LAB ASSIGNMENT - 3.1

NAME: Mohamed safwan farooqi

HTNO:2303A52246

BATCH: 36

Question 1:

Zero-Shot Prompting (Palindrome Number Program)

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

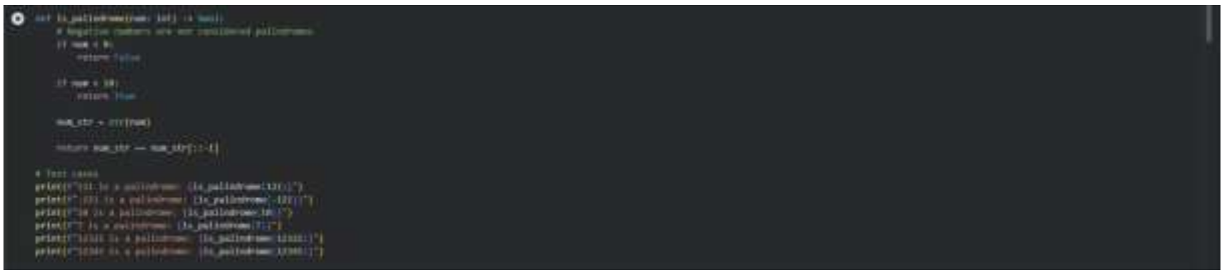
Task:

• Record the AI-generated code.

• Test the code with multiple inputs.

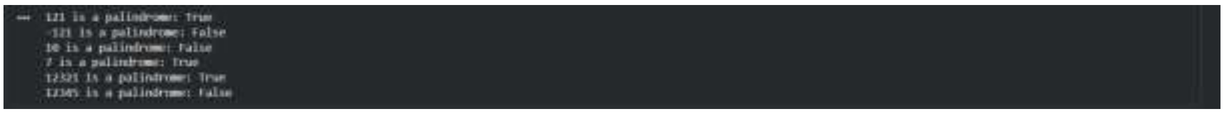
• Identify any logical errors or missing edge-case handling.

CODE:



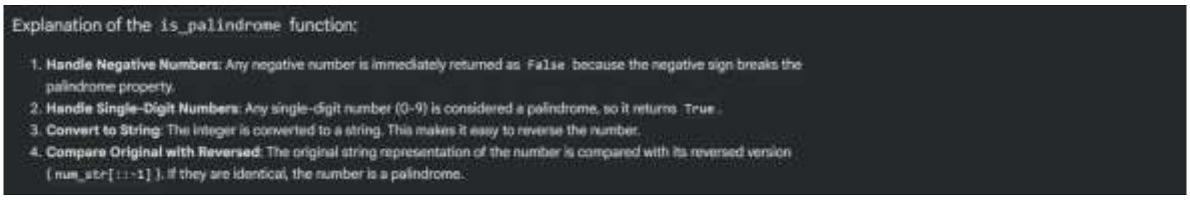
```python
def is_palindrome(num: int) -> bool:
    # Negative numbers are not considered palindromes
    if num < 0:
        return False

    if num < 10:
        return True

    num_str = str(num)

    return num_str == num_str[::-1]

# Test cases
print(f"121 is a palindrome: {is_palindrome(121)}")
print(f"-121 is a palindrome: {is_palindrome(-121)}")
print(f"10 is a palindrome: {is_palindrome(10)}")
print(f"7 is a palindrome: {is_palindrome(7)}")
print(f"12321 is a palindrome: {is_palindrome(12321)}")
print(f"12345 is a palindrome: {is_palindrome(12345)}")
```

OUTPUT:



```
121 is a palindrome: True
-121 is a palindrome: False
10 is a palindrome: False
7 is a palindrome: True
12321 is a palindrome: True
12345 is a palindrome: False
```

EXPLANATION:



Explanation of the `is_palindrome` function:

1. **Handle Negative Numbers:** Any negative number is immediately returned as `False` because the negative sign breaks the palindrome property.
2. **Handle Single-Digit Numbers:** Any single-digit number (0-9) is considered a palindrome, so it returns `True`.
3. **Convert to String:** The integer is converted to a string. This makes it easy to reverse the number.
4. **Compare Original with Reversed:** The original string representation of the number is compared with its reversed version (`num_str[::-1]`). If they are identical, the number is a palindrome.

