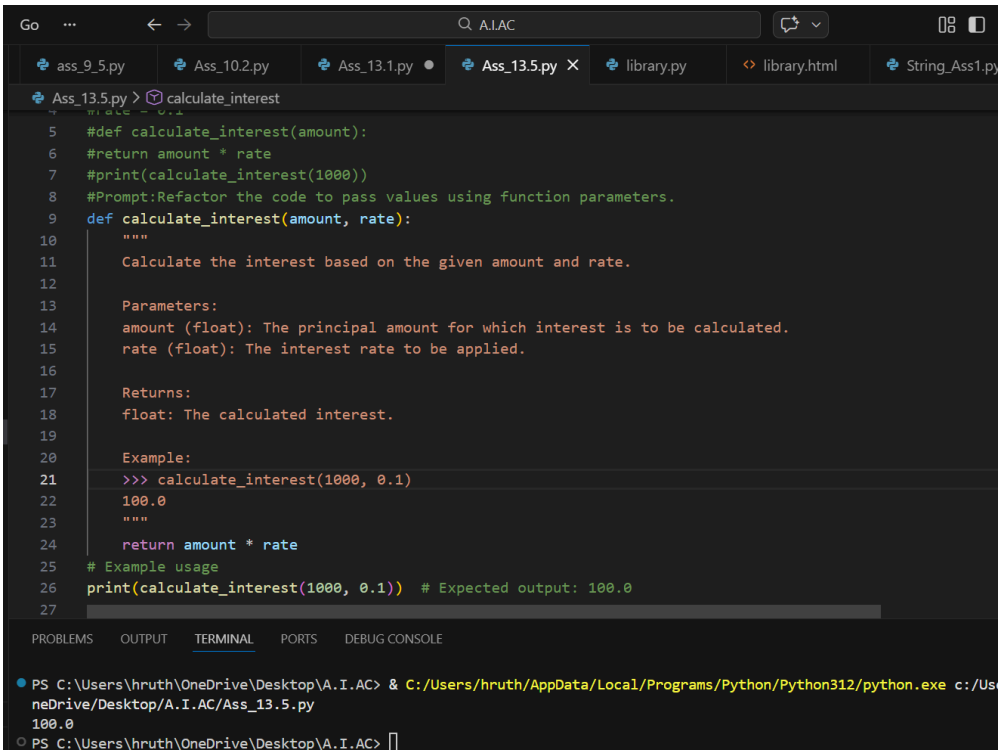
AI Assisted Coding Assignment-13.5

> 2303A51543 BT: 29 VELDI.HRUTHIKA 27.02.2026

💡 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

```
#2303A51543
#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 code to pass values using function parameters.
def calculate_interest(amount, rate):
    """
    Calculate the interest based on the given amount and rate.
    Parameters:
    amount (float): The principal amount for which interest is to be calculated.
    rate (float): The interest rate to be applied.
    Returns:
    float: The calculated interest.
    Example:
    >>> calculate_interest(1000, 0.1)
    100.0
    """
    return amount * rate
# Example usage
print(calculate_interest(1000, 0.1))  # Expected output: 100.0
```


View
Go
A.I.AC
ass_9_5.py
Ass_10.2.py
Ass_13.1.py
Ass_13.5.py
library.py
library.html
Ass_13.5.py
#Sample Legacy Code:
#rate = 0.1
#def calculate_interest(amount):
#return amount * rate
#print(calculate_interest(1000))
#Prompt:Refactor the code to pass values using function parameters.
def calculate_interest(amount, rate):
"""
Calculate the interest based on the given amount and rate.
Parameters:
amount (float): The principal amount for which interest is to be calculated.
rate (float): The interest rate to be applied.
Returns:
float: The calculated interest.
Example:
>>> calculate_interest(1000, 0.1)
100.0
"""
return amount * rate

