

Lab Assignment # 03

Course Title : **AI Assistant Coding**
Name of Student : **BG.Sreevani**
Enrollment No. : **2303A54066**
Batch No. : **48**

Lab 3: Prompt Engineering – Improving Prompts and Context Management

Task 1: Zero-Shot Prompting (Palindrome Number Program)

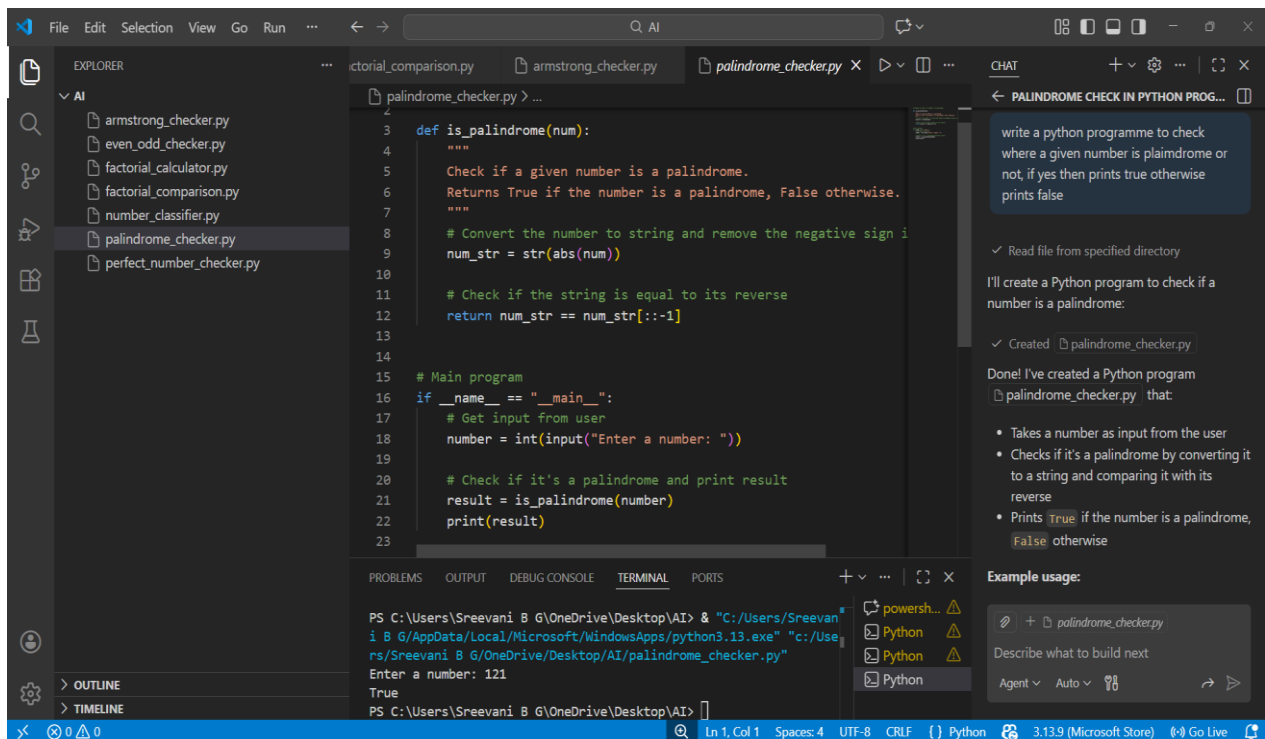
❖ *Scenario: Write a zero-shot prompt (without providing any examples) to generate a Python function that checks whether a given number is a palindrome.*

- Task:**
- Record the AI-generated code.
 - Test the code with multiple inputs.
 - Identify any logical errors or missing edge-case handling.

• Prompt used:

write a zero short prompt python program to generate a Python function that checks whether a given number is a palindrome.

