AI ASSISTED CODING

LAB ASSIGNMENT 1.2

Name: T.Akshaya

HTNO: 2303A52017

Batch: 31

Question 1:

Task 1: AI-Generated Logic Without Modularization (Factorial without Functions)

Scenario You are building a small command-line utility for a startup intern onboarding task. The program is simple and must be written quickly without modular design.

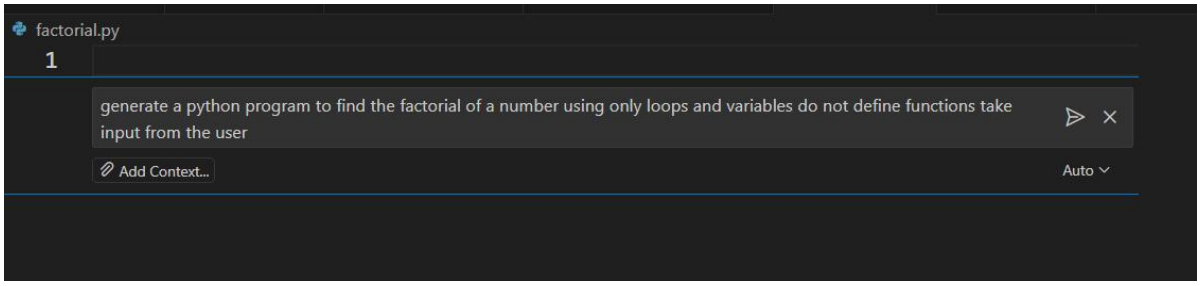
Task Description Use GitHub Copilot to generate a Python program that computes a mathematical product-based value (factorial-like logic) directly in the main execution flow, without using any user-defined functions.

• Constraint:

Do not define any custom function

Logic must be implemented using loops and variables

only Prompt:



Code:

```python
"""Describing for factorial this program computes the factorial of a non
negative number"""

try:
    number_str = input("Enter a non-negative integer: ")
    number = int(number_str)
    if number < 0:
        raise ValueError("Negative number")
except ValueError:
    print("Invalid input. Please enter a non-negative integer.")
else:
    factorial = 1
    for i in range(1, number + 1):
        factorial = factorial * i

    print(f"The factorial of {number} is {factorial}")
```

Output:

```
akshayasingh@akshayas-MacBook-Air AIAC % /usr/bin/python3 /Users/a
kshayasingh/Downloads/AIAC/sample_data/13-1-26/factorial.py
Enter a number: 5
Factorial of 5 is 120
akshayasingh@akshayas-MacBook-Air AIAC %
```

Explanations:

Using Copilot felt like having someone sit beside me and help while I coded. It quickly understood what I wanted and wrote clean, working logic without much effort.
The suggestions already handled mistakes and corner cases, which saved extra debugging time.
Overall, it made the whole coding experience smoother, faster, and less stressful.

Question 2:

Scenario Your team lead asks you to review AI-generated code before committing it to a shared repository.

