|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE** | | | | | **DEPARTMENT OF COMPUTER SCIENCE ENGINEERING** | | | | |
| **Program Name:** B. Tech | | | | **Assignment Type: Lab** | | | **Academic Year:**2025-2026 | | |
| **Course Coordinator Name** | | | | Dr. Rishabh Mittal | | | | | |
| **Instructor(s) Name** | | | | |  | | --- | | Mr. S Naresh Kumar | | Ms. B. Swathi | | Dr. Sasanko Shekhar Gantayat | | Mr. Md Sallauddin | | Dr. Mathivanan | | Mr. Y Srikanth | | Ms. N Shilpa | | Dr. Rishabh Mittal (Coordinator) | | Dr. R. Prashant Kumar | | Mr. Ankushavali MD | | Mr. B Viswanath | | Ms. Sujitha Reddy | | Ms. A. Anitha | | Ms. M.Madhuri | | Ms. Katherashala Swetha | | Ms. Velpula sumalatha | | Mr. Bingi Raju | | | | | | |
| **CourseCode** | | | 23CS002PC304 | **Course Title** | | AI Assisted Coding | | | |
| **Year/Sem** | | | III/II | **Regulation** | | R23 | | | |
| **Date and Day**  **of Assignment** | | | **Week7 – Friday** | **Time(s)** | | 23CSBTB01 To 23CSBTB52 | | | |
| **Duration** | | | 2 Hours | **Applicable to**  **Batches** | | All batches | | | |
| **Assignment Number: 13.5**(Present assignment number)/**24**(Total number of assignments) | | | | | | | | | |
|  | | | | | | | | | |
|  | **Q.No.** | **Question** | | | | | | ***Expected Time***  ***to complete*** |  |
|  | 1 | **Lab 13: Code Refactoring – Improving Legacy Code with AI Suggestions**  **Lab Objectives:**   * Identify code smells and inefficiencies in legacy Python scripts. * Use AI-assisted coding tools to **refactor** for readability, maintainability, and performance. * Apply **modern Python best practices** while ensuring output correctness.   **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.  **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 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 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 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 6**  **• 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 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 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 | | | | | | Week7 - Friday |  |

**Task 1: (Refactoring – Removing Global Variables)**

**Prompt:**

Refactor the following Python code to eliminate unnecessary global variables by passing values as function parameters. Improve modularity and testability while maintaining the same output.

**Code:**

def calculate\_interest(amount, rate):

    """

    Calculate interest based on amount and rate.

    Parameters:

    amount (float): Principal amount

    rate (float): Interest rate

    Returns:

    float: Calculated interest

    """

    return amount \* rate

def main():

    """

    Main function to demonstrate interest calculation.

    Refactored to eliminate global variables.

    """

    # Example Usage

    interest\_rate = 0.1  # Local variable within the main function

    principal\_amount = 1000

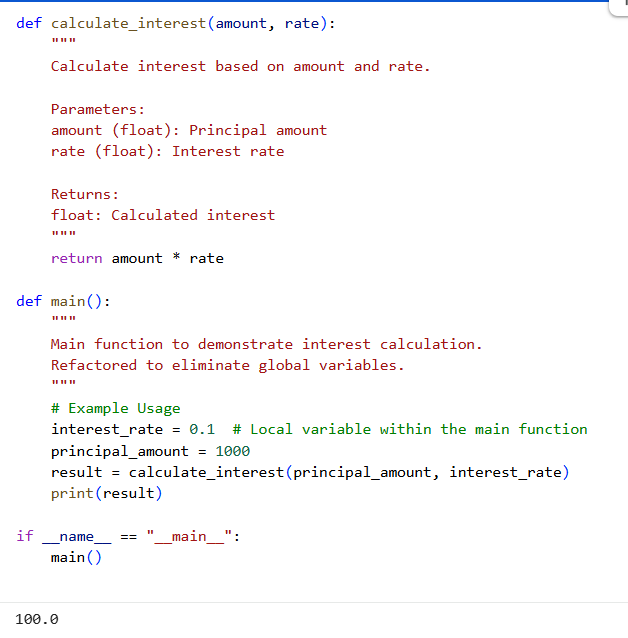
    result = calculate\_interest(principal\_amount, interest\_rate)

    print(result)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Sample Input/Output:**



**Explaination**

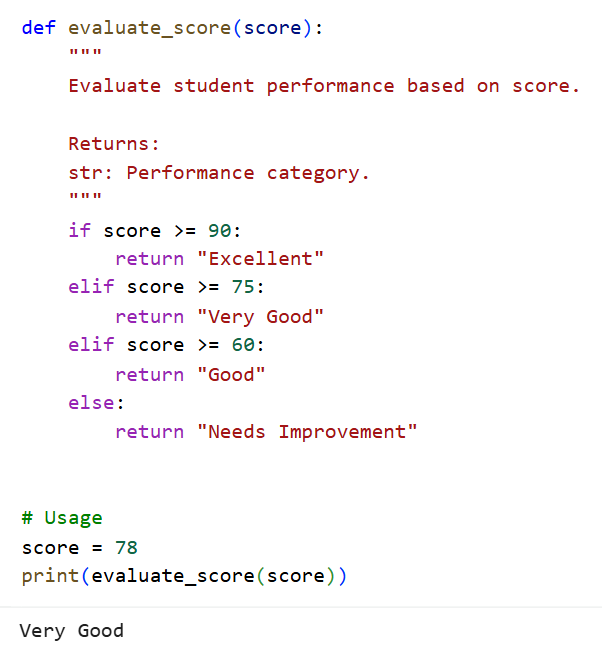
This Python code refactors an interest calculation. The calculate\_interest function takes amount and rate as parameters, ensuring it's self-contained and reusable. The main logic is encapsulated in a main() function, where interest\_rate and principal\_amount are now local variables. This eliminates global variables, making the code more modular and easier to test independently. The if \_\_name\_\_ == "\_\_main\_\_": block ensures main() runs when the script is executed.

**Task 2: (Refactoring Deeply Nested Conditionals)**

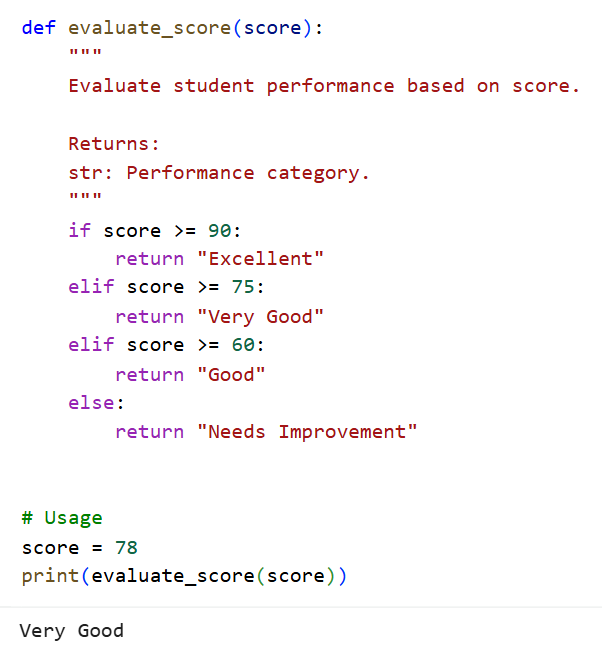
**Prompt:**

Refactor the following deeply nested if–elif–else structure into a cleaner and more readable format using guard clauses or simplified logic. Improve readability, logical clarity, and maintainability while preserving the same output.

**Code:**

****

**Sample Input/Output:**

****

**Explanation:**

* The deeply nested conditional structure was flattened using elif statements.
* This eliminates multiple indentation levels, improving readability.
* The logic flow is now linear and easier to understand.
* A separate function evaluate\_score() improves modularity.
* Returning values instead of directly printing enhances testability.
* The structure is now easier to modify if new score categories are added.
* The behavior and output remain exactly the same as the legacy code.
* The refactored version follows clean coding and maintainability principles.

**Task 3: (Refactoring Repeated File Handling Code)**

**Prompt:**

Refactor the following repeated file handling code to follow the DRY principle using a reusable function and context managers.Improve readability and ensure proper file handling using with open().

**Code:**

def read\_file(filename):

    """

    Read and print the contents of a file.

    Parameters:

    filename (str): Name of the file to read.

    """

    try:

        with open(filename, "r") as file:

            print(file.read())

    except FileNotFoundError:

        print(f"File '{filename}' not found.")

# Usage

read\_file("data1.txt")

read\_file("data2.txt")

**Explanation:**

* The repeated open-read-close logic was refactored into a reusable function read\_file().
* The DRY (Don’t Repeat Yourself) principle is applied by eliminating duplicated code.
* The with open() context manager automatically handles file closing.
* This prevents resource leaks and ensures safer file operations.
* Error handling using try-except improves robustness.
* The function accepts the filename as a parameter, increasing flexibility.
* The structure is now modular and easier to maintain.

**Task 4: (Optimizing Search Logic)**

**Prompt:**

Refactor the following inefficient linear search code by using a more appropriate data structure to improve time complexity. Justify the data structure choice and maintain the same output behavior.

**Code:**

# Using a set for faster lookup

users = {"admin", "guest", "editor", "viewer"}

name = input("Enter username: ")

if name in users:

    print("Access Granted")

else:

    print("Access Denied")

**Sample Input/Output:**

****

**Explanation:**

* The original implementation used a list and linear search with O(n) time complexity.
* A set data structure is used in the refactored version.
* Set membership checking (in) has average O(1) time complexity.
* This significantly improves performance for large datasets.
* The explicit loop and boolean flag were removed to simplify logic.
* The new version is shorter, cleaner, and easier to read.
* Output behavior remains exactly the same as the legacy code.
* Choosing the correct data structure improves efficiency and scalability.

**Task 5: (Refactoring Procedural Code into OOP Design)**

**Prompt:**

Refactor the following procedural salary calculation code into a class-based OOP design applying encapsulation principles. Create a class with appropriate attributes and methods while maintaining the same output.

**Code:**

class EmployeeSalaryCalculator:

"""

A class to calculate employee net salary after tax deduction.

"""

def \_\_init\_\_(self, salary, tax\_rate=0.2):

"""

Initialize salary and tax rate.

Parameters:

salary (float): Employee's gross salary.

tax\_rate (float): Tax percentage (default is 20%).

"""

self.salary = salary

self.tax\_rate = tax\_rate

def calculate\_tax(self):

"""

Calculate tax based on salary.

Returns:

float: Calculated tax amount.

"""

return self.salary \* self.tax\_rate

def calculate\_net\_salary(self):

"""

Calculate net salary after deducting tax.

Returns:

float: Net salary.

"""

return self.salary - self.calculate\_tax()

# Usage

employee = EmployeeSalaryCalculator(50000)

print(employee.calculate\_net\_salary())

**Sample Input/Output:**



**Explanation:**

* The procedural logic was refactored into a class named EmployeeSalaryCalculator.
* Salary and tax rate are encapsulated as instance attributes.
* The \_\_init\_\_ constructor initializes object state.
* A separate method calculate\_tax() improves modularity.
* The calculate\_net\_salary() method computes final salary using internal logic.
* Encapsulation ensures data and behavior are bundled together.
* The class design makes the system scalable for future extensions.

**Task 6: (Refactoring for Performance Optimization)**

**Prompt:**

Refactor the following performance-heavy loop to improve efficiency using mathematical optimization or Python built-in functions. Maintain the same output and include time complexity comparison.

**Code:**

def sum\_of\_even\_numbers(limit):

    """

    Calculate sum of even numbers from 1 to limit (exclusive).

    Time Complexity:

    Optimized Version: O(1)

    Legacy Version: O(n)

    Returns:

    int: Sum of even numbers.

    """

    # Number of even numbers below limit

    n = (limit - 1) // 2

    # Sum of first n even numbers = n \* (n + 1)

    return n \* (n + 1)

# Usage

print(sum\_of\_even\_numbers(1000000))

**Sample Input/Output:**



**Explanation:**

* The legacy code used a loop with O(n) time complexity.
* It checked each number for evenness, which is inefficient for large ranges.
* The optimized version uses a mathematical formula to compute the sum directly.
* No iteration is required, making it significantly faster.
* The output remains exactly the same as the original logic
* The sum of first n even numbers equals n × (n + 1).
* This reduces time complexity from O(n) to O(1).
* This demonstrates how algorithmic thinking improves performance dramatically.

**Task 7: (Removing Hidden Side Effects)**

**Prompt:**

Refactor the following code to remove hidden side effects caused by modifying shared global mutable state. Rewrite the function to return updated values instead of mutating global variables.

**Code:**

def add\_item(data, x):

    """

    Return a new list with the item added.

    Parameters:

    data (list): Original list.

    x (int): Value to add.

    Returns:

    list: New updated list.

    """

    return data + [x]

# Usage

data = []

data = add\_item(data, 10)

data = add\_item(data, 20)

print(data)

**Sample Input/Output:**



**Explanation:**

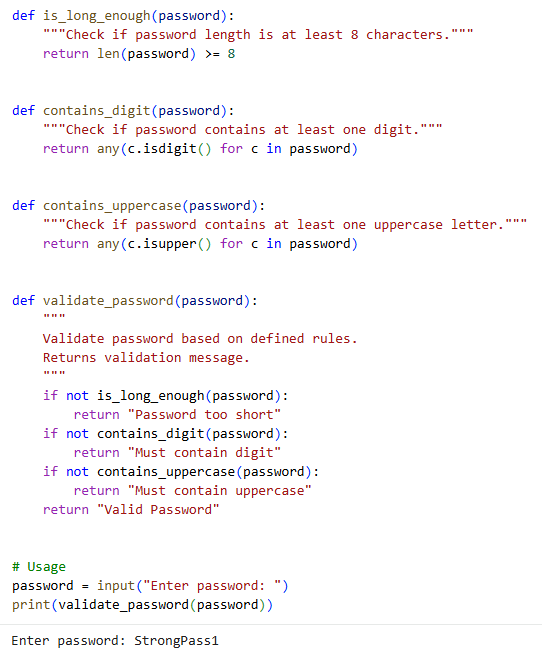
* It returns a new updated list instead of modifying the original.
* This follows a functional programming style.
* The code becomes more testable and easier to debug.
* Functions depending on global mutable state reduce predictability.
* The refactored version removes the global variable dependency.
* The function now accepts the list as a parameter.
* The legacy code modified a global list, creating hidden side effects.
* The output remains the same while improving reliability and maintainability.

**Task 8:**

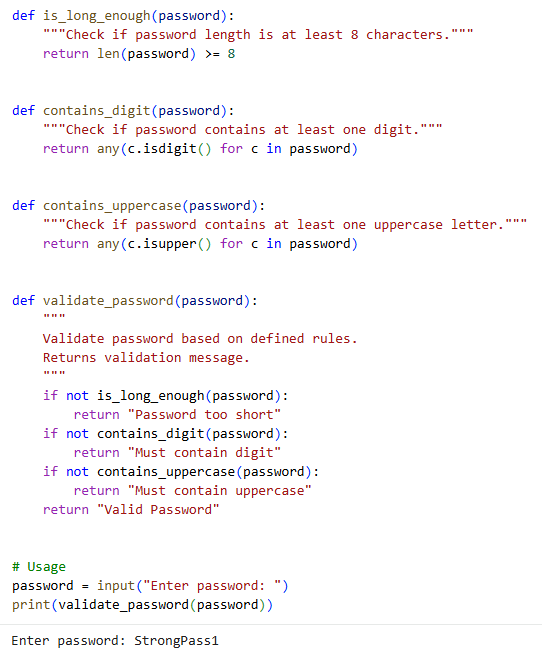
**Prompt:**

Refactor the following complex nested password validation logic into separate modular validation functions. Improve readability, maintainability, and testability while preserving the same behavior.

**Code:**

****

**Sample Input/Output:**



**Explanation:**

* The deeply nested validation logic was broken into separate helper functions.
* Each function now has a single responsibility, improving clarity.
* The main validate\_password() function orchestrates validation flow.
* This modular design improves readability significantly.
* Guard clauses replace nested conditions, reducing indentation levels.
* Future validation rules can be added easily without modifying complex logic.
* The output behavior remains identical while making the code maintainable.