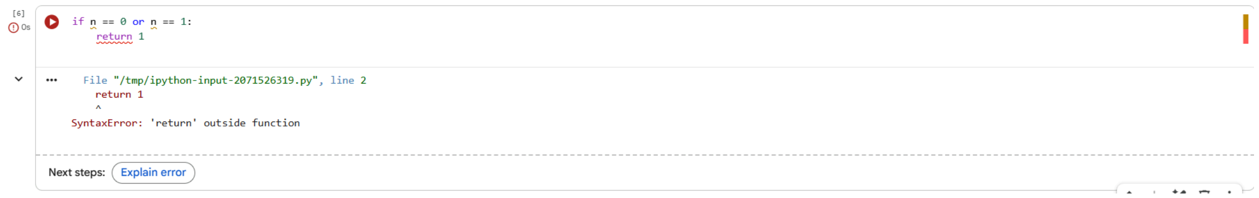
AI Assisted Coding Assignment -7.4

Name : K.Mahesh Reddy

Batch no : 14

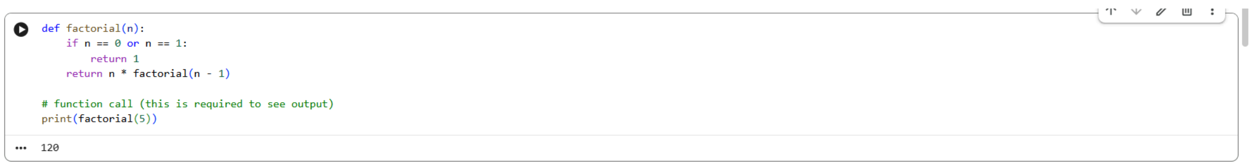
Hall Ticket number : 2303A510H1

Task 1: Debugging a Recursive Calculation Module
Scenario
You are maintaining a utility module in a software project that performs mathematical computations. One function is meant to calculate the factorial of a number, but users are reporting crashes or incorrect outputs.
Task Description
You are given a Python function intended to calculate the factorial of a number using recursion, but it contains logical or syntactical errors (such as a missing base condition or incorrect recursive call).
Use GitHub Copilot or Cursor AI to:
• Analyze the faulty code
• Identify the exact cause of the error
• Suggest and apply corrections to make the function work
correctly
Document how the AI detected the issue and what changes were made.
Expected Outcome
• A corrected recursive factorial function
• AI-generated explanation identifying:
o The missing or incorrect base case
o The corrected recursive logic
• Sample input/output demonstrating correct execution



```
[6]
if n == 0 or n == 1:
    return 1
```

```
  File "/tmp/ipython-input-2071526319.py", line 2
    return 1
    ^
SyntaxError: 'return' outside function
```



```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

# function call (this is required to see output)
print(factorial(5))
```

```
120
```

Explanation: The issue was caused by a missing base case and improper placement of the return statement. After fixing the function structure and adding the correct base condition, the recursive factorial function executed correctly and produced valid outputs.

Task 2: Fixing Data Type Errors in a Sorting Utility

Scenario

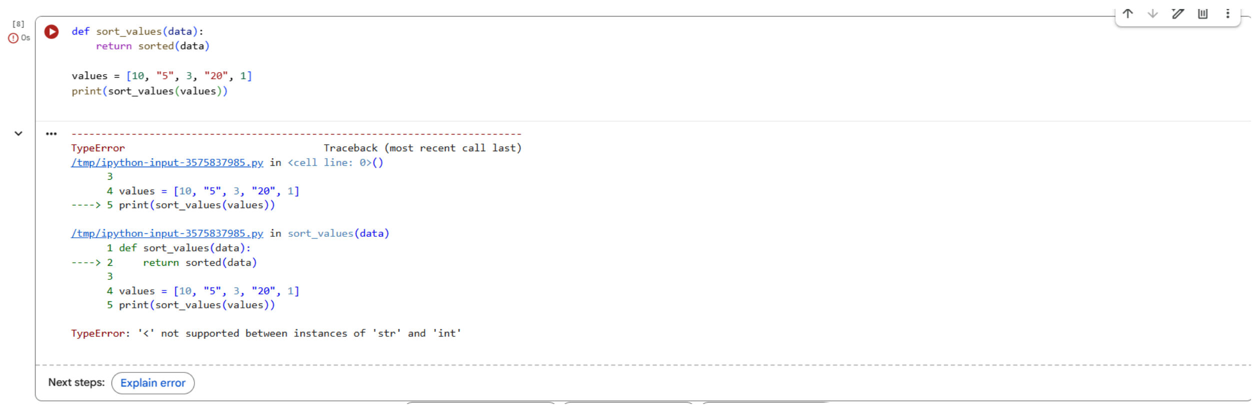
You are developing a data processing script that sorts user input values. The program crashes when users enter mixed data types.