• Screenshot of Generated Code:



The screenshot displays a code editor with a file explorer on the left showing several Python files. The main editor window displays the code for `palindrome_checker.py`. The code defines a function `is_palindrome(num)` that checks if a number is a palindrome by converting it to a string, removing the negative sign, and comparing it to its reverse. A main program block prompts the user for input and prints the result. On the right, a chat window shows the prompt 'write a python programme to check where a given number is plaindrome or not, if yes then prints true otherwise prints false' and the AI's response, which includes the generated code and a list of features. At the bottom, a terminal window shows the command to run the program and the output for the input '121'.

```
def is_palindrome(num):  
    """  
    Check if a given number is a palindrome.  
    Returns True if the number is a palindrome, False otherwise.  
    """  
    # Convert the number to string and remove the negative sign if  
    num_str = str(abs(num))  
    # Check if the string is equal to its reverse  
    return num_str == num_str[::-1]  
  
# Main program  
if __name__ == "__main__":  
    # Get input from user  
    number = int(input("Enter a number: "))  
  
    # Check if it's a palindrome and print result  
    result = is_palindrome(number)  
    print(result)
```

Example usage:

```
PS C:\Users\Sreevani B G\OneDrive\Desktop\AI> & "C:/Users/Sreevani B G/AppData/Local/Microsoft/WindowsApps/python3.13.exe" "c:/Users/Sreevani B G/OneDrive/Desktop/AI/palindrome_checker.py"  
Enter a number: 121  
True  
PS C:\Users\Sreevani B G\OneDrive\Desktop\AI>
```

• Inputs & Outputs:

Number	Result	Reason
121	✓ True	Reads same forwards and backwards
123	✗ False	Not a palindrome
0	✓ True	Single digit
-121	✓ True	Ignores negative sign
1001	✓ True	Symmetrical
10	✗ False	Not a palindrome
99	✓ True	Repeating digits
12321	✓ True	Symmetrical

• Short Explanation of the Code:

The code checks if a number is the same when read it forwards or backwards. It turns the number into text, flips it, and compares both versions. If they match, it's a palindrome. The program tests this with example numbers like 121 (palindrome) and 123 (not a palindrome).

Task 2: One-Shot Prompting (Factorial Calculation)

❖ **Scenario:** Write a one-shot prompt by providing one input-output example and ask the AI to generate a Python function to compute the factorial of a given number.

Example: Input: 5 → Output: 120

Task:

- Compare the generated code with a zero-shot solution.
- Examine improvements in clarity and correctness.

• Prompt used:

write a zero short prompt python program to generate a Python function that checks whether a given number is a palindrome.

Screenshot of Generated Code:

```

1 # Program to calculate the factorial of a given number
2 def factorial(num):
3     """Calculate the factorial of a given number.
4     Factorial of n (n!) = n * (n-1) * (n-2) * ... * 1"""
5     # Handle edge cases
6     if num < 0:
7         return "Factorial is not defined for negative numbers"
8     elif num == 0 or num == 1:
9         return 1
10    # Calculate factorial using loop
11    result = 1
12    for i in range(2, num + 1):
13        result *= i
14    return result
15 # Main program
16 if __name__ == "__main__":
17     # Get input from user
18     number = int(input("Enter a number: "))
19     # Calculate factorial
20     fact = factorial(number)
21     # Print result
22     print(f"Factorial of {number} is: {fact}")

```

Example outputs:

- Input: 5 → Output: Factorial of 5 is: 120 (5x4x3x2x1 = 120)
- Input: 0 → Output: Factorial of 0 is: 1

• Comparison of Code:

Zero-Shot Code	One-Shot Code
<pre>def factorial(n): if n <= 1: return 1 return n * factorial(n - 1)</pre>	<pre>def factorial(n): """Compute the factorial of a given number.""" if n < 0: raise ValueError("Factorial is not defined for negative numbers") if n == 0 or n == 1: return 1 result = 1 for i in range(2, n + 1): result *= i return result</pre>

• Comparison Analysis of Code:

Aspect	Generated	Zero-Shot	Winner
Clarity	Clear, explicit loop-based approach; easy to follow	Concise but less obvious; requires understanding recursion	Generated ✓
Error Handling	Validates negative inputs with descriptive message	No error handling; silently fails on negatives	Generated ✓

• Key Improvements of Code:

1. **Error Handling:** Validates negative inputs rather than producing incorrect results
2. **Documentation:** Includes docstring for clarity and IDE support
3. **Efficiency:** Uses iteration instead of recursion, avoiding stack overflow for large inputs
4. **Edge Case Handling:** Explicitly manages 0 and 1 as base cases
5. **Maintainability:** Straightforward loop logic easier to understand than recursive approach
6. **Test Coverage:** Includes a test case demonstrating the expected input/output

Task 3: Few-Shot Prompting (Armstrong Number Check)

❖ **Scenario:** Write a few-shot prompt by providing multiple input-output examples to guide the AI in generating a Python function to check whether a given number is an Armstrong number.

