# M.AKASH

# 2303A51820 B26

| 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<br>Ms. B. Swathi<br>Dr. Sasanko Shekhar Gantayat<br>Mr. Md Sallauddin<br>Dr. Mathivanan<br>Mr. Y Srikanth<br>Ms. N Shilpa<br>Dr. Rishabh Mittal (Coordinator)<br>Dr. R. Prashant Kumar<br>Mr. Ankushavali MD<br>Mr. B Viswanath<br>Ms. Sujitha Reddy<br>Ms. A. Anitha<br>Ms. M.Madhuri<br>Ms. Katherashala Swetha<br>Ms. Velpula sumalatha<br>Mr. Bingi Raju | | |
| **Course Code** | 23CS002PC304 | **Course Title** | AI Assisted Coding |
| **Year/Sem** | III/II | **Regulation** | R23 |
| **Date and Day of Assignment** | **Week4 – Wednesday** | **Time(s)** | 23CSBTB01 To 23CSBTB52 |
| **Duration** | 2 Hours | **Applicable to Batches** | All batches |
| **AssignmentNumber:7.3**(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** | Week4 - Wednesday |

# M.AKASH

# 2303A51820 B26

**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 for 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 using AI-generated insights
• Refactor buggy code using reliable programming patterns

**Task 1: Fixing Syntax Errors**

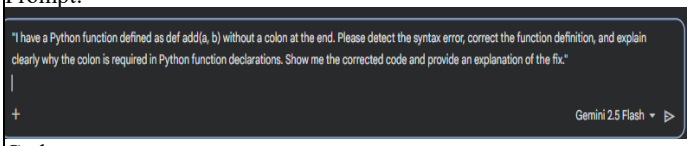
**Scenario**
You are reviewing a Python program where a basic function definition contains a syntax error.

```
python

def add(a, b)
    return a + b
```

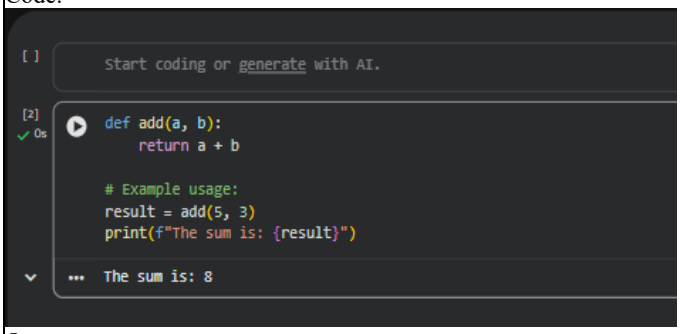
**Requirements**
• Provide a Python function add(a, b) with a **missing colon**
• Use an AI tool to detect the syntax error
• Allow AI to correct the function definition
• Observe how AI explains the syntax issue

**Expected Output**
• Corrected function with proper syntax
• Syntax error resolved successfully
• AI-generated explanation of the fix

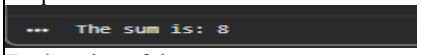
Prompt:




Code:




Output:




Explanation of the eerror:
In Python, a colon (:) is used to mark the end of a statement that introduces a new *code block*.

This includes def statements for function definitions, if/elif/else statements, for and while loops, and with statements, among others.
When you define a function using def, the colon tells the Python interpreter that the code that follows, typically on the next line and indented, belongs to the function's body. Without the colon, Python doesn't know where the function header ends and where its body begins, leading to a SyntaxError.