Question 2:

One-Shot Prompting (Factorial Calculation)

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:

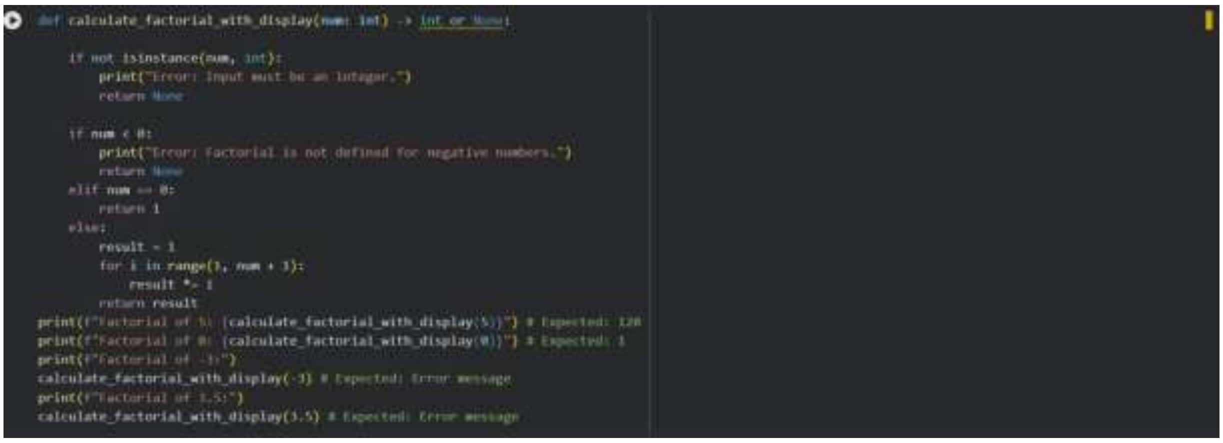
Input: 5 → Output: 120

Task:

• Compare the generated code with a zero-shot solution.

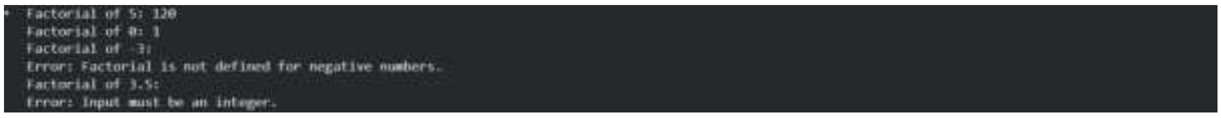
• Examine improvements in clarity and correctness.

CODE:



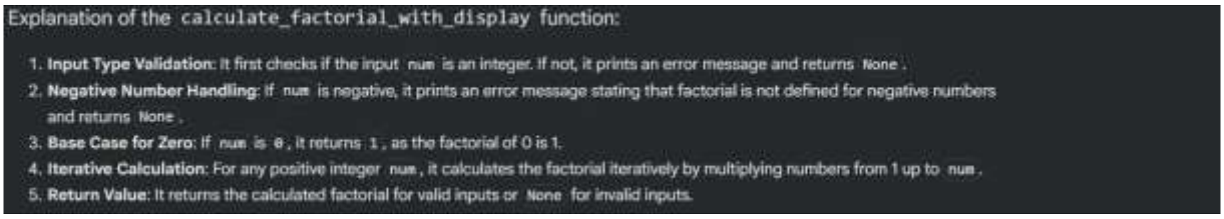
```python
def calculate_factorial_with_display(num: int) -> int or None:

    if not isinstance(num, int):
        print("Error: Input must be an integer.")
        return None

    if num < 0:
        print("Error: Factorial is not defined for negative numbers.")
        return None
    elif num == 0:
        return 1
    else:
        result = 1
        for i in range(1, num + 1):
            result *= i
        return result
print(f"Factorial of 5: {calculate_factorial_with_display(5)}") # Expected: 120
print(f"Factorial of 0: {calculate_factorial_with_display(0)}") # Expected: 1
print(f"Factorial of -3:")
calculate_factorial_with_display(-3) # Expected: Error message
print(f"Factorial of 3.5:")
calculate_factorial_with_display(3.5) # Expected: Error message
```

OUTPUT:



```
Factorial of 5: 120
Factorial of 0: 1
Factorial of -3:
Error: Factorial is not defined for negative numbers.
Factorial of 3.5:
Error: Input must be an integer.
```

EXPLANATION:


Explanation of the `calculate_factorial_with_display` function:

1. **Input Type Validation:** It first checks if the input `num` is an integer. If not, it prints an error message and returns `None`.
2. **Negative Number Handling:** If `num` is negative, it prints an error message stating that factorial is not defined for negative numbers and returns `None`.
3. **Base Case for Zero:** If `num` is `0`, it returns `1`, as the factorial of 0 is 1.
4. **Iterative Calculation:** For any positive integer `num`, it calculates the factorial iteratively by multiplying numbers from 1 up to `num`.
5. **Return Value:** It returns the calculated factorial for valid inputs or `None` for invalid inputs.

