Name:Ashfiya Jabeen

Ht.No:2303A52366

Batch-31

|  |  |  |  |
| --- | --- | --- | --- |
| **Assignment** | | | |
|  | **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.  ***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***    ***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***    ***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***    ***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***    ***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***    ***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***    ***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***    ***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. | Week3 - Monday | |  |
|  |  | **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***    ***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***    ***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.  ***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. |  | |  |