|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **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 | | | | | |
| **Name :** | | | P Akash | **Course Title** | | AI Assisted Coding | | | |
| **Year/Sem** | | | III/II | **Id Number:** | | 2303A54041 | | | |
| **Date and Day**  **of Assignment** | | | **Week4 – Friday** | **Time(s)** | | 23CSBTB01 To 23CSBTB52 | | | |
| **Duration** | | | 2 Hours | **Applicable to**  **Batches** | | All batches | | | |
| **Assignment Number: 7.5**(Present assignment number)/**24**(Total number of assignments) | | | | | | | | | |
|  | | | | | | | | | |
|  | **Q.No.** | **Question** | | | | | | ***Expected Time***  ***to complete*** |  |
|  | 1 | **Lab 7: Error Debugging with AI: Systematic approaches to finding and fixing bugs**  Lab Objectives:   * To identify and correct syntax, logic, and runtime errors in Python programs using AI tools. * To understand common programming bugs and AI-assisted debugging suggestions. * To evaluate how AI explains, detects, and fixes different types of coding errors. * To build confidence in using AI to perform structured debugging practices.   Lab Outcomes (LOs):  After completing this lab, students will be able to:   * Use AI tools to detect and correct syntax, logic, and runtime errors. * Interpret AI-suggested bug fixes and explanations. * Apply systematic debugging strategies supported by AI-generated insights.   Refactor buggy code using responsible and reliable programming patterns. | | | | | | Week4 -Monday |  |
|  |  | **Task 1 (Mutable Default Argument – Function Bug)**  Task: Analyze given code where a mutable default argument causes unexpected behavior. Use AI to fix it.  # Bug: Mutable default argument  def add\_item(item, items=[]):  items.append(item)  return items  print(add\_item(1))  print(add\_item(2))  Expected Output: Corrected function avoids shared list bug.  Prompt:  In Python, default arguments are evaluated **only once** at the time the function is defined. This is why using None as a placeholder is the "Gold Standard" fix—it forces the function to create a new object only when the function is actually executed.      Output:    Explanation:  When you use items=[], Python creates **one single list** in memory. Every time you call add\_item(), it keeps reusing that same list, "remembering" what you added last time. | | | | | |  |  |
|  |  | **Task 2 (Floating-Point Precision Error)**  Task: Analyze given code where floating-point comparison fails. Use AI to correct with tolerance.  # Bug: Floating point precision issue  def check\_sum():  return (0.1 + 0.2) == 0.3  print(check\_sum())  Expected Output: Corrected function  Prompt:  Explain why 0.1 + 0.2 doesn't equal 0.3 in Python and show me how to compare these floating-point numbers correctly using a small tolerance or 'epsilon'  Code:    Ouput: | | | | | |  |  |
|  |  | **Task 3 (Recursion Error – Missing Base Case)**  Task: Analyze given code where recursion runs infinitely due to missing base case. Use AI to fix.  # Bug: No base case  def countdown(n):  print(n)  return countdown(n-1)  countdown(5)  Expected Output : Correct recursion with stopping condition**.**  **Prompt:**  **Explain why this recursive function causes a 'RecursionError' and show me how to add a 'base case' to make it stop safely: [Insert Code]**  **Code:** | | | | | |  |  |
|  |  | Explanation:  Think of recursion like a person standing between two mirrors—the image repeats forever. In programming, "forever" isn't allowed because the computer eventually runs out of memory.  Output:    **Task 4 (Dictionary Key Error)**  Task: Analyze given code where a missing dictionary key causes error. Use AI to fix it.  # Bug: Accessing non-existing key  def get\_value():  data = {"a": 1, "b": 2}  return data["c"]  print(get\_value())  Expected Output: Corrected with .get() or error handling.  Prompt:  Show me how to safely access a dictionary key in Python so the program doesn't crash if the key is missing, and explain the difference between using .get() and a try-except block  Code: | | | | | |  |  |
