AI Assisted Coding

ASSIGNMENT 1.2

Name: V.Akshith
HT No: 2303A52433
Batch: 31

Question 1:

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

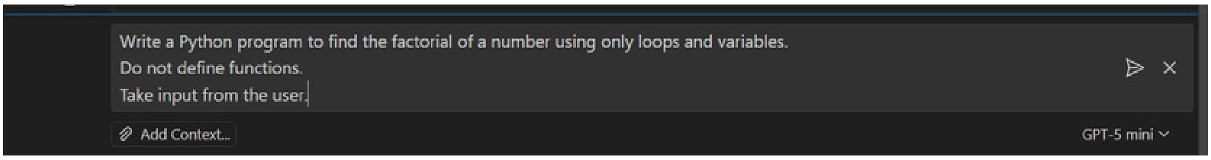
Scenario You are building a small command-line utility for a startup intern onboarding task. The program is simple and must be written quickly without modular design.

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:

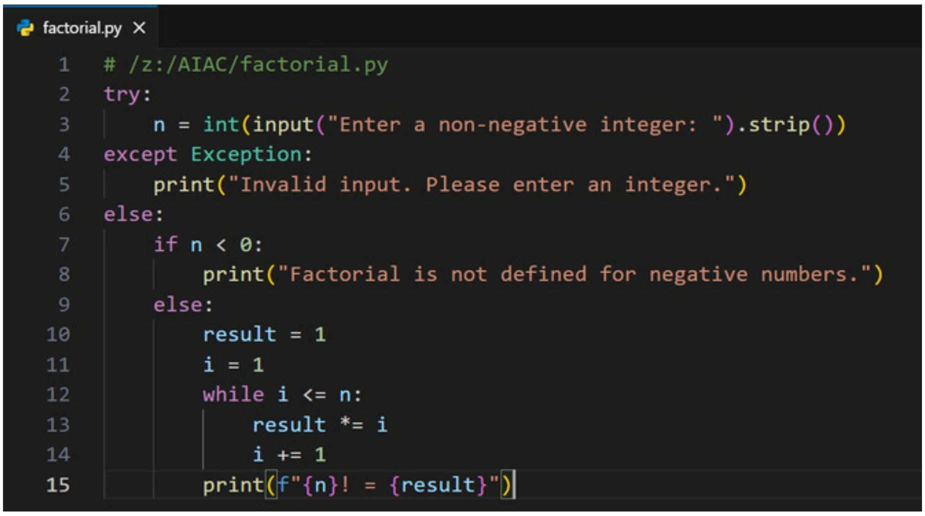
Do not define any custom function
Logic must be implemented using loops and variables only

Prompt:



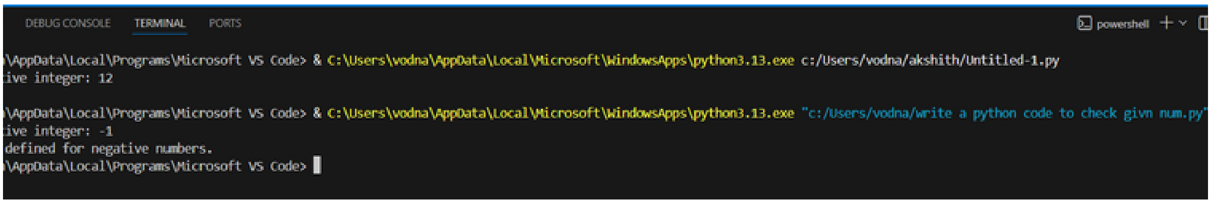
Write a Python program to find the factorial of a number using only loops and variables.
Do not define functions.
Take input from the user.

