NAME: N. Prudhvi

ROLLNO:2403A510G7

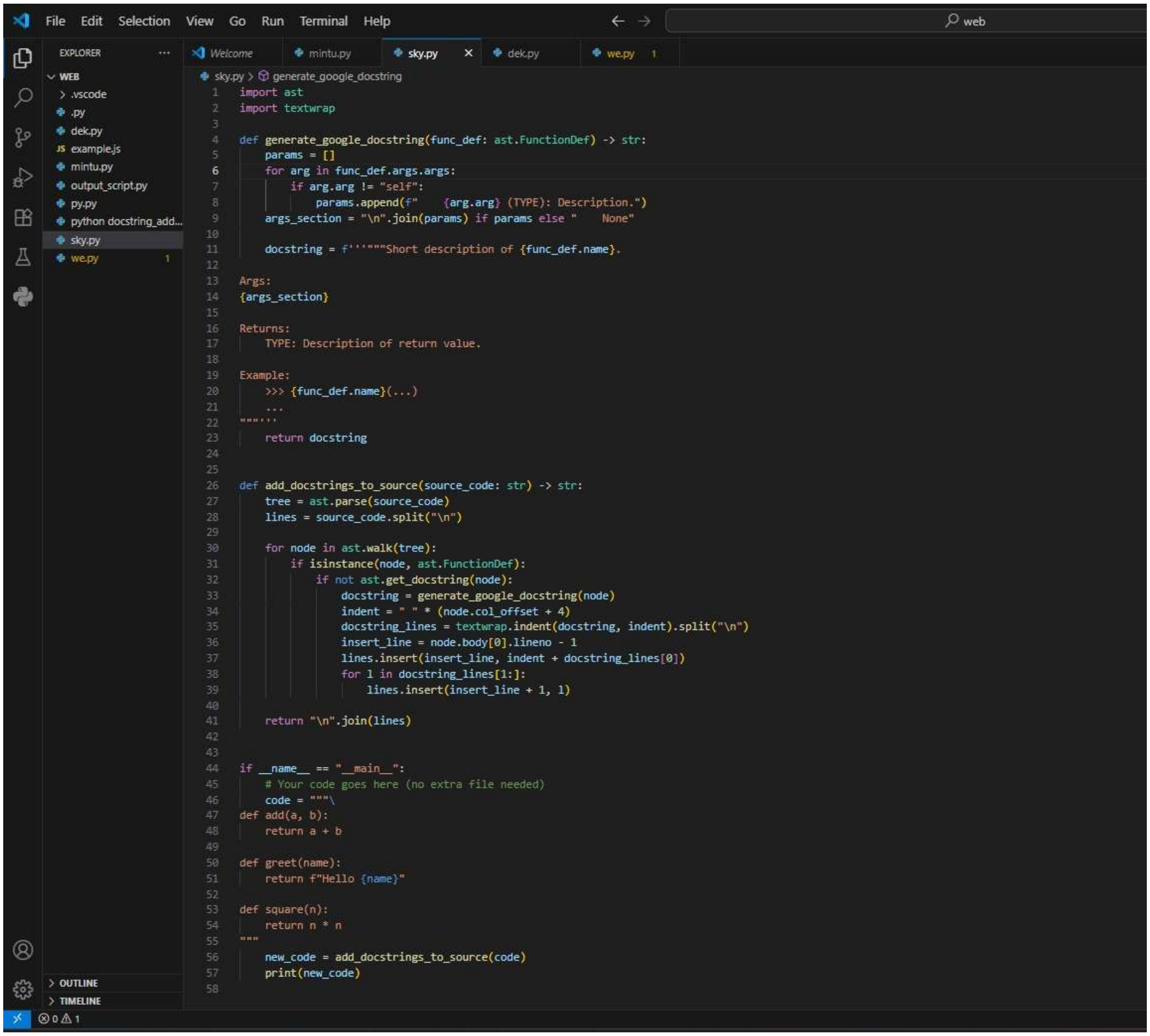
BATCHNO:06

Lab 9 – Documentation Generation (Modified Examples)

Task 1: Google-Style Docstrings for Python Functions

Prompt: “Add Google-style docstrings to all functions without input-output examples. Include function description, parameter types, return type, and example usage.”