|  |  | **Output:**    **Explanation:**  **A dictionary is like a real-life dictionary. If you try to look up a word that doesn't exist, Python "panics" and stops everything**  **Task 5 (Infinite Loop – Wrong Condition)**  Task: Analyze given code where loop never ends. Use AI to detect and fix it.  # Bug: Infinite loop  def loop\_example():  i = 0  while i < 5:  print(i)  Expected Output: Corrected loop increments i.  Prompt:  My 'while loop' is running forever and freezing my computer. Show me how to add a counter to make it stop after 5 rounds  Code:    Output:    Explanation:  **1. The Problem: The "Broken Timer"**  In the **Buggy Code**, you tell the computer: *"Keep printing 'i' as long as 'i' is less than 5."* \* You start with i = 0.   * But you **never change** the value of i. * Since 0 is always less than 5, the computer keeps printing 0 forever. It’s like a timer that never ticks forward. | | | | | |  |  |
|  |  | **Task 6 (Unpacking Error – Wrong Variables)**  Task: Analyze given code where tuple unpacking fails. Use AI to fix it.  # Bug: Wrong unpacking  a, b = (1, 2, 3)  Expected Output: Correct unpacking or using \_ for extra values.  Prompt:  Code:    **Output:**    **Explanation:**  **The Simple Explanation**  **Unpacking is like assigning seats; the Buggy Code crashes because there are 3 numbers but only 2 variable "chairs." The Fixed Code uses \*rest as a "catch-all bag" to safely store the extra numbers so the program doesn't break.** | | | | | |  |  |
|  |  | **Task 7 (Mixed Indentation – Tabs vs Spaces)**  **Task:** Analyze given code where mixed indentation breaks execution. Use AI to fix it.  # Bug: Mixed indentation  def func():  x = 5  y = 10  return x+y  Expected Output : Consistent indentation applied.  Prompt:  My Python code is crashing with an **IndentationError**; can you fix the alignment and show me how to use consistent **4-space indentation** for all blocks?" | | | | | |  |  |
|  |  | **Output:**    **Explanation:**  **Python uses empty space (indentation) to understand which lines of code belong together; the Buggy Code crashes because the lines are unevenly aligned or mixed. The Fixed Code ensures every line inside a function or loop starts at the exact same level, usually by using 4 spaces for each new block**  **Task 8 (Import Error – Wrong Module Usage)**  Task: Analyze given code with incorrect import. Use AI to fix.  # Bug: Wrong import  import maths  print(maths.sqrt(16))  Expected Output: Corrected to import math  Prompt:  My Python code crashes with a **ModuleNotFoundError** because I misspelled an import name; show me the correct name for the math module and the different ways to import its functions  Code:    **Output:**  **E**  **e** | | | | | |  |  |
|  |  | **ExplanationL**  **Python is very strict about names; the Buggy Code fails because it tries to import maths (with an 's') instead of the correct name, math. The Fixed Code shows that once you use the right name, you can either import the whole toolbox, pick specific tools like sqrt, or give the module a short nickname like m to save time**  **Task 9 (Unreachable Code – Return Inside Loop)**  **Task:** Analyze given code where a return inside a loop prevents full iteration. Use AI to fix it.  # Bug: Early return inside loop  def total(numbers):  for n in numbers:  return n  print(total([1,2,3]))  **Expected Output:** Corrected code accumulates sum and returns after loop.  **Code:**    **Output:**    **Prompt:**  **"My Python function is stopping after the very first item in a list instead of finishing the loop; show me how to move the return statement outside the loop so it processes everything.**  **Explamnation:**  **A return statement is like an "Exit" sign; as soon as the computer hits it, it leaves the function immediately. In the Buggy Code, the return is inside the loop, so it exits on the first item. The Fixed Code moves the return to the very end, allowing the loop to finish its job before handing back the final result.** | | | | | |  |  |
|  |  | **Task 10 (Name Error – Undefined Variable)**  Task: Analyze given code where a variable is used before being defined. Let AI detect and fix the error.  # Bug: Using undefined variable  def calculate\_area():  return length \* width  print(calculate\_area())  Requirements:   * Run the code to observe the error. * Ask AI to identify the missing variable definition. * Fix the bug by defining length and width as parameters. * Add 3 assert test cases for correctness.   Expected Output :   * Corrected code with parameters. * AI explanation of the bug.   Code:    Output:    Prompt:  **Task:** Analyze a function designed to calculate the area of a rectangle. The original code attempts to use variables length and width inside the function without defining them or passing them as arguments.  Explanation:  **What was the problem?** In the buggy version, the function calculate\_area() was like a calculator trying to solve a problem without any numbers. It asked for length and width, but those names didn't exist in the computer's memory yet. In Python, this triggers a NameError because the program searching for those "names" comes up empty-handed.  Successful execution of assertions. | | | | | |  |  |