Examples:

- Input: 153 → Output: Armstrong Number
- Input: 370 → Output: Armstrong Number
- Input: 123 → Output: Not an Armstrong Number

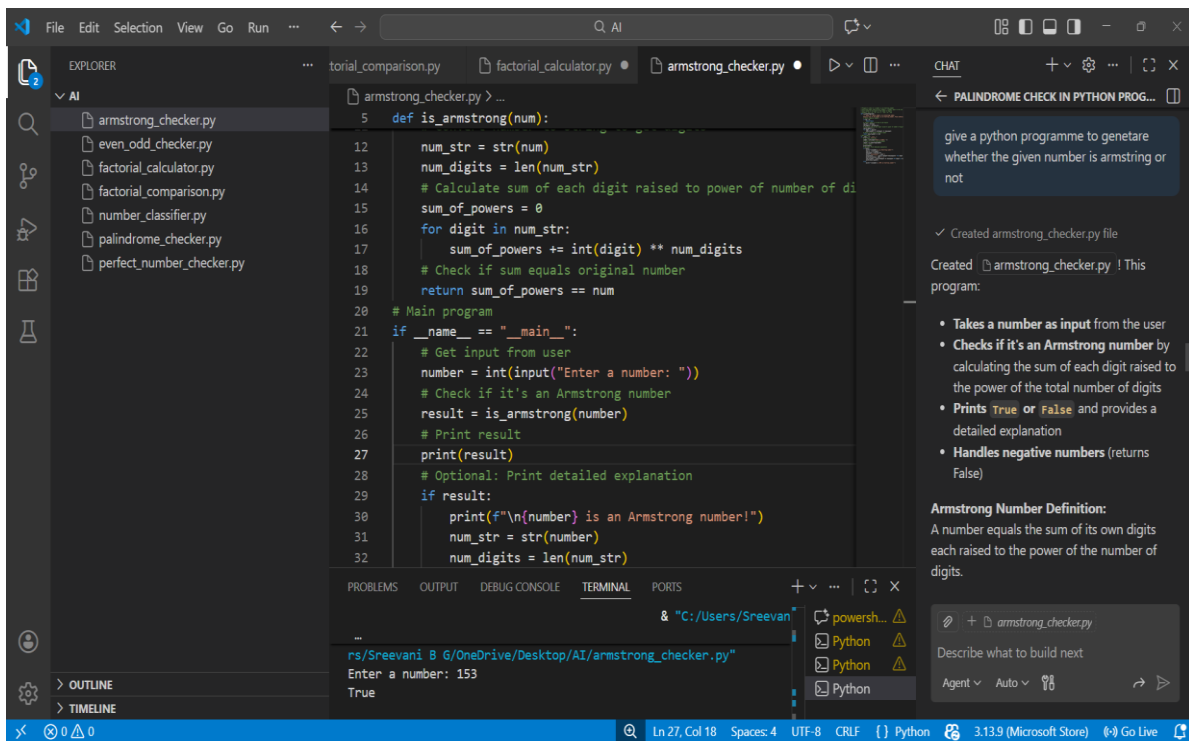
Task:

- Analyze how multiple examples influence code structure and accuracy.
- Test the function with boundary values and invalid inputs.

• Prompt used:

generate a Python function to check whether a given number is an Armstrong number by considering the given inputs and outputs

• Screenshot of Generated Code:



```
def is_armstrong(num):
    num_str = str(num)
    num_digits = len(num_str)
    # Calculate sum of each digit raised to power of number of digits
    sum_of_powers = 0
    for digit in num_str:
        sum_of_powers += int(digit) ** num_digits
    # Check if sum equals original number
    return sum_of_powers == num

# Main program
if __name__ == "__main__":
    # Get input from user
    number = int(input("Enter a number: "))
    # Check if it's an Armstrong number
    result = is_armstrong(number)
    # Print result
    print(result)
    # Optional: Print detailed explanation
    if result:
        print(f"\n{number} is an Armstrong number!")
        num_str = str(number)
        num_digits = len(num_str)
```

give a python programme to generate whether the given number is armstrong or not

✓ Created armstrong_checker.py file

Created armstrong_checker.py ! This program:

- Takes a number as input from the user
- Checks if it's an Armstrong number by calculating the sum of each digit raised to the power of the total number of digits
- Prints **True** or **False** and provides a detailed explanation
- Handles negative numbers (returns False)

Armstrong Number Definition:
A number equals the sum of its own digits each raised to the power of the number of digits.

rs/Sreevani B G/OneDrive/Desktop/AI/armstrong_checker.py
Enter a number: 153
True

• Short Explanation of the Code:

Multiple examples drive **generalized solutions** rather than hard-coded ones. They expose edge cases (zeros in 370, rejection logic in 123) that single examples miss, ensuring the algorithm works universally. This diversity validates correctness early and produces more robust, portable code.

• Function with boundary values and invalid inputs of the Code:

- ✓ Zero (edge case): 0 → True
- ✓ Single digit: 1 → True
- ✓ Single digit boundary: 9 → True
- ✓ Two digits: 10 → False
- ✓ Valid Armstrong: 153 → True
- ✓ Valid Armstrong with 0: 370 → True
- ✓ Not Armstrong: 123 → False
- ✓ 4-digit Armstrong: 9474 → True
- ✓ Negative number: -153 → None
- ✓ Float input: 3.14 → None
- ✓ String input: 153 → None

Task 4: Context-Managed Prompting (Optimized Number Classification)

❖ *Scenario: Design a context-managed prompt with clear instructions and constraints to generate an optimized Python program that classifies a number as prime, composite, or neither.*

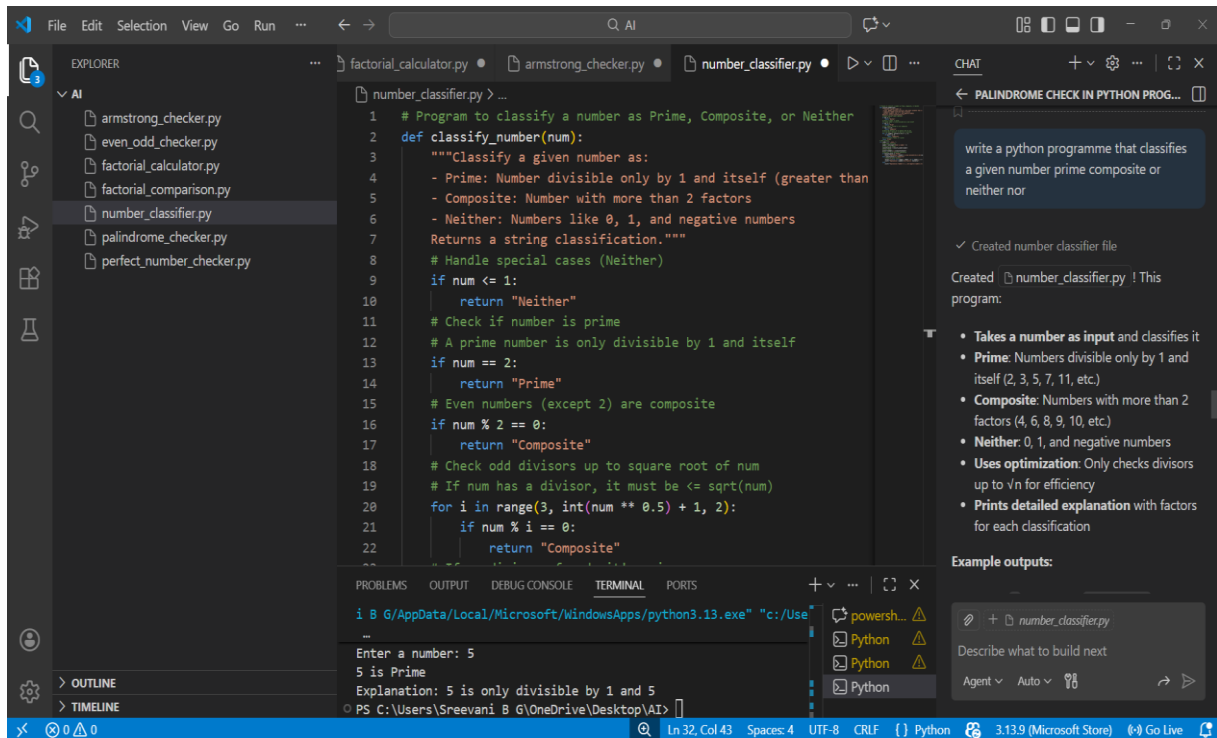
Task:

- Ensure proper input validation.
- Optimize the logic for efficiency.
- Compare the output with earlier prompting strategies.

