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*

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:
```

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:

```
100.0
```

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:

```
def evaluate_score(score):
    """
    Evaluate student performance based on score.

    Returns:
    str: Performance category.
    """
    if score >= 90:
        return "Excellent"
    elif score >= 75:
        return "Very Good"
    elif score >= 60:
        return "Good"
    else:
        return "Needs Improvement"


# Usage
score = 78
print(evaluate_score(score))
```

Sample Input/Output:

```
Very Good
```

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:

```
Enter username: admin
Access Granted
```

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:

```
40000.0
```

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:

```
249999500000
```

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:

```
[10, 20]
```

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:

```python
def is_long_enough(password):
    """Check if password length is at least 8 characters."""
    return len(password) >= 8


def contains_digit(password):
    """Check if password contains at least one digit."""
    return any(c.isdigit() for c in password)


def contains_uppercase(password):
    """Check if password contains at least one uppercase letter."""
    return any(c.isupper() for c in password)


def validate_password(password):
    """
    Validate password based on defined rules.
    Returns validation message.
    """
    if not is_long_enough(password):
        return "Password too short"
    if not contains_digit(password):
        return "Must contain digit"
    if not contains_uppercase(password):
        return "Must contain uppercase"
    return "Valid Password"


# Usage
password = input("Enter password: ")
print(validate_password(password))
```

Sample Input/Output:

```
Enter password: StrongPass1
```

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.