Code:



```python
import ast
import textwrap

def generate_google_docstring(func_def: ast.FunctionDef) -> str:
    params = []
    for arg in func_def.args.args:
        if arg.arg != "self":
            params.append(f"    {arg.arg} (TYPE): Description.")
    args_section = "\n".join(params) if params else "    None"

    docstring = f'''"""Short description of {func_def.name}.

Args:
{args_section}

Returns:
    TYPE: Description of return value.

Example:
    >>> {func_def.name}(...)
    ...
"""'''
    return docstring


def add_docstrings_to_source(source_code: str) -> str:
    tree = ast.parse(source_code)
    lines = source_code.split("\n")

    for node in ast.walk(tree):
        if isinstance(node, ast.FunctionDef):
            if not ast.get_docstring(node):
                docstring = generate_google_docstring(node)
                indent = " " * (node.col_offset + 4)
                docstring_lines = textwrap.indent(docstring, indent).split("\n")
                insert_line = node.body[0].lineno - 1
                lines.insert(insert_line, indent + docstring_lines[0])
                for l in docstring_lines[1:]:
                    lines.insert(insert_line + 1, l)

    return "\n".join(lines)


if __name__ == "__main__":
    # Your code goes here (no extra file needed)
    code = """\
def add(a, b):
    return a + b

def greet(name):
    return f"Hello {name}"

def square(n):
    return n * n
"""
    new_code = add_docstrings_to_source(code)
    print(new_code)
```

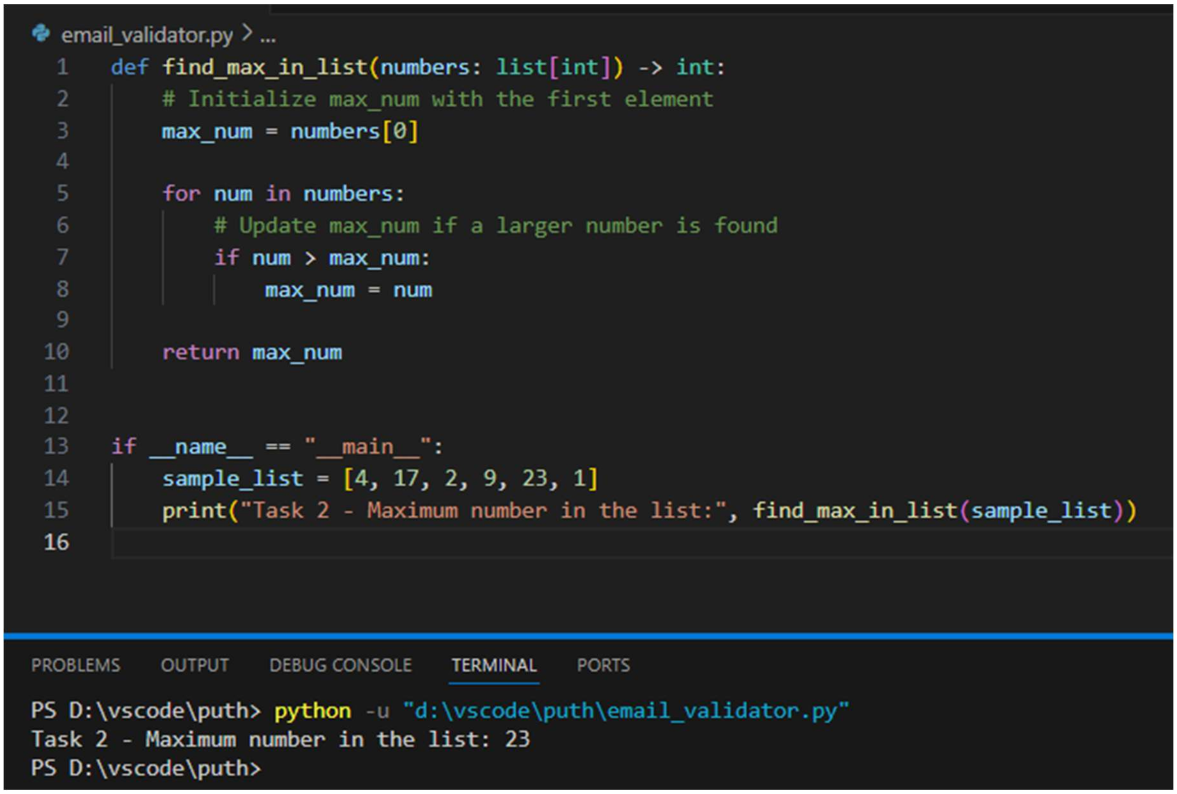
Output:

Observation: The function now has a professional Google-style docstring, improving clarity and usability.

