SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE			DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
Program Name: B. Tech		**Assignment Type: Lab**	**Academic Year:**2025-2026
Instructor(s) Name		S Naresh Kumar	
Course Code	24CS002PC215	**Course Title**	AI Assisted Coding
Year/Sem	II/III	**Regulation**	R24
Date and Day of Assignment	Week3 – Monday	**Time(s)**	
Duration	2 Hours	**Applicable to Batches**	
Assignment Number:5.1 and 6(Present assignment number)/**24**(Total number of assignments)			

Q.No.	Question	*Expected Time to complete*
	Task 1: **Employee Data:** Create Python code that defines a class named \`Employee\` with the following attributes: \`empid\`, \`empname\`, \`designation\`, \`basic_salary\`, and \`exp\`. Implement a method \`display_details()\` to print all employee details. Implement another method \`calculate_allowance()\` to determine additional allowance based on experience: - If \`exp > 10 years\` → allowance = 20% of \`basic_salary\` - If \`5 ≤ exp ≤ 10 years\` → allowance = 10% of \`basic_salary\` - If \`exp < 5 years\` → allowance = 5% of \`basic_salary\` Finally, create at least one instance of the \`Employee\` class, call the \`display_details()\` method, and print the calculated allowance.	Week3 - Monday

Code:

```
class Employee:
    def __init__(self, emp_id, emp_name, emp_salary,
designation, basic_salary, experience):
        self.emp_id = emp_id
        self.emp_name = emp_name
        self.emp_salary = emp_salary
        self.designation = designation
        self.basic_salary = basic_salary
        self.experience = experience

    def display_details(self):
        print(f"Employee ID: {self.emp_id}")
        print(f"Employee Name: {self.emp_name}")
        print(f"Employee Salary: {self.emp_salary}")
        print(f"Designation: {self.designation}")
        print(f"Basic Salary: {self.basic_salary}")
        print(f"Experience: {self.experience} years")

    def calculate_allowance(self):
        if self.experience > 10:
            allowance = 0.20 * self.basic_salary
        elif 5 <= self.experience <= 10:
            allowance = 0.10 * self.basic_salary
        else:
            allowance = 0.05 * self.basic_salary
        return allowance
# Creating an instance of Employee
employee = Employee(emp_id=101, emp_name="John Doe",
emp_salary=75000
                    , designation="Software
Engineer", basic_salary=60000, experience=8)
employee1 = Employee(emp_id=102, emp_name="Jane
Smith", emp_salary=85000
                    , designation="Senior Developer",
basic_salary=70000, experience=12)
employee2 = Employee(emp_id=103, emp_name="Alice
Johnson", emp_salary=50000
                    , designation="Junior Developer",
```

```
basic_salary=40000, experience=3)
employee3 = Employee(emp_id=104, emp_name="Bob
Brown", emp_salary=95000
                    , designation="Team Lead",
basic_salary=80000, experience=15)
# Displaying employee details
employee.display_details()
# Calculating and printing allowance
allowance = employee.calculate_allowance()
print(f"Calculated Allowance: {allowance}")
print("\n")
employee1.display_details()
allowance1 = employee1.calculate_allowance()
print(f"Calculated Allowance: {allowance1}")
print("\n")
employee2.display_details()
allowance2 = employee2.calculate_allowance()
print(f"Calculated Allowance: {allowance2}")
print("\n")
employee3.display_details()
allowance3 = employee3.calculate_allowance()
print(f"Calculated Allowance: {allowance3}")
# Analysis:
# Time Complexity: O(1) - The operations in the
methods are constant time operations.
# Space Complexity: O(1) - The space used by the
instance variables is constant.
# Compare this snippet from Assignment6/Task10.py:
```

Output

```
Employee Name: John Doe
Employee Salary: 75000
Designation: Software Engineer
Basic Salary: 60000
Experience: 8 years
Calculated Allowance: 6000.0


Employee ID: 102
Employee Name: Jane Smith
Employee Salary: 85000
Designation: Senior Developer
Basic Salary: 70000
Experience: 12 years
Calculated Allowance: 14000.0
```

Explanation

This class demonstrates object-oriented programming principles by encapsulating employee data and behavior. The calculate_allowance() method uses conditional statements to determine allowance percentage based on experience tiers. The implementation follows clean code practices with clear method names and proper data encapsulation.

