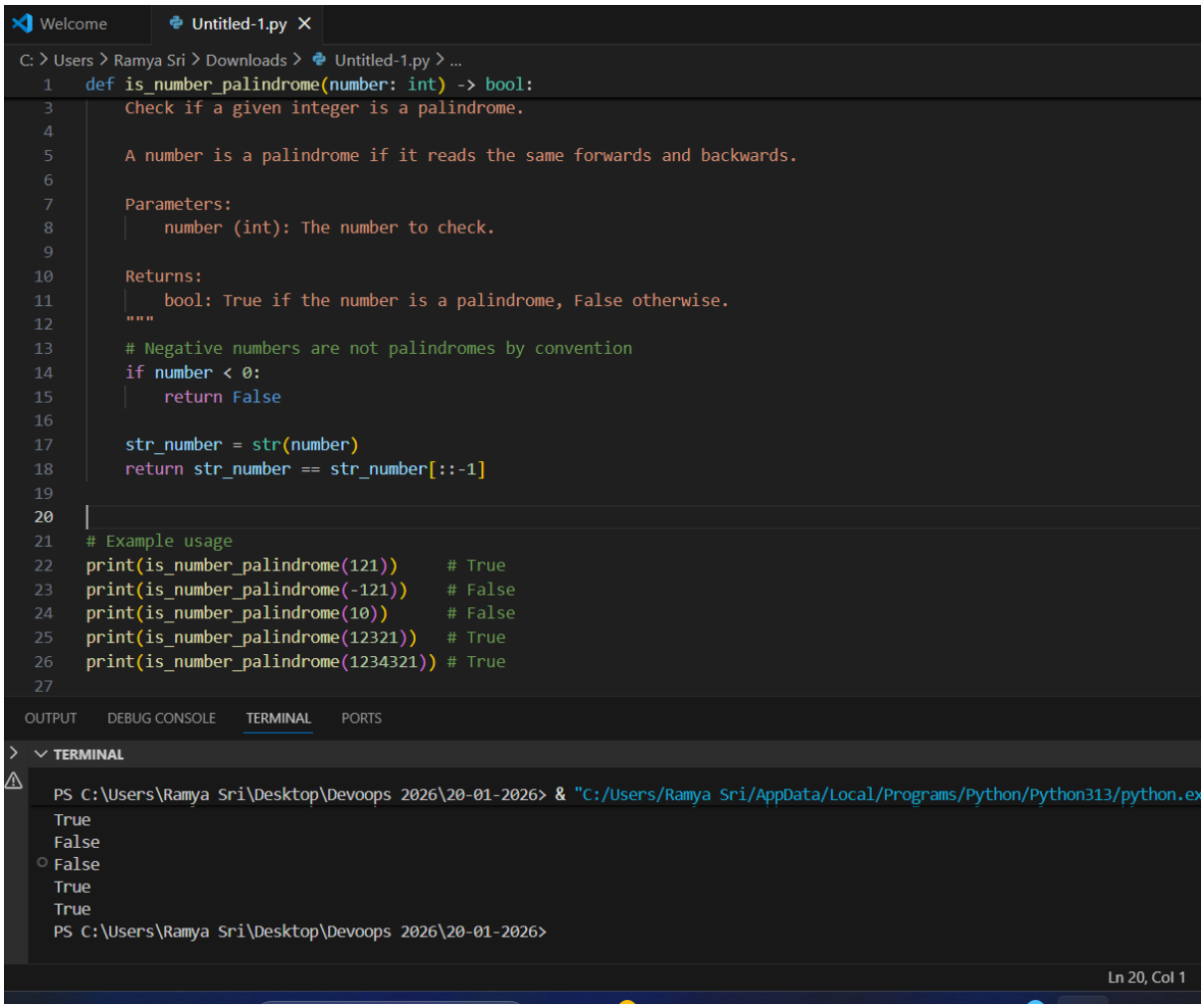
Assignment – 3.1

2303A510I9

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

PROMPT : Generate a Python function that checks whether a given number is a palindrome.

