

# **AI ASSISTED CODING ASS-3.1**

**2303A51642**

**B-28**

## **Experiment – Prompt Engineering Techniques Task**

### **Description**

**Design and refine prompts using different prompting strategies to generate Python programs for basic computational problems**

### **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.**

### **Objective**

To use zero-shot prompting to generate a Python function that checks whether a given number is a palindrome, then evaluate the generated code by testing it with multiple inputs and identifying logical issues or missing edge-case handling.

### **Step 1: Zero-Shot Prompt**

**Zero-shot prompt (no examples provided):**

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

## Step 2: AI-Generated Code

```
def is_palindrome(num):
    temp = num
    reverse_num = 0

    while temp > 0:
        digit = temp % 10
        reverse_num = reverse_num * 10 + digit
        temp = temp // 10

    return num == reverse_num
```

## Step 3: Testing the Code with Multiple Inputs

Input	Expected Output	Actual Output	Result
121	True	True	Correct
123	False	False	Correct
7	True	True	Correct
1221	True	True	Correct
10	False	False	Correct
0	True	False	Incorrect
-121	False	False (by coincidence)	Issue

## Step 4: Identification of Logical Errors and Missing Edge Cases Observed

### Issues

#### 1. Input = 0

- o Expected: True (0 is a palindrome)
- o Actual: False

- **Reason:**  
The while `temp > 0` loop does not execute for `temp = 0`, leaving `reverse_num = 0`, but the logic is not explicitly designed for this case.

## 2. Negative Numbers

- Example: -121
- The function returns False, but **not intentionally**.
- **Reason:**  
The loop condition `temp > 0` fails immediately for negative numbers, and no explicit negative check is included.

## 3. Missing Input Validation

- No handling for:
  - Non-integer inputs
  - Negative values (should explicitly return False)

### Improved Version (After Analysis)

```
def is_palindrome(num):  
    if num < 0:  
        return False  
  
    original = num  
    reverse_num = 0  
  
    while num > 0:  
        reverse_num = reverse_num * 10 + num % 10  
        num //= 10  
  
    return original == reverse_num
```

### Conclusion

- Zero-shot prompting successfully generated a working solution for standard cases.
- The AI-generated code lacked edge-case handling for:

- Zero
- Negative numbers
- Manual testing is essential to identify logical gaps.
- Zero-shot prompts are effective for simple problems but require human review and refinement for robustness.

## 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.**

### Objective

To use **one-shot prompting** to generate a Python function that computes the factorial of a number, and to compare the generated code with a **zero-shot solution** in terms of **clarity and correctness**.

### Step 1: One-Shot Prompt

**One-shot prompt** (includes one input-output example):

Example:

Input: 5

Output: 120

Now write a Python function that computes the factorial of a given number. The function should return the factorial value.

### Step 2: AI-Generated Code (One-Shot)

```
def factorial(n):
    if n < 0:
        return None

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

    return result
```

### Step 3: Zero-Shot Generated Code (For Comparison)

```
def factorial(n):
    fact = 1
    while n > 0:
        fact = fact * n
        n -= 1
    return fact
```

### Step 4: Testing the One-Shot Generated Code

Input	Expected Output	Actual Output	Result
5	120	120	Correct
1	1	1	Correct
0	1	1	Correct
3	6	6	Correct
-4	Error / Invalid	None	Handled

### Step 5: Comparison Between Zero-Shot and One-Shot Solutions

Aspect	Zero-Shot Code	One-Shot Code
Readability	Moderate	High
Input Validation	Not handled	Negative input handled
Handling of 0!	Works implicitly	Explicit and correct
Logical Clarity	Basic	Clear and structured
Robustness	Limited	Improved

### Step 6: Improvements Observed with One-Shot Prompting

1. The one-shot example helped the model **understand expected behavior** clearly.
2. The generated code is **more readable and structured**.

3. Edge cases like **negative inputs** are handled explicitly.
4. The function logic closely matches the mathematical definition of factorial.

## Conclusion

- **Zero-shot prompting** produces a basic working solution but may miss input validation.
- **One-shot prompting** improves:
  - Code clarity
  - Correctness
  - Edge-case handling
- Providing a single example significantly enhances the quality of AI-generated code.

## 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.**

## Objective

To use **few-shot prompting** by providing multiple input–output examples to generate a Python function that checks whether a given number is an **Armstrong number**, and to analyze how examples influence **code structure, accuracy, and robustness**.

### Step 1: Few-Shot Prompt

**Few-shot prompt** (multiple examples provided): Examples:

Input: 153

Output: Armstrong Number Input:

370

Output: Armstrong Number

Input: 123

Output: Not an Armstrong Number

Now write a Python function that checks whether a given number is an Armstrong number. The function should return True if it is an Armstrong number and False otherwise.