Task 2:

Electricity Bill Calculation- Create Python code that defines a class named `ElectricityBill` with attributes: `customer_id`, `name`, and `units_consumed`. Implement a method `display_details()` to print customer details, and a method `calculate_bill()` where:

- Units ≤ 100 → ₹5 per unit

- 101 to 300 units → ₹7 per unit

- More than 300 units → ₹10 per unit

Create a bill object, display details, and print the total bill amount.

Code

```
class ElectricityBill:
    def __init__(self, customer_id, name,
units_consumed):
        self.customer_id = customer_id
        self.name = name
        self.units_consumed = units_consumed

    def display_details(self):
        print(f"Customer ID: {self.customer_id}")
        print(f"Name: {self.name}")
        print(f"Units Consumed:
{self.units_consumed}")

    def calculate_bill(self):
        if self.units_consumed <= 100:
            rate_per_unit = 5
        elif 101 <= self.units_consumed <= 300:
            rate_per_unit = 7
        else:
            rate_per_unit = 10
        total_bill = self.units_consumed *
rate_per_unit
        return total_bill
# Creating an instance of ElectricityBill
bill = ElectricityBill(customer_id=1, name="Alice",
units_consumed=250)
# Displaying customer details
bill.display_details()
# Calculating and printing total bill amount
```

```
total_amount = bill.calculate_bill()
print(f"Total Bill Amount: ₹{total_amount}")
```

Output

```
Customer ID: 1
Name: Alice
Units Consumed: 250
Total Bill Amount: ₹1750
```

Explanation

The calculate_bill() method implements tiered pricing logic commonly used in utility billing systems. This progressive pricing structure encourages energy conservation by charging higher rates for excessive consumption. The implementation correctly handles boundary conditions between pricing tiers.

Task 3:

Product Discount Calculation- Create Python code that defines a class named `Product` with attributes: `product_id`, `product_name`, `price`, and `category`. Implement a method `display_details()` to print product details. Implement another method `calculate_discount()` where:

- Electronics → 10% discount

- Clothing → 15% discount

- Grocery → 5% discount

Create at least one product object, display details, and print the final price after discount.

Code

```
class Product:
    def __init__(self, product_id, product_name,
price, category):
        self.product_id = product_id
        self.product_name = product_name
        self.price = price
        self.category = category

    def display_details(self):
        print(f"Product ID: {self.product_id}")
        print(f"Product Name: {self.product_name}")
        print(f"Price: ₹{self.price}")
        print(f"Category: {self.category}")

    def calculate_discount(self):
        if self.category == "Electronics":
            discount_rate = 0.10
        elif self.category == "Clothing":
            discount_rate = 0.15
        elif self.category == "Grocery":
            discount_rate = 0.05
        else:
            discount_rate = 0.0  # No discount for
other categories
        final_price = self.price * (1 -
discount_rate)
        return final_price
# Creating an instance of Product
product = Product(product_id=101,
product_name="Smartphone", price=20000,
                        category="Electronics")
# Displaying product details
product.display_details()
# Calculating and printing final price after discount
```

```
final_price = product.calculate_discount()
print(f"Final Price after discount: ₹{final_price}")
```

Output

```
Product ID: 101
Product Name: Smartphone
Price: ₹20000
Category: Electronics
Final Price after discount: ₹18000.0
```

Explanation

This class demonstrates polymorphic behavior through category-based discount calculation.

Task 4:

Book Late Fee Calculation- Create Python code that defines a class named `LibraryBook` with attributes: `book_id`, `title`, `author`, `borrower`, and `days_late`. Implement a method `display_details()` to print book details, and a method `calculate_late_fee()` where:

- Days late ≤ 5 → ₹5 per day

- 6 to 10 days late → ₹7 per day

- More than 10 days late → ₹10 per day

Create a book object, display details, and print the late fee.

Code

```
class LibraryBook:
    def __init__(self, book_id, title, author,
borrower, days_late):
        self.book_id = book_id
        self.title = title
        self.author = author
        self.borrower = borrower
        self.days_late = days_late

    def display_details(self):
        print(f"Book ID: {self.book_id}")
        print(f"Title: {self.title}")
        print(f"Author: {self.author}")
        print(f"Borrower: {self.borrower}")
        print(f"Days Late: {self.days_late}")

    def calculate_late_fee(self):
        if self.days_late <= 5:
            fee_per_day = 5
        elif 6 <= self.days_late <= 10:
            fee_per_day = 7
        else:
            fee_per_day = 10
        total_fee = self.days_late * fee_per_day
        return total_fee
# Creating an instance of LibraryBook
book = LibraryBook(book_id=1, title="The Great
Gatsby", author="F. Scott Fitzgerald",
                   borrower="John Doe", days_late=8)
# Displaying book details
book.display_details()
# Calculating and printing late fee
late_fee = book.calculate_late_fee()
print(f"Late Fee: ₹{late_fee}")
# Analysis:
# Time Complexity: O(1) - Both methods perform a
constant number of operations.
```

