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	**Applicableto Batches**	
AssignmentNumber: **7.4**(Present assignment number)/**24**(Total number of assignments)			

Q.No.	Question	*ExpectedTime 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.	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:

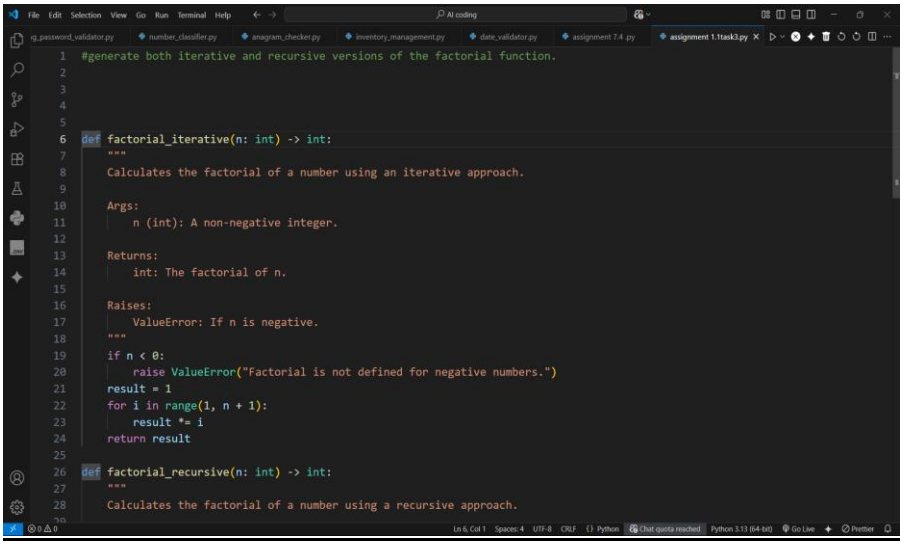
- Use AI tools to detect and correct syntax, logic, and runtime errors.
- Interpret AI-suggested bug fixes and explanations.
- Apply systematic debugging strategies supported by AI-generated insights.
- Refactor buggy code using responsible and reliable programming patterns.

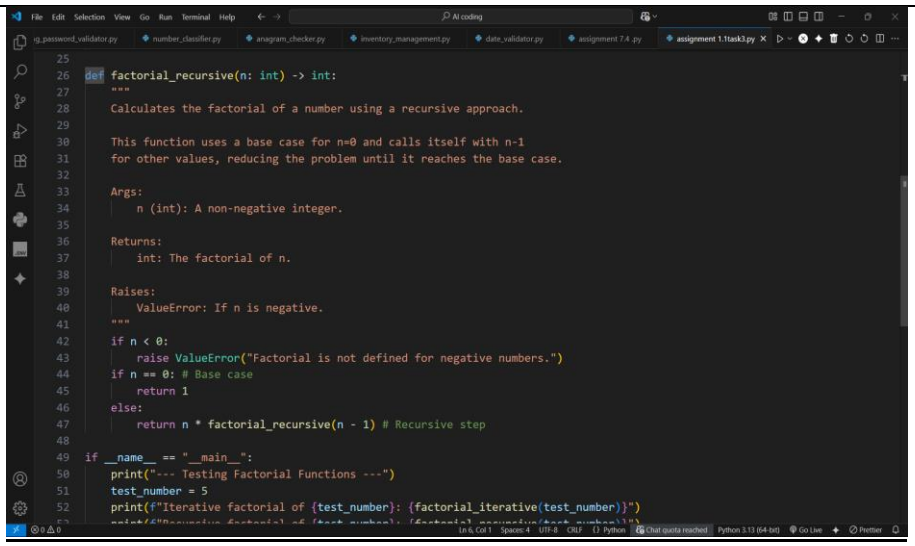
Task Description #1:
• Introduce a buggy Python function that calculates the factorial of a number using recursion. Use Copilot or Cursor AI to detect and fix the logical or syntax errors.