Code Screenshot:



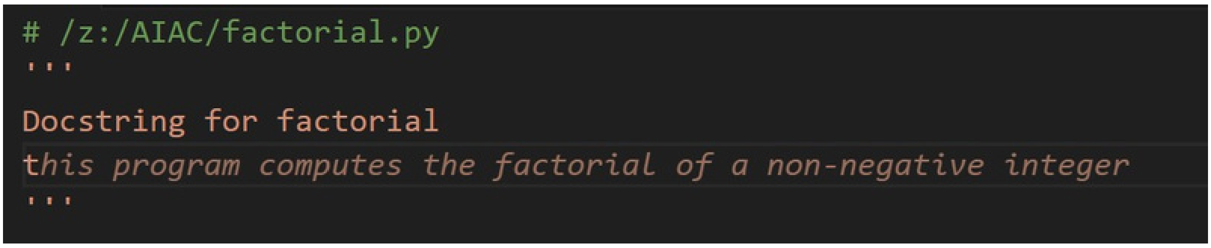
```python
# /z:/AIAC/factorial.py
try:
    n = int(input("Enter a non-negative integer: ").strip())
except Exception:
    print("Invalid input. Please enter an integer.")
else:
    if n < 0:
        print("Factorial is not defined for negative numbers.")
    else:
        result = 1
        i = 1
        while i <= n:
            result *= i
            i += 1
        print(f"{n}! = {result}")
```

Output(O/P):



```
\AppData\Local\Programs\Microsoft VS Code> & C:\Users\vodna\AppData\Local\Microsoft\WindowsApps\python3.13.exe c:/Users/vodna/akshith/Untitled-1.py
ive integer: 12

\AppData\Local\Programs\Microsoft VS Code> & C:\Users\vodna\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/vodna/write a python code to check givn num.py"
ive integer: -1
defined for negative numbers.
\AppData\Local\Programs\Microsoft VS Code> 
```

Copilot Suggestions:



```python
# /z:/AIAC/factorial.py
'''
Docstring for factorial
this program computes the factorial of a non-negative integer
'''
```

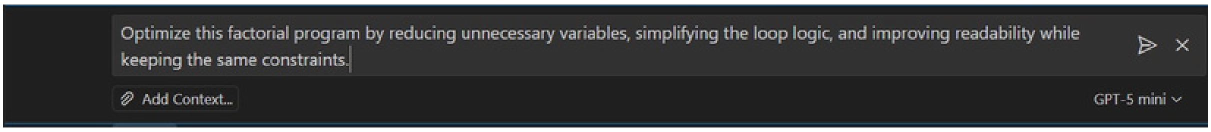
Explanation:

Copilot was very helpful for a beginner as
it quickly generated correct logic based on the prompt.
It reduced the time required to write the code by suggesting a complete solution.
The generated code followed best practices such as input validation and edge case handling. Copilot also helped in avoiding syntax errors and logical mistakes.
Overall, it made coding easier & faster.

Question 2:

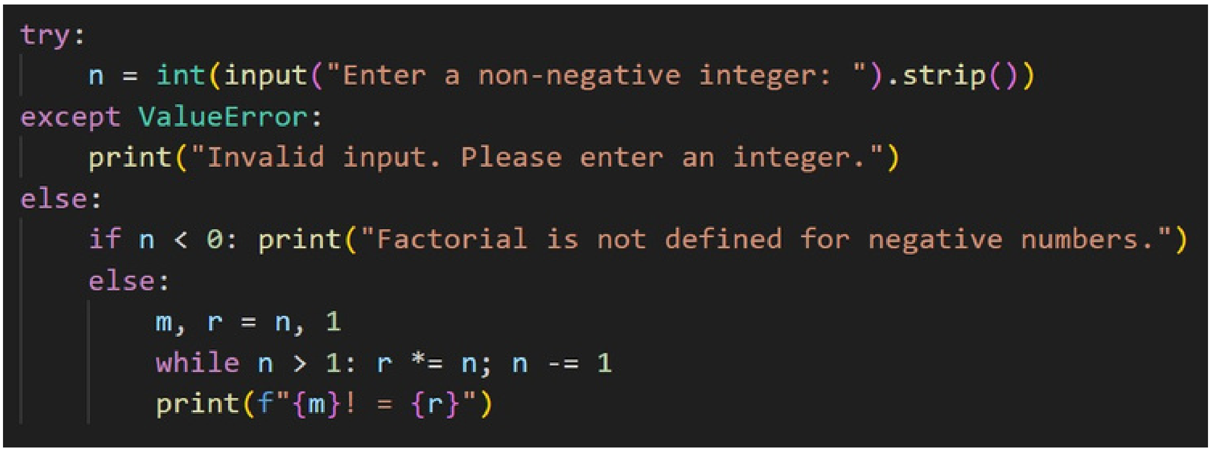
Scenario Your team lead asks you to review AI-generated code before committing it to a shared repository.
Task Description Analyze the code generated in Task 1 and use Copilot again to:
Reduce unnecessary variables
Improve loop clarity
Enhance readability and efficiency
Hint: Prompt Copilot with phrases like
“optimize this code”,
“simplify logic”, or
“make it more readable”