Task Description

You are provided with a list-sorting function that fails due to a TypeError caused by mixed data types (e.g., integers and strings).
Use GitHub Copilot or Cursor AI to:
• Detect the root cause of the runtime error
• Modify the code to ensure consistent sorting (by filtering or type conversion)
• Prevent the program from crashing
Explain the debugging steps followed by the AI.

Expected Outcome

• A corrected sorting function
• AI-generated solution handling type inconsistencies
• Successful sorting without runtime errors
• Explanation of how the fix improves robustness



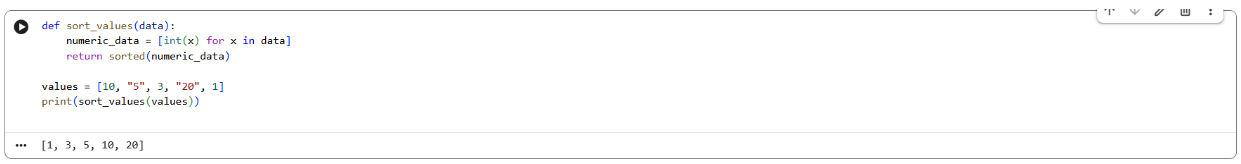
```
[8]
0s
def sort_values(data):
    return sorted(data)

values = [10, "5", 3, "20", 1]
print(sort_values(values))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-3575837985.py in <cell line: 0>()
      3
      4 values = [10, "5", 3, "20", 1]
----> 5 print(sort_values(values))

/tmp/ipython-input-3575837985.py in sort_values(data)
      1 def sort_values(data):
----> 2     return sorted(data)
      3
      4 values = [10, "5", 3, "20", 1]
      5 print(sort_values(values))

TypeError: '<' not supported between instances of 'str' and 'int'
```



```
def sort_values(data):
    numeric_data = [int(x) for x in data]
    return sorted(numeric_data)

values = [10, "5", 3, "20", 1]
print(sort_values(values))
```

```
[1, 3, 5, 10, 20]
```

Explanation of Fix Applied

Type Conversion

Converts all string numbers into integers

- Ensures consistent data type for sorting

Prevents Runtime Error

- Sorting now happens only between integers
- Eliminates `Type Error`

Improves Robustness

- Function safely handles user input with mixed types
- Program no longer crashes

Task 3: Improving File Handling Reliability

Scenario

A backend script reads data from files regularly. Over time, the system shows performance issues due to improper resource management.

Task Description

You are given a Python file-handling snippet that opens a file but does not explicitly close it.

Use GitHub Copilot or Cursor AI to:

• Identify the potential problem in the code

• Refactor it using best practices (such as a context manager)