PROMPT:

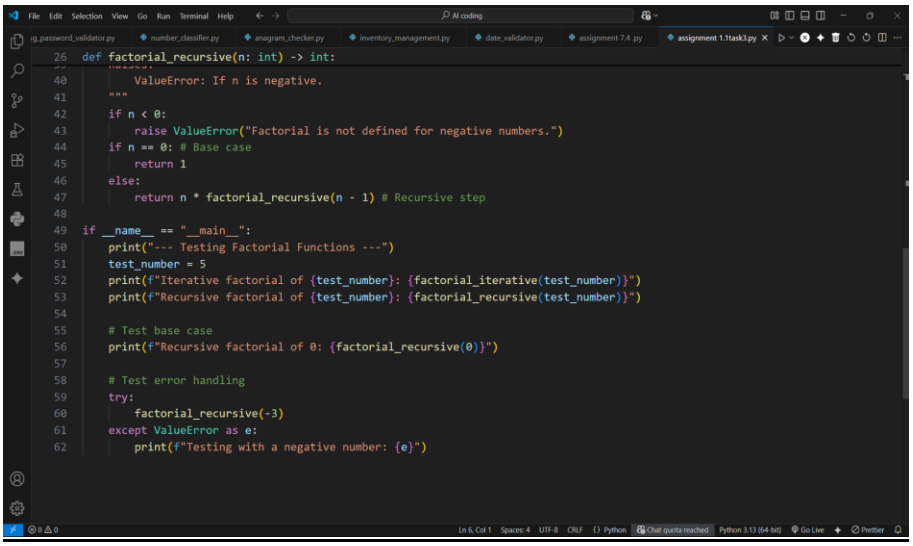
generate code of Python function that calculates the factorial of a number using recursion

CODE:





```python
def factorial_recursive(n: int) -> int:
    """
    Calculates the factorial of a number using a recursive approach.

    This function uses a base case for n=0 and calls itself with n-1
    for other values, reducing the problem until it reaches the base case.

    Args:
        n (int): A non-negative integer.

    Returns:
        int: The factorial of n.

    Raises:
        ValueError: If n is negative.
    """
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    if n == 0: # Base case
        return 1
    else:
        return n * factorial_recursive(n - 1) # Recursive step

if __name__ == "__main__":
    print("--- Testing Factorial Functions ---")
    test_number = 5
    print(f"Iterative factorial of {test_number}: {factorial_iterative(test_number)}")
```