Task 2: Inline Comments for Complex Logic

Prompt: “Add meaningful inline comments explaining complex logic parts only.”

Code:/Output:



```python
def find_max_in_list(numbers: list[int]) -> int:
    # Initialize max_num with the first element
    max_num = numbers[0]

    for num in numbers:
        # Update max_num if a larger number is found
        if num > max_num:
            max_num = num

    return max_num


if __name__ == "__main__":
    sample_list = [4, 17, 2, 9, 23, 1]
    print("Task 2 - Maximum number in the list:", find_max_in_list(sample_list))

```

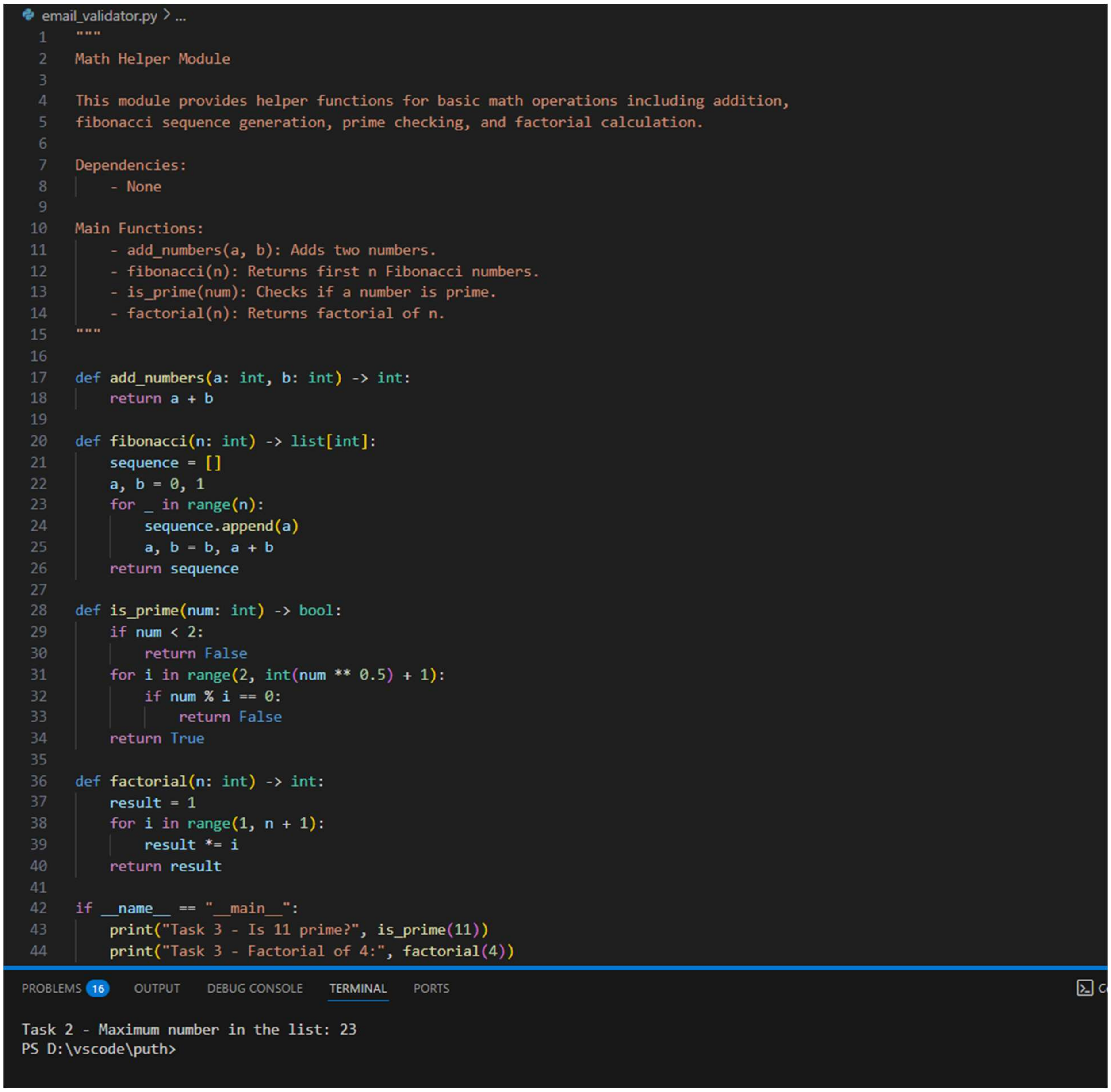
```
PS D:\vscode\puth> python -u "d:\vscode\puth\email_validator.py"
Task 2 - Maximum number in the list: 23
PS D:\vscode\puth>
```

