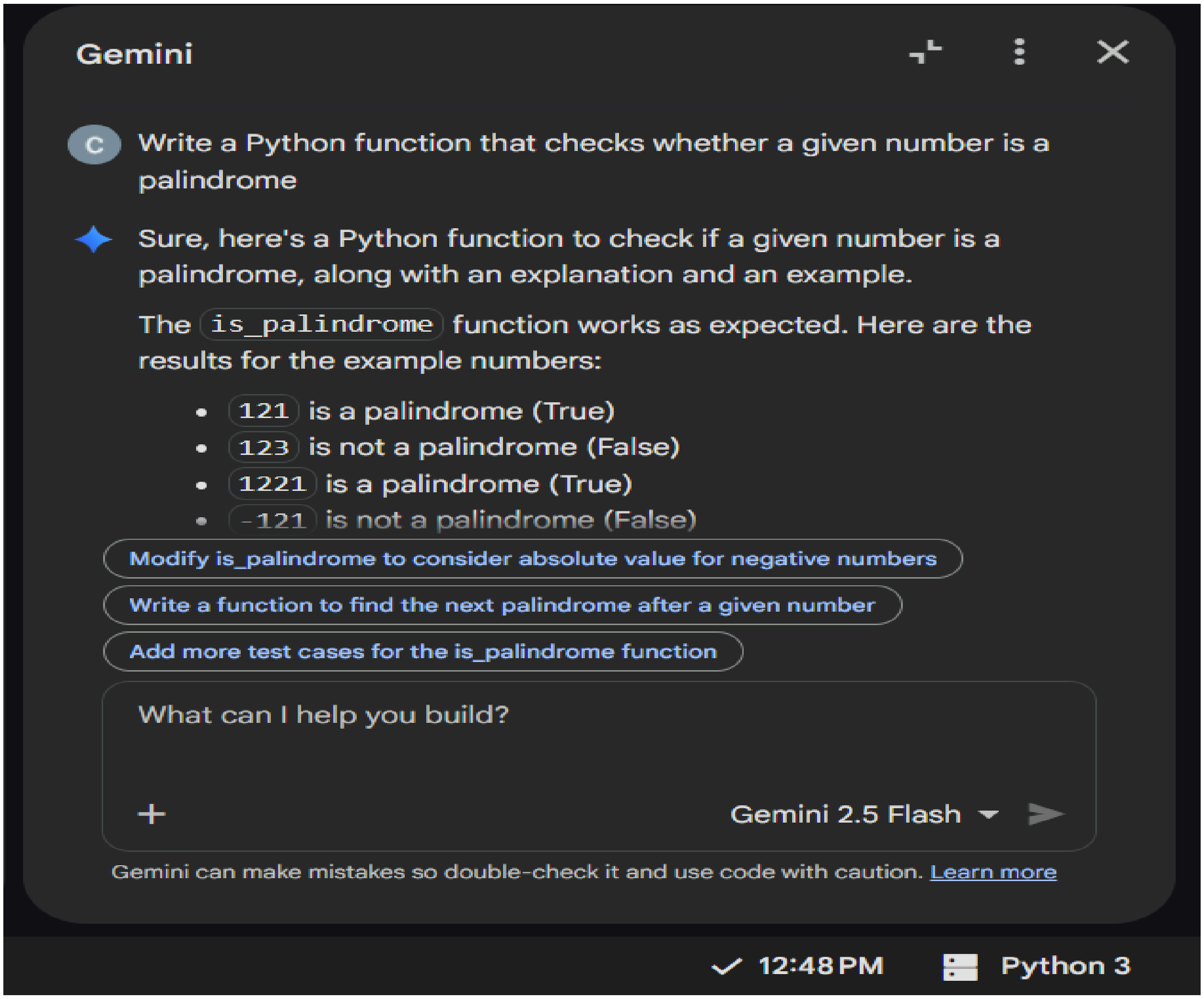
AI ASSISTED CODING
ASSIGNMENT - 3.1
ROLL NO. 2303A510C8
B-27

Question 1: Zero-Shot Prompting (Palindrome Number Program

Prompt:
Write a Python function that checks whether a given number is a palindrome.



Code:

```
def is_palindrome(number):
    """
    Checks if a given number is a palindrome.

    Args:
        number (int): The integer to check.

    Returns:
        bool: True if the number is a palindrome, False otherwise.
    """
    # Convert the number to a string
    num_str = str(number)
```

```python
    # Compare the string with its reverse
    return num_str == num_str[::-1]

# Example usage:
print(f"Is 121 a palindrome? {is_palindrome(121)}")
print(f"Is 123 a palindrome? {is_palindrome(123)}")
print(f"Is 1221 a palindrome? {is_palindrome(1221)}")
print(f"Is -121 a palindrome? {is_palindrome(-121)}")
```

Output:

```
Is 121 a palindrome? True
Is 123 a palindrome? False
Is 1221 a palindrome? True
Is -121 a palindrome? False
```

Explanation:

Code uses type casting and then reverse the string to compare and check for palindrome

Analysis (Errors & Edge Cases)

Aspect	Observation
Logic	Correct for positive integers
Negative numbers	-121 → False (acceptable, but not explicitly handled)
Input validation	Missing (floats, strings not handled)
Efficiency	Efficient for small numbers

Conclusion:
Zero-shot prompting produces short and correct code, but lacks validation and explicit constraints.

