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.

Task:

• Record the AI-generated code.

• Test the code with multiple inputs.

• Identify any logical errors or missing edge-case handling.

DAY3.PY > ...

```python
#Write a python that checks wheter a given number is a palindrome
def is_palindrome(number):
    # Convert the number to string to check for palindrome
    str_num = str(number)
    # Check if the string is equal to its reverse
    return str_num == str_num[::-1]
# Test the function with multiple inputs
test_numbers = [121, -121, 10, 12321, 0]
for num in test_numbers:
    result = is_palindrome(num)
    print(f"Is {num} a palindrome? {result}")

```

OUTPUT:

```
PS C:\Users\nsri1\OneDrive\Desktop\AI-Assisted Coding>
Is 121 a palindrome? True
Is -121 a palindrome? False
Is 10 a palindrome? False
Is 12321 a palindrome? True
Is 0 a palindrome? True
```

Code Analysis:

Simple & concise: Uses string slicing to reverse and compare.

Works for positive integers: Correctly identifies palindromes like 121, 12321, and 0.

Identify any logical errors or missing edge-case handling:

Negative numbers → -121 returns False but should be explicitly handled.

Non-integers → Floats/strings may give unexpected results.

Leading zeros → Integers strip them (010 → 10), so palindrome check may be misleading.

Empty/None input → Raises error, no handling.

Performance → String conversion inefficient for very large numbers.

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.

Example:

Input: 5 → Output: 120

Task:

• Compare the generated code with a zero-shot solution.

• Examine improvements in clarity and correctness.

```python
# Write a python function to compute the factorial of a given number.
# Example: Input:5--> Output: 120
def factorial(n):
    if n < 0:
        return "Factorial is not defined for negative numbers"
    elif n == 0 or n == 1:
        return 1
    else:
        result = 1
        for i in range(2, n + 1):
            result *= i
        return result
# Test the factorial function with input
test_number = 5
print(f"The factorial of {test_number} is {factorial(test_number)}")

```

OUTPUT:

```
PS C:\Users\nsri1\OneDrive\Desktop\AI-Assisted Coding>
The factorial of 5 is 120
```

Code Analysis:

Compare the generated code with a zero-shot solution

Zero-Shot: Often recursive (factorial(n) = n * factorial(n-1)), concise but may miss edge cases (negatives, large inputs → recursion depth errors).

One-Shot: With example (5 → 120), AI tends to generate iterative code using loops, adds base cases (0! = 1), and sometimes error handling for negatives.

Examine improvements in clarity and correctness.

Clarity: One-shot is step-by-step, easier for beginners.

Correctness: Handles edge cases better, avoids recursion limits, more robust for large numbers.

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.

Examples:

• Input: 153 → Output: Armstrong Number

• Input: 370 → Output: Armstrong Number

• Input: 123 → Output: Not an Armstrong Number

Task:

• Analyze how multiple examples influence code structure and accuracy.

• Test the function with boundary values and invalid inputs.

```python
# Write 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
def is_armstrong(number):
    # Convert the number to string to easily iterate over digits
    str_num = str(number)
    num_digits = len(str_num)
    sum_of_powers = sum(int(digit) ** num_digits for digit in str_num)
    return sum_of_powers == number
# Test the function with given inputs
test_numbers = [153, 370, 123]
for num in test_numbers:
    result = is_armstrong(num)
    if result:
        print(f"{num} is an Armstrong Number")
    else:
        print(f"{num} is Not an Armstrong Number")

```

OUTPUT:

```
PS C:\Users\nsri1\OneDrive\Desktop\AI-Assisted Coding>
153 is an Armstrong Number
370 is an Armstrong Number
123 is Not an Armstrong Number
```

Code Analysis:

Analyze how multiple examples influence code structure and accuracy.

Multiple examples (153, 370, 123) helped the AI capture the Armstrong rule correctly and structure the code clearly.

Accuracy improved since the AI saw both Armstrong and non-Armstrong cases.

Test the function with boundary values and invalid inputs.

- 0, 1, 9474 → Armstrong Numbers
- -153 → should be Not Armstrong (needs fix)
- "153", 2.5 → invalid inputs (TypeError)

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.

```python
# Write a Python program that classifies a number as prime, composite, or neither.
def classify_number(number):
    if number <= 1:
        return "Neither prime nor composite"
    for i in range(2, int(number**0.5) + 1):
        if number % i == 0:
            return "Composite"
    return "Prime"
# Test the function with multiple inputs
test_numbers = [1, 2, 4, 7, 10, 13]
for num in test_numbers:
    classification = classify_number(num)
    print(f"{num} is classified as: {classification}")

```

OUTPUT:

```
PS C:\Users\nsri1\OneDrive\Desktop\AI-Assisted Coding>
1 is classified as: Neither prime nor composite
2 is classified as: Prime
4 is classified as: Composite
7 is classified as: Prime
10 is classified as: Composite
13 is classified as: Prime
```

Code Analysis:

Handles edge cases (n <= 1 → Neither).

Optimized check: divisibility only up to √n.

Special handling for 2 and even numbers improves efficiency.

Weaknesses:

No type validation (non-integers may break).

Returns "Neither" but wording could be standardized.

Context-Managed Prompting:

- **Clear instructions** → AI knows to handle negatives, 0, and 1 properly.
- **Constraints** → Forces optimized loop (up to √n) instead of naive full check.
- **Accuracy** → Ensures correct classification across all cases.
- **Robustness** → Type safety prevents invalid inputs

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.

Task:

• Record the AI-generated code.

• Test the program with multiple inputs.

• Identify any missing conditions or inefficiencies in the logic.

```
63  # Write a Python function that checks whether a given number is a perfect number.
64  def is_perfect_number(number):
65      if number <= 0:
66          return False
67      sum_of_divisors = sum(i for i in range(1, number) if number % i == 0)
68      return sum_of_divisors == number
69  # Test the function with multiple inputs
70  test_numbers = [6, 28, 12, 15, 496]
71  for num in test_numbers:
72      result = is_perfect_number(num)
73      if result:
74          print(f"{num} is a Perfect Number")
75      else:
76          print(f"{num} is Not a Perfect Number")
77
```

OUTPUT:

```
PS C:\Users\nsri1\OneDrive\Desktop\AI-Assisted Coding>
6 is a Perfect Number
28 is a Perfect Number
12 is Not a Perfect Number
15 is Not a Perfect Number
496 is a Perfect Number
```

Code Analysis:

1.Simple and clear logic.

2.Correctly identifies small perfect numbers (6, 28, 496).

3.Excludes non-positive numbers.

Identify any missing conditions or inefficiencies in the logic.

Inefficient → Loops through all numbers up to n-1; should only check divisors up to √n.

- **No type safety** → Breaks if input is float/string.
- **Edge case** → 1 returns False but not explicitly handled.
- **Negatives/zero** → Excluded only by <= 0 check.

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.

Examples:

• Input: 8 → Output: Even

• Input: 15 → Output: Odd

• Input: 0 → Output: Even

Task:

• Analyze how examples improve input handling and output clarity.

• Test the program with negative numbers and non-integer inputs.

```python
# Write 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
def check_even_odd(number):
    if not isinstance(number, int):
        return "Invalid input. Please enter an integer."
    if number % 2 == 0:
        return "Even"
    else:
        return "Odd"
# Test the function with give inputs
test_numbers = [8, 15, 0]
for num in test_numbers:
    result = check_even_odd(num)
    print(f"The number {num} is: {result}")

```

OUTPUT:

```
PS C:\Users\nsri1\OneDrive\Desktop\AI-Assisted Coding>
The number 8 is: Even
The number 15 is: Odd
The number 0 is: Even
```

Code Analysis:

Examples (8 → Even, 15 → Odd, 0 → Even) guide the AI to generalize correctly, add **input validation**, and give **clear outputs**.

They improve consistency: every case returns a clear message ("Even", "Odd", or "Invalid").

Test the program with negative numbers and non-integer inputs.

print(check_even_odd(-4)) # Even

print(check_even_odd(-7)) # Odd

print(check_even_odd(3.5)) # Invalid input

print(check_even_odd("10")) # Invalid input