### Step 2: AI-Generated Code (Few-Shot)

```
def is_armstrong(num):
    if num < 0:
        return False

    digits = str(num)
    power = len(digits)
    total = 0

    for d in digits:
        total += int(d) ** power

    return total == num
```

### Step 3: Testing the Generated Code Normal

#### Test Cases

Input	Expected Output	Actual Output	Result
153	True	True	Correct
370	True	True	Correct
123	False	False	Correct
9474	True	True	Correct

## Boundary Value Testing

Input	Expected Output	Actual Output	Result
0	True ( $0^i = 0$ )	True	Correct
1	True	True	Correct
9	True	True	Correct
10	False	False	Correct

## Invalid Input Testing

Input	Expected Behavior	Actual Behavior	Observation
-153	False	False	Handled
"153"	Error / Reject	Type Error	Not handled
15.3	Error / Reject	Type Error	Not handled

## Step 4: Analysis – Influence of Few-Shot Prompting Impact

### on Code Structure

- The use of multiple examples encouraged:
  - Conversion of number to digits
  - Dynamic power calculation using digit count
- The logic closely follows the mathematical definition of an Armstrong number.

### Impact on Accuracy

- Few-shot prompting resulted in:
  - Correct handling of standard Armstrong numbers
  - Proper handling of boundary values like 0 and 1
  - Explicit handling of negative numbers

### Comparison with Zero-Shot / One-Shot

- **Zero-shot:** Likely to miss digit-length handling or edge cases.
- **One-shot:** Better clarity but still limited generalization.
- **Few-shot:** Most accurate and structured solution.

## **Limitations Observed**

- No explicit validation for:
  - Non-integer inputs
  - Floating-point numbers
- Additional checks would be required for production-level robustness.

## **Conclusion**

- **Few-shot prompting** significantly improves:
  - Logical correctness
  - Code readability
  - Edge-case handling
- Providing multiple examples helps the AI infer **general rules**, not just patterns.
- Few-shot prompting is the **most reliable technique** for generating accurate Python programs for computational problems.

## **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.**

### **Objective**

To design a **context-managed prompt** with explicit instructions and constraints that generates an **optimized and robust Python program** to classify a number as:

- **Prime**
- **Composite**
- **Neither**

The solution must ensure **input validation**, **efficient logic**, and allow **comparison with earlier prompting strategies**.

### Step 1: Context-Managed Prompt

**Context-managed prompt** (clear role, rules, constraints, and output expectations):

You are an expert Python developer.

Task:

Write an optimized Python function that classifies a given number as Prime,

Composite, or Neither.

Constraints:

1. Validate input – the input must be an integer.
2. Numbers less than or equal to 1 should be classified as "Neither".
3. Use an efficient algorithm (check divisibility only up to  $\sqrt{n}$ ).
4. Return only one of these strings: "Prime", "Composite", or "Neither".
5. The function should be clean, readable, and efficient.

### Step 2: AI-Generated Code (Context-Managed)

```
import math

def classify_number(n):
    if not isinstance(n, int):
        return "Invalid Input"

    if n <= 1:
        return "Neither"

    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return "Composite"

    return "Prime"
```

## Step 3: Testing the Function Normal

### Test Cases

Input	Expected Output	Actual Output	Result
2	Prime	Prime	Correct
7	Prime	Prime	Correct
10	Composite	Composite	Correct
15	Composite	Composite	Correct

### Boundary Value Testing

Input	Expected Output	Actual Output	Result
0	Neither	Neither	Correct
1	Neither	Neither	Correct
-5	Neither	Neither	Correct

### Invalid Input Testing

Input	Expected Behavior	Actual Output	Observation
7.5	Reject	Invalid Input	Handled
"10"	Reject	Invalid Input	Handled
None	Reject	Invalid Input	Handled

## Step 4: Optimization Analysis

### Efficiency

- Divisibility checks are limited to  $\sqrt{n}$ , reducing time complexity from  $O(n)$  to  $O(\sqrt{n})$ .
- Early termination on first factor improves performance.

## Input Validation

- Explicit type checking prevents runtime errors.
- Clearly distinguishes **invalid input** from valid numeric cases.

## Step 5: Comparison with Earlier Prompting Strategies

Aspect	Zero-Shot	One-Shot	Few-Shot	Context- Managed
Input Validation	Missing	Partial	Partial	Complete
Efficiency	Basic loop	Improved	Improved	Optimized
Readability	Moderate	Good	Very Good	Excellent
Reliability	Low	Medium	High	Very High

## Conclusion

- Context-managed prompting produces the most reliable and production-ready code.
- Explicit constraints guide the AI to:
  - Handle edge cases correctly
  - Optimize performance
  - Maintain clean structure
- Compared to zero-shot, one-shot, and few-shot prompting, context-managed prompting offers superior correctness, efficiency, and robustness.

## 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.**

## Objective

To use **zero-shot prompting** to generate a Python function that checks whether a given number is a **perfect number**, then evaluate the generated code by testing it with multiple inputs and identifying **missing conditions or inefficiencies**.