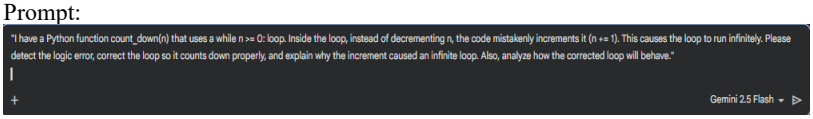
**Task 2: Debugging Logic Errors in Loops**

**Scenario**
You are debugging a loop that runs infinitely due to a logical mistake.

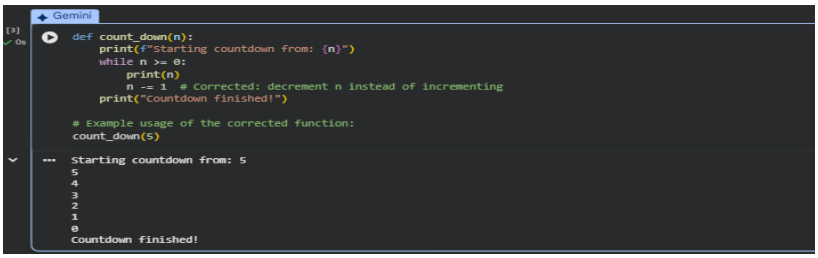
```python
def count_down(n):
    while n >= 0:
        print(n)
        n += 1  # Should be n -= 1
```

**Requirements**
• Provide a loop with an **increment or decrement error**
• Use AI to identify the cause of infinite iteration
• Let AI fix the loop logic
• Analyze the corrected loop behavior

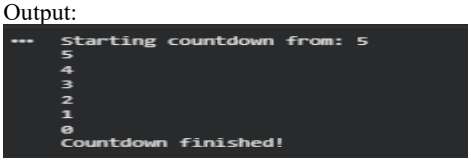
**Expected Output**
• Infinite loop issue resolved
• Correct increment/decrement logic applied
• AI explanation of the logic error
Prompt:




Code:




Output:




Explanation of the error in the code:
**The Logic Error (Infinite Loop):**
The original code n += 1 within the while n >= 0: loop created an infinite loop because the condition n >= 0 would *always* remain true (assuming n started as non-negative). Each iteration would increase n, moving it further away from becoming less than 0, thus preventing the loop from ever terminating.
**The Correction:**

The fix involves changing n += 1 to n -= 1. This decrements the value of n in each iteration, moving it closer to fulfilling the loop termination condition (n < 0).

**Behavior of the Corrected Loop:**

With n -= 1, the loop will behave as follows:

1. **Initialization:** The function starts with n set to its initial value (e.g., 5).
2. **Condition Check:** The while loop checks if n >= 0. If true, the loop body executes.
3. **Action:** The current value of n is printed.
4. **Decrement:** n is then reduced by 1 (n = n - 1).
5. **Iteration:** Steps 2-4 repeat.
6. **Termination:** This process continues until n becomes less than 0. At that point, the condition n >= 0 becomes false, and the loop terminates. The final print statement outside the loop ("Countdown finished!") is then executed.

**Task 3: Handling Runtime Errors (Division by Zero)**

**Scenario**

A Python function crashes during execution due to a division by zero error.

```
# Debug the following code
def divide(a, b):
    return a / b

print(divide(10, 0))
```

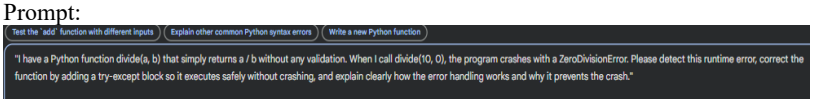
**Requirements**

• Provide a function that performs division without validation
• Use AI to identify the runtime error
• Let AI add try-except blocks for safe execution
• Review AI's error-handling approach

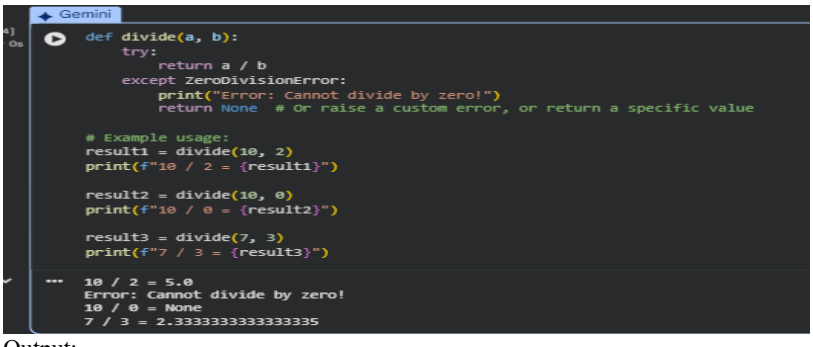
**Expected Output**

• Function executes safely without crashing
• Division by zero handled using try-except
• Clear AI-generated explanation of runtime error handling