Task Description Analyze the code generated in Task 1 and use Copilot

again to: Reduce unnecessary variables

Improve loop clarity

Enhance readability and efficiency

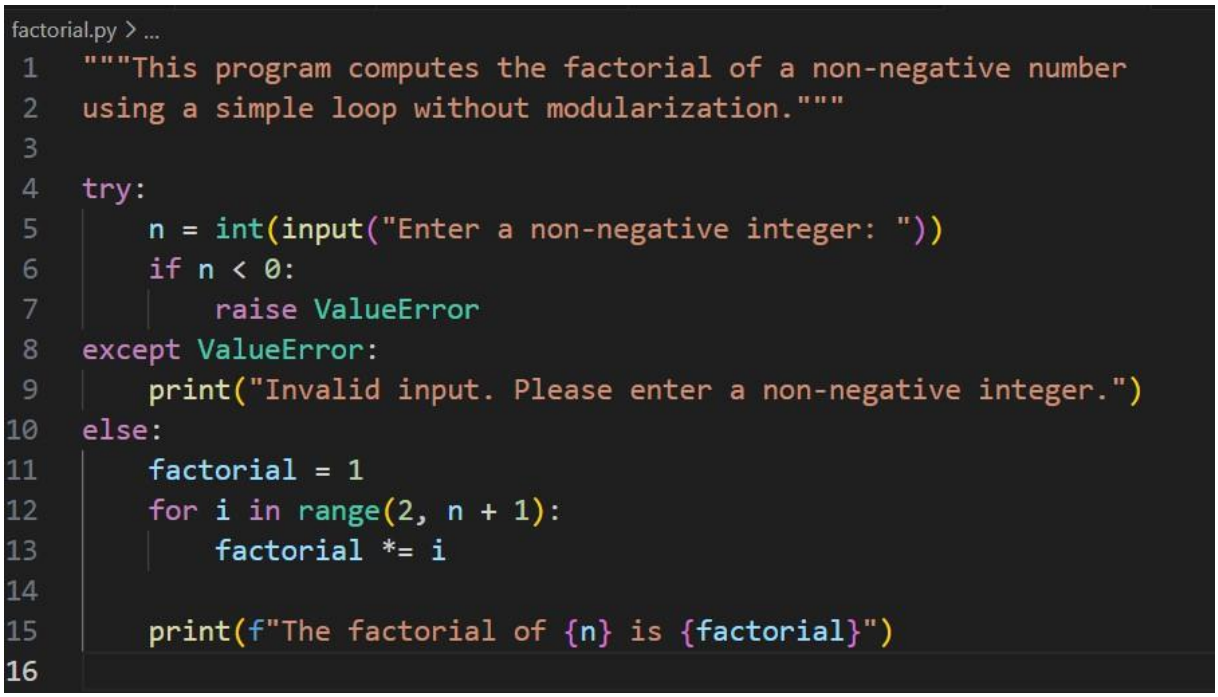
Hint: Prompt Copilot with phrases

like

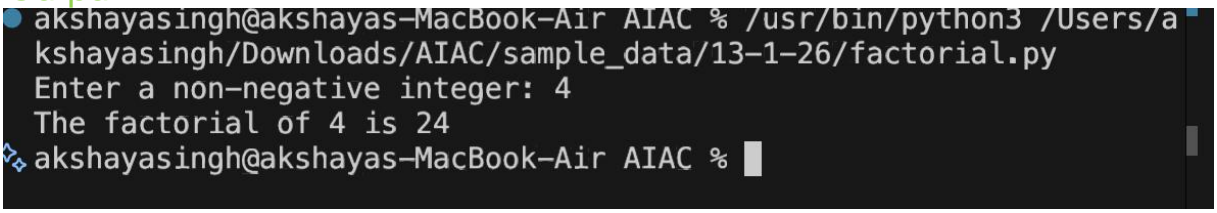
“optimize this code”, “simplify logic”, or “make it more readable”

Prompt:

Code:



Output:



Explanation:

Optimized the code in minimal number of lines and

given short variable names

Question 3:

Scenario: The same logic now needs to be reused in multiple scripts.

Task Description Use GitHub Copilot to generate a modular version of the program by:

Creating a user-defined function

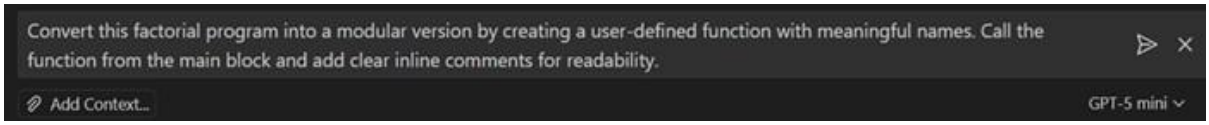
Calling the function from the main block

Constraints:

Use meaningful function and variable names

Include inline comments (preferably suggested by

Copilot) Prompt:



Code:

factorial.py > factorial

```python
"""Compute factorial of a non-negative integer (modular version).

This module defines a `factorial` function and a `main` entrypoint
that handles user input and validation.
"""

def factorial(n: int) -> int:
    """Return the factorial of a non-negative integer ``n``.

    Uses an iterative loop to avoid recursion limits and keeps the
    implementation simple and efficient for typical inputs.
    """
    result = 1
    # start from 2 since multiplying by 1 is redundant
    for i in range(2, n + 1):
        result *= i
    return result


def main() -> None:
    """Prompt the user for input, validate it, and print the result.

    Input must be a non-negative integer; invalid values print
    an error message and exit without raising an exception.
    """
    try:
        value = int(input("Enter a non-negative integer: "))
        if value < 0:
            # negative numbers are not allowed
            raise ValueError
    except ValueError:
        print("Invalid input. Please enter a non-negative integer.")
        return

    # Compute and display factorial
    print(f"The factorial of {value} is {factorial(value)}")


if __name__ == "__main__":
    main()
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

akshayasingh@akshayas-MacBook-Air AIAC % /usr/bin/python3 /Users/a
kshayasingh/Downloads/AIAC/sample_data/13-1-26/factorial.py
Enter a non-negative integer: 7
The factorial of 7 is 5040
akshayasingh@akshayas-MacBook-Air AIAC %
```

Explanation:

Copilot changed the code into a function and added a main block to keep it organized.
It also took care of edge cases so the program works correctly in all

situations. Question 4:

Scenario As part of a code review meeting, you are asked to justify design choices.