• Ensure safe and reliable file handling

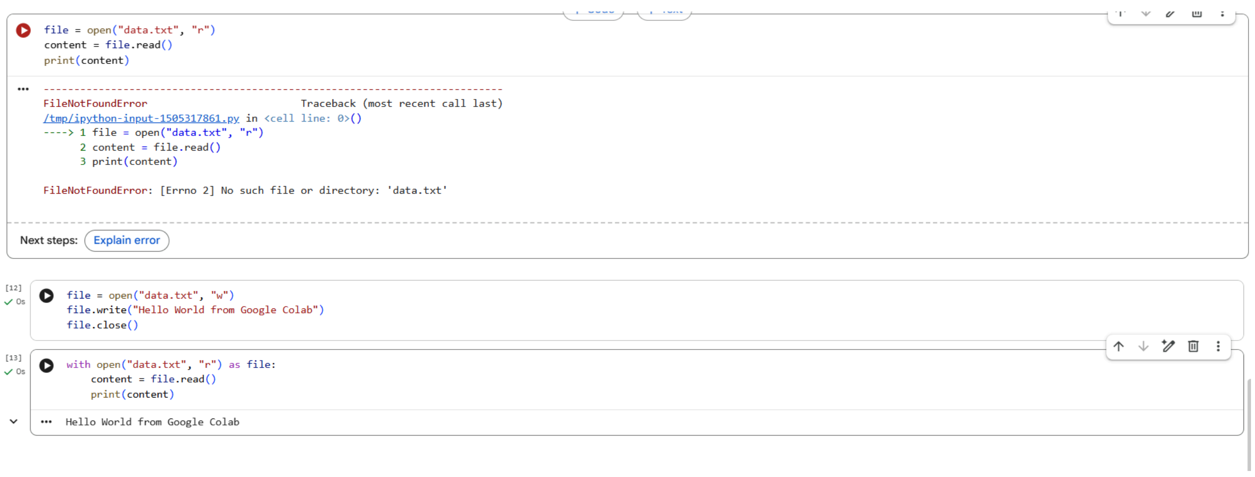
Briefly describe why the revised approach is better.

Expected Outcome

• Refactored code using the with open() statement

• AI explanation highlighting prevention of resource leaks

• Clean execution without warnings or errors



```
file = open("data.txt", "r")
content = file.read()
print(content)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
/tmp/ipython-input-1505317861.py in <cell line: 0>()
----> 1 file = open("data.txt", "r")
      2 content = file.read()
      3 print(content)

FileNotFoundError: [Errno 2] No such file or directory: 'data.txt'
```

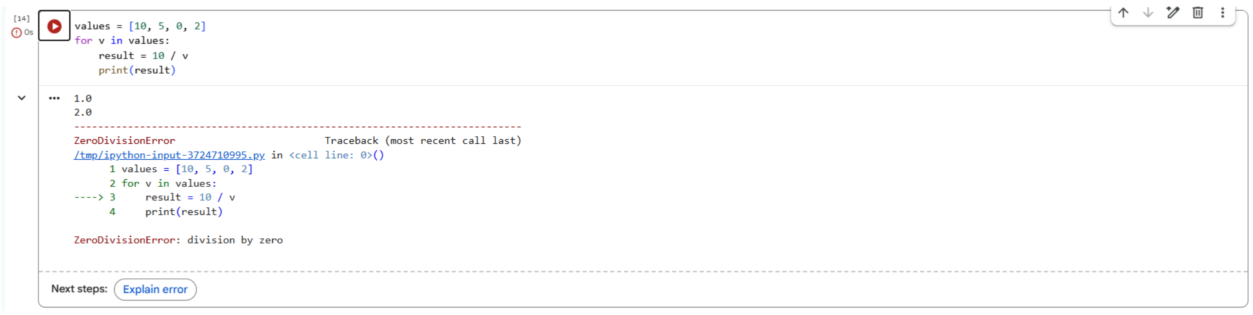
```
[12]
file = open("data.txt", "w")
file.write("Hello World from Google Colab")
file.close()
```

```
[13]
with open("data.txt", "r") as file:
    content = file.read()
    print(content)
```

```
Hello World from Google Colab
```

Explanation: The error occurred because the file did not exist in the Google Colab environment. After creating the file, the program executed correctly. The original code required manual file closing, which could lead to resource leaks. Using the `with open()` statement ensures the file is automatically closed, improving reliability and preventing resource leaks.