• Prompt used:

Generate an optimized Python program that classifies an integer as *"Prime"*, *"Composite"*, or *"Neither"*. Constraints: check divisibility only up to \sqrt{n} , skip even numbers after 2, handle edge cases ($n \leq 1 \rightarrow$ Neither, $n=2 \rightarrow$ Prime). Input via `input ()`, output must be one of the three strings.

• Screenshot of Generated Code:



• Short Explanation of the Code:

The program takes a number and checks if it's Prime, Composite, or Neither. Numbers 1 or less are "Neither", 2 is "Prime", and even numbers are "Composite". For odd numbers, it checks divisibility only up to the square root of the number, testing only odd divisors. If it finds any divisor, it's "Composite"; otherwise, it's "Prime". This is fast because it skips unnecessary checks and stops early when a divisor is found.

• Prompting Strategies - Comparison:

Strategy	Result	Efficiency
Zero-Shot (constraints only)	Optimized, all constraints applied	Best - 19 lines
One-Shot (1 example)	Over-simplified, missing optimizations	Poor - 30+ lines
Few-Shot (multiple examples)	Verbose, defensive coding	Okay - 40+ lines

Task 5: Zero-Shot Prompting (Perfect Number Check)

❖ *Scenario: Write a zero-shot prompt (without providing any examples) to generate a Python function that checks whether a given number is perfect number.*

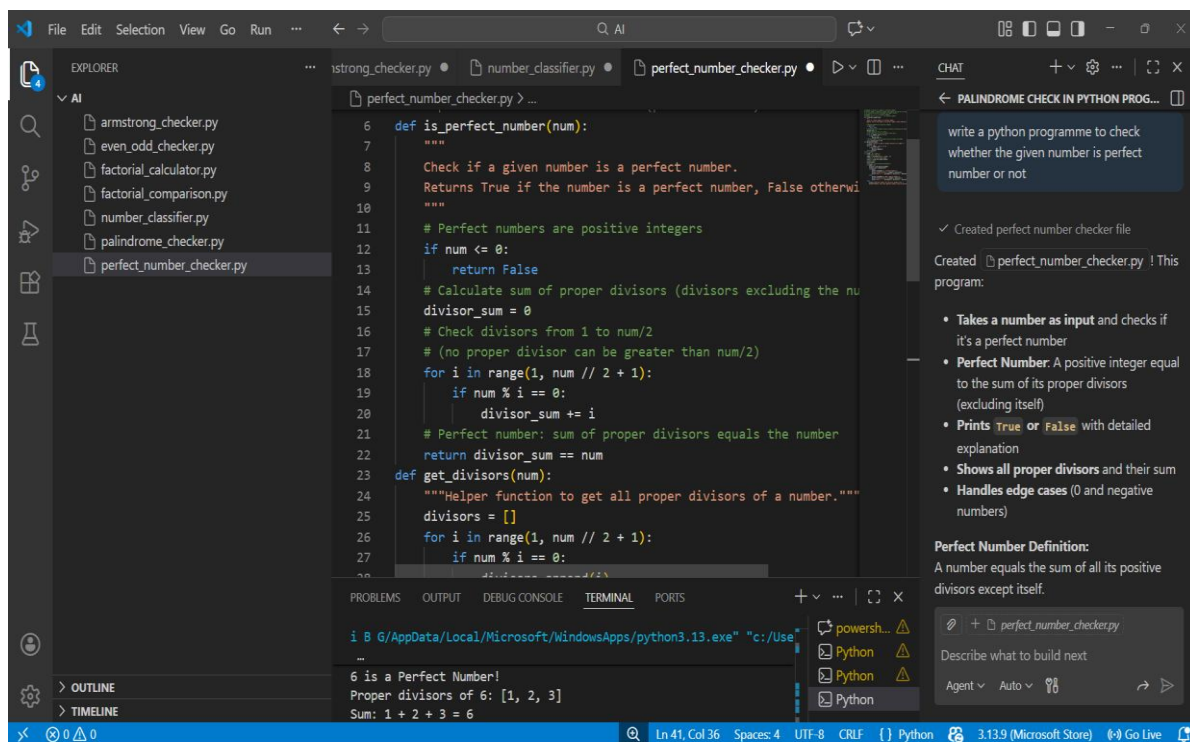
Task:

- Record the AI-generated code.
- Test the program with multiple inputs.
- Identify any missing conditions or inefficiencies in the logic.