Prompt:



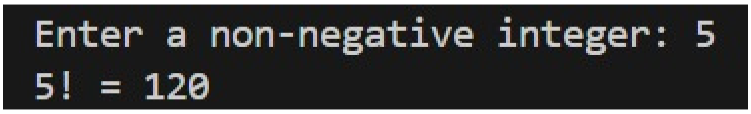
Optimize this factorial program by reducing unnecessary variables, simplifying the loop logic, and improving readability while keeping the same constraints.

Code Screenshot:



```python
try:
    n = int(input("Enter a non-negative integer: ").strip())
except ValueError:
    print("Invalid input. Please enter an integer.")
else:
    if n < 0: print("Factorial is not defined for negative numbers.")
    else:
        m, r = n, 1
        while n > 1: r *= n; n -= 1
        print(f"{m}! = {r}")
```

Output(O/P):



```
Enter a non-negative integer: 5
5! = 120
```

Explanation:

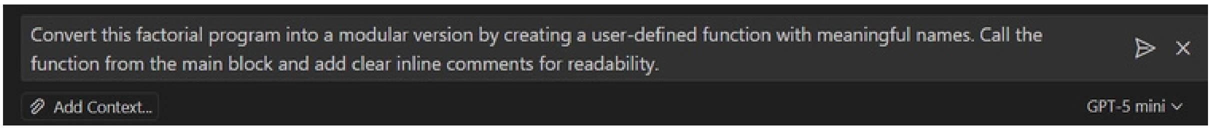
Optimized the code in minimal number of lines and short variable names

---

Question 3:

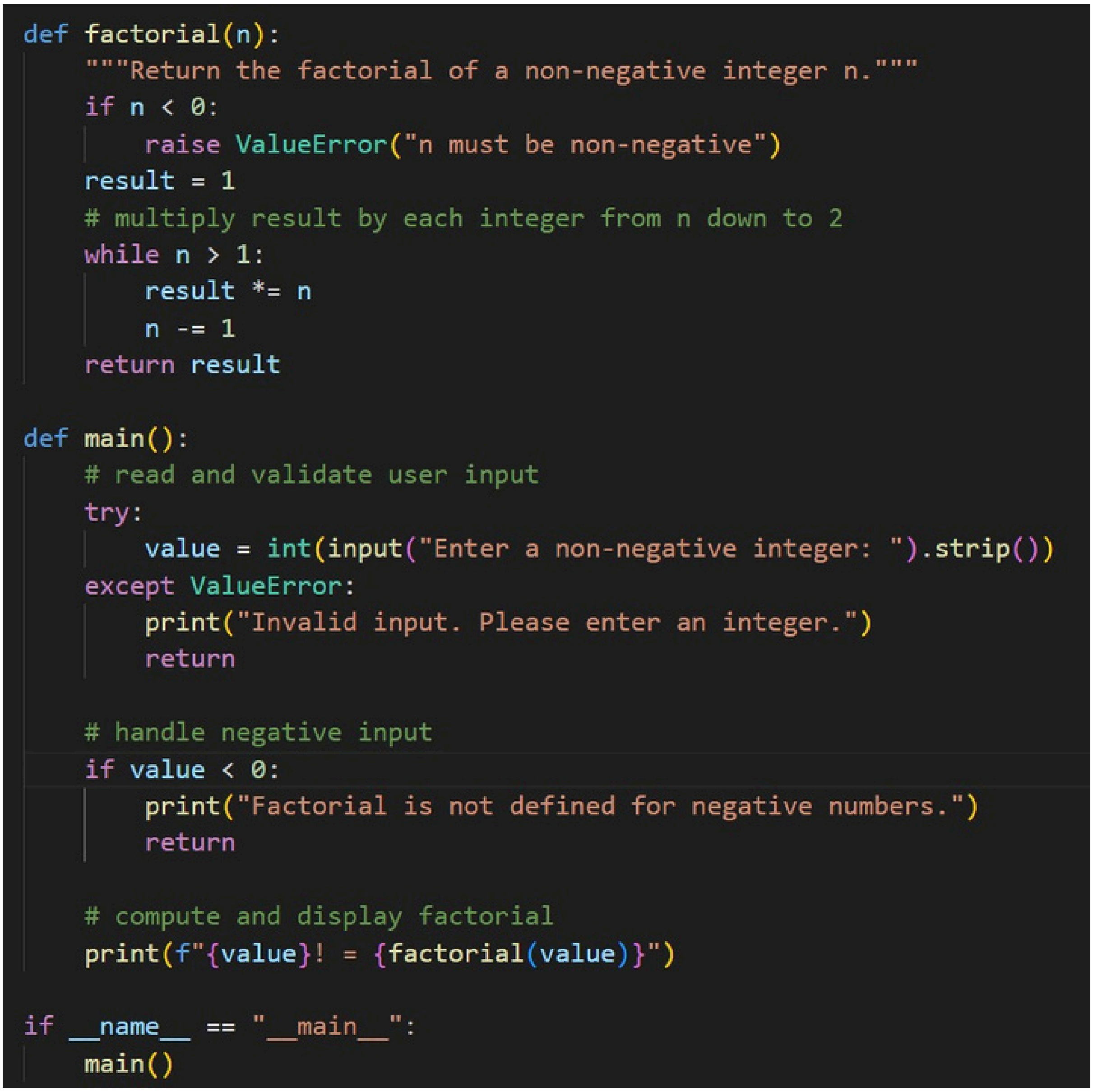
Scenario: The same logic now needs to be reused in multiple scripts.
Task Description Use GitHub Copilot to generate a modular version of the program by:
Creating a user-defined function
Calling the function from the main block
Constraints:
Use meaningful function and variable names
Include inline comments (preferably suggested by Copilot)

Prompt:



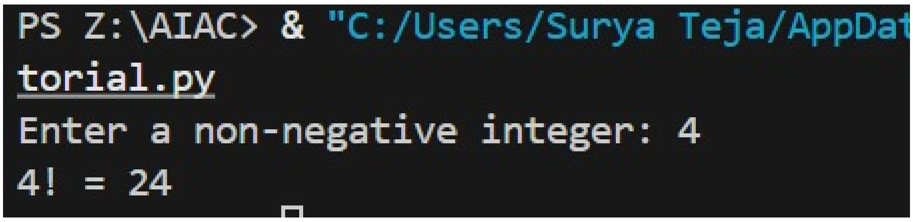
Convert this factorial program into a modular version by creating a user-defined function with meaningful names. Call the function from the main block and add clear inline comments for readability.

Code Screenshot:



```python
def factorial(n):
    """Return the factorial of a non-negative integer n."""
    if n < 0:
        raise ValueError("n must be non-negative")
    result = 1
    # multiply result by each integer from n down to 2
    while n > 1:
        result *= n
        n -= 1
    return result

def main():
    # read and validate user input
    try:
        value = int(input("Enter a non-negative integer: ").strip())
    except ValueError:
        print("Invalid input. Please enter an integer.")
        return

    # handle negative input
    if value < 0:
        print("Factorial is not defined for negative numbers.")
        return

    # compute and display factorial
    print(f"{value}! = {factorial(value)}")

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

Output(O/P):