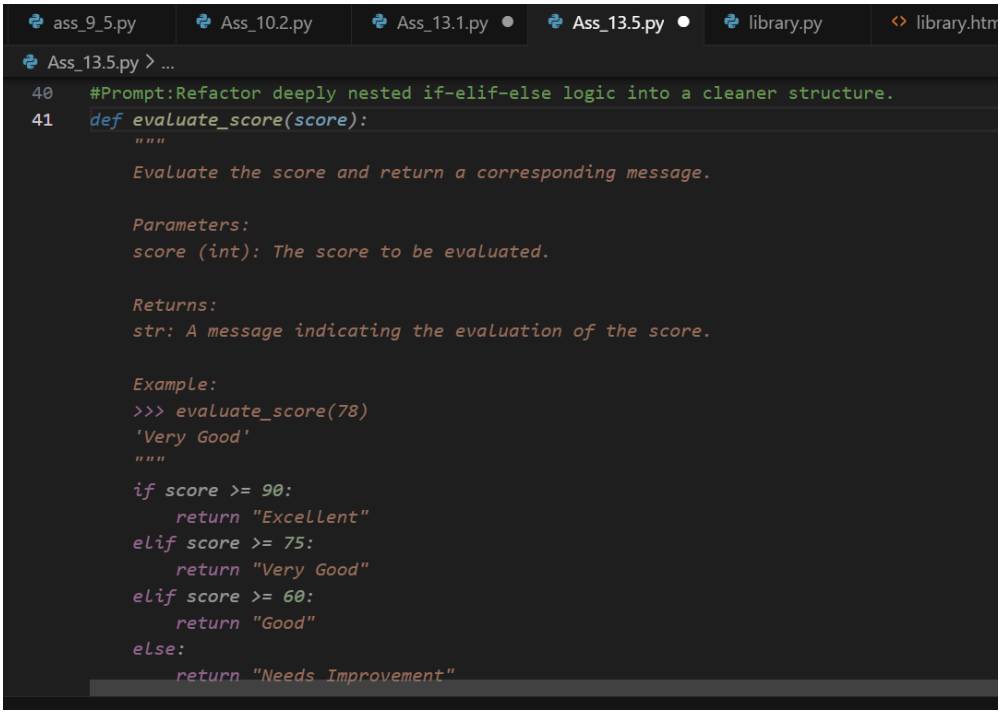

Go
A.I.AC
ass_9_5.py
Ass_10.2.py
Ass_13.1.py
Ass_13.5.py
library.py
library.html
String_Ass1.py
Ass_13.5.py > calculate_interest
#def calculate_interest(amount):
#return amount * rate
#print(calculate_interest(1000))
#Prompt:Refactor the code to pass values using function parameters.
def calculate_interest(amount, rate):
"""
Calculate the interest based on the given amount and rate.
Parameters:
amount (float): The principal amount for which interest is to be calculated.
rate (float): The interest rate to be applied.
Returns:
float: The calculated interest.
Example:
>>> calculate_interest(1000, 0.1)
100.0
"""
return amount * rate
Example usage
print(calculate_interest(1000, 0.1)) # Expected output: 100.0
PROBLEMS
OUTPUT
TERMINAL
PORTS
DEBUG CONSOLE
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC> & C:/Users/hruth/AppData/Local/Programs/Python/Python312/python.exe c:/Users/hruth/OneDrive/Desktop/A.I.AC/Ass_13.5.py
100.0
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC>

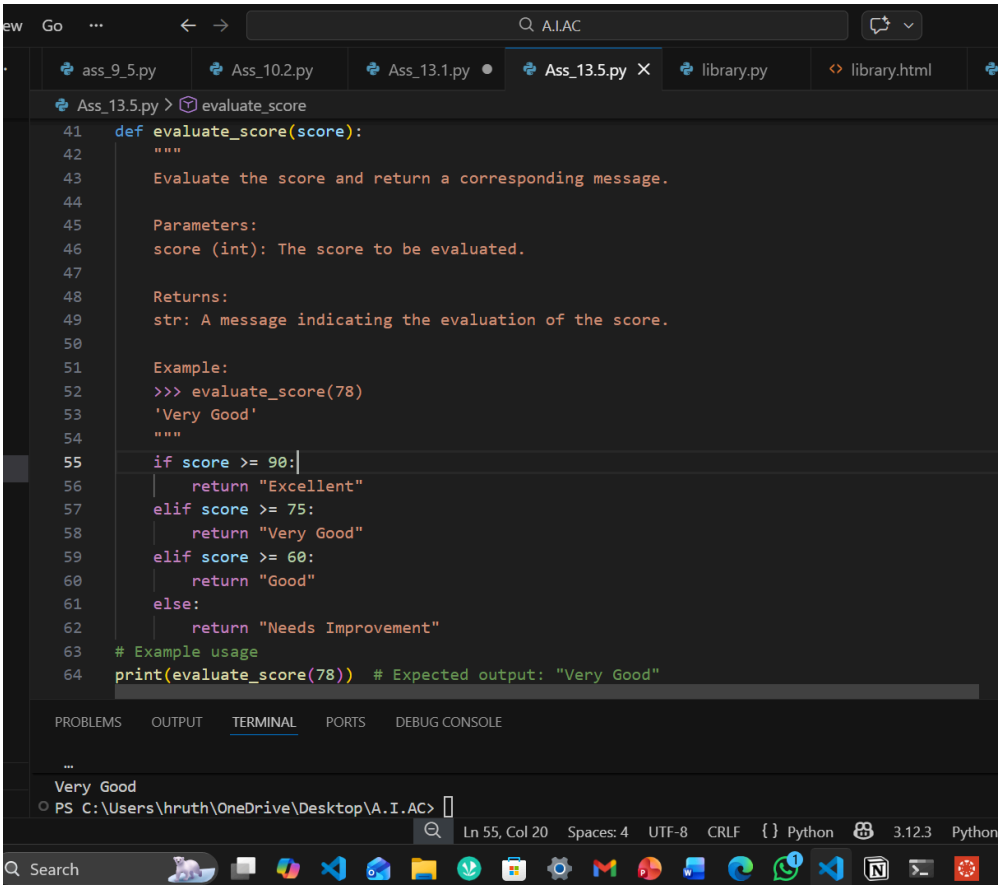
Task Description #2 : (Refactoring Deeply Nested Conditionals)
• Task: Use AI to refactor deeply nested if–elif–else logic into a cleaner structure.
• Focus Areas:
o Readability
o Logical simplification
o Maintainability
Legacy Code:
score = 78
if score >= 90:
print("Excellent")
else:
if score >= 75:
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.

```
#Task Description #2 : (Refactoring Deeply Nested Conditionals)
#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 deeply nested if–elif–else logic into a cleaner structure.
def evaluate_score(score):
    """
    Evaluate the score and return a corresponding message.
    Parameters:
    score (int): The score to be evaluated.
    Returns:
    str: A message indicating the evaluation of the score.

    Example:
    >>> evaluate_score(78)
    'Very Good'
    """
    if score >= 90:
```

```
        return "Excellent"
    elif score >= 75:
        return "Very Good"
    elif score >= 60:
        return "Good"
    else:
        return "Needs Improvement"
# Example usage
print(evaluate_score(78))  # Expected output: "Very Good"
```

