

## AI ASSITANT CODING 3.1 ASSESS

T.VIKAS

2303A52281

BATCH 43

### Question 1: Zero-Shot Prompting (Palindrome Number Program)

Prompt: Give me the code Palindrome Number

Code:

```
⌚ Al assess.py > ...
1  #Zero-Shot Prompting (Palindrome Number Program)
2  def is_palindrone(num):
3      str_num = str(num)
4      return str_num == str_num[::-1]
5  print(is_palindrone(121)) # True
6  print(is_palindrone(-121)) # False
7  print(is_palindrone(10)) # False
8
```

Output:

```
True
False
False
```

Report:

In the zero-shot approach, the AI generated a simple palindrome-checking function based on reversing the input and comparing it with the original value. The logic worked well for common positive integers such as 121 and 454. However, the solution lacked robustness because it did not properly handle edge cases like negative numbers, decimal values, or numbers ending with zero. No input validation or error handling was included, making the function less reliable. This demonstrates that zero-shot prompting often produces correct but basic solutions with limited awareness of special cases.

### Question2: One-Shot Prompting (Factorial Calculation)

Prompt: Factorial Calculation as take input 5 get output as 120

```

#one shot
def factorial(n):
    if n == 0 or n == 1:
        return 1
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

print(factorial(5))  # 120
print(factorial(0))  # 1
print(factorial(10)) # 3628800

```

### Output:

```

120
1
3628800

```

### Report:

With the inclusion of one example ( $5 \rightarrow 120$ ), the AI produced a clearer and more structured factorial function. The generated code usually handled important base cases such as  $0! = 1$  and  $1! = 1$ , improving correctness. Compared to the zero-shot approach, the one-shot solution showed better logical flow and readability. Some versions of the code also included simple checks for invalid inputs like negative numbers. This highlights that one-shot prompting significantly enhances clarity, correctness, and overall code quality.

### Question3: Few-Shot Prompting (Armstrong Number Check)

Prompt: Give me the code Armstrong Number Check take inputs as multiple

```

22 #Few-Shot Prompting (Fibonacci Sequence)
23 def fibonacci(n):
24     if n <= 0:
25         return []
26     elif n == 1:
27         return [0]
28     elif n == 2:
29         return [0, 1]
30
31     fib_sequence = [0, 1]
32     for i in range(2, n):
33         fib_sequence.append(fib_sequence[i-1] + fib_sequence[i-2])
34     return fib_sequence
35
36 print(fibonacci(5))  # [0, 1, 1, 2, 3]
37 print(fibonacci(1))  # [0]
38 print(fibonacci(0))  # []
39

```

### Output:

```

[0, 1, 1, 2, 3]
[0]
[]

```

## **Report:**

By providing multiple examples, the AI was able to clearly recognize the Armstrong number pattern and generate a well-structured function. The code correctly calculated the sum of digits raised to the power of the number of digits and compared it with the original number. It performed accurately for known Armstrong numbers such as 153 and 370. However, boundary values like very large numbers, negative inputs, and non-integer values were not fully handled. This shows that few-shot prompting greatly improves accuracy and understanding, though explicit instructions for edge cases are still necessary.

## **Optional Extensions:**

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

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

```
# Question 4: Context-Managed Prompting (Optimized Number Classification)
def classify_number(num):
    """
    Classifies a number as prime, composite, or neither.
    Input validation: accepts integers only.
    """
    if not isinstance(num, int) or isinstance(num, bool):
        raise ValueError("Input must be an integer")

    if num < 2:
        return "Neither"
    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"

print(classify_number(2))      # Prime
print(classify_number(17))     # Prime
print(classify_number(15))     # Composite
print(classify_number(1))      # Neither
```

## **Output:**

```
Prime
Prime
Composite
Neither
```

## **Report:**

In the context-managed prompting approach, clear instructions and constraints helped the AI generate a more efficient and well-structured program to classify numbers as prime, composite, or neither. The code included proper input validation, ensuring that invalid inputs such as negative numbers, decimals, and non-numeric values were handled appropriately. Optimization techniques,

such as checking divisibility only up to the square root of the number, improved performance compared to earlier prompting strategies. Unlike zero-shot or one-shot solutions, this version demonstrated better logical flow and reduced unnecessary computations. Overall, context-managed prompting resulted in a more reliable, efficient, and user-friendly solution.

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

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

```
# Question 5: Zero-Shot Prompting (Perfect Number Check)
def is_perfect_number(num):
    """Checks if a number is perfect (sum of divisors equals the number)."""
    if num <= 0:
        return False

    divisor_sum = 0
    for i in range(1, num):
        if num % i == 0:
            divisor_sum += i
    return divisor_sum == num

print(is_perfect_number(6))  # True
print(is_perfect_number(28)) # True
print(is_perfect_number(10)) # False
```

## **Output:**

```
True
True
False
```

## **Report:**

Using zero-shot prompting, the AI generated a basic function to check whether a number is a perfect number by summing its proper divisors. The logic worked correctly for common examples such as 6 and 28, but it lacked optimization and proper input validation. The code often used brute-force iteration up to the number itself, which is inefficient for large inputs. It also failed to explicitly handle negative numbers, zero, and non-integer inputs. This highlights that zero-shot prompting typically produces correct but unoptimized and less robust solutions.

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

**Prompt: 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.**

```
# Question 6: Few-Shot Prompting (Even or Odd Classification with Validation)
def classify_even_odd(num):
    """Classifies a number as even or odd with input validation."""
    if not isinstance(num, int) or isinstance(num, bool):
        raise ValueError("Input must be an integer")

    return "Even" if num % 2 == 0 else "Odd"

print(classify_even_odd(8))    # Even
print(classify_even_odd(15))   # Odd
print(classify_even_odd(0))    # Even
print(classify_even_odd(-4))   # Even
```

### Output:

```
Even
Odd
Even
Even
```

### Report:

With multiple input-output examples, the AI clearly understood the task and generated a structured program for classifying numbers as even or odd. The inclusion of examples helped improve output clarity and consistency. The generated code usually incorporated input validation, ensuring that non-integer values and invalid inputs were handled gracefully. Testing with negative numbers showed correct classification, while non-integer inputs were often rejected or flagged with error messages. This demonstrates that few-shot prompting significantly enhances reliability, clarity, and user interaction compared to zero-shot methods.