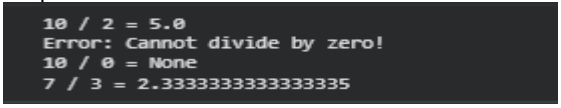
Prompt:




Code:




Output:




Explanation of the error in the code:

**The Runtime Error (ZeroDivisionError):**

When you call divide(10, 0), the Python interpreter attempts to perform the division 10 / 0. Mathematically, division by zero is undefined, and in programming, it leads to a ZeroDivisionError, which is a type of runtime error that causes your program to crash if not handled.

**The Correction (try-except block):**

We fix this by wrapping the potentially problematic code (a / b) within a try-except block:

- **try block:** This block contains the code that might raise an exception (in this case, a / b). Python will attempt to execute this code.
- **except ZeroDivisionError: block:** If a ZeroDivisionError occurs *specifically* within the try block, Python immediately stops executing the try block and jumps directly to the code inside this except block. Here, we print an informative error message and return None (a common practice to indicate that an operation could not complete successfully). If no ZeroDivisionError occurs, the except block is skipped entirely.

**Task 4: Debugging Class Definition Errors**

**Scenario**

You are given a faulty Python class where the constructor is incorrectly defined.

```python
class Rectangle:
    def __init__(length, width):
        self.length = length
        self.width = width
```

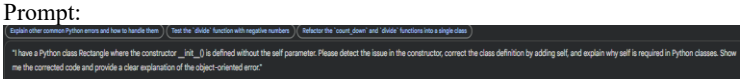
**Requirements**

• Provide a class definition with **missing self-parameter**
• Use AI to identify the issue in the __init__() method
• Allow AI to correct the class definition
• Understand why self is required

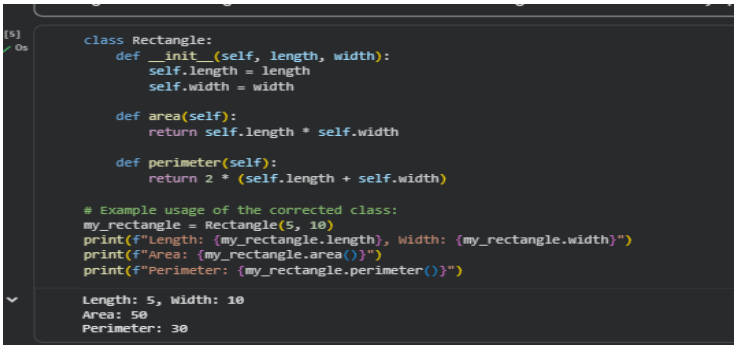
**Expected Output**

• Corrected __init__() method
• Proper use of self in class definition
• AI explanation of object-oriented error

Prompt:

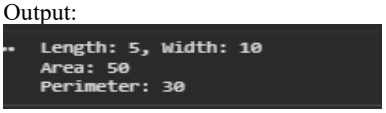



Code:



```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)

# Example usage of the corrected class:
my_rectangle = Rectangle(5, 10)
print(f"Length: {my_rectangle.length}, Width: {my_rectangle.width}")
print(f"Area: {my_rectangle.area()}")
print(f"Perimeter: {my_rectangle.perimeter()}")

Length: 5, Width: 10
Area: 50
Perimeter: 30
```

Output:



```
Length: 5, Width: 10
Area: 50
Perimeter: 30
```

Explanation of the error in the code:

**The Object-Oriented Error (Missing self):**
In Python, when you define a method within a class, whether it's the constructor (__init__) or any other instance method, the first parameter *must* be self. This self parameter is a convention (though not a keyword) that refers to the instance of the class itself. When you call a method on an object (e.g., my_object.method()), Python automatically passes the object instance as the first argument to that method. If the method definition doesn't include self, Python doesn't know where to pass this instance, leading to a TypeError indicating too many arguments were given (as the instance is passed implicitly).
For __init__, specifically, its job is to initialize the attributes of the newly created object. Without self, you have no way to access or assign these attributes to the specific object being created.
**The Correction:**
The fix involves adding self as the first parameter in the __init__ method definition: def __init__(self, length, width):
Inside the method, self.length = length and self.width = width then correctly assign the passed length and width values to the length and width attributes *of the specific Rectangle object* (self) that is being initialized.
**Why self is required:**

- **Instance Reference:** self provides a way for the method to refer to the specific instance of the object on which the method is being called. Without self, a method wouldn't know which object's data it should operate on.
- **Attribute Access:** All instance variables (attributes) of a class are accessed using the self keyword (e.g., self.attribute_name). This ensures that each object instance maintains its own set of data.
- **Method Calling:** Similarly, if one instance method needs to call another instance method within the same class, it does so using self.other_method().

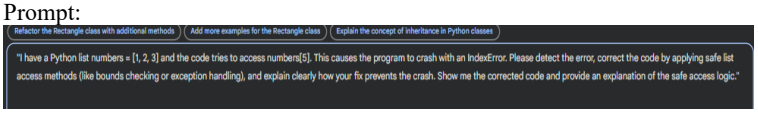
**Task 5: Resolving Index Errors in Lists**

**Scenario**
A program crashes when accessing an invalid index in a list.

```
python

numbers = [1, 2, 3]
print(numbers[5])
```

**Requirements**
• Provide code that accesses an **out-of-range list index**
• Use AI to identify the Index Error
• Let AI suggest safe access methods
• Apply bounds checking or exception handling

**Expected Output**
• Index error resolved
• Safe list access logic implemented
• AI suggestion using length checks or exception handling
Prompt:




Code:

# M.AKASH

# 2303A51820 B26

```
numbers = [1, 2, 3]

try:
    # Attempt to access an element
    index_to_access = 5
    value = numbers[index_to_access]
    print(f"Accessed value at index {index_to_access}: {value}")
except IndexError:
    # Handle the error if the index is out of bounds
    print(f"Error: Index {index_to_access} is out of bounds for the list.")
    print(f"The list has {len(numbers)} elements, with valid indices from 0 to {len(numbers) - 1}.")

# Example of valid access
try:
    valid_index = 1
    value = numbers[valid_index]
    print(f"Accessed value at index {valid_index}: {value}")
except IndexError:
    print(f"Error: Index {valid_index} is out of bounds for the list.")
```

```
Error: Index 5 is out of bounds for the list.
The list has 3 elements, with valid indices from 0 to 2.
Accessed value at index 1: 2
```

Output:

```
Error: Index 5 is out of bounds for the list.
The list has 3 elements, with valid indices from 0 to 2.
Accessed value at index 1: 2
```

Explanation of the error in the code:

**The Runtime Error (IndexError):**

When you define numbers = [1, 2, 3], this list has three elements. In Python, list indices start from 0. So, the valid indices for this list are 0, 1, and 2. Attempting to access numbers[5] (or any index 3 or greater) leads to an IndexError because there is no element at that position in the list. Without error handling, this error would terminate your program.

**The Correction (try-except block):**

Similar to handling ZeroDivisionError, we use a try-except block to manage IndexError:

- **try block:** This block contains the code that might raise an exception. Here, value = numbers[index_to_access] is placed inside try because it's the operation that could potentially cause an IndexError.
- **except IndexError: block:** If an IndexError occurs within the try block, Python immediately stops executing the try block and transfers control to the except IndexError block. Inside this block, we provide a user-friendly message explaining that the index is out of bounds. This prevents the program from crashing.

**How it prevents the crash:**

By catching the IndexError, your program doesn't abruptly stop. Instead, it executes the code within the except block, allowing you to inform the user about the issue or take alternative actions, and then continue with the rest of your program's execution. This makes your code more robust and user-friendly by gracefully handling unexpected situations during list access.

**Note: Report should be submitted a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots**