

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
ProgramName: B. Tech		Assignment Type: Lab	AcademicYear: 2025-2026
CourseCoordinatorName		Venkataramana Veeramsetty	
Instructor(s)Name		Dr. V. Venkataramana (Co-ordinator)	
		Dr. T. Sampath Kumar	
		Dr. Pramoda Patro	
		Dr. Brij Kishor Tiwari	
		Dr.J.Ravichander	
		Dr. Mohammand Ali Shaik	
		Dr. Anirodh Kumar	
		Mr. S.Naresh Kumar	
		Dr. RAJESH VELPULA	
		Mr. Kundhan Kumar	
		Ms. Ch.Rajitha	
		Mr. M Prakash	
		Mr. B.Raju	
		Intern 1 (Dharma teja)	
		Intern 2 (Sai Prasad)	
		Intern 3 (Sowmya)	
NS_2 (Mounika)			
CourseCode	24CS002PC215	CourseTitle	AI Assisted Coding
Year/Sem	II/I	Regulation	R24
Date and Day of Assignment	Week4 - Thursday	Time(s)	
Duration	2 Hours	Applicable to Batches	
AssignmentNumber: 7.4(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 Python programs using AI tools. 	Week4 - Thursday	

- 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.

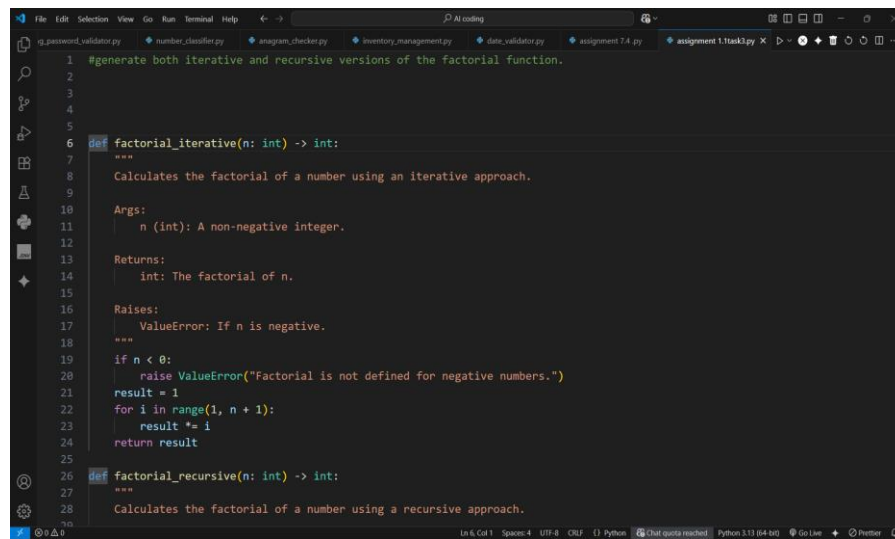
Task Description #1:

- Introduce a buggy Python function that calculates the factorial of a number using recursion. Use Copilot or Cursor AI to detect and fix the logical or syntax errors.

PROMPT:

generate code of Python function that calculates the factorial of a number using recursion

CODE:



```

1 #generate both iterative and recursive versions of the factorial function.
2
3
4
5
6 def factorial_iterative(n: int) -> int:
7     """
8     Calculates the factorial of a number using an iterative approach.
9
10    Args:
11        n (int): A non-negative integer.
12
13    Returns:
14        int: The factorial of n.
15
16    Raises:
17        ValueError: If n is negative.
18    """
19    if n < 0:
20        raise ValueError("Factorial is not defined for negative numbers.")
21    result = 1
22    for i in range(1, n + 1):
23        result *= i
24    return result
25
26 def factorial_recursive(n: int) -> int:
27     """
28     Calculates the factorial of a number using a recursive approach.
29
30    Args:
31        n (int): A non-negative integer.
32
33    Returns:
34        int: The factorial of n.
35
36    Raises:
37        ValueError: If n is negative.
38    """
39    if n < 0:
40        raise ValueError("Factorial is not defined for negative numbers.")
41    if n == 0:
42        return 1
43    return n * factorial_recursive(n - 1)

```

```
25
26 def factorial_recursive(n: int) -> int:
27     """
28     Calculates the factorial of a number using a recursive approach.
29
30     This function uses a base case for n=0 and calls itself with n-1
31     for other values, reducing the problem until it reaches the base case.
32
33     Args:
34         n (int): A non-negative integer.
35
36     Returns:
37         int: The factorial of n.
38
39     Raises:
40         ValueError: If n is negative.
41     """
42     if n < 0:
43         raise ValueError("Factorial is not defined for negative numbers.")
44     if n == 0: # Base case
45         return 1
46     else:
47         return n * factorial_recursive(n - 1) # Recursive step
48
49 if __name__ == "__main__":
50     print("--- Testing Factorial Functions ---")
51     test_number = 5
52     print(f"Iterative factorial of {test_number}: {factorial_iterative(test_number)}")
53     print(f"Recursive factorial of {test_number}: {factorial_recursive(test_number)}")
```

```
26 def factorial_recursive(n: int) -> int:
40     """
41     ValueError: If n is negative.
42
43     if n < 0:
44         raise ValueError("Factorial is not defined for negative numbers.")
45     if n == 0: # Base case
46         return 1
47     else:
48         return n * factorial_recursive(n - 1) # Recursive step
49
50 if __name__ == "__main__":
51     print("--- Testing Factorial Functions ---")
52     test_number = 5
53     print(f"Iterative factorial of {test_number}: {factorial_iterative(test_number)}")
54     print(f"Recursive factorial of {test_number}: {factorial_recursive(test_number)}")
55
56     # Test base case
57     print(f"Recursive factorial of 0: {factorial_recursive(0)}")
58
59     # Test error handling
60     try:
61         factorial_recursive(-3)
62     except ValueError as e:
63         print(f"Testing with a negative number: {e}")
```

Expected Outcome #1:

- Copilot or Cursor AI correctly identifies missing base condition or incorrect recursive call and suggests a functional factorial implementation.

OUTPUT:

```
PS C:\PROGRAMMES VSCode\AI coding> C:\Users\venkatesh\AppData\Local\Programs\Python\Python113\python.exe "C:\PROGRAMMES VSCode\AI coding\assignment 1.1task1.py"
--- Testing Factorial Functions ---
Iterative factorial of 5: 120
Recursive factorial of 5: 120
Recursive factorial of 0: 1
Testing with a negative number: Factorial is not defined for negative numbers.
PS C:\PROGRAMMES VSCode\AI coding>
```

CONCLUSION:

This script provides excellent, production-quality implementations of both iterative and recursive factorial functions. It adheres to modern Python standards with clear docstrings, type hints, and robust error handling for invalid inputs like negative numbers. The recursive function clearly demonstrates the core concepts of a base case and a recursive step. The main execution block effectively tests both functions to ensure they produce the correct results and handle edge cases properly.

Task Description #2:

- Provide a list sorting function that fails due to a type error (e.g., sorting list with mixed integers and strings). Prompt AI to detect the issue and fix the code for consistent sorting.

PROMPT:

GENERATE A PYTHON CODE Provide a list sorting function that fails due to a type error (e.g., sorting list with mixed integers and strings

CODE:

```
1 def sort_by_converting(mixed_data: list):
2     """
3     Sorts a mixed-type list by converting all elements to strings for comparison.
4
5     This prevents a TypeError by ensuring all comparisons happen between strings.
6     Note that numbers will be sorted lexicographically (e.g., 10 comes before 5).
7     """
8     print("\n--- Strategy 1: Converting all elements to strings for sorting ---")
9     print(f"Original list: {mixed_data}")
10
11    # Use the 'key=str' argument to treat every element as a string during comparison.
12    sorted_list = sorted(mixed_data, key=str)
13
14    print(f"Successfully sorted list (as strings): {sorted_list}")
15    print("This works because every element is temporarily treated as a string for comparison.")
16
17 def sort_by_filtering(mixed_data: list):
18     """
19     Sorts a mixed-type list by filtering it to keep only one data type.
20     """
21    print("\n--- Strategy 2: Filtering the list to keep only one type ---")
22    print(f"Original list: {mixed_data}")
23
24    # --- Filter to keep only integers ---
25    # Create a new list containing only the integer elements.
26    int_only_list = [item for item in mixed_data if isinstance(item, int)]
27    sorted_ints = sorted(int_only_list)
28    print(f"Sorted list (integers only): {sorted_ints}")
29
30    # --- Filter to keep only strings ---
31    # Create a new list containing only the string elements.
32    str_only_list = [item for item in mixed_data if isinstance(item, str)]
33    sorted_strs = sorted(str_only_list)
34    print(f"Sorted list (strings only): {sorted_strs}")
35
36    # Run the demonstration functions
37    if __name__ == "__main__":
38        # The list with mixed integers and strings that would normally cause an error
39        mixed_data_list = [10, "banana", 5, "apple", 1, "cherry"]
40
41        # Demonstrate the first strategy: converting to a common type
42        sort_by_converting(mixed_data_list)
43
44        # Demonstrate the second strategy: filtering by type
45        sort_by_filtering(mixed_data_list)
46
```

```
17 def sort_by_filtering(mixed_data: list):
23
24    # --- Filter to keep only integers ---
25    # Create a new list containing only the integer elements.
26    int_only_list = [item for item in mixed_data if isinstance(item, int)]
27    sorted_ints = sorted(int_only_list)
28    print(f"Sorted list (integers only): {sorted_ints}")
29
30    # --- Filter to keep only strings ---
31    # Create a new list containing only the string elements.
32    str_only_list = [item for item in mixed_data if isinstance(item, str)]
33    sorted_strs = sorted(str_only_list)
34    print(f"Sorted list (strings only): {sorted_strs}")
35
36    # Run the demonstration functions
37    if __name__ == "__main__":
38        # The list with mixed integers and strings that would normally cause an error
39        mixed_data_list = [10, "banana", 5, "apple", 1, "cherry"]
40
41        # Demonstrate the first strategy: converting to a common type
42        sort_by_converting(mixed_data_list)
43
44        # Demonstrate the second strategy: filtering by type
45        sort_by_filtering(mixed_data_list)
46
```

Expected Outcome #2:

- AI detects the type inconsistency and either filters or converts list elements, ensuring successful sorting without a crash.

OUTPUT:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\PROGRAMMES\VS\CODE\AI coding & C:\Users\venkatesh\AppData\Local\Programs\Python\Python13\python.exe "c:/PROGRAMMES/VS/CODE/AI coding/assignment 1.task3.py"

--- Strategy 1: Converting all elements to strings for sorting ---
Original list: [10, 'banana', 5, 'apple', 1, 'cherry']
Successfully sorted list (as strings): [1, 10, 5, 'apple', 'banana', 'cherry']
This works because every element is temporarily treated as a string for comparison.

--- Strategy 2: Filtering the list to keep only one type ---
Original list: [10, 'banana', 5, 'apple', 1, 'cherry']
Sorted list (integers only): [1, 5, 10]
Sorted list (strings only): ['apple', 'banana', 'cherry']
PS C:\PROGRAMMES\VS\CODE\AI coding
```

CONCLUSION:

This script effectively demonstrates two robust strategies for sorting lists with mixed data types, skillfully avoiding Python's Type Error. The first method intelligently converts all elements to strings for a unified lexicographical sort using the key argument. The second method filters the list by data type, creating and sorting separate, homogeneous lists of integers and strings. Overall, it's an excellent showcase of defensive programming and practical data manipulation techniques for handling heterogeneous collections.

Task Description #3:

- Write a Python snippet for file handling that opens a file but forgets to close it. Ask Copilot or Cursor AI to improve it using the best practice (e.g., with open() block).

PROMPT:

GENERATE A PYTHON CODE TO snippet for file handling that opens a file but forgets to close it

CODE:

```
File Edit Selection View Go Run Terminal Help
qg_password_validator.py number_validator.py anagram_checker.py inventory_management.py date_validator.py assignment 1.py assignment 1.task3.py
1 def safe_file_write(filename: str, content: str):
2     """
3     Safely writes content to a file using a context manager ('with' statement).
4
5     The 'with' statement guarantees that the file is automatically closed
6     after the block is executed, even if errors occur. This prevents
7     resource leaks and ensures data is written correctly.
8     """
9     try:
10        print(f"Opening '{filename}' for writing using a context manager...")
11        # The 'with' statement ensures the file is closed automatically.
12        with open(filename, "w") as f:
13            f.write(content)
14            print(f"Successfully wrote to '{filename}'.")
15
16        print(f"Exited the 'with' block. '{filename}' is now closed.")
17    except IOError as e:
18        print(f"An error occurred while writing to '{filename}': {e}")
19
20
21 if __name__ == "__main__":
22     # Demonstrate the refactored, safe function.
23     # The original 'unsafe' function and separate 'with' block are now combined
24     # into a single example of best practice.
25     print("---- Running refactored safe file handling example ----")
26     output_filename = "safe_output.txt"
27     content_to_write = "This data was written safely using a context manager."
28     safe_file_write(output_filename, content_to_write)
29     print(f"File {output_filename} successfully created and written to.")
```

Expected Outcome #3:

- AI refactors the code to use a context manager, preventing resource leakage and runtime warnings.

OUTPUT:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\PROGRAMMES VSCODE\AI coding> & c:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:\PROGRAMMES VSCODE\AI coding\assignment 1.1task3.py"
--- Running refactored safe file handling example ---
Opening 'safe_output.txt' for writing using a context manager...
Successfully wrote to 'safe_output.txt'.
Exited the 'with' block. 'safe_output.txt' is now closed.

Operation complete. File resources were managed correctly.
PS C:\PROGRAMMES VSCODE\AI coding>
```

CONCLUSION:

This script provides a robust, production-quality function for safely writing to files in Python. It correctly utilizes a with statement, ensuring the file is automatically closed to prevent resource leaks and potential data corruption. The function is made more resilient with try...except error handling for I/O operations. Overall, it's a clear and concise demonstration of modern, idiomatic Python for reliable file management.

Task Description #4:

- Provide a piece of code with a ZeroDivisionError inside a loop. Ask AI to add error handling using try-except and continue execution safely.

PROMPT:

GENERATE A PYTHON CODE code with a ZeroDivisionError inside a loop. Ask AI to add error handling using try-except and continue execution safely.

CODE:

```
File Edit Selection View Go Run Terminal Help
AI coding
umber_classifier.py anagram_checker.py inventory_management.py date_validator.py assignment 7.4.py assignment 1.1task3.py division_error_handling.py
1 def safe_division_in_loop():
2     """
3     Demonstrates handling a ZeroDivisionError within a loop using try-except.
4
5     This function iterates through a list of divisors and attempts to perform
6     a division. When a ZeroDivisionError is encountered, it is caught,
7     a message is printed, and the loop continues to the next iteration
8     safely without crashing.
9     """
10    print("--- Running safe division loop with error handling ---")
11    numerator = 100
12    divisors = [10, 5, 2, 0, 4, 8]
13
14    for divisor in divisors:
15        try:
16            # Attempt the potentially problematic division
17            result = numerator / divisor
18            print(f"{numerator} / {divisor} = {result}")
19        except ZeroDivisionError:
20            # If a ZeroDivisionError occurs, this block is executed.
21            print(f"Error: Cannot divide {numerator} by zero. Skipping this iteration.")
22            # The 'continue' keyword explicitly moves to the next item in the loop.
23            continue
24
25    print("\nLoop finished successfully because the error was handled.")
26
27
28 if __name__ == "__main__":
29     safe_division_in_loop()
30
In 1 Cell | Space 4 | UTF-8 | GBK | Python | ChatGPT (Python 3.12.6.64-bit) | Go Live | Preview
```

Expected Outcome #4:

- Copilot adds a try-except block around the risky operation, preventing crashes and printing a meaningful error message.

OUTPUT:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\PROGRAMMES\VS\CODE\AI> coding & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES/VS/CODE/AI/coding/division_error_handling.py"
--- Running safe division loop with error handling ---
100 / 10 = 10.0
100 / 5 = 20.0
100 / 2 = 50.0
Error: Cannot divide 100 by zero. Skipping this iteration.
100 / 4 = 25.0
100 / 8 = 12.5

Loop finished successfully because the error was handled.
PS C:\PROGRAMMES\VS\CODE\AI>
```

CONCLUSION:

This script provides an excellent demonstration of handling `ZeroDivisionError` within a loop using a `try-except` block. By catching the specific exception, it prevents a fatal crash and ensures the program's continuity and stability. The use of the `continue` statement is key, allowing the loop to safely skip the problematic iteration and proceed. Overall, it's a clear, practical example of defensive programming, essential for creating reliable and resilient applications.

Task Description #5:

- Include a buggy class definition with incorrect `__init__` parameters or attribute references. Ask AI to analyze and correct the constructor and attribute usage.

PROMPT:

GENERATE A PYTHON CODE TO Include a buggy class definition with incorrect `init` parameters or attribute references.

CODE:

```
1 class User:
2     """
3     A buggy class definition to demonstrate common __init__ and attribute errors.
4
5     This class contains two intentional bugs:
6     1. In the __init__ method, the 'name' parameter is assigned to a
7       misspelled attribute 'self.user_name'.
8     2. The 'display_profile' method then tries to access 'self.name', which
9       was never created, leading to an AttributeError.
10    """
11    def __init__(self, name: str, email: str):
12        """
13        Initializes a User object with a bug.
14
15        BUG: The 'name' parameter is incorrectly assigned to 'self.user_name'.
16        """
17        print(f"Initializing user with name='{name}'...")
18        self.user_name = name # <-- Bug is here: Should be self.name
19        self.email = email
20
21    def display_profile(self):
22        """
23        Attempts to display the user's profile, triggering the bug.
24
25        BUG: This method tries to access 'self.name', but it was never defined
26        in __init__ due to the typo. This will raise an AttributeError.
27        """
28        print("Attempting to display profile...")
29
30    # This line will fail because 'self.name' does not exist
```

```
1 class User:
21     def display_profile(self):
27         """
28         print("Attempting to display profile...")
29         # This line will fail because 'self.name' does not exist.
30         return f"Username: {self.name}, Email: {self.email}"
31
32
33 if __name__ == "__main__":
34     print("--- Demonstrating an AttributeError from a buggy class ---\n")
35
36     # Create an instance of the buggy User class
37     user_instance = User("Alice", "alice@example.com")
38
39     try:
40         # Call the method that will fail
41         user_instance.display_profile()
42     except AttributeError as e:
43         print(f"Caught an expected error: {e}")
44         print("This error occurred because the 'display_profile' method tried to access 'self.name', but the '__init__' method
```

Expected Outcome #5:

- Copilot identifies mismatched parameters or missing self references and rewrites the class with accurate initialization and usage.

OUTPUT:

```
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:\PROGRAMMES VSCODE\AI coding\buggy_class.py"
--- Demonstrating an AttributeError from a buggy class ---

Initializing user with name='Alice'...
Attempting to display profile...
Caught an expected error: 'User' object has no attribute 'name'
This error occurred because the 'display_profile' method tried to access 'self.name', but the '__init__' method incorrectly saved it as 'self.user_name'.
PS C:\PROGRAMMES VSCODE\AI coding>
```

CONCLUSION:

This script provides an excellent, clear demonstration of a common `AttributeError` in object-oriented Python. It effectively illustrates how a simple typo in an attribute name within the `__init__` method can cause runtime failures in other parts of the class. The code wisely uses a `try...except` block not just to prevent a crash, but also to explain the exact nature of the bug. Overall, it's a perfect, self-contained lesson on the importance of consistent attribute naming and proper error handling

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

Evaluation Criteria:

Criteria	Max Marks
Logic	0.5
Type mismatch in list elements during sorting	0.5
Resource	0.5
Runtime	0.5
Syntax	0.5
Total	2.5 Marks