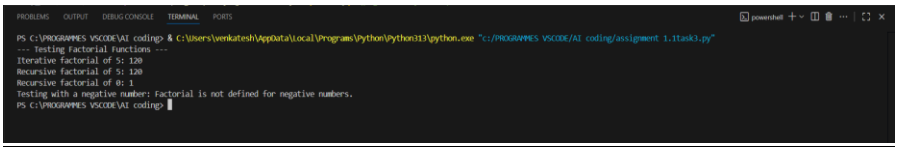


```python
def factorial_recursive(n: int) -> int:
        ValueError: If n is negative.
    """
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    if n == 0: # Base case
        return 1
    else:
        return n * factorial_recursive(n - 1) # Recursive step

if __name__ == "__main__":
    print("--- Testing Factorial Functions ---")
    test_number = 5
    print(f"Iterative factorial of {test_number}: {factorial_iterative(test_number)}")
    print(f"Recursive factorial of {test_number}: {factorial_recursive(test_number)}")

    # Test base case
    print(f"Recursive factorial of 0: {factorial_recursive(0)}")

    # Test error handling
    try:
        factorial_recursive(-3)
    except ValueError as e:
        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\Python313\python.exe "c:/PROGRAMMES VSCODE/AI coding/assignment 1.1task3.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.

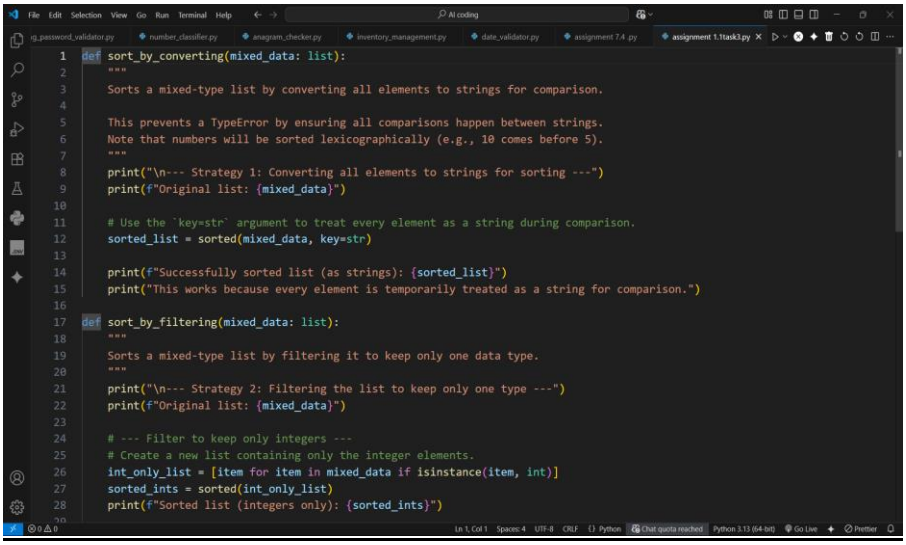
Task Description #2:

• 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**.**

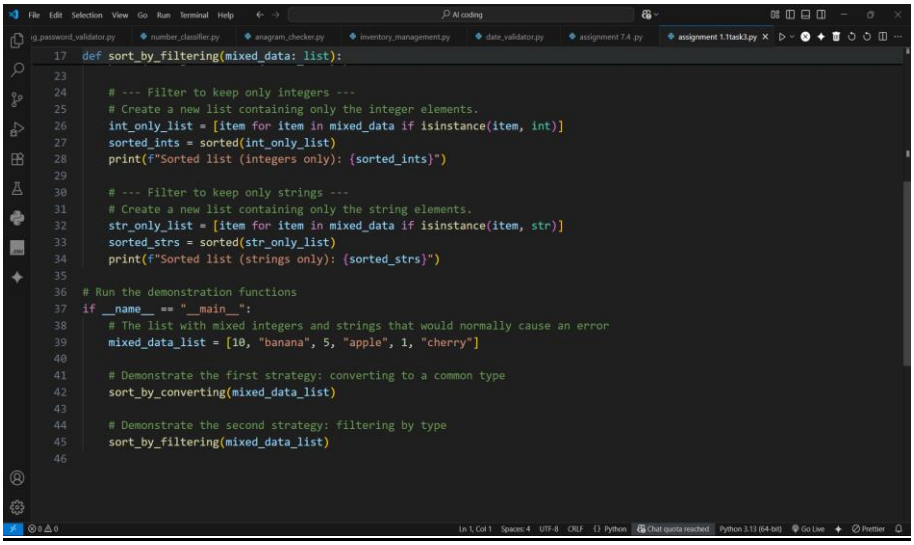
PROMPT:

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:



```python
def sort_by_converting(mixed_data: list):
    """
    Sorts a mixed-type list by converting all elements to strings for comparison.

    This prevents a TypeError by ensuring all comparisons happen between strings.
    Note that numbers will be sorted lexicographically (e.g., 10 comes before 5).
    """
    print("\n--- Strategy 1: Converting all elements to strings for sorting ---")
    print(f"Original list: {mixed_data}")

    # Use the `key=str` argument to treat every element as a string during comparison.
    sorted_list = sorted(mixed_data, key=str)

    print(f"Successfully sorted list (as strings): {sorted_list}")
    print("This works because every element is temporarily treated as a string for comparison.")

def sort_by_filtering(mixed_data: list):
    """
    Sorts a mixed-type list by filtering it to keep only one data type.
    """
    print("\n--- Strategy 2: Filtering the list to keep only one type ---")
    print(f"Original list: {mixed_data}")