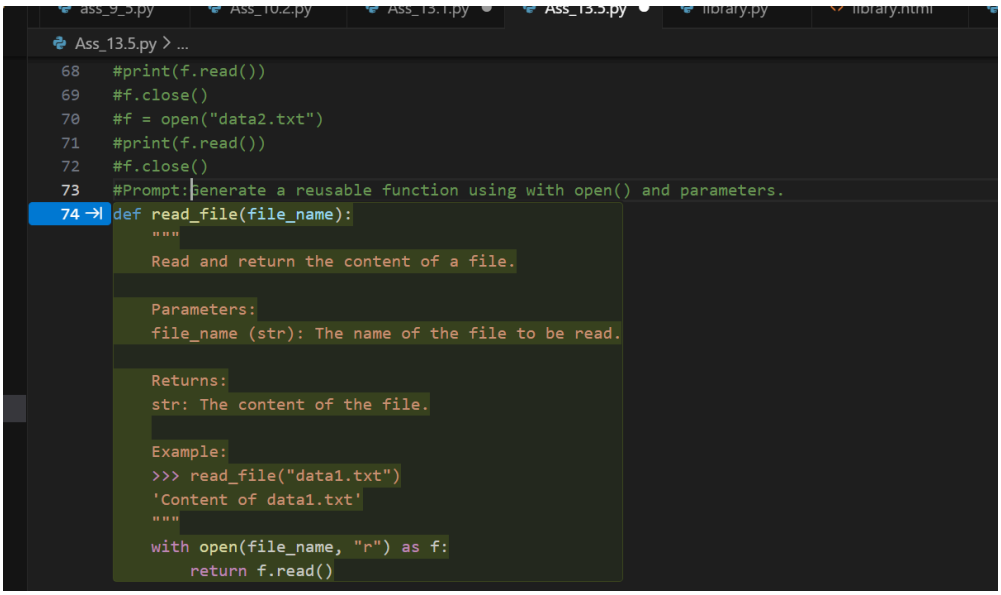


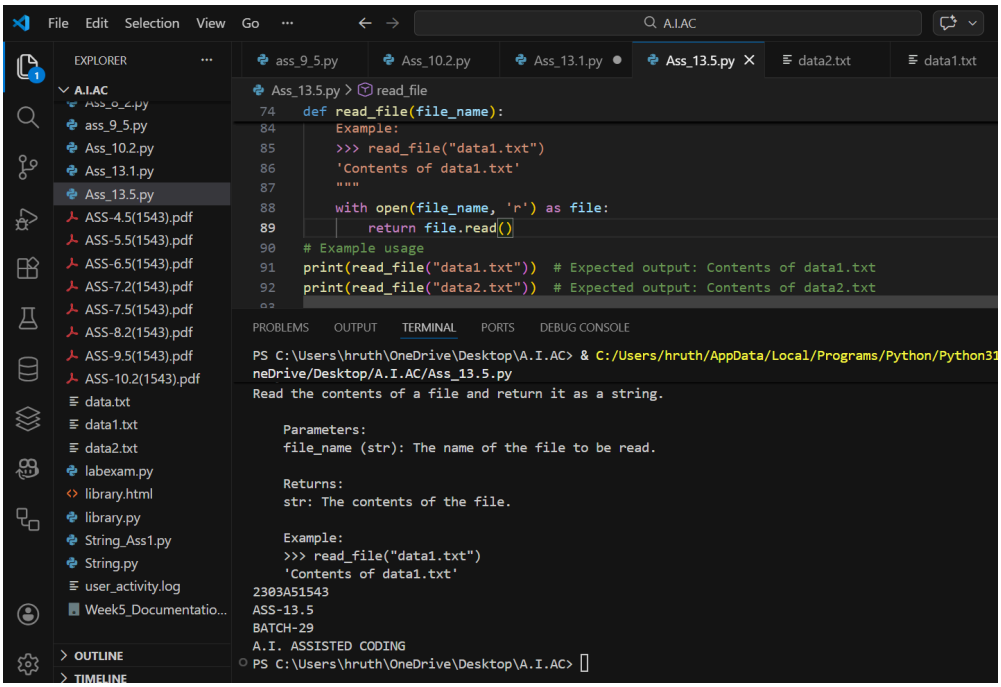


Task 3 (Refactoring Repeated File Handling Code)
• Task: Use AI to refactor repeated file open/read/close logic.
• Focus Areas:
o DRY principle
o Context managers
o Function reuse
Legacy Code:
f = open("data1.txt")
print(f.read())
f.close()
f = open("data2.txt")
print(f.read())
f.close()
Expected Outcome:
o Reusable function using with open() and parameters.

```
#Task 3 (Refactoring Repeated File Handling Code)
#f = open("data1.txt")
#print(f.read())
#f.close()
#f = open("data2.txt")
#print(f.read())
```

```
#f.close()
#Prompt:Generate a reusable function using with open() and parameters.
def read_file(file_name):
    """
    Read the contents of a file and return it as a string.
    Parameters:
    file_name (str): The name of the file to be read.
    Returns:
    str: The contents of the file.
    Example:
    >>> read_file("data1.txt")
    'Contents of data1.txt'
    """
    with open(file_name, 'r') as file:
        return file.read()
# Example usage
print(read_file("data1.txt"))  # Expected output: Contents of data1.txt
print(read_file("data2.txt"))  # Expected output: Contents of data2.txt
```