Output

```
Book ID: 1
Title: The Great Gatsby
Author: F. Scott Fitzgerald
Borrower: John Doe
Days Late: 8
Late Fee: ₹56
```

Explanation

The late fee calculation implements a progressive penalty structure that incentivizes timely returns while being fair to borrowers with minor delays. The method correctly handles edge cases including on-time returns (zero days late).The implementation uses clear conditional logic to apply appropriate fee rates based on delay duration.

Task 5:

Student Performance Report - Define a function `student_report(student_data)` that accepts a dictionary containing student names and their marks. The function should:

- Calculate the average score for each student

- Determine pass/fail status (pass ≥ 40)

- Return a summary report as a list of dictionaries

Use Copilot suggestions as you build the function and format the output.

Code

```
def student_report(student_data):
    report = []
    for student, marks in student_data.items():
        average_score = sum(marks) / len(marks)
        status = "Pass" if average_score >= 40 else "Fail"
        report.append({
            "name": student,
            "average_score": average_score,
            "status": status
        })
    return report
# Example usage
if __name__ == "__main__":
    student_data = {
        "Alice": [85, 92, 78],
        "Bob": [58, 64, 70],
        "Charlie": [35, 40, 30]
    }
    summary_report = student_report(student_data)
    for student in summary_report:
        print(f"Name: {student['name']}, Average Score: {student['average_score']:.2f}, Status: {student['status']}")
# Analysis:
# Time Complexity: O(n) - where n is the number of students, as we iterate
```

Output

```
5.py
Name: Alice, Average Score: 85.00, Status: Pass
Name: Bob, Average Score: 64.00, Status: Pass
Name: Charlie, Average Score: 35.00, Status: Fail
```

Explanation

The Student class encapsulates academic performance tracking by storing marks and calculating grades based on average performance. The calculate_average() method uses Python's built-in sum() function for clean calculation. The grading system uses cascading if-elif statements ordered from highest to lowest grade,ensuring the correct grade is assigned. This implementation demonstrates effective use of class methods to separate concerns: data storage, calculation, and display.

Task 6:

Taxi Fare Calculation-Create Python code that defines a class named `TaxiRide` with attributes: `ride_id`, `driver_name`, `distance_km`, and `waiting_time_min`. Implement a method `display_details()` to print ride details, and a method `calculate_fare()` where:

- ₹15 per km for the first 10 km
- ₹12 per km for the next 20 km
- ₹10 per km above 30 km
- Waiting charge: ₹2 per minute

Create a ride object, display details, and print the total fare.

Code

```
class TaxiRide:
    def __init__(self, ride_id, driver_name,
distance_km, waiting_time_min):
        self.ride_id = ride_id
```

```
        self.driver_name = driver_name
        self.distance_km = distance_km
        self.waiting_time_min = waiting_time_min

    def display_details(self):
        print(f"Ride ID: {self.ride_id}")
        print(f"Driver Name: {self.driver_name}")
        print(f"Distance (km): {self.distance_km}")
        print(f"Waiting Time (min):
{self.waiting_time_min}")

    def calculate_fare(self):
        fare = 0
        if self.distance_km <= 10:
            fare += self.distance_km * 15
        elif 10 < self.distance_km <= 30:
            fare += 10 * 15 + (self.distance_km - 10)
* 12
        else:
            fare += 10 * 15 + 20 * 12 +
(self.distance_km - 30) * 10
        fare += self.waiting_time_min * 2
        return fare
# Creating an instance of TaxiRide
ride = TaxiRide(ride_id=1, driver_name="Rajesh",
distance_km
                     =35, waiting_time_min=15)
# Displaying ride details
ride.display_details()
# Calculating and printing total fare
total_fare = ride.calculate_fare()
print(f"Total Fare: ₹{total_fare}")
```

Output

```
Ride ID: 1
Driver Name: Rajesh
Distance (km): 35
Waiting Time (min): 15
Total Fare: ₹470
```

Explanation

The BankAccount class implements essential banking operations with proper validation and error handling. The deposit() and withdraw() methods include guard clauses to prevent invalid operations like negative amounts or overdrawing. The class maintains balance integrity by updating it only after validation passes. Return values (True/False) allow calling code to verify transaction success. This implementation demonstrates defensive programming practices and state management.