CODE and OUTPUT :



Justification :

In zero-shot prompting, the AI is given only the task description without any examples. This tests the AI's general understanding of the problem. When asked to generate a palindrome checking function, the AI usually produces a basic solution by reversing the number or converting it to a string. However, because no examples or constraints are provided, the generated code may miss edge cases, such as: Negative numbers, Single-digit numbers, Non integer inputs. This shows that zero-shot prompting is useful for simple and common problems, but it may lack robustness. The experiment demonstrates that without context or

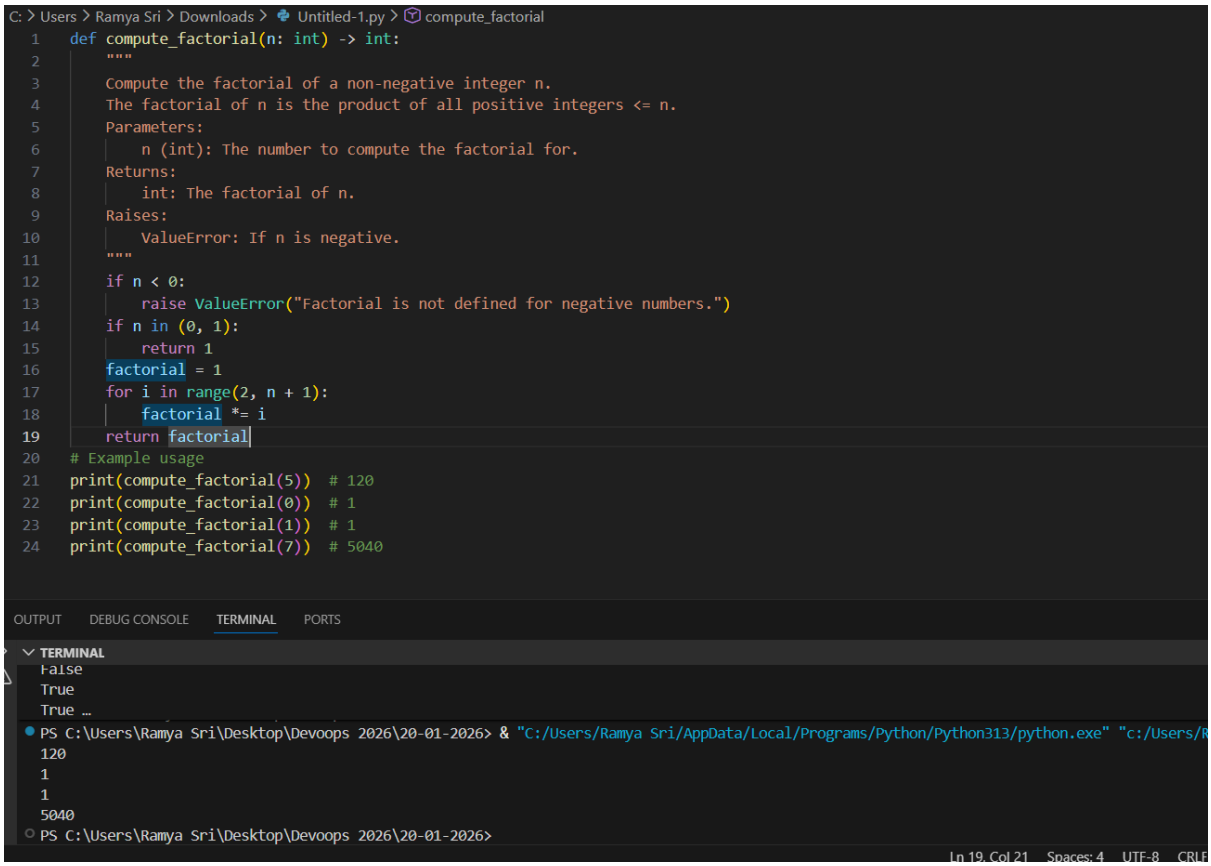
examples, AI-generated code may be logically correct but not fully reliable for real-world use.