```
PS Z:\AIAC> & "C:/Users/Surya Teja/AppDat
torial.py
Enter a non-negative integer: 4
4! = 24
```

Explanation:

Copilot Modified The Code To Function Type
& main block is created
Handled Edge Cases Well

Question 4:

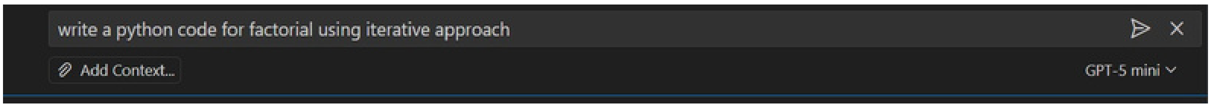
Scenario As part of a code review meeting, you are asked to justify design choices.
Task Description Compare the non-function and function-based Copilot-generated programs on the following criteria:
Logic clarity
Reusability
Debugging ease
Suitability for large projects
AI dependency risk

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	Not suitable for large projects	Suitable for large projects
AI dependency risk	High risk of blind copying	Lower risk due to clear structure

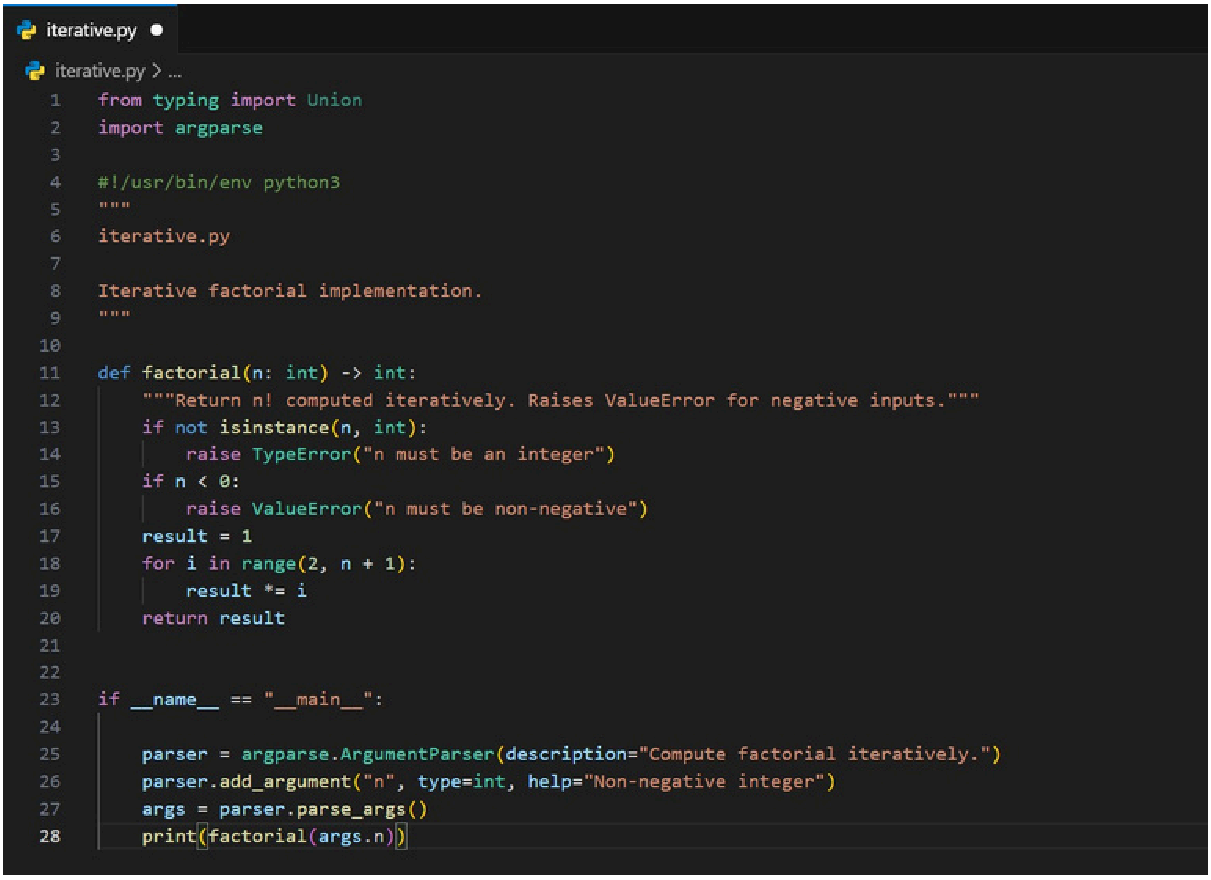
Question 5: Scenario:
Your mentor wants to test how well AI understands different computational paradigms.
Task Description:
Prompt Copilot to generate:
An iterative version of the logic A recursive version of the same logic Constraints: Both implementations must produce identical outputs Students must not manually write the code first

Prompt 1:



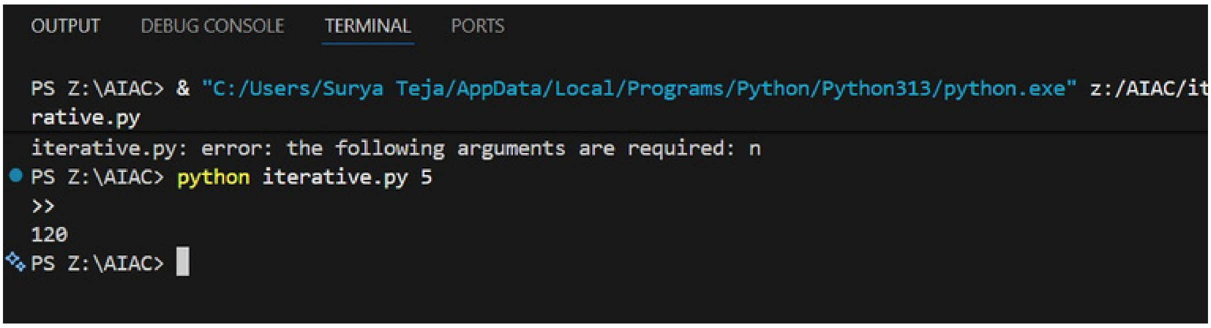
write a python code for factorial using iterative approach

Code:



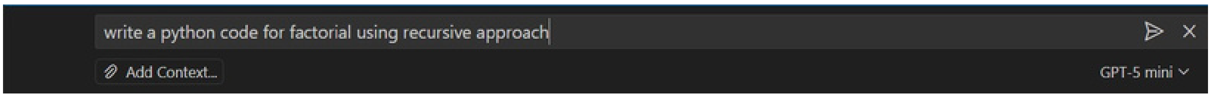
```python
from typing import Union
import argparse