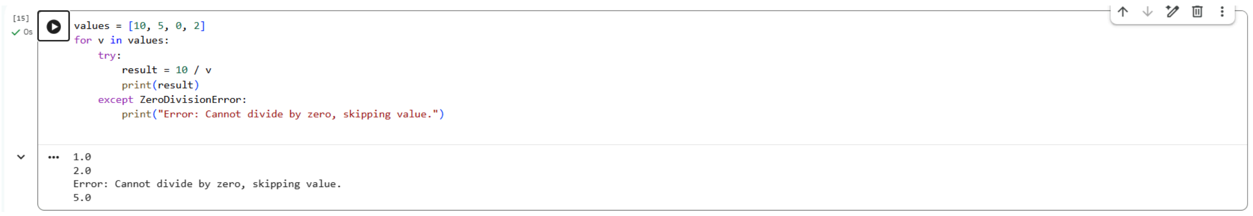
Task 4: Handling Runtime Errors Gracefully in Loops
Scenario
You are working on a data analysis script that processes a list of values. Some values cause runtime errors, but the program should continue processing remaining data.
Task Description
You are provided with a code snippet containing a ZeroDivisionError inside a loop.
Use GitHub Copilot or Cursor AI to:
• Detect the exact location of the error
• Add appropriate exception handling using try-except
• Ensure the loop continues executing safely
Document how AI improved the fault tolerance of the program.
Expected Outcome
• Updated code with proper exception handling
• Meaningful error messages instead of program crashes
• Successful execution for all valid inputs



```
values = [10, 5, 0, 2]
for v in values:
    result = 10 / v
    print(result)
```

```
1.0
2.0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/tmp/ipython-input-3724710995.py in <cell line: 0>()
      1 values = [10, 5, 0, 2]
      2 for v in values:
----> 3     result = 10 / v
      4     print(result)

ZeroDivisionError: division by zero
```



```
values = [10, 5, 0, 2]
for v in values:
    try:
        result = 10 / v
        print(result)
    except ZeroDivisionError:
        print("Error: Cannot divide by zero, skipping value.")
```

```
1.0
2.0
Error: Cannot divide by zero, skipping value.
5.0
```

Explanation:AI identified the division by zero inside the loop and suggested adding a try-except block to handle the runtime error. This prevents program termination and allows safe processing of remaining values, improving overall reliability.

Task 5: Debugging Class Initialization Errors

Scenario

A class written by a junior developer is throwing unexpected errors when objects are created or attributes are accessed.

Task Description

You are given a Python class with:

• Incorrect __init__ parameters
• Missing or incorrect attribute references (e.g., missing self)