Observation: Inline comments explain the non-trivial logic of updating the Fibonacci sequence in a clear way.

Task 3: Module-Level Documentation

Prompt: “Write a module-level docstring summarizing the purpose, dependencies, and main functions.”

Code:/Output:



```python
"""
Math Helper Module

This module provides helper functions for basic math operations including addition,
fibonacci sequence generation, prime checking, and factorial calculation.

Dependencies:
    - None

Main Functions:
    - add_numbers(a, b): Adds two numbers.
    - fibonacci(n): Returns first n Fibonacci numbers.
    - is_prime(num): Checks if a number is prime.
    - factorial(n): Returns factorial of n.
"""

def add_numbers(a: int, b: int) -> int:
    return a + b

def fibonacci(n: int) -> list[int]:
    sequence = []
    a, b = 0, 1
    for _ in range(n):
        sequence.append(a)
        a, b = b, a + b
    return sequence

def is_prime(num: int) -> bool:
    if num < 2:
        return False
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            return False
    return True

def factorial(n: int) -> int:
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

if __name__ == "__main__":
    print("Task 3 - Is 11 prime?", is_prime(11))
    print("Task 3 - Factorial of 4:", factorial(4))
```

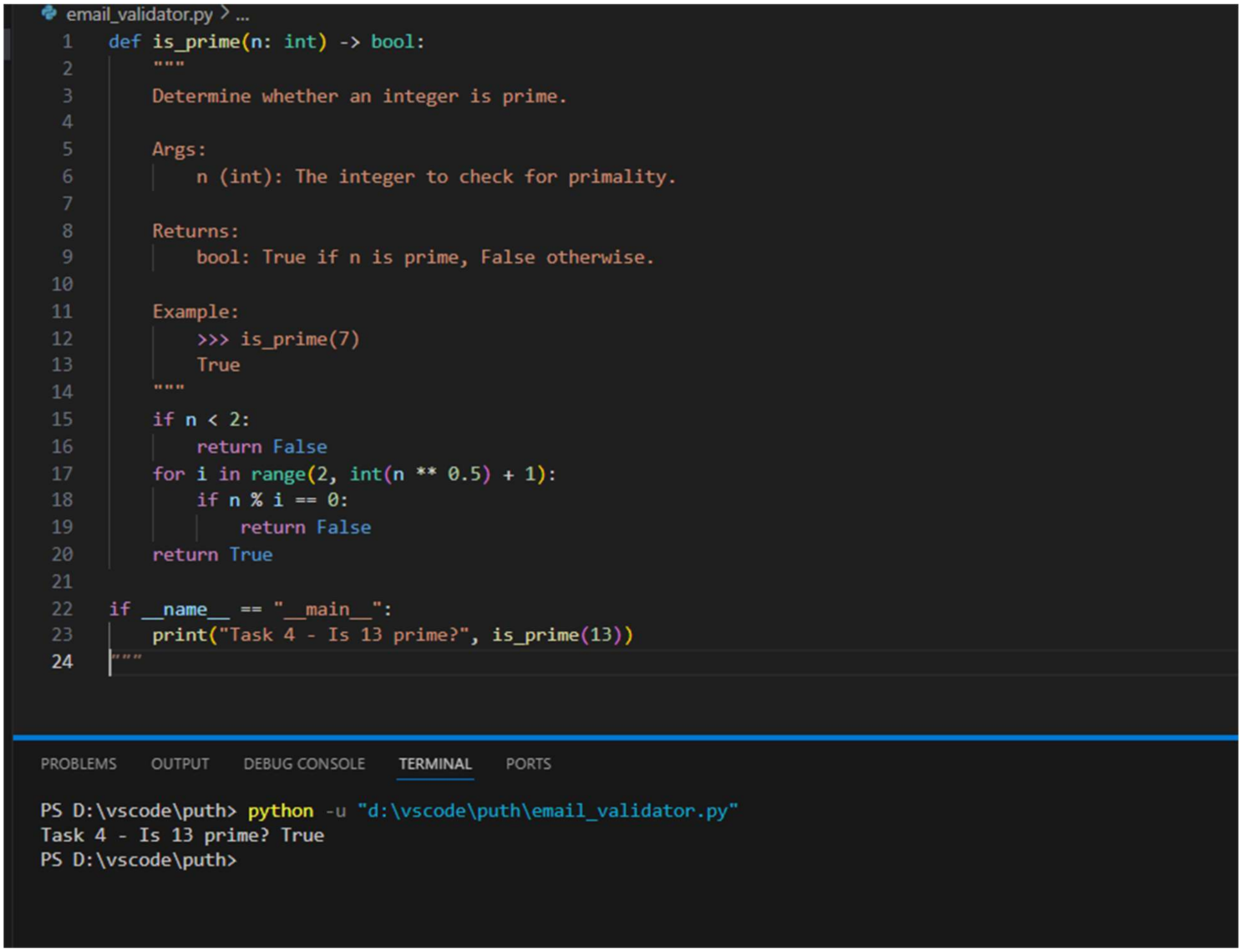
```
Task 2 - Maximum number in the list: 23
PS D:\vscode\puth>
```

