

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	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: <ul style="list-style-type: none"> To identify and correct syntax, logic, and runtime errors in 	Week4 - Monday	

	<p>Python programs using AI tools.</p> <ul style="list-style-type: none"> • 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. <p>Lab Outcomes (LOs):</p> <p>After completing this lab, students will be able to:</p> <ul style="list-style-type: none"> • 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. <p>Refactor buggy code using responsible and reliable programming patterns.</p>	
	<p>Task 1 (Mutable Default Argument – Function Bug)</p> <p>Task: Analyze given code where a mutable default argument causes unexpected behavior. Use AI to fix it.</p> <pre># Bug: Mutable default argument def add_item(item, items=[]): items.append(item) return items print(add_item(1)) print(add_item(2))</pre> <p>Expected Output: Corrected function avoids shared list bug.</p> <p>Prompt:</p> <p>"Analyze this Python function for bugs. It uses a mutable default argument, which causes unexpected behavior. Explain the issue, why it happens, and provide a corrected version that avoids the shared list bug."</p>	

Code

```
day3.py > corrected_function
1 #generate a python code forAnalyze this Python function for bugs. It uses a mutable default argument, which causes
2 def buggy_function(items=[]):
3     items.append("new_item")
4     return items
5
6 # The issue: The default argument 'items=[]' is a mutable object. When the function is called multiple times without
7 # This leads to unexpected behavior where modifications to the list persist between function calls.
8
9 # Corrected version:
10 def corrected_function(items=None):
11     if items is None:
12         items = []
13     items.append("new_item")
14     return items
```

Code Output

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\konda\Downloads\AI Assistant> & C:/Users/konda/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/konda/
stant/day3.py"
PS C:\Users\konda\Downloads\AI Assistant> 5
5
```

Explanation:

The bug is a classic mutable default argument issue. In Python, default arguments are evaluated once when the function is defined, not each time it's called. So `items=[]` creates a single shared list across all calls. When you call `add_item(1)`, it appends to this shared list, returning `[1]`. The second call `add_item(2)` appends to the same list, returning `[1, 2]` instead of just `[2]`.

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

Fix the floating-point comparison error using a tolerance value and explain why direct comparison fails.

Code

```
# generate a pthon code Analyze given code where floating-point comparison fails
def analyze_floating_point():
    a = 0.1
    b = 0.2
    c = 0.3

    print(f"a = {a}")
    print(f"b = {b}")
    print(f"c = {c}")
    print(f"a + b = {a + b}")
    print(f"a + b == c: {a + b == c}")
    print(f"abs((a + b) - c) = {abs((a + b) - c)}")

Click to add a breakpoint
14 analyze_floating_point()
```

Code Output

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python
PS C:\Users\konda\Downloads\AI Assistant> & c:/Users/konda/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/konda/Downloads/AI Assistant/analyze_floating_point.py"
a = 0.1
b = 0.2
c = 0.3
a + b = 0.30000000000000004
a + b == c: False
abs((a + b) - c) = 5.551115123125783e-17
PS C:\Users\konda\Downloads\AI Assistant>
```

Explanation

Floating-point numbers are stored approximately in memory. Using a tolerance (epsilon) checks whether values are *close enough*, avoiding precision errors.

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

Identify the recursion error caused by a missing base case and fix the code.

Code

```
day3.py > ...
1 #GENERATE A PYTHON CODE FOR Analyze given code where recursion runs infinitely due to missing base case. Use AI to fix.
2 def factorial(n):
3     # Base case: if n is 0 or 1, return 1
4     if n == 0 or n == 1:
5         return 1
6     # Recursive case: n! = n * (n-1)!
7     return n * factorial(n - 1)
8 # Example usage
9 print(factorial(5)) # Output: 120
```

Code Output

```
PS
120 Focus folder in explorer (ctrl + click) instant> & C:/Users/konda/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/konda/Downloads/AI Ass
```

Explanation

Without a base case, recursion runs infinitely.
The condition if $n < 0$: return stops the recursion safely.