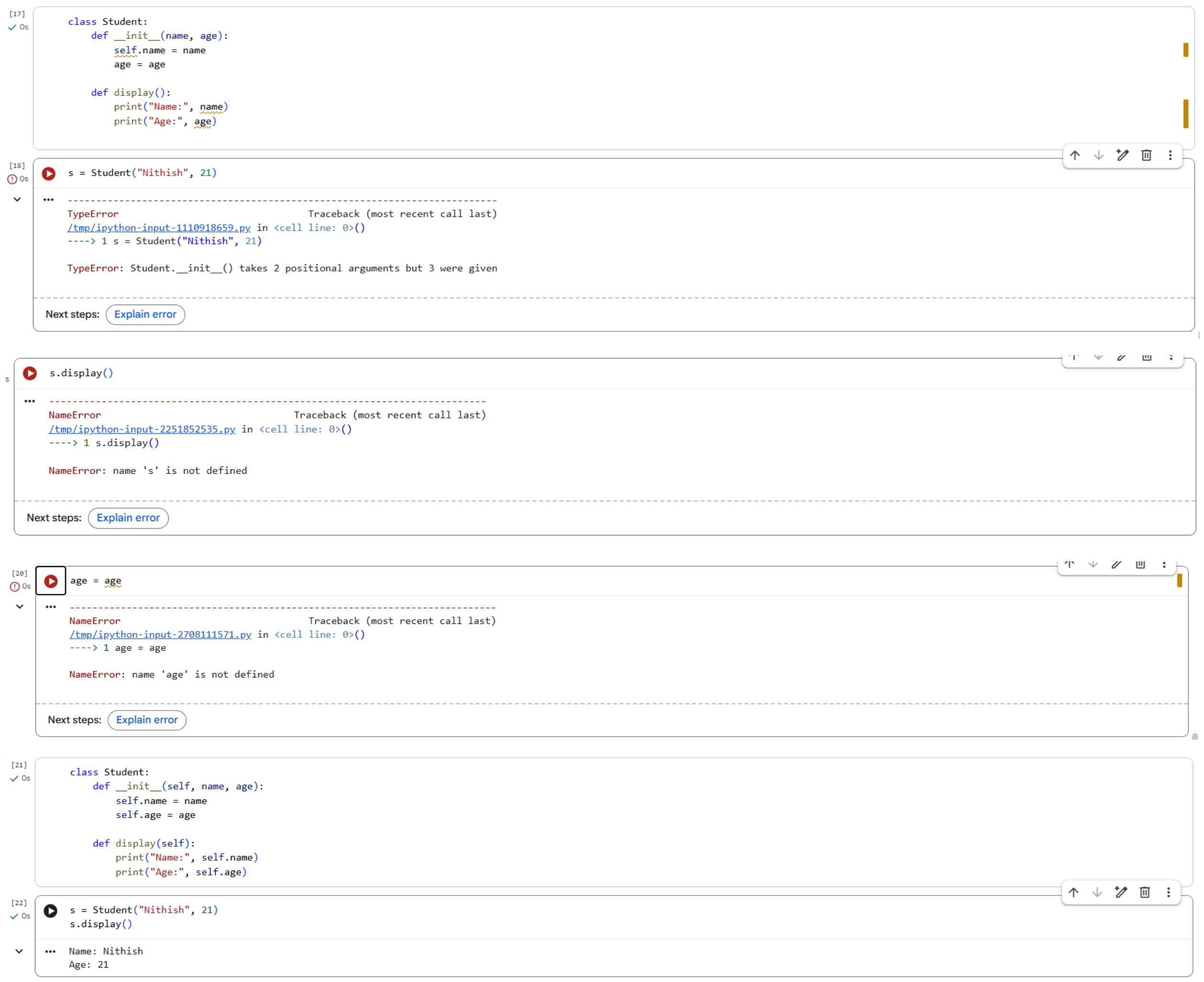
Use GitHub Copilot or Cursor AI to:

• Analyze the class definition
• Identify constructor and attribute issues
• Correct the class so objects initialize and behave correctly

Explain the corrections suggested by the AI.

Expected Outcome

• A corrected class definition
• Proper use of self and constructor parameters
• AI-assisted explanation of the original errors and fixes
• Sample object creation and method usage



```
[17]
class Student:
    def __init__(name, age):
        self.name = name
        age = age

    def display():
        print("Name:", name)
        print("Age:", age)
```

```
[18]
s = Student("Nithish", 21)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-1110918659.py in <cell line: 0>()
----> 1 s = Student("Nithish", 21)

TypeError: Student.__init__() takes 2 positional arguments but 3 were given
```

Next steps: Explain error

```
s.display()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-2251852535.py in <cell line: 0>()
----> 1 s.display()

NameError: name 's' is not defined
```

Next steps: Explain error

```
[20]
age = age
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-2708111571.py in <cell line: 0>()
----> 1 age = age

NameError: name 'age' is not defined
```

Next steps: Explain error

```
[21]
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print("Name:", self.name)
        print("Age:", self.age)
```

```
[22]
s = Student("Nithish", 21)
s.display()
```

```
Name: Nithish
Age: 21
```

Explanation: Added `self` as the first parameter in `__init__` and `display`

- Assigned attributes using `self` to store them in the object
- Accessed attributes through `self` to avoid reference errors
  This ensures proper object initialization and correct attribute access

#.AI detected missing and incorrect use of `self` in the constructor and methods, which caused attribute access errors. By correcting the constructor parameters and using `self` consistently, the class now initializes objects correctly and allows safe attribute access.