Question 2: One-Shot Prompting (Factorial Calculation)

Prompt:

Write a Python function to calculate the factorial of a number.
Example: Input: 5 gives Output: 120

Code:

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

    Args:
        n (int): The non-negative integer to calculate the factorial for.

    Returns:
        int: The factorial of n.

    Raises:
        ValueError: If n is a negative integer.
    """
    if not isinstance(n, int) or n < 0:
        raise ValueError("Factorial is defined only for non-negative
integers.")

    if n == 0:
        return 1

    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

# Example usage:
print(f"Factorial of 0: {factorial(0)}")
print(f"Factorial of 1: {factorial(1)}")
print(f"Factorial of 5: {factorial(5)}")
print(f"Factorial of 7: {factorial(7)}")

# Example with a negative number (will raise an error)
```

```python
try:
    print(f"Factorial of -3: {factorial(-3)}")
except ValueError as e:
    print(f"Error: {e}")
```

Output:

```
Factorial of 0: 1
Factorial of 1: 1
Factorial of 5: 120
Factorial of 7: 5040
Error: Factorial is defined only for non-negative integers.
```

Explanation:

This function calculates the factorial of a number using an iterative approach.

It first validates the input, ensuring that `n` is a non-negative integer; otherwise, it raises a `ValueError`.

If `n` is `0`, it returns `1` (by definition of factorial).

For positive integers, it multiplies numbers from `1` to `n` and returns the final result.

Comparison: Zero-Shot vs One-Shot

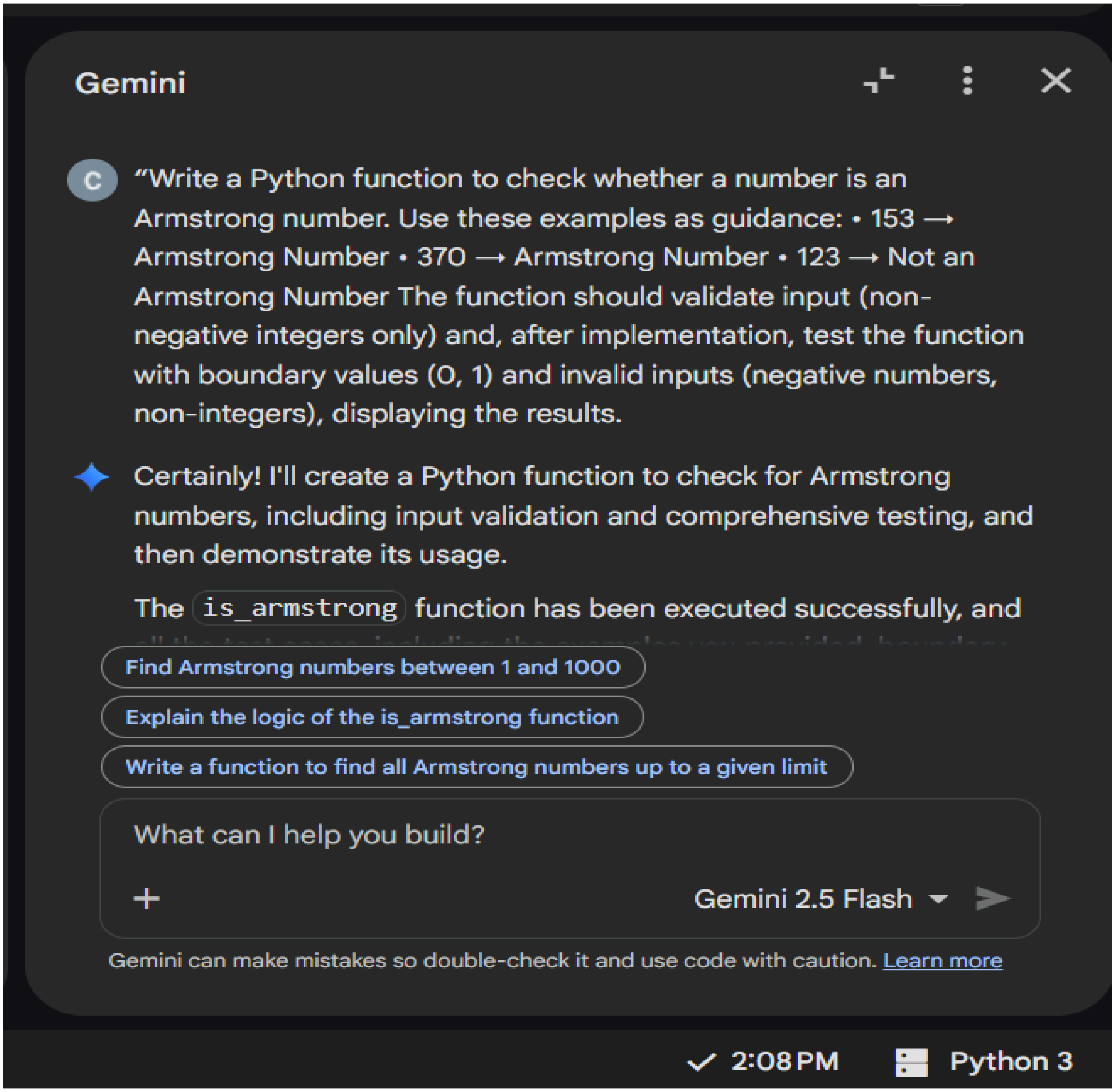
Feature	Zero-Shot	One-Shot
Input validation	No	Yes
Clarity	Medium	High
Edge cases	Missing	Handled
Readability	Basic	Improved

Conclusion:
Providing one example improves correctness, validation, and clarity.

Question 3: Few-Shot Prompting (Armstrong Number Check)
Prompt:

Write a Python function to check whether a number is an Armstrong number.
Use these examples as guidance:
• 153 → Armstrong Number
• 370 → Armstrong Number
• 123 → Not an Armstrong Number

The function should validate input (non-negative integers only) and, after implementation, test the function with boundary values (0, 1) and invalid inputs (negative numbers, non-integers), displaying the results.



Code:

```
def is_armstrong(number):
    """
    Checks if a given number is an Armstrong number.
```

```
    An Armstrong number is a number that is the sum of its own digits each
raised

    to the power of the number of digits.


    Args:

        number (int): The integer to check.


    Returns:

        bool: True if the number is an Armstrong number, False otherwise.


    Raises:

        ValueError: If the input is not a non-negative integer.

    """

    if not isinstance(number, int) or number < 0:

        raise ValueError("Input must be a non-negative integer.")


    # Special case for 0 and single-digit numbers

    if number < 10:

        return True


    num_str = str(number)

    num_digits = len(num_str)

    armstrong_sum = 0
```

```
    for digit_char in num_str:

        digit = int(digit_char)

        armstrong_sum += digit ** num_digits


    return armstrong_sum == number


# Test cases provided by the user

print(f"Is 153 an Armstrong number? {is_armstrong(153)}") # Expected: True

print(f"Is 370 an Armstrong number? {is_armstrong(370)}") # Expected: True

print(f"Is 123 an Armstrong number? {is_armstrong(123)}") # Expected:
False


# Test with boundary values

print(f"Is 0 an Armstrong number? {is_armstrong(0)}")   # Expected: True

print(f"Is 1 an Armstrong number? {is_armstrong(1)}")   # Expected: True

print(f"Is 9 an Armstrong number? {is_armstrong(9)}")   # Expected: True

print(f"Is 10 an Armstrong number? {is_armstrong(10)}")  # Expected: False


# Test with other known Armstrong numbers

print(f"Is 371 an Armstrong number? {is_armstrong(371)}") # Expected: True

print(f"Is 407 an Armstrong number? {is_armstrong(407)}") # Expected: True


# Test with invalid inputs

print("\nTesting with invalid inputs:")
```

```python
try:
    print(f"Is -153 an Armstrong number? {is_armstrong(-153)}")
except ValueError as e:
    print(f"Error for -153: {e}")

try:
    print(f"Is 12.5 an Armstrong number? {is_armstrong(12.5)}")
except ValueError as e:
    print(f"Error for 12.5: {e}")

try:
    print(f"Is 'abc' an Armstrong number? {is_armstrong('abc')}")
except ValueError as e:
    print(f"Error for 'abc': {e}")
```

Output:

```
Is 153 an Armstrong number? True
Is 370 an Armstrong number? True
Is 123 an Armstrong number? False
Is 0 an Armstrong number? True
Is 1 an Armstrong number? True
Is 9 an Armstrong number? True
Is 10 an Armstrong number? False
Is 371 an Armstrong number? True
Is 407 an Armstrong number? True