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

Fix the dictionary key error using safe access methods and explain the solution.

Code

```
days.py 7 ...
1  def get_value():
2      data = {"a": 1, "b": 2}
3      return data.get("c", "Key not found")
4
5  print(get_value())
6  |
```

Code Output

```
0
PS C:\Users\konda\Downloads\AI Assistant>
Key not found
PS C:\Users\konda\Downloads\AI Assistant>
```

Explanation

Accessing a missing key using `data["c"]` raises a `KeyError`. Using `.get()` prevents crashes and allows a default value.

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

Detect and fix the infinite loop caused by a missing increment statement.

Code

```
day3.py > ...
1  def loop_example():
2      i = 0
3      while i < 5:
4          print(i)
5          i += 1
6
7  loop_example()
8
```

Code Output

```
PS C:\Users\konda\Downloads\AI Assis
0
1
2
3
4
PS C:\Users\konda\Downloads\AI Assis
```

Explanation

The loop never ended because i was not updated.
Incrementing i ensures the loop eventually terminates.

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.

	<p>Prompt</p> <p>Fix the tuple unpacking error by adjusting variables correctly.</p> <p>Code</p> <pre> y3.py > ... #generate a python code for Fix the tuple unpacking error by adjusting vari data = ("Alice", 30, "Engineer") name, age, profession = data print(f"Name: {name}, Age: {age}, Profession: {profession}") # The tuple unpacking is already correct in this code. # If there was an error, it might have been due to incorrect number of vari </pre> <p>Code Output</p> <pre> ● PS C:\Users\konda\Downloads\AI Assistant> & C:/Users Name: Alice, Age: 30, Profession: Engineer ○ PS C:\Users\konda\Downloads\AI Assistant> </pre> <p>Explanation</p> <p>Tuple unpacking requires the number of variables to match the number of values.</p> <p>Adding the correct number of variables fixes the error.</p>	
	<p>Task 7 (Mixed Indentation – Tabs vs Spaces)</p> <p>Task: Analyze given code where mixed indentation breaks execution. Use AI to fix it.</p> <p># Bug: Mixed indentation</p> <pre> def func(): x = 5 y = 10 return x+y </pre> <p>Expected Output : Consistent indentation applied.</p> <p>Prompt</p> <p>Identify and fix the indentation error caused by inconsistent spacing.</p>	

Code

```
day3.py x
day3.py > ...
1 #generate a python code for Identify and fix the indentation error caused by inconsistent
2 def example_function():
3     print("This line is correctly indented with 4 spaces.")
4     if True:
5         print("This line is also correctly indented with 4 spaces.")
6     else:
7         print("This line is also correctly indented with 4 spaces.")
8     print("This line is correctly indented with 4 spaces.")
9 example_function()
```

Code Output

```
PS C:\Users\konda\Downloads\AI Assistant> & C:/Users/konda/AppData/Local/Programs/Python/Python39-64/Python.exe day3.py
This line is correctly indented with 4 spaces.
This line is also correctly indented with 4 spaces.
This line is correctly indented with 4 spaces.
PS C:\Users\konda\Downloads\AI Assistant>
```

Explanation

Python relies on consistent indentation.

Mixing tabs and spaces causes IndentationError.

Using uniform spaces fixes the issue.

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

Fix the import error by using the correct Python module name.

Code

```
day3.py
1  import math
2
3  print(math.sqrt(16))
4  print(math.sqrt(25))
5  print(math.sqrt(2))
6
7  (function) def print(
8      *values: object,
9      sep: str | None = " ",
10     end: str | None = "\n",
11     file: SupportsWrite[str] | None = None,
12     flush: Literal[False] = False
13 ) -> None
```

Code Output

```
4.0
5.0
1.4142135623730951
1.7320508075688772
2.0
2.23606797749979
2.449489742783178
2.6457513110645907
2.8284271247461903
3.0
3.1622776601683795
PS C:\Users\konda\Downloads\AI Assistant> █
```

Explanation

The module name is math, not maths.

Correct imports prevent ModuleNotFoundError.