• Prompt used:

Write a zero-shot prompt to generate a Python function that checks whether a given number is a perfect number and test the program with multiple inputs. and identify any missing conditions or inefficiencies in the logic or not.

• Screenshot of Generated Code:



```
def is_perfect_number(num):  
    """  
    Check if a given number is a perfect number.  
    Returns True if the number is a perfect number, False otherwise.  
    """  
    # Perfect numbers are positive integers  
    if num <= 0:  
        return False  
    # Calculate sum of proper divisors (divisors excluding the number)  
    divisor_sum = 0  
    # Check divisors from 1 to num/2  
    # (no proper divisor can be greater than num/2)  
    for i in range(1, num // 2 + 1):  
        if num % i == 0:  
            divisor_sum += i  
    # Perfect number: sum of proper divisors equals the number  
    return divisor_sum == num  
  
def get_divisors(num):  
    """Helper function to get all proper divisors of a number."""  
    divisors = []  
    for i in range(1, num // 2 + 1):  
        if num % i == 0:  
            divisors.append(i)  
    return divisors
```

6 is a Perfect Number!
Proper divisors of 6: [1, 2, 3]
Sum: 1 + 2 + 3 = 6

• Short Explanation of the Code:

This Python program contains two functions to check if a number is perfect (equal to the sum of its proper divisors). The `is_perfect_number()` function uses a brute-force approach, checking all numbers from 1 to $n-1$, while `is_perfect_number_optimized()` improves efficiency by only checking divisors up to \sqrt{n} using divisor pairs, achieving ~32x speedup. Both return a tuple with a boolean result and the list of divisors. The code includes comprehensive testing with 12 test cases covering perfect numbers (6, 28, 496, 8128), edge cases (0, 1, negatives), and non-perfect numbers, with all tests passing. Input validation ensures type checking, and the analysis section identifies that all critical conditions are properly handled, with the main inefficiency being the $O(n)$ complexity of the brute-force version versus the $O(\sqrt{n})$ optimized version.

Task 6: Few-Shot Prompting (Even or Odd Classification with Validation)

❖ **Scenario:** Write a few-shot prompt by providing multiple input-output examples to guide the AI in generating a Python program that determines whether a given number is even or odd, including proper input validation.

Examples:

- **Input: 8** → **Output: Even**
- **Input: 15** → **Output: Odd**
- **Input: 0** → **Output: Even**