Testing with invalid inputs:
Error for -153: Input must be a non-negative integer.
Error for 12.5: Input must be a non-negative integer.
Error for 'abc': Input must be a non-negative integer.
```

Explanation:

The function checks whether a number is an Armstrong number after validating that the input is a non-negative integer.

It extracts each digit, raises it to the power of the total number of digits, and sums the results.

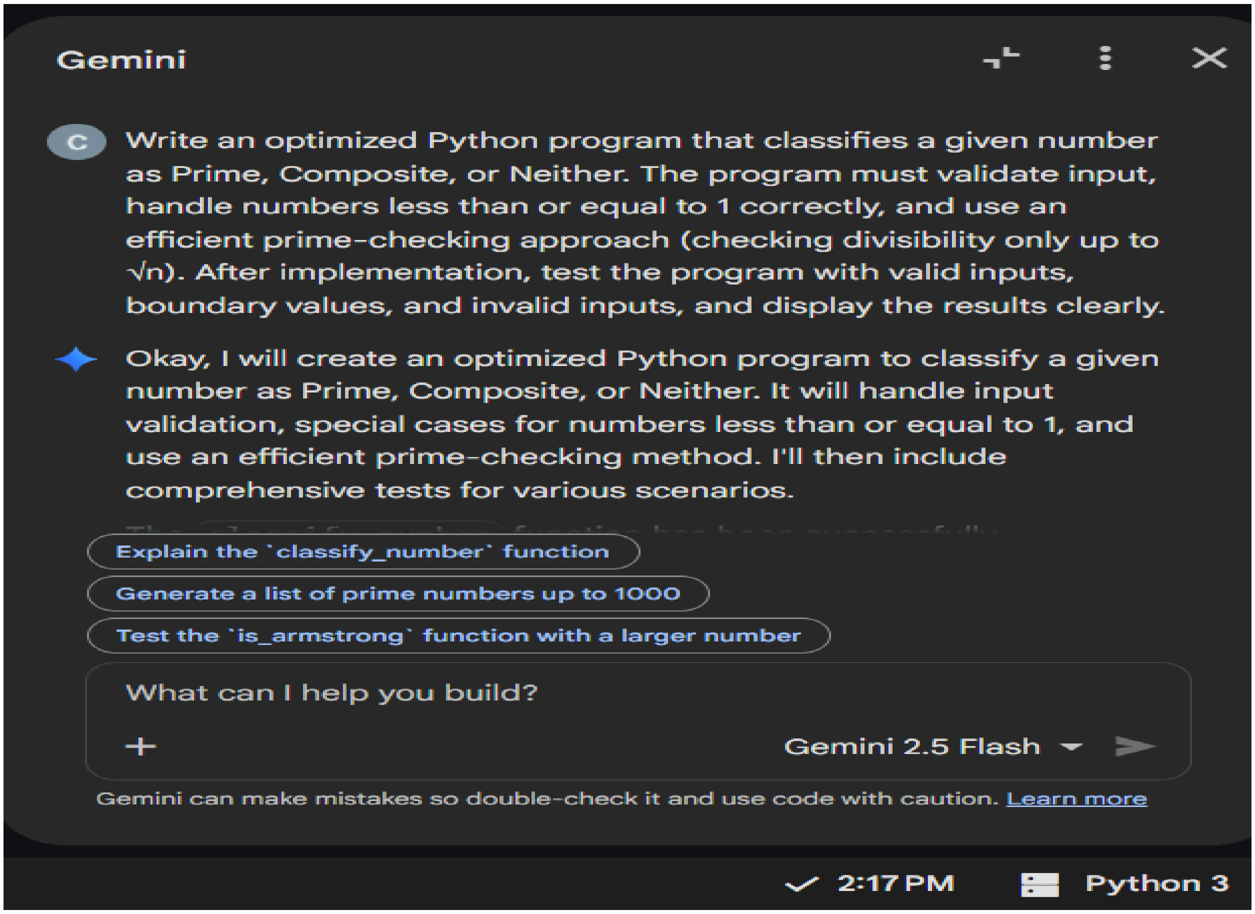
If this sum equals the original number, the function returns `True`; otherwise, it returns `False`.

Analysis: Effect of Multiple Examples

Providing multiple examples helps the AI correctly understand the Armstrong number formula.
It improves code accuracy, ensures proper digit-based computation, and leads to a clear and reliable implementation.

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

Prompt:
Write an optimized Python program that classifies a given number as **Prime**, **Composite**, or **Neither**.
The program must validate input, handle numbers less than or equal to 1 correctly, and use an efficient prime-checking approach (checking divisibility only up to √n).
After implementation, test the program with valid inputs, boundary values, and invalid inputs, and display the results clearly.



Code:

```
import math

def classify_number(number):
    """
    Classifies a given number as Prime, Composite, or Neither.

    Args:
        number (int): The integer to classify.

    Returns:
        str: 'Prime', 'Composite', or 'Neither' based on the
classification.