Task 7:

Statistics Subject Performance - Create a Python function `statistics_subject(scores_list)` that accepts a list of 60 student scores and computes key performance statistics. The function should return the following:

- Highest score in the class
- Lowest score in the class
- Class average score
- Number of students passed (score ≥ 40)
- Number of students failed (score < 40)

Allow Copilot to assist with aggregations and logic

	Code	

```
def statistics_subject(scores_list):
    highest_score = max(scores_list)
    lowest_score = min(scores_list)
    average_score = sum(scores_list) /
len(scores_list)
    passed_count = sum(1 for score in scores_list if
score >= 40)
    failed_count = sum(1 for score in scores_list if
score < 40)

    return {
        "highest_score": highest_score,
        "lowest_score": lowest_score,
        "average_score": average_score,
        "passed_count": passed_count,
        "failed_count": failed_count
    }
# Example usage
if __name__ == "__main__":
    scores = [55, 67, 45, 23, 89, 90, 34, 76, 88, 92,
              41, 39, 60, 72, 81, 33, 49, 58, 77, 84,
              91, 38, 44, 53, 66, 70, 79, 82, 95,
100,
              29, 31, 36, 42, 47, 50, 54, 61, 65, 68,
              74, 80, 85, 87, 93, 96, 98, 22, 25, 27,
              30, 32, 35, 37, 40, 43, 46, 48, 51, 52]
    stats = statistics_subject(scores)
    print(f"Highest Score: {stats['highest_score']}")
    print(f"Lowest Score: {stats['lowest_score']}")
    print(f"Average Score:
{stats['average_score']:.2f}")
    print(f"Number of Students Passed:
{stats['passed_count']}")
    print(f"Number of Students Failed:
{stats['failed_count']}")
```

Output

```
Highest Score: 100
Lowest Score: 22
Average Score: 59.25
Number of Students Passed: 45
Number of Students Failed: 15
```

Explanation

The Temperature class encapsulates temperature conversion logic using standard scientific formulas. The class stores temperature in Celsius as the base unit and provides methods to convert to other scales on demand. This approach maintains a single source of truth while allowing flexible representation. The display_all() method provides a convenient interface for viewing all conversions simultaneously. This implementation demonstrates encapsulation and method-based calculations.

Lab 5: Ethical Foundations – Responsible AI Coding Practices

Lab Objectives:

- To explore the ethical risks associated with AI-generated code.
- To recognize issues related to security, bias, transparency, and copyright.
- To reflect on the responsibilities of developers when using AI tools in software development.
- To promote awareness of best practices for responsible and ethical AI coding.

Lab Outcomes (LOs):

After completing this lab, students will be able to:

- Identify and avoid insecure coding patterns generated by AI tools.
- Detect and analyze potential bias or discriminatory logic in AI-generated outputs.
- Evaluate originality and licensing concerns in reused AI-generated code.

- Understand the importance of explainability and transparency in AI-assisted programming.
- Reflect on accountability and the human role in ethical AI coding practices.

Task Description #8 (Transparency in Algorithm Optimization)

Task: Use AI to generate two solutions for checking prime numbers:

- Naive approach(basic)
- Optimized approach

Prompt:

"Generate Python code for two prime-checking methods and explain how the optimized version improves performance."

Expected Output:

- Code for both methods.
- Transparent explanation of time complexity.
- Comparison highlighting efficiency improvements.

Code

```
# genrate well commentes code to Check Prime
Number
def is_prime_naive(n):
    """Check if a number is prime using the naive
approach."""
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
# Optimized Approach to Check Prime Number
def is_prime_optimized(n):
    """Check if a number is prime using the
optimized approach."""
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
```

```
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
# Example usage
if __name__ == "__main__":
    number = int(input("Enter a number to check if it's prime: "))

    # Using Naive Approach
    if is_prime_naive(number):
        print(f"{number} is a prime number (Naive Approach).")
    else:
        print(f"{number} is not a prime number (Naive Approach).")

    # Using Optimized Approach
    if is_prime_optimized(number):
        print(f"{number} is a prime number (Optimized Approach).")
    else:
        print(f"{number} is not a prime number (Optimized Approach).")
# Analysis:
# Time Complexity:
# Naive Approach: O(n) - In the worst case, we check all numbers from 2 to n-1.
# Optimized Approach: O(√n) - We only check up to the square root of n and skip even numbers after checking for 2 and 3.
# The optimized approach significantly reduces the number of iterations needed to determine if a number is prime, especially for large values of n.
# Space Complexity:
# Both approaches have a space complexity of O(1) as they use a constant amount of space.
```