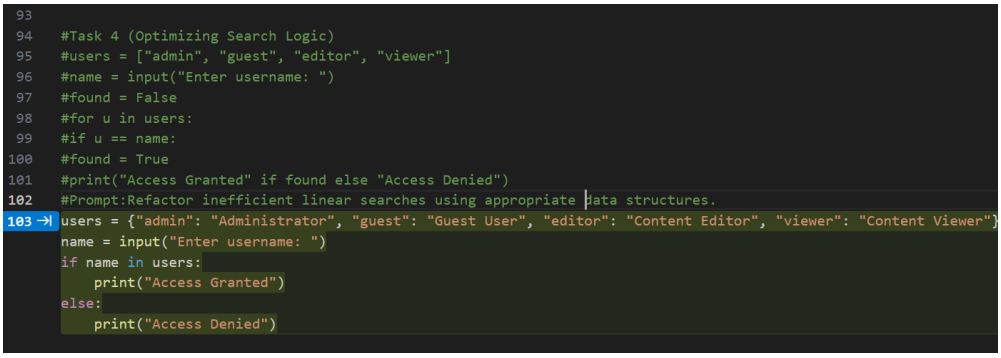


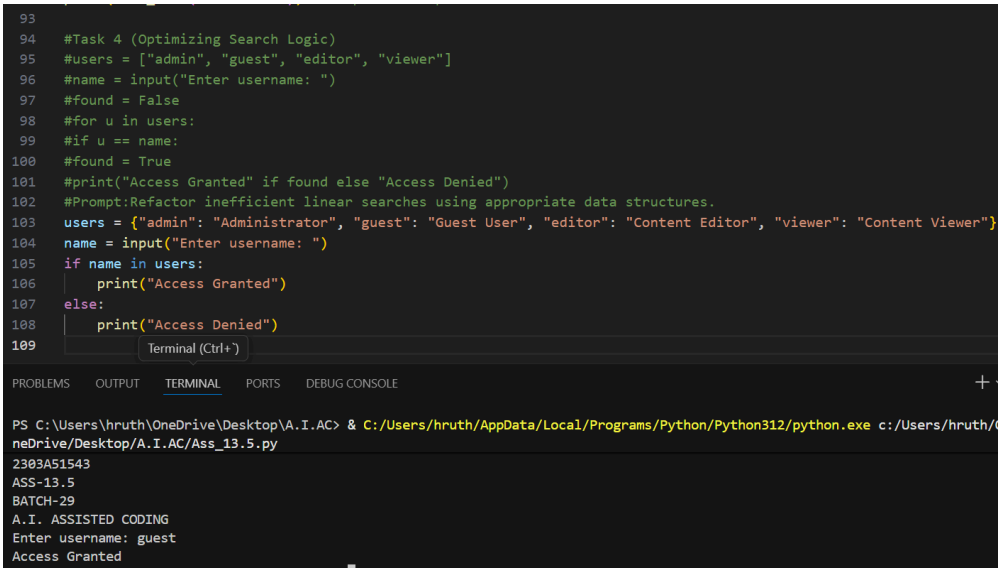


Task 4 (Optimizing Search Logic)
• Task: Refactor inefficient linear searches using appropriate data structures.
• Focus Areas:
o Time complexity
o Data structure choice
Legacy Code:
users = ["admin", "guest", "editor", "viewer"]
name = input("Enter username: ")
found = False
for u in users:
if u == name:
found = True
print("Access Granted" if found else "Access Denied")
Expected Outcome:
o Use of sets or dictionaries with complexity justification.

```
#Task 4 (Optimizing Search Logic)
#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 inefficient linear searches using appropriate data structures.