    Raises:
        ValueError: If the input is not an integer.
    """
```

```python
    if not isinstance(number, int):
        raise ValueError("Input must be an integer.")

    if number <= 1:
        return "Neither"
    elif number == 2:
        return "Prime"
    elif number % 2 == 0:
        return "Composite"

    # Check for divisibility from 3 up to sqrt(number) with a step of 2
    # (only odd numbers need to be checked after 2)
    for i in range(3, int(math.sqrt(number)) + 1, 2):
        if number % i == 0:
            return "Composite"
    return "Prime"
# Test cases for classify_number function
print("--- Valid Inputs ---")
print(f"2 is: {classify_number(2)}")      # Expected: Prime
print(f"7 is: {classify_number(7)}")      # Expected: Prime
print(f"10 is: {classify_number(10)}")    # Expected: Composite
print(f"17 is: {classify_number(17)}")    # Expected: Prime
print(f"25 is: {classify_number(25)}")    # Expected: Composite
print(f"101 is: {classify_number(101)}") # Expected: Prime
print(f"999 is: {classify_number(999)}") # Expected: Composite

print("\n--- Boundary Values ---")
print(f"0 is: {classify_number(0)}")      # Expected: Neither
print(f"1 is: {classify_number(1)}")      # Expected: Neither
print(f"-5 is: {classify_number(-5)}")    # Expected: Neither
print(f"-1 is: {classify_number(-1)}")    # Expected: Neither

print("\n--- Invalid Inputs ---")
try:
    print(f"1.5 is: {classify_number(1.5)}")
except ValueError as e:
    print(f"Error for 1.5: {e}")

try:
    print(f"'abc' is: {classify_number('abc')}")
```

```python
except ValueError as e:
    print(f"Error for 'abc': {e}")

try:
    print(f"None is: {classify_number(None)}")
except ValueError as e:
    print(f"Error for None: {e}")
```

Output:

```
    print(f"Error for None: {e}")

--- Valid Inputs ---
2 is: Prime
7 is: Prime
10 is: Composite
17 is: Prime
25 is: Composite
101 is: Prime
999 is: Composite

--- Boundary Values ---
0 is: Neither
1 is: Neither
-5 is: Neither
-1 is: Neither

--- Invalid Inputs ---
Error for 1.5: Input must be an integer.
Error for 'abc': Input must be an integer.
Error for None: Input must be an integer.
```

Explanation:

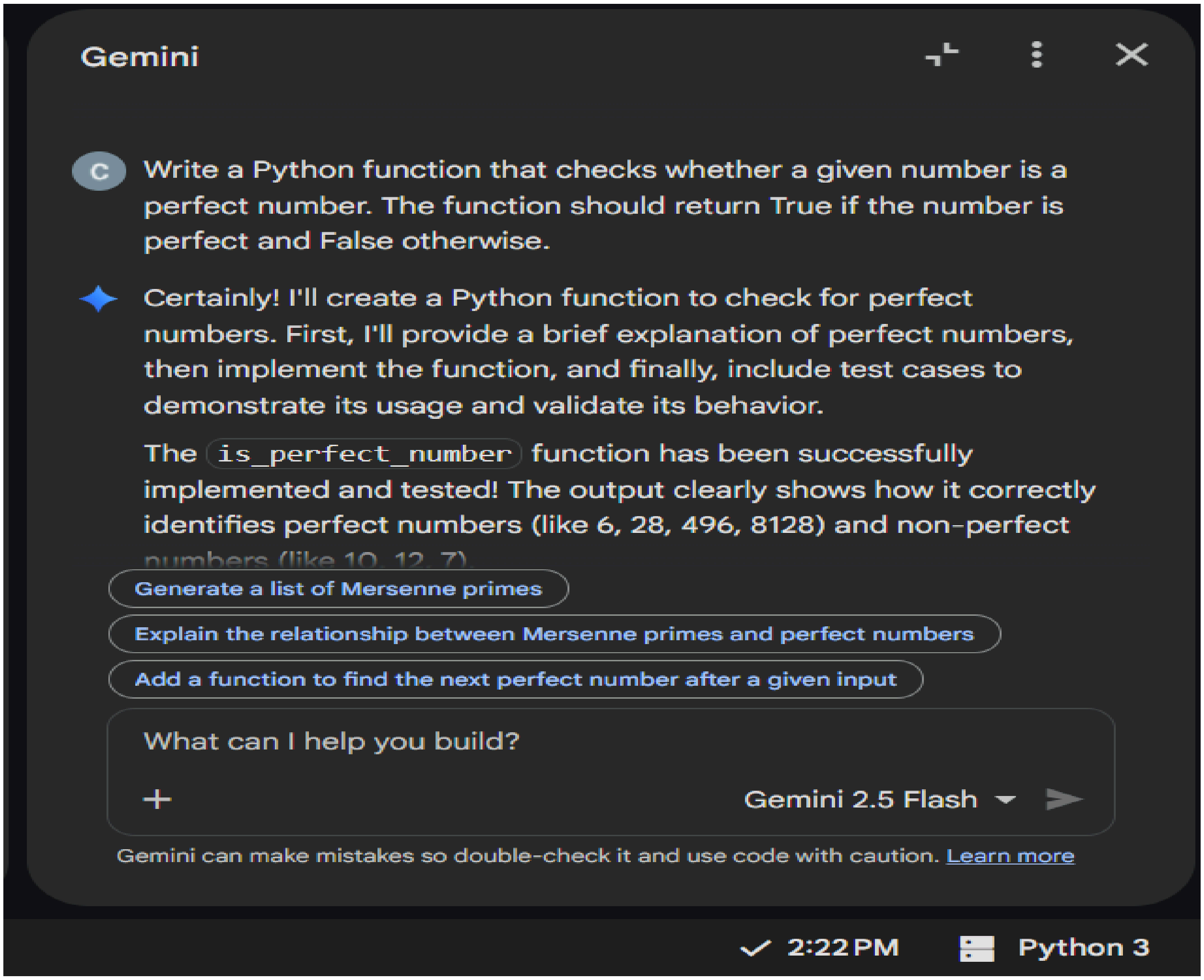
The function first validates that the input is an integer.
Numbers less than or equal to 1 are classified as **Neither**, while 2 is directly classified as **Prime**.
Even numbers greater than 2 are marked **Composite**.
For other numbers, divisibility is checked only up to √n and only for odd numbers, making the algorithm efficient.
If no divisor is found, the number is classified as **Prime**.

Comparison with Earlier Prompting Strategies

Compared to zero-shot and few-shot prompting, context-managed prompting produces **more optimized and structured output**.
Earlier strategies often lacked efficiency or complete validation, whereas this approach ensures **correct classification, input validation, and better performance**.
Overall, context-managed prompting results in **more reliable and production-quality code**.

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

Prompt:
Write a Python function that checks whether a given number is a perfect number.
The function should return `True` if the number is perfect and `False` otherwise.



Code:

```
def is_perfect_number(number):
    """
    Checks if a given number is a perfect number.

    A perfect number is a positive integer that is equal to the sum of its
    proper positive divisors (divisors excluding the number itself).

    Args:
        number (int): The integer to check.

    Returns:
        bool: True if the number is a perfect number, False otherwise.

    Raises:
```

```
        ValueError: If the input is not a positive integer.
    """
    if not isinstance(number, int) or number <= 0:
        raise ValueError("Input must be a positive integer.")

    if number == 1:
        return False # 1 has no proper divisors, sum is 0, not 1

    sum_of_divisors = 1 # Start with 1, as 1 is always a divisor
    for i in range(2, int(number**0.5) + 1):
        if number % i == 0:
            sum_of_divisors += i
            if i * i != number:
                sum_of_divisors += number // i

    return sum_of_divisors == number

# Example usage and test cases
print("--- Perfect Number Checks ---")
print(f"Is 6 a perfect number? {is_perfect_number(6)}")      # Expected:
True (1 + 2 + 3 = 6)
print(f"Is 28 a perfect number? {is_perfect_number(28)}")    # Expected:
True (1 + 2 + 4 + 7 + 14 = 28)
print(f"Is 496 a perfect number? {is_perfect_number(496)}") # Expected:
True
print(f"Is 8128 a perfect number? {is_perfect_number(8128)}") # Expected:
True

print("\n--- Non-Perfect Number Checks ---")
print(f"Is 10 a perfect number? {is_perfect_number(10)}")    # Expected:
False (1 + 2 + 5 = 8)
print(f"Is 12 a perfect number? {is_perfect_number(12)}")    # Expected:
False (1 + 2 + 3 + 4 + 6 = 16)
print(f"Is 7 a perfect number? {is_perfect_number(7)}")      # Expected:
False (1)

print("\n--- Boundary and Invalid Inputs ---")
print(f"Is 1 a perfect number? {is_perfect_number(1)}")      # Expected:
False (as per definition)
```

```python
try:
    print(f"Is 0 a perfect number? {is_perfect_number(0)}")
except ValueError as e:
    print(f"Error for 0: {e}")

try:
    print(f"Is -5 a perfect number? {is_perfect_number(-5)}")
except ValueError as e:
    print(f"Error for -5: {e}")

try:
    print(f"Is 12.5 a perfect number? {is_perfect_number(12.5)}")
except ValueError as e:
    print(f"Error for 12.5: {e}")

try:
    print(f"Is 'abc' a perfect number? {is_perfect_number('abc')}")
except ValueError as e:
    print(f"Error for 'abc': {e}")
```

Output:

```
--- Perfect Number Checks ---
Is 6 a perfect number? True
Is 28 a perfect number? True
Is 496 a perfect number? True
Is 8128 a perfect number? True

--- Non-Perfect Number Checks ---
Is 10 a perfect number? False
Is 12 a perfect number? False
Is 7 a perfect number? False

--- Boundary and Invalid Inputs ---
Is 1 a perfect number? False
Error for 0: Input must be a positive integer.
Error for -5: Input must be a positive integer.
Error for 12.5: Input must be a positive integer.
Error for 'abc': Input must be a positive integer.
```

Explanation:

The function first checks whether the input is a positive integer, raising an error for invalid inputs.

It returns `False` for 1 since it is not a perfect number.

The function then calculates the sum of all proper divisors by checking divisors only up to √n, adding both the divisor and its pair when found.
Finally, it compares the sum of divisors with the original number to determine if it is perfect.

Missing Conditions or Inefficiencies in the Logic

- The code efficiently reduces time complexity by checking divisors only up to √n.
- However, it does not stop early if the divisor sum exceeds the number, which could improve performance.
- Large input values may still be computationally expensive due to iteration.
- Apart from minor optimization opportunities, the logic correctly handles all required conditions.

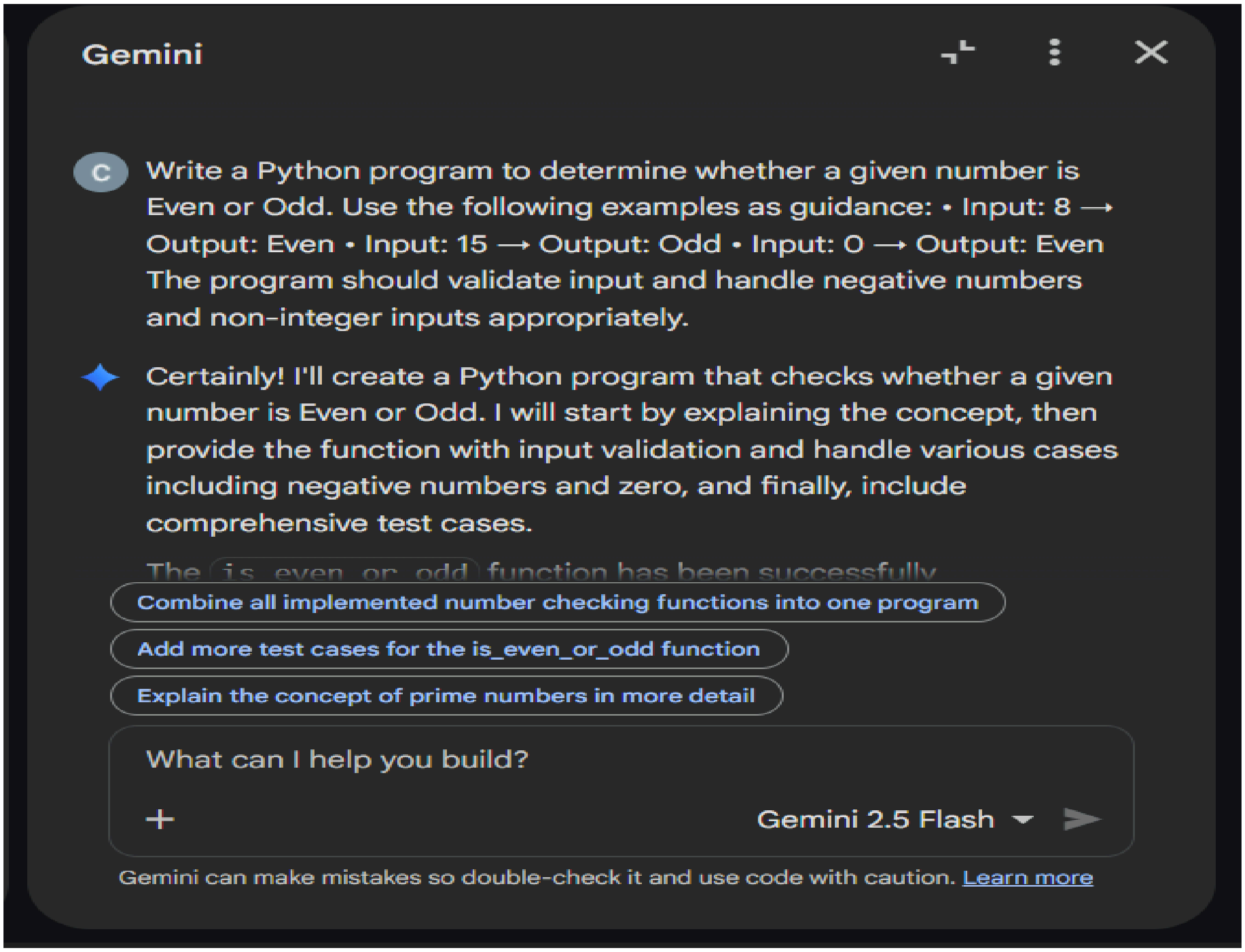
Question 6: Few-Shot Prompting (Even or Odd Classification with Validation)

Prompt:

Write a Python program to determine whether a given number is Even or Odd.
Use the following examples as guidance:
• Input: 8 → Output: Even

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

The program should validate input and handle negative numbers and non-integer inputs appropriately.

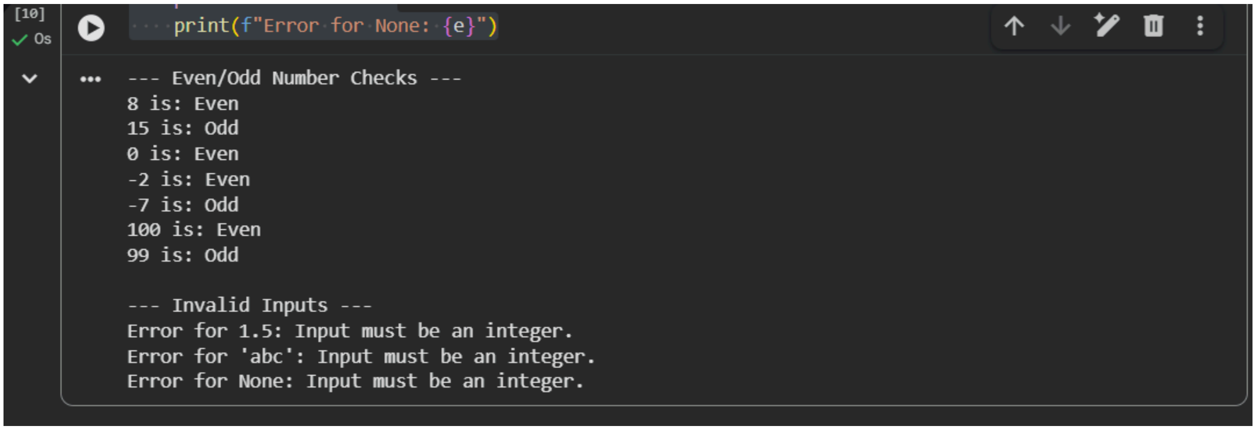


Code:

```
def is_even_or_odd(number):
    """
    Determines if a given number is Even or Odd.

    Args:
        number (int): The integer to check.

    Returns:
```

```python
        str: 'Even' if the number is even, 'Odd' if the number is odd.

    Raises:
        ValueError: If the input is not an integer.
    """
    if not isinstance(number, int):
        raise ValueError("Input must be an integer.")

    if number % 2 == 0:
        return "Even"
    else:
        return "Odd"
# Example usage and test cases
print("--- Even/Odd Number Checks ---")
print(f"8 is: {is_even_or_odd(8)}")     # Expected: Even
print(f"15 is: {is_even_or_odd(15)}")   # Expected: Odd
print(f"0 is: {is_even_or_odd(0)}")     # Expected: Even
print(f"-2 is: {is_even_or_odd(-2)}")    # Expected: Even
print(f"-7 is: {is_even_or_odd(-7)}")    # Expected: Odd
print(f"100 is: {is_even_or_odd(100)}") # Expected: Even
print(f"99 is: {is_even_or_odd(99)}")    # Expected: Odd

print("\n--- Invalid Inputs ---")
try:
    print(f"1.5 is: {is_even_or_odd(1.5)}")
except ValueError as e:
    print(f"Error for 1.5: {e}")

try:
    print(f"'abc' is: {is_even_or_odd('abc')}")
except ValueError as e:
    print(f"Error for 'abc': {e}")

try:
    print(f"None is: {is_even_or_odd(None)}")
except ValueError as e:
    print(f"Error for None: {e}")
```

Output:



```
[10]
0s
    print(f"Error for None: {e}")

--- Even/Odd Number Checks ---
8 is: Even
15 is: Odd
0 is: Even
-2 is: Even
-7 is: Odd
100 is: Even
99 is: Odd

--- Invalid Inputs ---
Error for 1.5: Input must be an integer.
Error for 'abc': Input must be an integer.
Error for None: Input must be an integer.
```

Explanation:

The program first checks whether the input is an integer to ensure valid input.
It then uses the modulo operator to determine if the number is divisible by 2.
If divisible, the number is classified as Even; otherwise, it is classified as Odd.
Invalid inputs are handled gracefully using input validation.

Analysis: Effect of Examples

Providing multiple examples helps the AI clearly understand the expected behavior for different cases.
It improves input handling, ensures correct classification of 0 and negative numbers, and enhances output clarity.

Conclusion from this assignment:

Prompt design directly impacts the **accuracy and quality** of AI-generated code.

Zero-shot prompting gives basic solutions but may lack validation or optimization.

One-shot and few-shot prompting improve correctness, readability, and handling of edge cases.

Context-managed prompting produces **optimized, reliable, and fully validated programs**.

Well-structured prompts lead to **more efficient, robust, and production-ready code**.