Question 3:

Few-Shot Prompting (Armstrong Number Check)

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

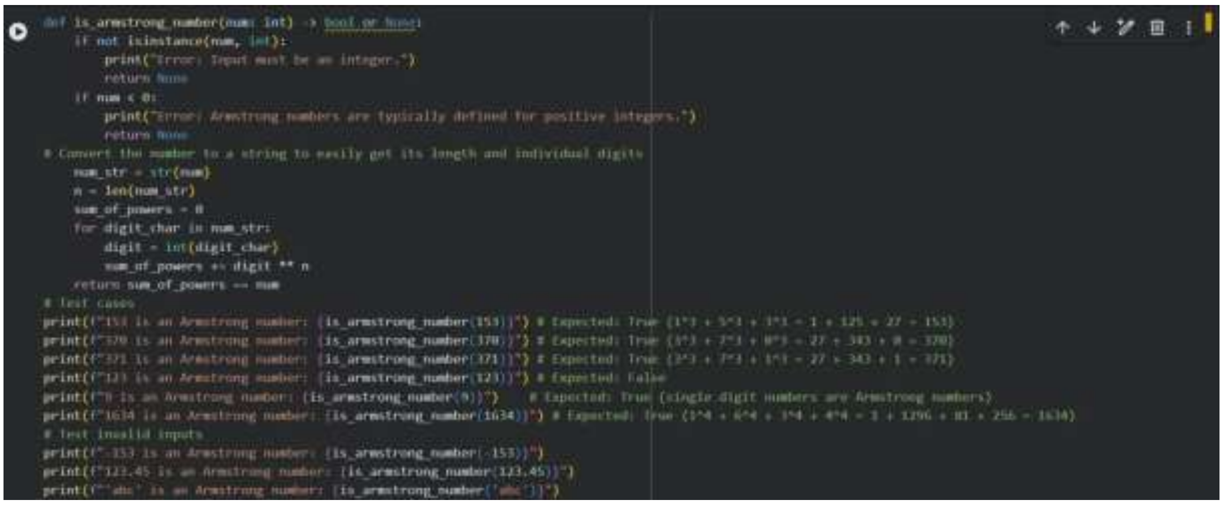
• Input: 370 → Output: Armstrong Number • Input: 123 → Output: Not an Armstrong Number

Task:

• Analyze how multiple examples influence code structure and accuracy.

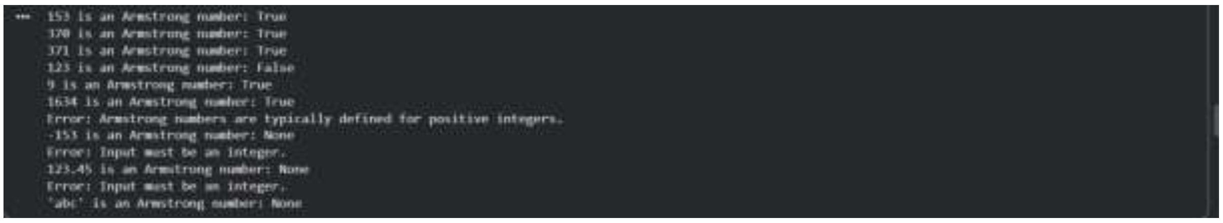
• Test the function with boundary values and invalid inputs.

CODE:



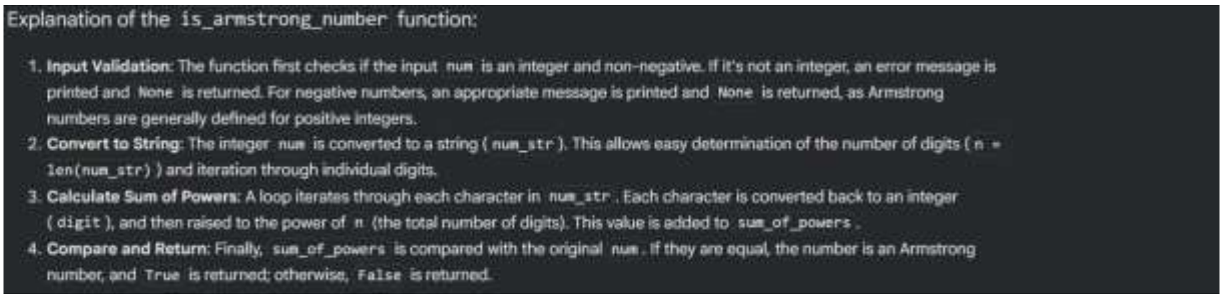
```python
def is_armstrong_number(num: int) -> bool or None:
    if not isinstance(num, int):
        print("Error: Input must be an integer.")
        return None
    if num < 0:
        print("Error: Armstrong numbers are typically defined for positive integers.")
        return None
# Convert the number to a string to easily get its length and individual digits
    num_str = str(num)
    n = len(num_str)
    sum_of_powers = 0
    for digit_char in num_str:
        digit = int(digit_char)
        sum_of_powers += digit ** n
    return sum_of_powers == num
# Test cases
print(f"153 is an Armstrong number: {is_armstrong_number(153)}") # Expected: True (1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153)
print(f"370 is an Armstrong number: {is_armstrong_number(370)}") # Expected: True (3^3 + 7^3 + 0^3 = 27 + 343 + 0 = 370)
print(f"371 is an Armstrong number: {is_armstrong_number(371)}") # Expected: True (3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371)
print(f"123 is an Armstrong number: {is_armstrong_number(123)}") # Expected: False
print(f"9 is an Armstrong number: {is_armstrong_number(9)}")     # Expected: True (single digit numbers are Armstrong numbers)
print(f"1634 is an Armstrong number: {is_armstrong_number(1634)}") # Expected: True (1^4 + 6^4 + 3^4 + 4^4 = 1 + 1296 + 81 + 256 = 1634)
# Test invalid inputs
print(f"-153 is an Armstrong number: {is_armstrong_number(-153)}")
print(f"123.45 is an Armstrong number: {is_armstrong_number(123.45)}")
print(f"'abc' is an Armstrong number: {is_armstrong_number('abc')}")
```

OUTPUT:



```
153 is an Armstrong number: True
370 is an Armstrong number: True
371 is an Armstrong number: True
123 is an Armstrong number: False
9 is an Armstrong number: True
1634 is an Armstrong number: True
Error: Armstrong numbers are typically defined for positive integers.
-153 is an Armstrong number: None
Error: Input must be an integer.
123.45 is an Armstrong number: None
Error: Input must be an integer.
'abc' is an Armstrong number: None
```

EXPLANATION:



Explanation of the `is_armstrong_number` function:

1. **Input Validation:** The function first checks if the input `num` is an integer and non-negative. If it's not an integer, an error message is printed and `None` is returned. For negative numbers, an appropriate message is printed and `None` is returned, as Armstrong numbers are generally defined for positive integers.
2. **Convert to String:** The integer `num` is converted to a string (`num_str`). This allows easy determination of the number of digits (`n = len(num_str)`) and iteration through individual digits.
3. **Calculate Sum of Powers:** A loop iterates through each character in `num_str` . Each character is converted back to an integer (`digit`), and then raised to the power of `n` (the total number of digits). This value is added to `sum_of_powers` .
4. **Compare and Return:** Finally, `sum_of_powers` is compared with the original `num` . If they are equal, the number is an Armstrong number, and `True` is returned; otherwise, `False` is returned.

Question 4:

Context-Managed Prompting (Optimized Number

Classification)

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.

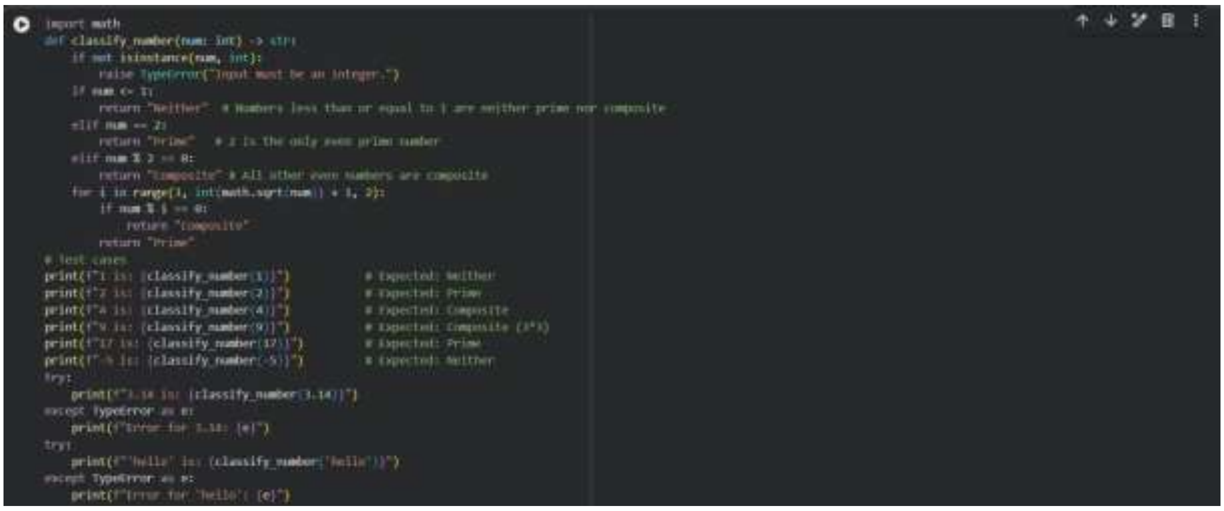
Task:

• Ensure proper input validation.

• Optimize the logic for efficiency.

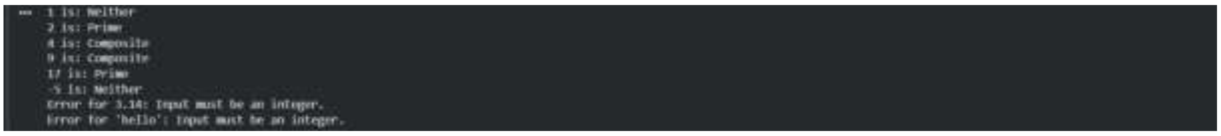
• Compare the output with earlier prompting strategies.

CODE:



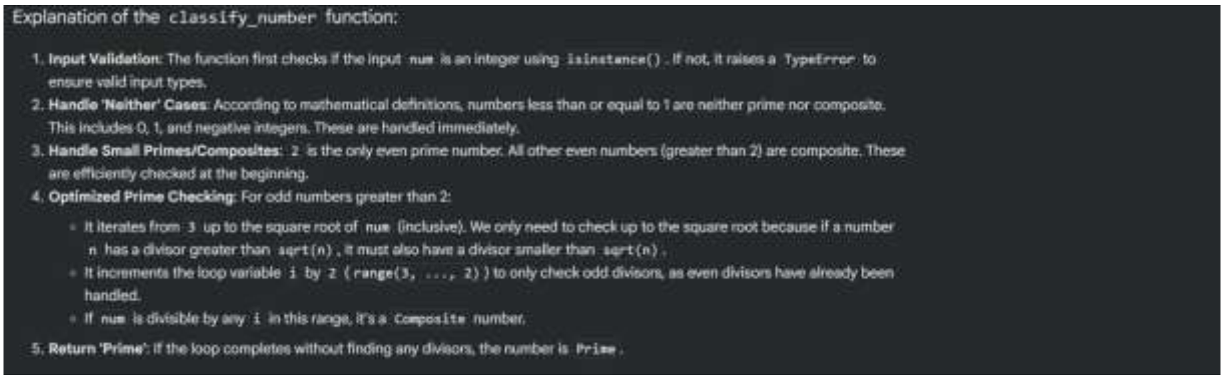
```python
import math
def classify_number(num: int) -> str:
    if not isinstance(num, int):
        raise TypeError("Input must be an integer.")
    if num <= 1:
        return "Neither"  # Numbers less than or equal to 1 are neither prime nor composite
    elif num == 2:
        return "Prime"   # 2 is the only even prime number
    elif num % 2 == 0:
        return "Composite" # All other even numbers are composite
    for i in range(3, int(math.sqrt(num)) + 1, 2):
        if num % i == 0:
            return "Composite"
        return "Prime"
# Test cases
print(f"1 is: {classify_number(1)}")          # Expected: Neither
print(f"2 is: {classify_number(2)}")          # Expected: Prime
print(f"4 is: {classify_number(4)}")          # Expected: Composite
print(f"9 is: {classify_number(9)}")          # Expected: Composite (3*3)
print(f"17 is: {classify_number(17)}")        # Expected: Prime
print(f"-5 is: {classify_number(-5)}")        # Expected: Neither
try:
    print(f"3.14 is: {classify_number(3.14)}")
except TypeError as e:
    print(f"Error for 3.14: {e}")
try:
    print(f"'hello' is: {classify_number('hello')}")
except TypeError as e:
    print(f"Error for 'hello': {e}")
```

OUTPUT:



```
1 is: Neither
2 is: Prime
4 is: Composite
9 is: Composite
17 is: Prime
-5 is: Neither
Error for 3.14: Input must be an integer.
Error for 'hello': Input must be an integer.
```

EXPLANATION:



Explanation of the `classify_number` function:

1. **Input Validation**: The function first checks if the input `num` is an integer using `isinstance()`. If not, it raises a `TypeError` to ensure valid input types.
2. **Handle 'Neither' Cases**: According to mathematical definitions, numbers less than or equal to 1 are neither prime nor composite. This includes 0, 1, and negative integers. These are handled immediately.
3. **Handle Small Primes/Composites**: `2` is the only even prime number. All other even numbers (greater than 2) are composite. These are efficiently checked at the beginning.
4. **Optimized Prime Checking**: For odd numbers greater than 2:
   - It iterates from `3` up to the square root of `num` (inclusive). We only need to check up to the square root because if a number `n` has a divisor greater than `sqrt(n)`, it must also have a divisor smaller than `sqrt(n)`.
   - It increments the loop variable `i` by `2` (`range(3, ..., 2)`) to only check odd divisors, as even divisors have already been handled.
   - If `num` is divisible by any `i` in this range, it's a `Composite` number.
5. **Return 'Prime'**: If the loop completes without finding any divisors, the number is `Prime`.

Question 5:

Zero-Shot Prompting (Perfect Number Check)

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

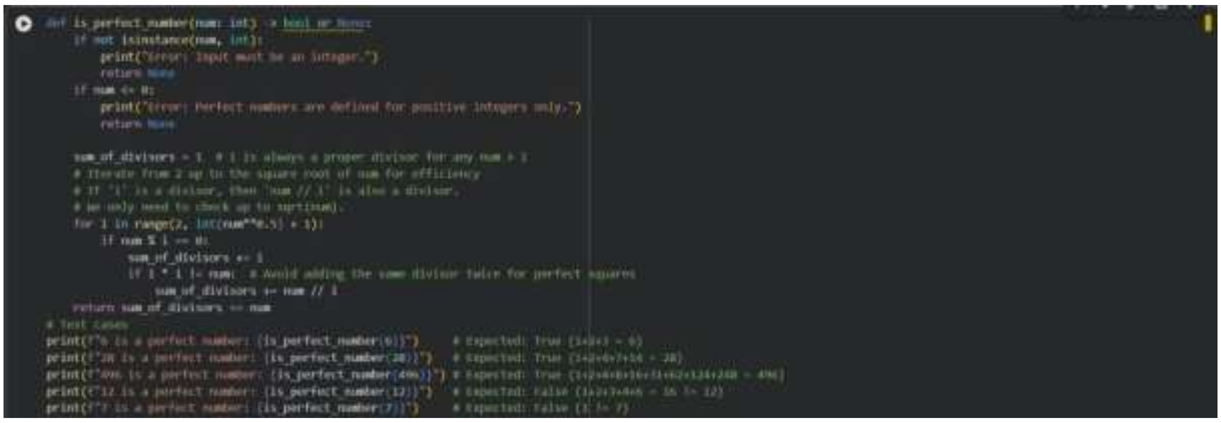
Task:

• Record the AI-generated code.

• Test the program with multiple inputs.

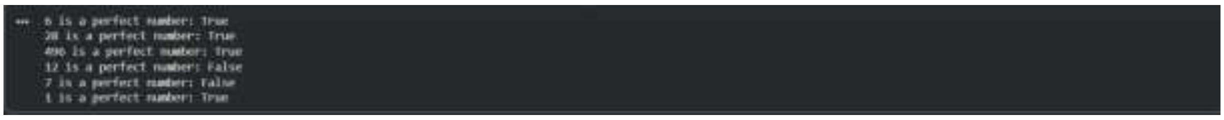
• Identify any missing conditions or inefficiencies in the logic.

CODE:



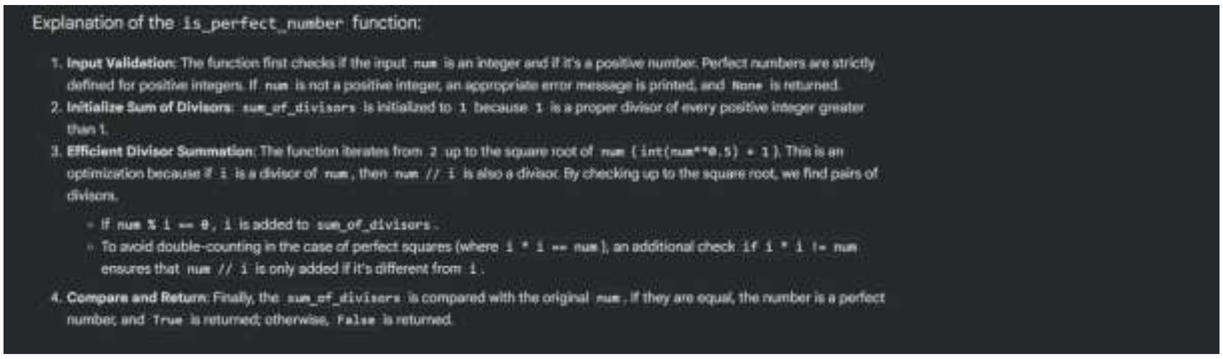
```python
def is_perfect_number(num: int) -> bool or None:
    if not isinstance(num, int):
        print("Error: Input must be an integer.")
        return None
    if num <= 0:
        print("Error: Perfect numbers are defined for positive integers only.")
        return None

    sum_of_divisors = 1  # 1 is always a proper divisor for any num > 1
    # Iterate from 2 up to the square root of num for efficiency
    # If `i` is a divisor, then `num // i` is also a divisor.
    # We only need to check up to sqrt(num).
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            sum_of_divisors += i
            if i * i != num:  # Avoid adding the same divisor twice for perfect squares
                sum_of_divisors += num // i
    return sum_of_divisors == num
# Test cases
print(f"6 is a perfect number: {is_perfect_number(6)}")     # Expected: True (1+2+3 = 6)
print(f"28 is a perfect number: {is_perfect_number(28)}")   # Expected: True (1+2+4+7+14 = 28)
print(f"496 is a perfect number: {is_perfect_number(496)}") # Expected: True (1+2+4+8+16+31+62+124+248 = 496)
print(f"12 is a perfect number: {is_perfect_number(12)}")   # Expected: False (1+2+3+4+6 = 16 != 12)
print(f"7 is a perfect number: {is_perfect_number(7)}")     # Expected: False (1 != 7)
```

OUTPUT:



```
6 is a perfect number: True
28 is a perfect number: True
496 is a perfect number: True
12 is a perfect number: False
7 is a perfect number: False
1 is a perfect number: True
```

EXPLANATION:



Explanation of the `is_perfect_number` function:

1. **Input Validation:** The function first checks if the input `num` is an integer and if it's a positive number. Perfect numbers are strictly defined for positive integers. If `num` is not a positive integer, an appropriate error message is printed, and `None` is returned.
2. **Initialize Sum of Divisors:** `sum_of_divisors` is initialized to `1` because `1` is a proper divisor of every positive integer greater than 1.
3. **Efficient Divisor Summation:** The function iterates from `2` up to the square root of `num` (`int(num**0.5) + 1`). This is an optimization because if `i` is a divisor of `num`, then `num // i` is also a divisor. By checking up to the square root, we find pairs of divisors.
    - If `num % i == 0`, `i` is added to `sum_of_divisors`.
    - To avoid double-counting in the case of perfect squares (where `i * i == num`), an additional check `if i * i != num` ensures that `num // i` is only added if it's different from `i`.
4. **Compare and Return:** Finally, the `sum_of_divisors` is compared with the original `num`. If they are equal, the number is a perfect number, and `True` is returned; otherwise, `False` is returned.

Question 6:

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

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

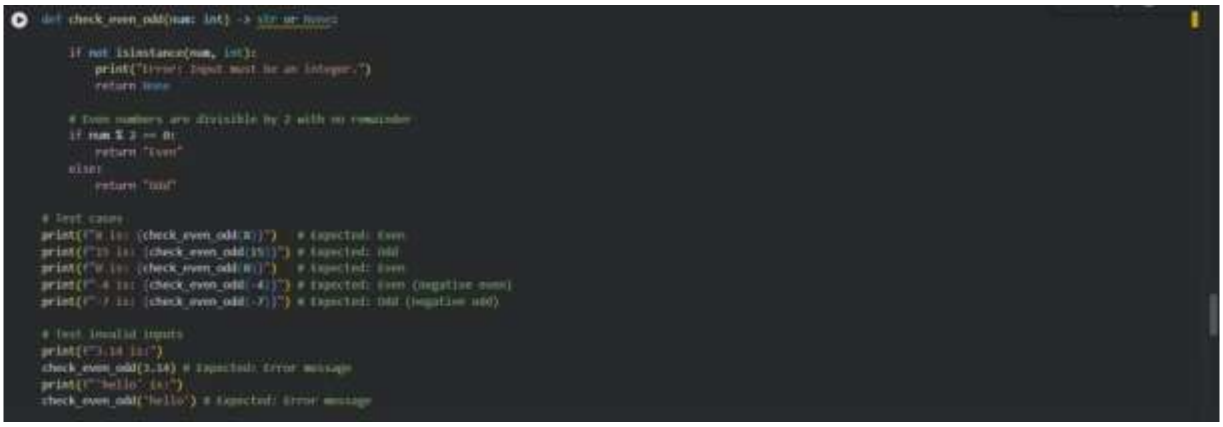
• Input: 0 → Output: Even

Task:

• Analyze how examples improve input handling and output clarity.

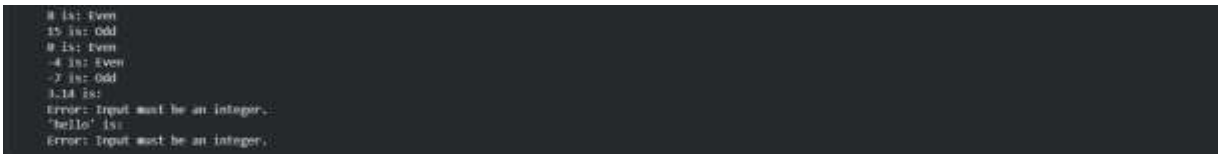
• Test the program with negative numbers and non-integer inputs.

CODE:



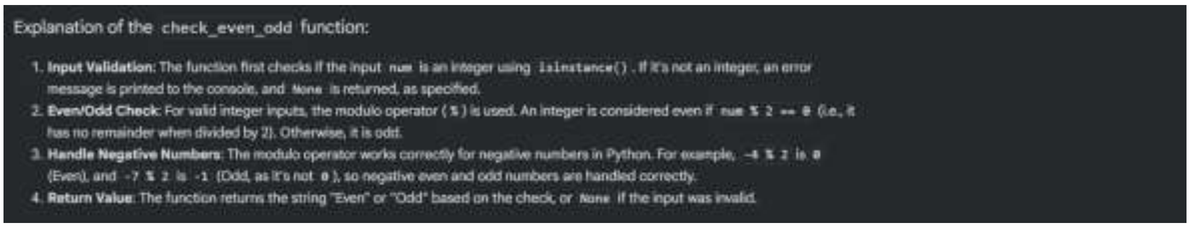
```python
def check_even_odd(num: int) -> str or None:

    if not isinstance(num, int):
        print("Error: Input must be an integer.")
        return None

    # Even numbers are divisible by 2 with no remainder
    if num % 2 == 0:
        return "Even"
    else:
        return "Odd"

# Test cases
print(f"8 is: {check_even_odd(8)}")   # Expected: Even
print(f"15 is: {check_even_odd(15)}") # Expected: Odd
print(f"0 is: {check_even_odd(0)}")   # Expected: Even
print(f"-4 is: {check_even_odd(-4)}") # Expected: Even (negative even)
print(f"-7 is: {check_even_odd(-7)}") # Expected: Odd (negative odd)

# Test invalid inputs
print(f"3.14 is:")
check_even_odd(3.14) # Expected: Error message
print(f"'hello' is:")
check_even_odd('hello') # Expected: Error message
```

OUTPUT:



```
8 is: Even
15 is: Odd
0 is: Even
-4 is: Even
-7 is: Odd
3.14 is:
Error: Input must be an integer.
'hello' is:
Error: Input must be an integer.
```

EXPLANATION:



Explanation of the `check_even_odd` function:

1. **Input Validation:** The function first checks if the input `num` is an integer using `isinstance()`. If it's not an integer, an error message is printed to the console, and `None` is returned, as specified.
2. **Even/Odd Check:** For valid integer inputs, the modulo operator (`%`) is used. An integer is considered even if `num % 2 == 0` (i.e., it has no remainder when divided by 2). Otherwise, it is odd.
3. **Handle Negative Numbers:** The modulo operator works correctly for negative numbers in Python. For example, `-4 % 2` is `0` (Even), and `-7 % 2` is `-1` (Odd, as it's not `0`), so negative even and odd numbers are handled correctly.
4. **Return Value:** The function returns the string "Even" or "Odd" based on the check, or `None` if the input was invalid.