|  |  | **Task 11 (Type Error – Mixing Data Types Incorrectly)**  Task: Analyze given code where integers and strings are added incorrectly. Let AI detect and fix the error.  # Bug: Adding integer and string  def add\_values():  return 5 + "10"  print(add\_values())  Requirements:   * Run the code to observe the error. * AI should explain why int + str is invalid. * Fix the code by type conversion (e.g., int("10") or str(5)). * Verify with 3 assert cases.   Expected Output #6:   * Corrected code with type handling. * AI explanation of the fix.   Successful test validation.  Code:    Explanation:  **What was the problem?** In the buggy code, you were trying to do 5 + "10". Python is a "strongly typed" language, which means it won't guess what you want to do. It sees a **number** (5) and a **piece of text** ("10") and gets confused: *"Do you want to do math, or do you want to join these words together?"* Instead of guessing, it throws a TypeError.  Prompt:  **Task:** Analyze code that attempts to perform an addition between an integer and a string, which normally causes a TypeError  Output: | | | | | |  |  |
|  |  | **Task 12 (Type Error – String + List Concatenation)**  Task: Analyze code where a string is incorrectly added to a list.  # Bug: Adding string and list  def combine():  return "Numbers: " + [1, 2, 3]  print(combine())  Requirements:   * Run the code to observe the error. * Explain why str + list is invalid. * Fix using conversion (str([1,2,3]) or " ".join()). * Verify with 3 assert cases.   Expected Output:   * Corrected code * Explanation * Successful test validation       Prompt:  Write a function to combine a string label with a list of data. Fix the error where Python prevents adding a string to a list by either converting the list to a string or using .join() to format the elements."  Explanation:   * **The Problem:** Python cannot use the + operator to join a string and a list directly; it results in a TypeError. * **The Fix:** We convert the list to a string using str(lst) or use .join() to merge the individual elements into a single piece of text.   **Task 13 (Type Error – Multiplying String by Float)**  Task: Detect and fix code where a string is multiplied by a float.  # Bug: Multiplying string by float  def repeat\_text():  return "Hello" \* 2.5  print(repeat\_text())  Requirements:   * Observe the error. * Explain why float multiplication is invalid for strings. * Fix by converting float to int. * Add 3 assert test cases.       Prompt:  "Create a function that repeats a string a specific number of times. Fix the bug where passing a decimal (float) causes a failure by explicitly converting the multiplier to an integer."  E  **Task 14 (Type Error – Adding None to Integer)**  Task: Analyze code where None is added to an integer.  # Bug: Adding None and integer  def compute():  value = None  return value + 10  print(compute())  Requirements:   * Run and identify the error. * Explain why NoneType cannot be added. * Fix by assigning a default value. * Validate using asserts.       **Task 15 (Type Error – Input Treated as String Instead of Number)**  Task: Fix code where user input is not converted properly.  # Bug: Input remains string  def sum\_two\_numbers():  a = input("Enter first number: ")  b = input("Enter second number: ")  return a + b  print(sum\_two\_numbers())  Requirements:   * Explain why input is always string. * Fix using int() conversion. * Verify with assert test cases.       Prompt:  Write a function to sum two values that might be received as strings (like from a user input field). Ensure the function converts both inputs to integers before performing the addition  Explaantion:   **The Problem:** If you add "2" + "3" as strings, Python gives you "23" (text joining) instead of 5 (math).   **The Fix:** We wrap the inputs in int() to turn the text into real numbers so that the + operator performs actual addition. | | | | | |  |  |