#!/usr/bin/env python3
"""
iterative.py

Iterative factorial implementation.
"""

def factorial(n: int) -> int:
    """Return n! computed iteratively. Raises ValueError for negative inputs."""
    if not isinstance(n, int):
        raise TypeError("n 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


if __name__ == "__main__":

    parser = argparse.ArgumentParser(description="Compute factorial iteratively.")
    parser.add_argument("n", type=int, help="Non-negative integer")
    args = parser.parse_args()
    print(factorial(args.n))
```

Output:



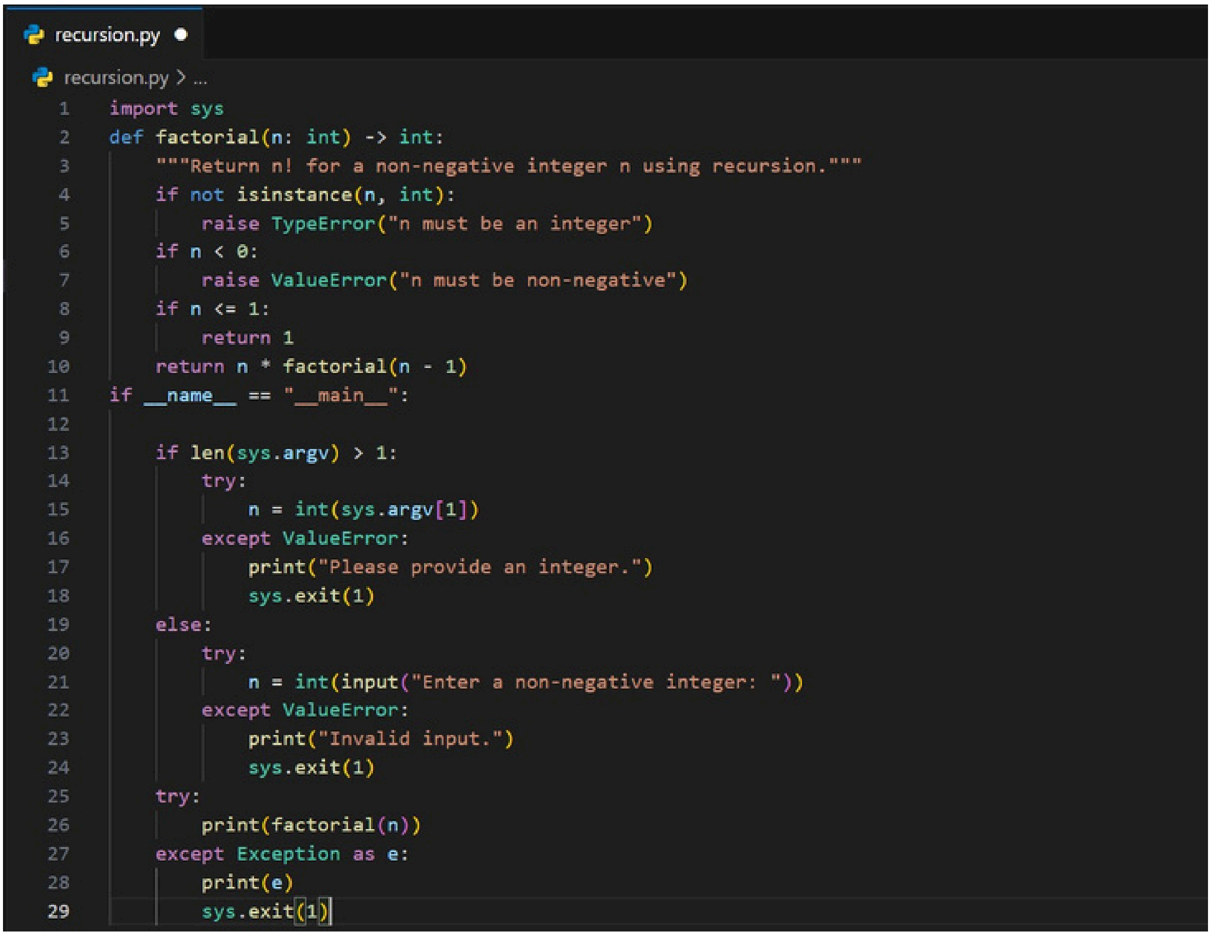
```
PS Z:\AIAC> & "C:/Users/Surya Teja/AppData/Local/Programs/Python/Python313/python.exe" z:/AIAC/it
rative.py
iterative.py: error: the following arguments are required: n
PS Z:\AIAC> python iterative.py 5
>>
120
PS Z:\AIAC> █
```

Prompt 2:



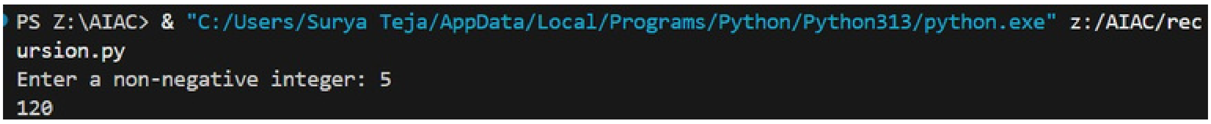
write a python code for factorial using recursive approach

Code:



```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:



```
PS Z:\AIAC> & "C:/Users/Surya Teja/AppData/Local/Programs/Python/Python313/python.exe" z:/AIAC/rec
ursion.py
Enter a non-negative integer: 5
120
```

Iterative version:

The program starts with an initial value and repeatedly updates it using a loop until the condition is satisfied. The control flow stays within a single function and finishes once the loop ends.

Recursive version:

The program calls itself with a smaller input each time. Each call waits on the stack until the base condition is reached. After that, results are returned 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