users = {"admin": "Administrator", "guest": "Guest User", "editor": "Content Editor", "vi
ewer": "Content Viewer"}
```

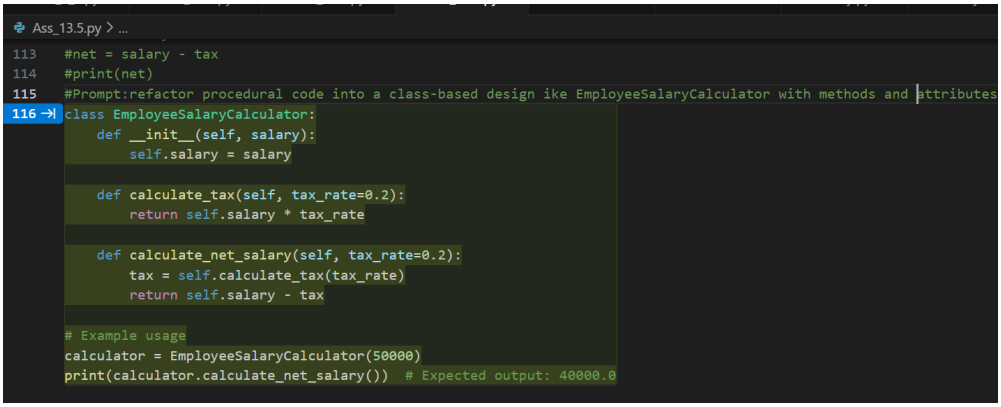
```
name = input("Enter username: ")
if name in users:
    print("Access Granted")
else:
    print("Access Denied")
```

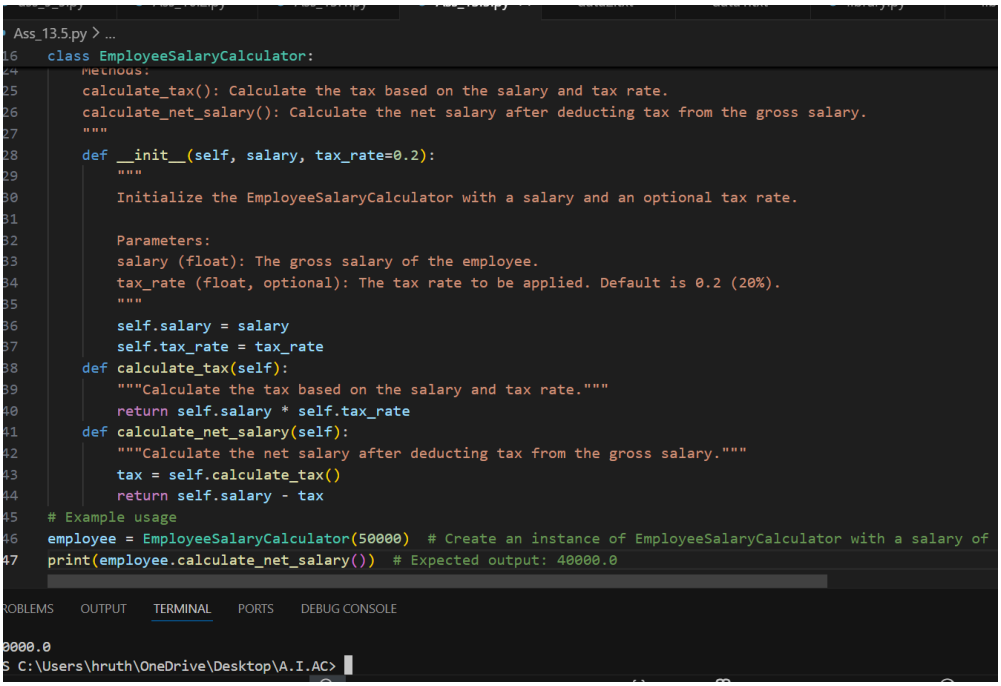




Task 5 (Refactoring Procedural Code into OOP Design)
• Task: Use AI to refactor procedural code into a class-based design.
• Focus Areas:
o Object-Oriented principles
o Encapsulation
Legacy Code:
salary = 50000
tax = salary * 0.2
net = salary - tax
print(net)
Expected Outcome:
o A class like EmployeeSalaryCalculator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