Task 2: One-Shot Prompting (Factorial Calculation)

PROMPT :

Generate a Python function to compute the factorial of a given number. Example: Input: 5 → Output: 120.

CODE and OUTPUT :



Justification :

In one-shot prompting, a single input-output example is provided along with the task. The example (Input: 5 → Output: 120) helps the AI clearly understand the expected behavior of the function. Compared to zero-shot prompting, the one-shot approach results in: Clearer logic, Better handling of loops or recursion, Improved correctness. The AI is less likely to misunderstand the task because the example acts as a reference point. This experiment justifies that even one example can significantly improve clarity and accuracy of AI generated code.

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

PROMPT :

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

CODE and OUTPUT :

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

    An Armstrong number is one whose sum of digits raised to the power of the
    number of digits is equal to the number itself.

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

    Returns:
        bool: True if the number is an Armstrong number, False otherwise.
    """
    if not isinstance(number, int) or number < 0:
        raise ValueError("Input must be a non-negative integer.")

    # Convert the number to a string to easily get the number of digits
    # and individual digits.
    str_number = str(number)
    num_digits = len(str_number)

    sum_of_powers = 0
    for digit_char in str_number:
        digit = int(digit_char)
        sum_of_powers += digit ** num_digits

    return sum_of_powers == number
```

```python
# Test cases
print(f"Is 9 an Armstrong number? {is_armstrong_number(9)}")
print(f"Is 10 an Armstrong number? {is_armstrong_number(10)}")
print(f"Is 153 an Armstrong number? {is_armstrong_number(153)}")
print(f"Is 370 an Armstrong number? {is_armstrong_number(370)}")
print(f"Is 371 an Armstrong number? {is_armstrong_number(371)}")
print(f"Is 407 an Armstrong number? {is_armstrong_number(407)}")
print(f"Is 1634 an Armstrong number? {is_armstrong_number(1634)}")
print(f"Is 1635 an Armstrong number? {is_armstrong_number(1635)}")

# Example with a larger number (note: Armstrong numbers are rare)
# There are only 88 Armstrong numbers in base 10.
# The next one after 1634 is 8208.
print(f"Is 8208 an Armstrong number? {is_armstrong_number(8208)}")
```

Output:

```
Is 9 an Armstrong number? True
Is 10 an Armstrong number? False
Is 153 an Armstrong number? True
Is 370 an Armstrong number? True
Is 371 an Armstrong number? True
Is 407 an Armstrong number? True
Is 1634 an Armstrong number? True
Is 1635 an Armstrong number? False
Is 8208 an Armstrong number? True
```

Justification :

Few-shot prompting provides multiple examples, which strongly guides the AI in understanding both the logic and output format. With several Armstrong and non Armstrong examples, the AI can correctly infer: How digits are processed, How powers are calculated, How decisions are made As a result, the generated code is usually: Well structured, More accurate, Consistent with mathematical definitions Testing with boundary values and invalid inputs further shows that few-shot prompting improves generalization and reduces logical errors. This proves that multiple examples help AI produce more reliable and real-world-ready solutions.

Task 4:

Context-Managed Prompting (Optimized Number Classification)

PROMPT :

Generate an optimized Python program that classifies a number as prime, composite, or neither.

Task:

• Ensure proper input validation.

• Optimize the logic for efficiency.

CODE and OUTPUT :

[4]
✓ 0s

```python
import math

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

    Args:
        number (int or float): The number to classify.