Observation: The module-level docstring provides a helpful overview of the file’s purpose and functionality.

Task 4: Convert Comments to Structured Docstrings

Prompt: “Transform existing inline comments into structured Google-style docstrings.”

Code:/Output:



```python
def is_prime(n: int) -> bool:
    """
    Determine whether an integer is prime.

    Args:
        n (int): The integer to check for primality.

    Returns:
        bool: True if n is prime, False otherwise.

    Example:
        >>> is_prime(7)
        True
    """
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True

if __name__ == "__main__":
    print("Task 4 - Is 13 prime?", is_prime(13))
"""
```

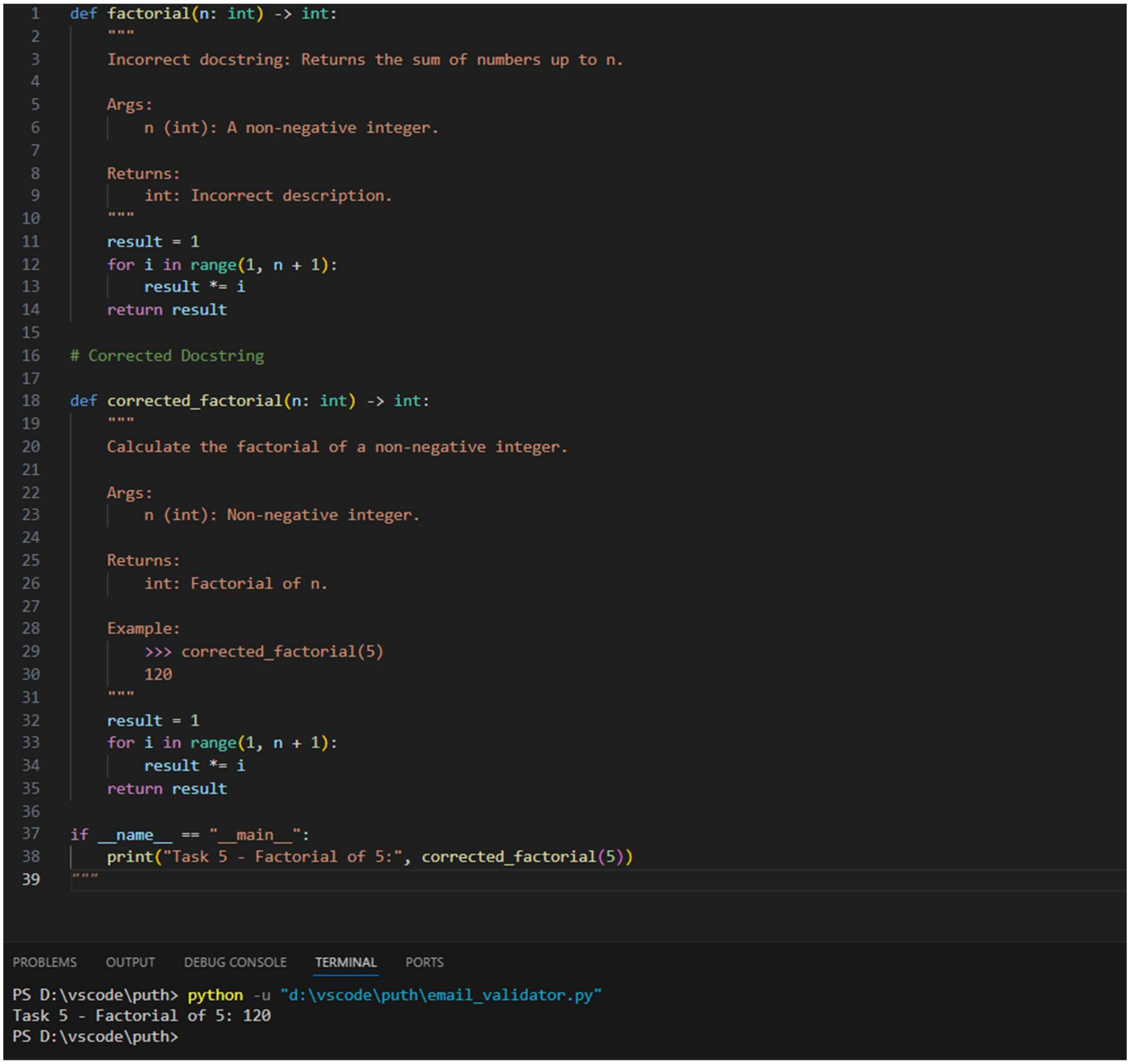
```
PS D:\vscode\puth> python -u "d:\vscode\puth\email_validator.py"
Task 4 - Is 13 prime? True
PS D:\vscode\puth>
```

Observation: Structured docstring provides a clear, standard format for function documentation improving usability and maintainability.

Task 5: Review and Correct Docstrings

Prompt: “Identify and fix inaccuracies in existing docstrings.”

Code:



```python
def factorial(n: int) -> int:
    """
    Incorrect docstring: Returns the sum of numbers up to n.

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

    Returns:
        int: Incorrect description.
    """
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

# Corrected Docstring

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

    Args:
        n (int): Non-negative integer.

    Returns:
        int: Factorial of n.

    Example:
        >>> corrected_factorial(5)
        120
    """
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

if __name__ == "__main__":
    print("Task 5 - Factorial of 5:", corrected_factorial(5))
"""
```