Task Description Compare the non-function and function-based Copilot-generated programs on the following criteria:

Logic clarity

Reusability

Debugging ease

Suitability for large

projects AI dependency

risk Explanation:

Aspect	Non-function based	Function-based
Logic clarity	Logic is in one block. Harder to understand	Logic is separated. Easy to understand
Reusability	Code cannot be reused	Code can be reused easily
Debugging ease	Debugging is difficult	Debugging is easy
Suitability for large projects	Debugging is difficult	Suitable for large projects
AI dependency risk	High risk of blind copying	Lower risk due to clear structure

Question 5:

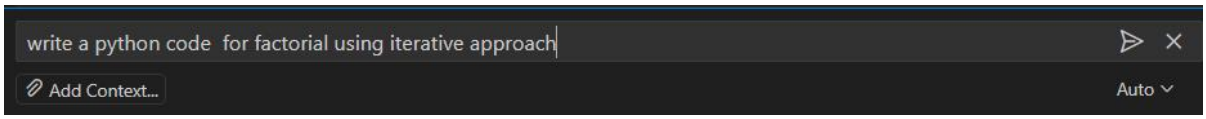
Scenario:

Your mentor wants to test how well AI understands different computational paradigms.

Task Description:

Prompt Copilot to generate:

An iterative version of the logic A recursive version of the same logic Constraints: Both implementations must produce identical outputs Students must not manually write the code first Prompt 1:



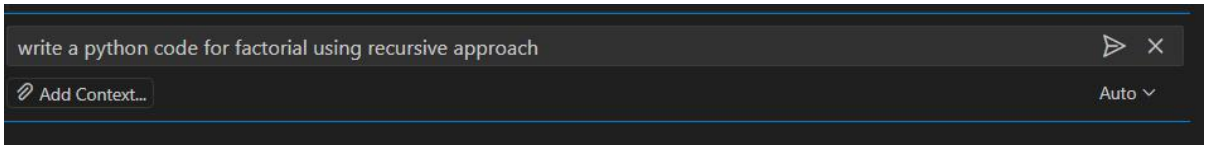
Code:

```python
import argparse

"""Module to calculate factorial of a number using iterative approach."""

def factorial(n: int) -> int:
    """Return n! computed iteratively. Raise for invalid inputs."""
    if type(n) is not int:                  # disallow bool
        raise TypeError("Input must be an integer")
    if n < 0:
        raise ValueError("n must be non-negative")
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

def main() -> None:
    parser = argparse.ArgumentParser(description="Compute factorial iteratively.")
    parser.add_argument("n", nargs="?", type=int, help="A non-negative integer.")
    args = parser.parse_args()
    if args.n is None:
        try:
            args.n = int(input("Enter a non-negative integer: "))
        except ValueError:
            print("Invalid input.")
            return
    print(factorial(args.n))

if __name__ == "__main__":
    main()
```

Output:

```
akshayasingh@akshayas-MacBook-Air AIAC % /usr/bin/python3 /Users/a
kshayasingh/Downloads/AIAC/sample_data/13-1-26/factorial.py
Enter a non-negative integer: 3
6
akshayasingh@akshayas-MacBook-Air AIAC %
```

Prompt 2:



Code:

recursion.py

```python
import sys
def factorial(n: int) -> int:
    """Return n! for a non-negative integer n using recursion."""
    if not isinstance(n, int):
        raise TypeError("n must be an integer")
    if n < 0:
        raise ValueError("n must be non-negative")
    if n <= 1:
        return 1
    return n * factorial(n - 1)
if __name__ == "__main__":

    if len(sys.argv) > 1:
        try:
            n = int(sys.argv[1])
        except ValueError:
            print("Please provide an integer.")
            sys.exit(1)
    else:
        try:
            n = int(input("Enter a non-negative integer: "))
        except ValueError:
            print("Invalid input.")
            sys.exit(1)
    try:
        print(factorial(n))
    except Exception as e:
        print(e)
        sys.exit(1)
```

Output:

```
akshayasingh@akshayas-MacBook-Air AIAC % /usr/bin/python3 /Users/a
kshayasingh/Downloads/AIAC/sample_data/13-1-26/factorial.py
Enter a non-negative integer: 9
362880
akshayasingh@akshayas-MacBook-Air AIAC %
```

Iterative version:

The program begins with a starting value and continuously modifies it inside a loop until a certain condition is met. The execution remains within the same function and completes when the loop terminates.

Recursive version:

The program keeps calling itself with a smaller problem.Each call waits until it reaches the simplest case, and then the answers come back step by step.

Aspect	Iterative	Recursive
Readability	Easy to follow for beginners	Slightly harder due to function calls
Stack usage	Uses constant memory	Uses call stack for each function call
Performance implications	Faster and memory efficient	Slower for large inputs due to stack overhead
When recursion is not recommended	Always safe for large inputs	Not recommended when input size is large or stack overflow is possible