## HTNO :- 2403A52045

# Assignment – 7.4

### 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.

### CODE

```
def factorial(n):
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")
    elif n == 0 or n == 1:   # ✅ Correct base case
        return 1
    else:
        return n * factorial(n - 1)  # ✅ Correct recursive step
print(factorial(5))   # 120
print(factorial(0))   # 1
print(factorial(1))   # 1
```

### OUTPUT

```
120
1
1
720
```

### OBSERVATION

The recursive factorial function works correctly. The output discrepancy (720) indicates that the code executed in your environment had one more print(factorial(6)) call than the snippet you posted.

### 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.

## CODE

```
def buggy_sort_list(items):
    return sorted(items)


# Fixed version: Convert everything to integers before sorting
def fixed_sort_list(items):
    return sorted(int(x) for x in items)
# Example input
data = [10, "5", 3, "20", 7]
print("---- Buggy Version ----")
try:
    print(buggy_sort_list(data))
except TypeError as e:
    print("Error:", e)
print("\n---- Fixed Version ----")
print(fixed_sort_list(data))
```

## OUTPUT

```
[3, 7, 10]
```

## OBSERVATION

1.The buggy version fails because Python cannot compare integers and strings during sorting.

2.The fixed version should convert all values to integers, producing [3, 5, 7, 10, 20].

3.If your output is [3, 7, 10], that means the AI applied filtering instead of conversion (ignoring strings instead of converting them).

4.Both approaches are valid fixes, but they give different results

### 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).

## CODE

```
def read_file(filename):
    with open(filename, "r") as file:
        content = file.read()
    return content
# Example usage
print(read_file("example.txt"))
```

## OUTPUT

```
Hello AI
This is a test file.
```

## OBSERVATION

1.**Best practice followed:**

- Using with open() prevents resource leakage, unlike open() without close().

2.**Safe execution:**

- No need to manually close the file; avoids runtime warnings.

3.**Readable and concise:**

- The context manager makes the code clean and Pythonic.

4.**Limitation:**

- Reads the entire file at once; for very large files, reading line by line might be more memory-efficient

## 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

## CODE

```
numbers = [10, 5, 0, 2, 0, 4]
for n in numbers:
    try:
        result = 100 / n
```

```
        print(f"100 / {n} = {result}")
    except ZeroDivisionError:
        print(f"Error: Cannot divide by zero for n = {n}. Skipping...")
        continue
```

## OUTPUT

```
100 / 10 = 10.0
100 / 5 = 20.0
Error: Cannot divide by zero for n = 0. Skipping...
100 / 2 = 50.0
Error: Cannot divide by zero for n = 0. Skipping...
100 / 4 = 25.0
```

## OBSERVATION

1.**Error handling works correctly:**

- Division by zero does not crash the program.
- A meaningful message is printed for zero values.

2.**Execution continues safely:**

- Loop continues for the remaining numbers after an error.

3.**Readable and maintainable:**

- Clear separation of normal execution and error handling.

4.**Best practice:**

- Using try-except inside a loop is ideal when some operations might fail but others should still execute

## 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.

## CODE

```
class Student:
```

```
    def __init__(self, name, age):   # ✅ Correct parameters
        self.name = name          # ✅ Proper attribute assignment
        self.age = age


    def display(self):
        print(f"Name: {self.name}, Age: {self.age}")
# Example usage
s1 = Student("Alice", 20)
s1.display()
```

## OUTPUT

Name: Alice, Age: 20

## OBSERVATION

1.**Constructor correctly defined:**

- Includes self as the first parameter.
- Attributes self.name and self.age are properly initialized.

2.**Attribute references correct:**

- display method correctly accesses attributes with self.

3.**No runtime errors:**

- Instantiating the class and calling display works as expected.

4.**Good coding practice:**

- Proper use of self ensures instance-specific data is maintained.
- Code is readable and maintainable.