Assignment – 3.1

2303A51927

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

PROMPT :

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

CODE :

```python
#generate a Python function that checks whether a given number is a palindrome.
def is_number_palindrome(number):
    """
    Check if a given number is a palindrome.

    A number is a palindrome if it reads the same forwards and backwards.

    Parameters:
    number (int): The number to check.

    Returns:
    bool: True if the number is a palindrome, False otherwise.
    """
    # Convert the number to string to easily reverse it
    str_number = str(number)

    # Compare the string with its reverse
    return str_number == str_number[::-1]
    cleaned_s = ''.join(char.lower() for char in s if char.isalnum())
    return cleaned_s == cleaned_s[::-1]
# Example Usage:
print(is_number_palindrome(121))  # True
print(is_number_palindrome(-121)) # False
print(is_number_palindrome(10))   # False
print(is_number_palindrome(12321)) # True
print(is_number_palindrome(1234321)) # True
```

OUTPUT :

```
.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '62967' '--' 'D:\AI\AI-3.1.py'
True
False
False
True
True
PS D:\AI>
```

Justification :

Zero-shot prompting involves providing the AI with only a clear problem statement, without offering any examples or additional context. This method is used to assess how well the AI can interpret and solve a task on its own. When the AI is asked to write a function to check whether a number is a palindrome, it generally produces a simple and direct approach, such as reversing the number or comparing a string with its reverse. However, due to the lack of explicit constraints or sample cases, the generated solution may fail to handle certain edge cases, including negative numbers, single-digit inputs, or non-integer values. This indicates that although zero-shot prompting is effective for basic and commonly known problems, it

may result in solutions that are not fully robust. The experiment demonstrates that without sufficient context or guidance, AI-generated code can appear correct but may not be reliable for practical, real-world scenarios.

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 :

```python
#generate a Python function to compute the factorial of a given number. Example: Input: 5 → Output: 120.
def compute_factorial(n):
    """
    Compute the factorial of a given number.

    The factorial of a non-negative integer n is the product of all positive integers less than or equal to n.

    Parameters:
    n (int): The number to compute the factorial for.

    Returns:
    int: The factorial of the number.
    """
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    elif n == 0 or n == 1:
        return 1
    else:
        factorial = 1
        for i in range(2, n + 1):
            factorial *= i
        return factorial
# Example Usage:
print(compute_factorial(5))  # Output: 120
print(compute_factorial(0))  # Output: 1
print(compute_factorial(1))  # Output: 1
print(compute_factorial(7))  # Output: 5040
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python Debug Console

OUTPUT:

```
.0-win32-x64\bundled\libs\debugpy\launcher' '60726' '--' 'D:\AI\AI-3.1.py'
120
1
1
5040
PS D:\AI>
```

Justification :

In one-shot prompting, the problem statement is supported by a single sample input and its corresponding output. An example such as Input: 5 → Output: 120 gives the AI a concrete reference for understanding what the function is expected to do. Compared to zero-shot prompting, this method typically produces more structured logic, better use of loops or recursive techniques, and more accurate results. Since the example serves as a guideline, the

chances of the AI misinterpreting the task are reduced. This experiment shows that providing even one illustrative example can noticeably improve the precision and clarity 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 :

```
57
58  #generating a Python function to check whether a given number is an Armstrong number. Examples:
59  #Input: 153 → Output: Armstrong Number
60  #Input: 370 → Output: Armstrong Number
61  #Input: 123 → Output: Not an Armstrong Number
62  def is_armstrong_number(number):
63      """
64      Check if a given number is an Armstrong number.
65      An Armstrong number (or narcissistic number) is a number that is equal to the sum of its own digits each raised
66      Parameters:
67      number (int): The number to check.
68      Returns:
69      bool: True if the number is an Armstrong number, False otherwise.
70      """
71      # Convert the number to string to easily iterate over digits
72      str_number = str(number)
73      num_digits = len(str_number)
74      # Calculate the sum of each digit raised to the power of num_digits
75      sum_of_powers = sum(int(digit) ** num_digits for digit in str_number)
76
77      # Check if the sum of powers is equal to the original number
78      return sum_of_powers == number
79  # Example Usage:
80  print(is_armstrong_number(153))  # Output: True
81  print(is_armstrong_number(370))  # Output: True
82  print(is_armstrong_number(123))  # Output: False
83  print(is_armstrong_number(9474)) # Output: True
84  print(is_armstrong_number(9475)) # Output: False
```

OUTPUT:

```
True
True
False
True
False
PS D:\AI>
```

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 :

```
AI-3.1.py > classify_number
#generate an optimized Python program that classifies a number as prime, composite, or neither. Task:
#Ensure proper input validation.
#Optimize the logic for efficiency.
def classify_number(num):
    """Classify a number as prime, composite, or neither.
    Parameters:
    num (int): The number to classify.
    Returns:
    str: "Prime", "Composite", or "Neither"."""
    # Input validation
    if not isinstance(num, int):
        return "Input must be an integer."
    if num < 0:
        return "Neither"
    if num == 0 or num == 1:
        return "Neither"
    # Check for primality
    if num == 2:
        return "Prime"
    if num % 2 == 0:
        return "Composite"

    for i in range(3, int(num**0.5) + 1, 2):
        if num % i == 0:
            return "Composite"
    return "Prime"
# Example Usage:
print(classify_number(11))  # Output: Prime
print(classify_number(15))  # Output: Composite
print(classify_number(1))   # Output: Neither
print(classify_number(0))   # Output: Neither
```

OUTPUT:

```
.0-win32-x64\bundled\libs\debugpy\launcher' '62266' '--' 'D:\AI\AI-3.1.py'
Prime
Composite
Neither
Neither
```

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 :

```python
#generate a Python function that checks whether a given number is a perfect number.
def is_perfect_number(number):
    """
    Check 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, excluding itsel

    Parameters:
    number (int): The number to check.

    Returns:
    bool: True if the number is a perfect number, False otherwise.
    """
    if number <= 0:
        return False

    # Calculate the sum of proper divisors
    sum_of_divisors = 0
    for i in range(1, number // 2 + 1):
        if number % i == 0:
            sum_of_divisors += i

    # Check if the sum of divisors equals the original number
    return sum_of_divisors == number
# Example Usage:
print(is_perfect_number(6))    # True
print(is_perfect_number(28))   # True
print(is_perfect_number(12))   # False
print(is_perfect_number(496))  # True
```

OUTPUT :

```
.0-win32-x64\bundled\libs\debugpy\launcher' '51384' '--' 'D:\AI\AI-3.1.py'
True
True
False
True
PS D:\AI>
```

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 :

```python
#generating a Python program that determines whether a given number is even or odd, including proper input validatic
#Examples:
#Input: 8 → Output: Even
#Input: 15 → Output: Odd
#Input: 0 → Output: Even
def check_even_odd(number):
    """
    Determine whether a given number is even or odd.

    Parameters:
    number (int): The number to check.

    Returns:
    str: "Even" if the number is even, "Odd" if the number is odd.
    """
    # Input validation
    if not isinstance(number, int):
        return "Input must be an integer."

    # Check if the number is even or odd
    if number % 2 == 0:
        return "Even"
    else:
        return "Odd"
# Example Usage:
print(check_even_odd(8))    # Output: Even
print(check_even_odd(15))   # Output: Odd
print(check_even_odd(0))    # Output: Even
print(check_even_odd(-4))   # Output: Even
```

OUTPUT :

```
.0-win32-x64\bundled\libs\debugpy\launcher' '51813' '--' 'D:\AI\AI-3.1.py'
Even
Odd
Even
Even
```

Justification : By supplying several examples, especially those that cover edge cases such as zero, few-shot prompting enables the AI to better grasp the expected results, input validation needs, and the handling of various numerical scenarios. As a result, the generated program

typically contains well-defined conditional statements, appropriate output responses, and enhanced support for negative values. When the code is tested with non-integer inputs, it generally performs more reliably than solutions produced using zero-shot prompting. This clearly demonstrates that few-shot prompting greatly enhances robustness, improves clarity in outputs, and strengthens overall input handling.