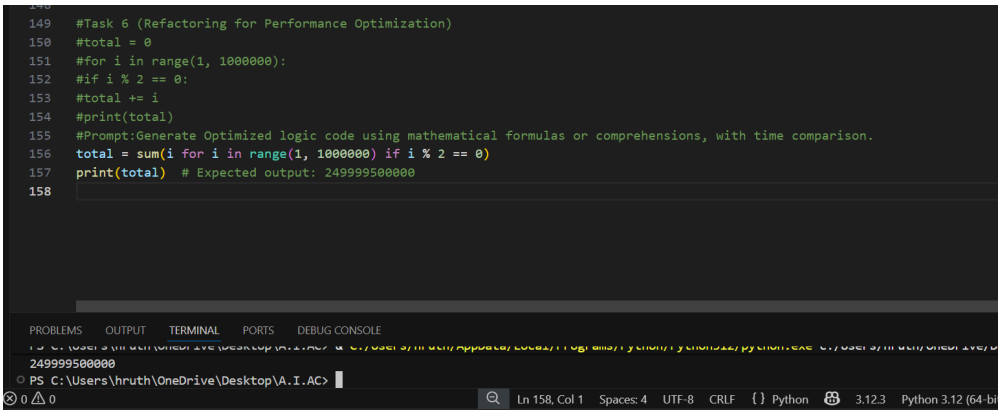
```
#Task 5 (Refactoring Procedural Code into OOP Design)
#salary = 50000
#tax = salary * 0.2
#net = salary - tax
#print(net)
#Prompt:refactor procedural code into a class-based design ike EmployeeSalaryCalculator w
ith methods and attributes.
class EmployeeSalaryCalculator:
    """
    A class to calculate the net salary of an employee after tax deductions.
    Attributes:
    salary (float): The gross salary of the employee.
    tax_rate (float): The tax rate to be applied on the salary.
    Methods:
    calculate_tax(): Calculate the tax based on the salary and tax rate.
    calculate_net_salary(): Calculate the net salary after deducting tax from the gross s
alary.
    """
    def __init__(self, salary, tax_rate=0.2):
        """
        Initialize the EmployeeSalaryCalculator with a salary and an optional tax rate.
        Parameters:
        salary (float): The gross salary of the employee.
        tax_rate (float, optional): The tax rate to be applied. Default is 0.2 (20%).
        """
        self.salary = salary
        self.tax_rate = tax_rate
    def calculate_tax(self):
        """Calculate the tax based on the salary and tax rate."""
        return self.salary * self.tax_rate
    def calculate_net_salary(self):
        """Calculate the net salary after deducting tax from the gross salary."""
        tax = self.calculate_tax()
        return self.salary - tax
# Example usage
employee = EmployeeSalaryCalculator(50000)  # Create an instance of EmployeeSalaryCalcula
tor with a salary of 50000
print(employee.calculate_net_salary())  # Expected output: 40000.0
```





💡 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.

```
#Task 6 (Refactoring for Performance Optimization)
#total = 0
#for i in range(1, 1000000):
#if i % 2 == 0:
#total += i
#print(total)
#Prompt:Generate Optimized logic code using mathematical formulas or comprehensions, with
time comparison.
total = sum(i for i in range(1, 1000000) if i % 2 == 0)
print(total)  # Expected output: 249999500000
```



Task 7 (Removing Hidden Side Effects)

- Task: Refactor code that modifies shared mutable state.
- Focus Areas:

o Functional-style refactoring
o Predictability

Legacy Code:

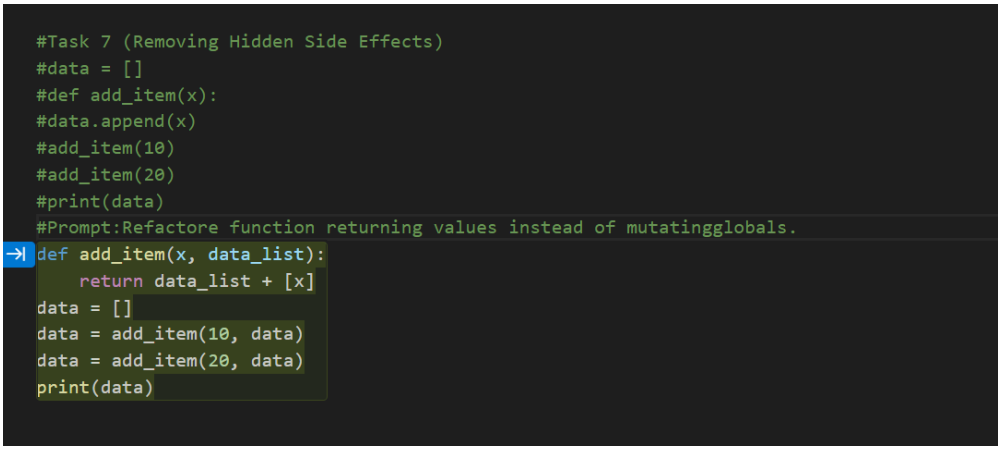
```
data = []
def add_item(x):
data.append(x)
add_item(10)
add_item(20)
print(data)
```

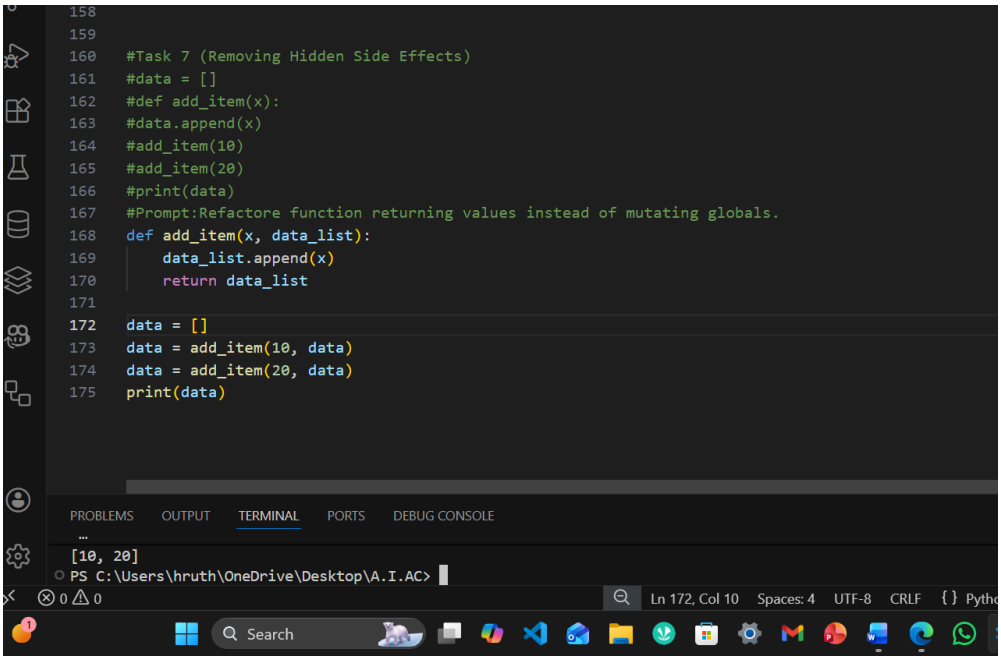
Expected Outcome:

o Refactored function returning values instead of mutating globals.

```
#Task 7 (Removing Hidden Side Effects)
#data = []
#def add_item(x):
#data.append(x)
#add_item(10)
#add_item(20)
#print(data)
#Prompt:Refactore function returning values instead of mutating globals.
```

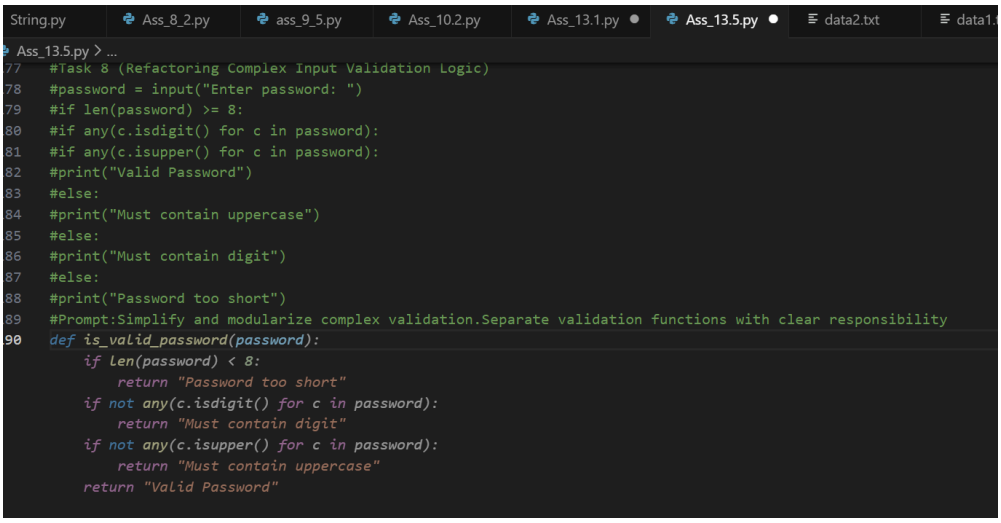
```
def add_item(x, data_list):
    data_list.append(x)
    return data_list
data = []
data = add_item(10, data)
data = add_item(20, data)
print(data)
```