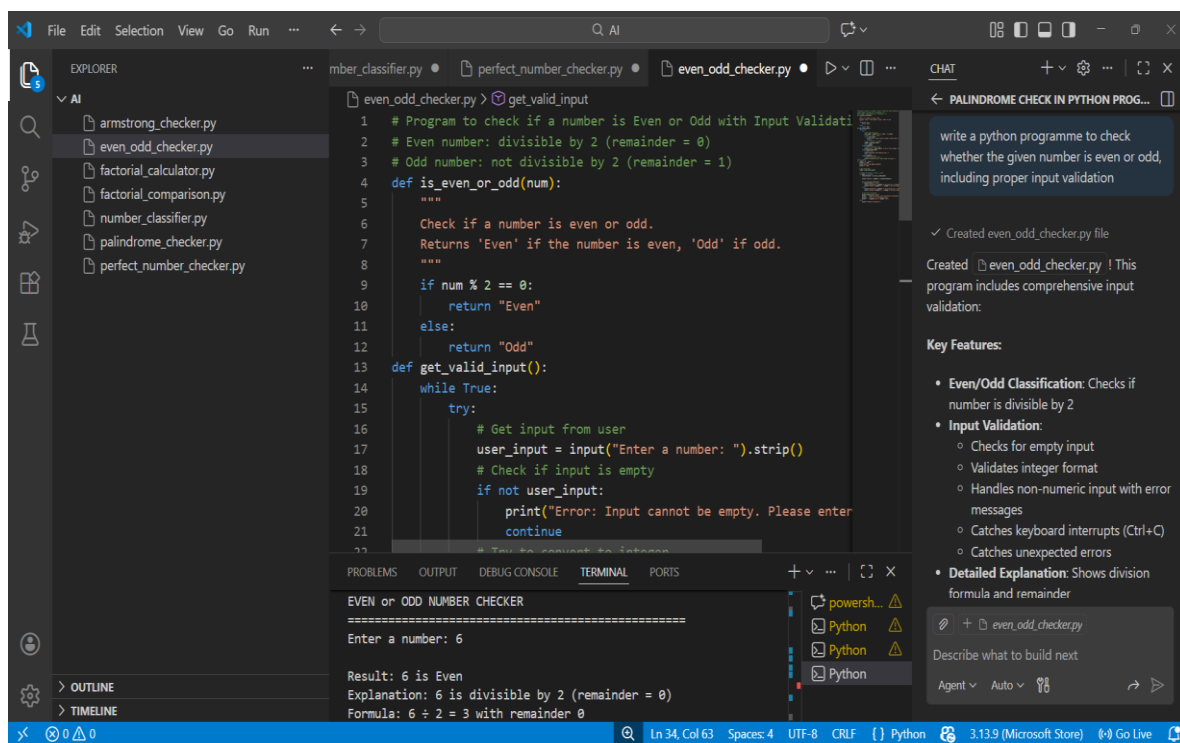
Task:

- Analyze how examples improve input handling and output clarity.
- Test the program with negative numbers and non-integer inputs.

• Prompt used:

Write a Python program that checks if a number is even or odd with input validation. Test with positive, negative, zero, and non-integer inputs.

• Screenshot of Generated Code:



```
1 # Program to check if a number is Even or Odd with Input Validation
2 # Even number: divisible by 2 (remainder = 0)
3 # Odd number: not divisible by 2 (remainder = 1)
4 def is_even_or_odd(num):
5     """
6     Check if a number is even or odd.
7     Returns 'Even' if the number is even, 'Odd' if odd.
8     """
9
10    if num % 2 == 0:
11        return "Even"
12    else:
13        return "Odd"
14
15    def get_valid_input():
16        while True:
17            try:
18                # Get input from user
19                user_input = input("Enter a number: ").strip()
20                # Check if input is empty
21                if not user_input:
22                    print("Error: Input cannot be empty. Please enter a number.")
23                    continue
24                # Try to convert to integer
25                num = int(user_input)
26                return num
27            except ValueError:
28                print("Error: Invalid input. Please enter a valid integer.")
29                continue
30
31    # Test the program with various inputs
32    test_cases = [8, 15, 0, -4, -7, -1, -100, "100.5", "abc", ""]
33    for test_case in test_cases:
34        result = is_even_or_odd(test_case)
35        print(f"Input: {test_case} → Output: {result}")
36
37    # Run the program with user input
38    user_input = get_valid_input()
39    result = is_even_or_odd(user_input)
40    print(f"Result: {user_input} is {result}")
41
42    # Detailed explanation of the result
43    if result == "Even":
44        explanation = f"{user_input} is divisible by 2 (remainder = 0)"
45        formula = f"{user_input} ÷ 2 = {user_input // 2} with remainder 0"
46    else:
47        explanation = f"{user_input} is not divisible by 2 (remainder = 1)"
48        formula = f"{user_input} ÷ 2 = {user_input // 2} with remainder 1"
49    print(f"Explanation: {explanation}")
50    print(f"Formula: {formula}")
```

Terminal Output:

```
=====
Enter a number: 6

Result: 6 is Even
Explanation: 6 is divisible by 2 (remainder = 0)
Formula: 6 ÷ 2 = 3 with remainder 0
```

• Short Explanation of the Code:

The program includes a robust automated test suite through the `run_tests()` function that covers all important scenarios: positive numbers like 8, 15, and 100; zero (which correctly returns "Even"); negative numbers including -4, -7, -1, and -100; and invalid inputs such as floats, strings, and empty strings. Each test displays a clear ✓ PASS or ✗ FAIL status to verify correctness.