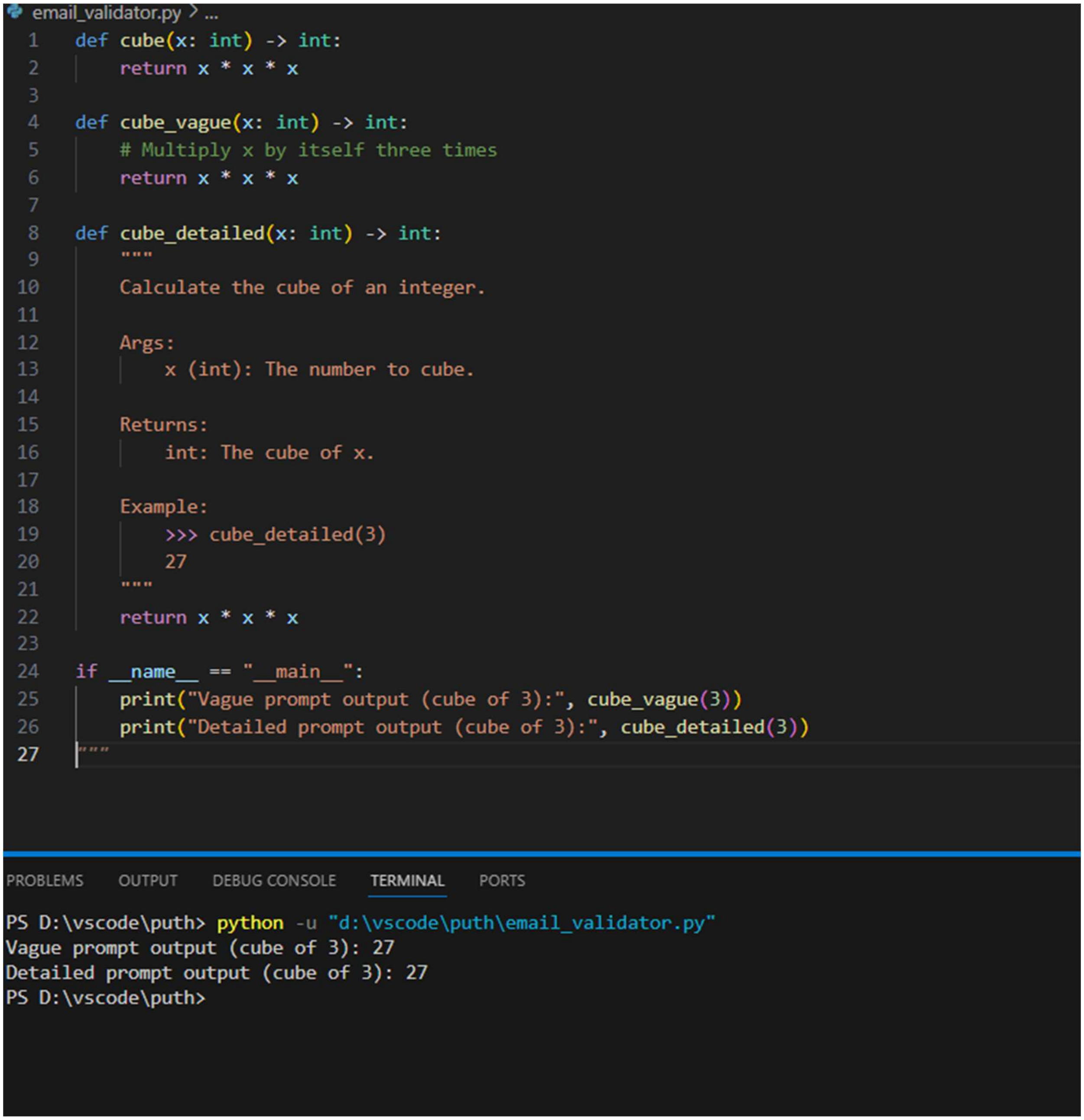
```
PS D:\vscode\puth> python -u "d:\vscode\puth\email_validator.py"
Task 5 - Factorial of 5: 120
PS D:\vscode\puth>
```

Observation: The corrected docstring now accurately reflects the function’s behavior following Google style.

Task 6: Prompt Comparison Experiment

Prompt: - Vague: “Add comments to this function.” - Detailed: “Add a Google-style docstring with description, parameters, return type, and example usage.”

Code:



```python
def cube(x: int) -> int:
    return x * x * x

def cube_vague(x: int) -> int:
    # Multiply x by itself three times
    return x * x * x

def cube_detailed(x: int) -> int:
    """
    Calculate the cube of an integer.

    Args:
        x (int): The number to cube.

    Returns:
        int: The cube of x.

    Example:
        >>> cube_detailed(3)
        27
    """
    return x * x * x

if __name__ == "__main__":
    print("Vague prompt output (cube of 3):", cube_vague(3))
    print("Detailed prompt output (cube of 3):", cube_detailed(3))
"""
```

```
PS D:\vscode\puth> python -u "d:\vscode\puth\email_validator.py"
Vague prompt output (cube of 3): 27
Detailed prompt output (cube of 3): 27
PS D:\vscode\puth>
```

Output:

```
Vague prompt output (cube of 3): 27
Detailed prompt output (cube of 3): 27
```

Observation: The detailed prompt produces a complete Google-style docstring improving function usability, while the vague prompt produces only a simple inline comment.