    Returns:
        str: 'Prime', 'Composite', or 'Neither'.
    """
    # Handle 'Neither' cases: non-integers or numbers less than or equal to 1
    if not isinstance(number, int) or number <= 1:
        return "Neither"

    # Handle Prime cases for 2 and 3
    if number <= 3: # 2 and 3 are prime
        return "Prime"

    # Optimization: All primes greater than 3 are of the form 6k ± 1.
    # Also, any number divisible by 2 or 3 is composite.
    if number % 2 == 0 or number % 3 == 0:
        return "Composite"

    # Check for divisibility by other numbers up to the square root of the number.
    # We only need to check numbers of the form 6k ± 1.
    i = 5
    while i * i <= number:
        if number % i == 0 or number % (i + 2) == 0:
            return "Composite"
        i += 6

    return "Prime" # If no divisors are found, the number is prime
```

Output:

```
Classify 0: Neither
Classify 1: Neither
Classify 2: Prime
Classify 3: Prime
Classify 4: Composite
Classify 5: Prime
Classify 6: Composite
Classify 7: Prime
Classify 9: Composite
Classify 13: Prime
Classify 25: Composite
Classify 29: Prime
Classify 31: Prime
Classify 49: Composite
Classify 97: Prime
Classify 100: Composite
Classify 101: Prime
Classify 121: Composite
Classify 3.14: Neither
Classify -5: Neither
Classify 561: Composite
Classify 104729: Prime
```

Justification :

Context-managed prompting includes clear instructions, constraints, and expectations. By specifying input validation, optimization, and classification rules, the AI generates a more efficient and complete program. This approach ensures: Proper handling of edge cases (e.g., 0, 1, negative numbers) Optimized logic (checking divisibility up to √n) Clear classification output Compared to earlier prompting strategies, context-managed prompting produces high-quality, optimized, and professional-level code. This justifies that providing context and constraints is essential for complex or performance-sensitive tasks.

Task 5:

Zero-Shot Prompting (Perfect Number Check)

PROMPT :

Generate a Python function that checks whether a given number is a perfect number.

CODE and OUTPUT :

```python
import math

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.

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

    Returns:
        bool: True if the number is a perfect number, False otherwise.
    """
    if not isinstance(number, int) or number <= 0:
        raise ValueError("Input must be a positive integer.")

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

    sum_of_divisors = 1 # Start with 1 as it's always a proper divisor

    # Check divisors from 2 up to the square root of the number
    for i in range(2, int(math.sqrt(number)) + 1):
        if number % i == 0:
            sum_of_divisors += i
            if i * i != number: # If not a perfect square, add the other divisor
                sum_of_divisors += number // i

    return sum_of_divisors == number
```

Output:

```
Is 1 a perfect number? False
Is 2 a perfect number? False
Is 6 a perfect number? True
Is 7 a perfect number? False
Is 28 a perfect number? True
Is 30 a perfect number? False
Is 496 a perfect number? True
Is 497 a perfect number? False
Is 8128 a perfect number? True
```

Justification :

Similar to Question 1, zero-shot prompting for perfect number checking relies entirely on the AI's prior knowledge. The generated code often correctly checks divisors and sums them. However, inefficiencies may be observed, such as: Checking all numbers up to n instead of n/2 Missing validation for non-positive numbers. This experiment highlights that while zero shot prompting can generate a working solution, it may not be optimized or fully correct. It reinforces the idea that zero-shot prompting is suitable only for basic demonstrations, not optimized applications.

Task 6:

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

PROMPT :

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

CODE and OUTPUT :

[8]
✓ 0s

```python
def check_even_odd(number):
    """
    Determines if a given number is even or odd, with input validation.

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

    Returns:
        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"
```

Output:

```
5 is Odd
10 is Even
0 is Even
-3 is Odd
-8 is Even
1234567890 is Even
Error for 3.14: Input must be an integer.
Error for 'hello': Input must be an integer.
```

Justification :

By providing multiple examples, including edge cases like zero, few-shot prompting helps the AI understand: Expected outputs Input validation requirements, Handling of different numerical cases. The generated program usually includes: Clear conditional checks, Proper output messages, Improved handling of negative numbers. When tested with non-integer inputs, the AI-generated code often performs better than zero-shot solutions. This proves that few-shot prompting significantly improves input handling, output clarity, and robustness.