    # --- Filter to keep only integers ---
    # Create a new list containing only the integer elements.
    int_only_list = [item for item in mixed_data if isinstance(item, int)]
    sorted_ints = sorted(int_only_list)
    print(f"Sorted list (integers only): {sorted_ints}")
```

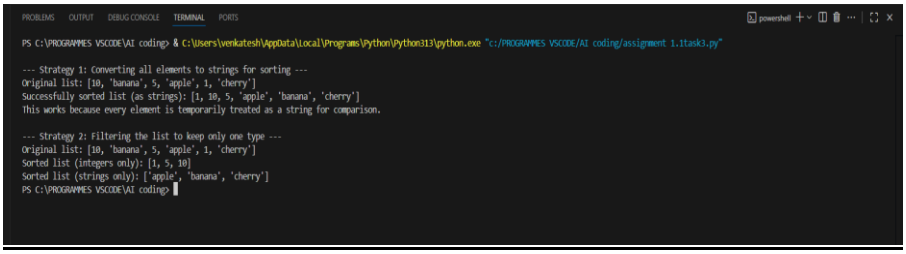


```python
def sort_by_filtering(mixed_data: list):

    # --- Filter to keep only integers ---
    # Create a new list containing only the integer elements.
    int_only_list = [item for item in mixed_data if isinstance(item, int)]
    sorted_ints = sorted(int_only_list)
    print(f"Sorted list (integers only): {sorted_ints}")

    # --- Filter to keep only strings ---
    # Create a new list containing only the string elements.
    str_only_list = [item for item in mixed_data if isinstance(item, str)]
    sorted_strs = sorted(str_only_list)
    print(f"Sorted list (strings only): {sorted_strs}")

# Run the demonstration functions
if __name__ == "__main__":
    # The list with mixed integers and strings that would normally cause an error
    mixed_data_list = [10, "banana", 5, "apple", 1, "cherry"]