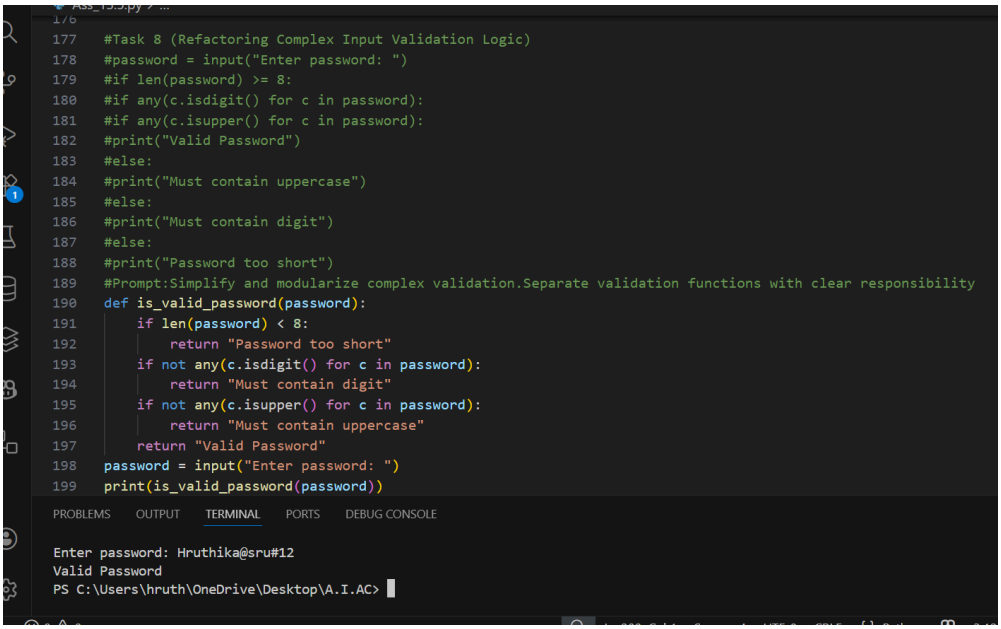




Task 8 (Refactoring Complex Input Validation Logic)
• Task: Use AI to simplify and modularize complex validation rules.
• Focus Areas:
o Readability
o Testability
Legacy Code:
password = input("Enter password: ")
if len(password) >= 8:
if any(c.isdigit() for c in password):
if any(c.isupper() for c in 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

```
#Task 8 (Refactoring Complex Input Validation Logic)
#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:Simplify and modularize complex validation.Separate validation functions with cle
ar responsibility
def is_valid_password(password):
    if len(password) < 8:
        return "Password too short"
    if not any(c.isdigit() for c in password):
        return "Must contain digit"
    if not any(c.isupper() for c in password):
        return "Must contain uppercase"
    return "Valid Password"
password = input("Enter password: ")
print(is_valid_password(password))
```


String.py
Ass_8_2.py
ass_9_5.py
Ass_10.2.py
Ass_13.1.py
Ass_13.5.py
data2.txt
Ass_13.5.py > ...
#Task 8 (Refactoring Complex Input Validation Logic)
#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:Simplify and modularize complex validation.Separate validation functions with clear responsibility
def is_valid_password(password):
if len(password) < 8:
return "Password too short"
if not any(c.isdigit() for c in password):
return "Must contain digit"
if not any(c.isupper() for c in password):
return "Must contain uppercase"
return "Valid Password"


#Task 8 (Refactoring Complex Input Validation Logic)
#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:Simplify and modularize complex validation.Separate validation functions with clear responsibility
def is_valid_password(password):
if len(password) < 8:
return "Password too short"
if not any(c.isdigit() for c in password):
return "Must contain digit"
if not any(c.isupper() for c in password):
return "Must contain uppercase"
return "Valid Password"
password = input("Enter password: ")
print(is_valid_password(password))
PROBLEMS
OUTPUT
TERMINAL
PORTS
DEBUG CONSOLE
Enter password: Hruthika@sru#12
Valid Password
PS C:\Users\hruth\OneDrive\Desktop\A.I.AC>