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

Example:  $6 \rightarrow 1 + 2 + 3 = 6$

### Step 1: Zero-Shot Prompt

**Zero-shot prompt** (no examples provided):

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.

### Step 2: AI-Generated Code

```
def is_perfect(num):
    total = 0
    for i in range(1, num):
        if num % i == 0:
            total += i
    return total == num
```

### Step 3: Testing the Program with Multiple Inputs Normal

#### Test Cases

Input	Expected Output	Actual Output	Result
6	True	True	Correct
28	True	True	Correct
12	False	False	Correct
10	False	False	Correct

## Boundary Value Testing

Input	Expected Output	Actual Output	Result
1	False	False	Correct
0	False	True	Incorrect
-6	False	True	Incorrect

## Invalid Input Testing

Input	Expected Behavior	Actual Behavior	Observation
6.5	Reject / Error	Incorrect result	Not handled
"28"	Reject / Error	Type Error	Not handled
None	Reject / Error	Type Error	Not handled

## Step 4: Identification of Missing Conditions and Inefficiencies

### Missing Conditions

#### 1. Zero and Negative Numbers

- The function incorrectly treats 0 and negative numbers as perfect.
- Perfect numbers are defined **only for positive integers greater than 1**.

#### 2. Input Validation

- No check for non-integer inputs such as strings or floating-point numbers.

### Inefficiencies

#### 1. Time Complexity

- The loop runs from 1 to num-1, resulting in **O(n)** time complexity.
- This is inefficient for large numbers.

#### 2. Unnecessary Iterations

- Divisors only need to be checked up to  $\text{num} // 2$  or, more efficiently,  $\sqrt{\text{num}}$ .

### Improved Version (After Analysis)

```
import math

def is_perfect(num):
    if not isinstance(num, int) or num <= 1:
        return False

    total = 1
    for i in range(2, int(math.sqrt(num)) + 1):
        if num % i == 0:
            total += i
            if i != num // i:
                total += num // i

    return total == num
```

### Conclusion

- **Zero-shot prompting** successfully generated a basic working solution.
- The generated code **lacked input validation** and had **inefficient logic**.
- Manual testing revealed errors for:
  - Zero
  - Negative numbers
- Optimizing divisor checks significantly improves performance.

### 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.**

## Objective

To use **few-shot prompting** with multiple input–output examples to generate a Python program that correctly determines whether a given number is **Even** or **Odd**, while also performing **proper input validation**. The goal is to analyze how examples improve **input handling** and **output clarity**.

### Step 1: Few-Shot Prompt

**Few-shot prompt** (multiple labeled examples provided): Examples:

Input: 8 Output:

Even Input: 15

Output: Odd

Input: 0 Output:

Even

Now write a Python function that determines whether a given input number is Even or Odd.

Requirements:

- Validate the input.
- The input must be an integer.
- Handle negative numbers correctly.
- Return only "Even", "Odd", or "Invalid Input"

### Step 2: AI-Generated Code (Few-Shot)

```
def even_or_odd(n):  
    if not isinstance(n, int):  
        return "Invalid Input"  
  
    if n % 2 == 0:  
        return "Even"  
    else:  
        return "Odd"
```

### **Step 3: Testing the Program**

#### **Standard Test Cases**

<b>Input</b>	<b>Expected Output</b>	<b>Actual Output</b>	<b>Result</b>
8	Even	Even	Correct
15	Odd	Odd	Correct
0	Even	Even	Correct
22	Even	Even	Correct

#### **Negative Number Testing**

<b>Input</b>	<b>Expected Output</b>	<b>Actual Output</b>	<b>Result</b>
-4	Even	Even	Correct
-7	Odd	Odd	Correct

#### **Non-Integer Input Testing**

<b>Input</b>	<b>Expected Output</b>	<b>Actual Output</b>	<b>Observation</b>
3.5	Invalid Input	Invalid Input	Handled
"10"	Invalid Input	Invalid Input	Handled
None	Invalid Input	Invalid Input	Handled

### **Step 4: Analysis – Impact of Few-Shot Prompting**

#### **Improvement in Input Handling**

- Providing examples encouraged the AI to:
  - Explicitly validate input type
  - Reject non-integer values safely
- Negative integers are handled correctly without additional logic.

#### **Improvement in Output Clarity**

- Output is restricted to **clear, predefined labels**:

- "Even"
  - "Odd"
  - "Invalid Input"
- No ambiguous or verbose output is produced.

### Comparison with Zero-Shot / One-Shot

- **Zero-shot:** Often omits input validation.
- **One-shot:** Improves clarity but may still miss invalid inputs.
- **Few-shot:** Produces the most reliable and user-friendly solution.

### Conclusion

- **Few-shot prompting** significantly improves:
  - Input validation
  - Output consistency
  - Code reliability
- The generated program correctly handles:
  - Positive integers
  - Negative integers
  - Zero
  - Invalid inputs
- Few-shot prompting is ideal for tasks requiring **both correctness and robustness.**