    # Demonstrate the first strategy: converting to a common type
    sort_by_converting(mixed_data_list)

    # Demonstrate the second strategy: filtering by type
    sort_by_filtering(mixed_data_list)
```

Expected Outcome #2:

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

OUTPUT:



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.

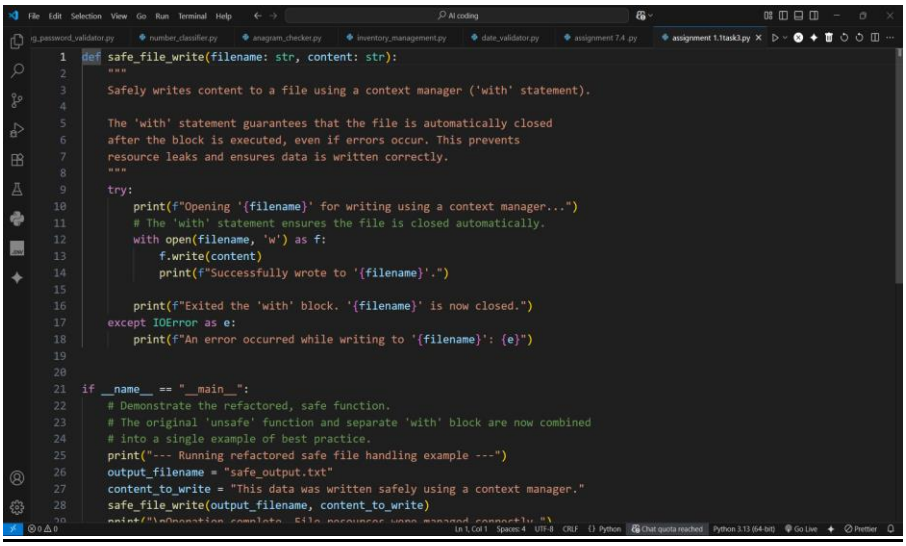
Task Description #3:

• 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

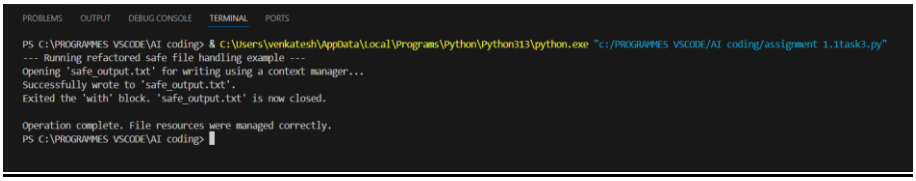
CODE:



Expected Outcome #3:

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

OUTPUT:



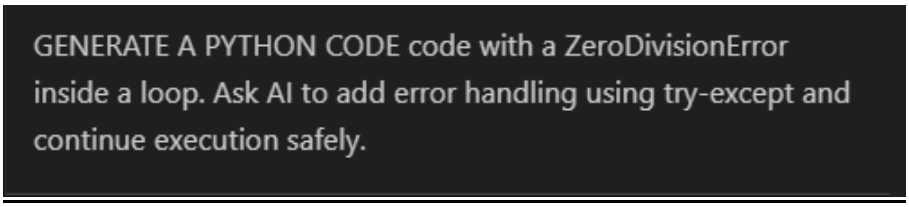
CONCLUSION:

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.

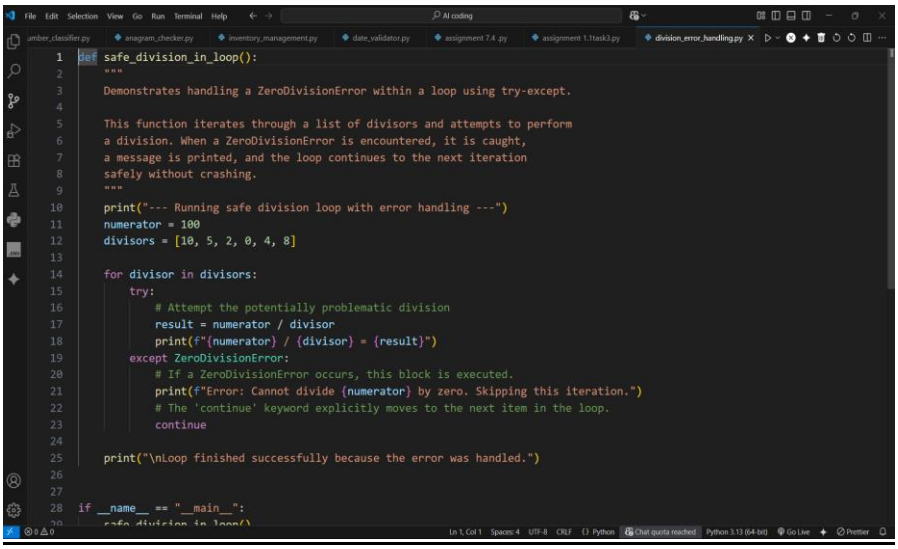
Task Description #4:

• 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:



CODE:



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 VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/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 VSCODE\AI coding>
```

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:

```python
class User:
    """
    A buggy class definition to demonstrate common __init__ and attribute errors.

    This class contains two intentional bugs:
    1. In the __init__ method, the 'name' parameter is assigned to a
       misspelled attribute `self.user_name`.
    2. The `display_profile` method then tries to access `self.name`, which
       was never created, leading to an AttributeError.
    """
    def __init__(self, name: str, email: str):
        """
        Initializes a User object with a bug.

        BUG: The 'name' parameter is incorrectly assigned to 'self.user_name'.
        """
        print(f"Initializing user with name='{name}'...")
        self.user_name = name  # <-- Bug is here: Should be self.name
        self.email = email

    def display_profile(self):
        """
        Attempts to display the user's profile, triggering the bug.

        BUG: This method tries to access 'self.name', but it was never defined
        in __init__ due to the typo. This will raise an AttributeError.
        """
        print("Attempting to display profile...")
        # This line will fail because 'self.name' does not exist
```

```python
class User:
    def display_profile(self):
        """
        print("Attempting to display profile...")
        # This line will fail because 'self.name' does not exist.
        return f"Username: {self.name}, Email: {self.email}"


if __name__ == "__main__":
    print("--- Demonstrating an AttributeError from a buggy class ---\n")

    # Create an instance of the buggy User class
    user_instance = User("Alice", "alice@example.com")

    try:
        # Call the method that will fail
        user_instance.display_profile()
    except AttributeError as e:
        print(f"Caught an expected error: {e}")
        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:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                                    powershell

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**