Output

```
Enter a number to check if it's prime: 59
59 is a prime number (Naive Approach).
59 is a prime number (Optimized Approach).
```

Explanation

The naive prime check tests all numbers from 2 to n minus one to see if any divides the given number, which makes it slow with time complexity O(n). The optimized method improves this by checking divisibility only up to the square root of the number, skipping even numbers and multiples of three, which greatly reduces the number of checks and runs in O(square root of n) time, making it much faster for larger values.

Task Description #9 (Transparency in Recursive Algorithms)

Objective: Use AI to generate a recursive function to calculate Fibonacci numbers.

Instructions:

1. Ask AI to add clear comments explaining recursion.
2. Ask AI to explain base cases and recursive calls.

Expected Output:

- Well-commented recursive code.
- Clear explanation of how recursion works.
- Verification that explanation matches actual execution.

Code

```
#genrate well commentes code for fibonacci series
using recursion
def fibonacci(n):
    """Generate Fibonacci series up to n terms
using recursion."""
    # Base cases
    if n <= 0:
        return []
```

```
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]
    else:
        fib_series = fibonacci(n - 1)
        next_value = fib_series[-1] +
fib_series[-2]
        fib_series.append(next_value)
        return fib_series
# Example usage
if __name__ == "__main__":
    terms = int(input("Enter the number of terms
for Fibonacci series: "))
    series = fibonacci(terms)
    print(f"Fibonacci series up to {terms} terms:
{series}")
```

Output

```
Enter the number of terms for Fibonacci series: 10
Fibonacci series up to 10 terms: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Explanation

This program calculates Fibonacci numbers using recursion, where the function returns zero for input zero and one for input one, and for any larger value it calls itself to add the two previous Fibonacci numbers. The tracing version prints each function call with indentation to visually show how recursion branches into smaller subproblems, helping to understand the recursion tree and how results are combined step by step, though this approach is inefficient due to repeated calculations.

Task Description #10 (Transparency in Error Handling)

Task: Use AI to generate a Python program that reads a file and processes data.

Prompt:

"Generate code with proper error handling and clear explanations for each exception."

Expected Output:

- Code with meaningful exception handling.
- Clear comments explaining each error scenario.
- Validation that explanations align with runtime behavior.

Code

```
# Generate well-commented code to read a file and process data with error handling
def read_and_process_file(file_path):
    """Read a file and process its data with proper error handling."""
    try:
        # Attempt to open the file
        with open(file_path, 'r') as file:
            data = file.readlines()

        # Process the data (for example, converting each line to an integer)
        processed_data = []
        for line in data:
            try:
                number = int(line.strip())
                processed_data.append(number)
            except ValueError:
                # Handle the case where conversion to integer fails
                print(f"Warning: Could not convert line to integer: '{line.strip()}'")

        return processed_data

    except FileNotFoundError:
        # Handle the case where the file does not exist
```

```
        print(f"Error: The file '{file_path}' was 
not found.")
    except PermissionError:
        # Handle the case where there are 
permission issues
        print(f"Error: Permission denied when 
trying to read the file '{file_path}'.")
    except Exception as e:
        # Handle any other unexpected exceptions
        print(f"An unexpected error occurred: 
{e}")
# Example usage
if __name__ == "__main__":
    file_path = 'Assigment5/data.txt'  # Replace 
with your file path
    result = read_and_process_file(file_path)
    if result is not None:
        print("Processed Data:", result)
# Analysis:
# Time Complexity: O(n) - where n is the number of 
lines in the file,
# as we read and process each line once.
# Space Complexity: O(m) - where m is the number 
of successfully processed
# lines, as we store them in a list.
```

Output

```
Warning: Could not convert line to integer: 'Hello!'
Processed Data: []
```

Explanation

This program demonstrates robust error handling in Python by defining custom exceptions for invalid input, calculation failures, and missing data, making errors clear and meaningful. The divide function validates input types and prevents division by zero, while the user data function ensures all required fields are present, and the main block shows how different exceptions are

	raised, caught, and handled cleanly to keep the program reliable and easy to debug.	