

Assignment-3.1

Name:M.Harshith

Ht.no:2303A51444

Batch.No:21

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.

Prompt used:

write a python code to check whether a number is a palindrome or not using functions, the number input should be taken from the user.

Code:

```
# write a python code to check whether a number is a palindrome or not, the number input should be taken from the user.
def is_palindrome(n):
    cleaned = ''.join(c.lower() for c in str(n) if c.isalnum())
    return cleaned == cleaned[::-1]

user_input = input("Enter a number: ")
if is_palindrome(user_input):
    print("The number is a palindrome.")
else:
    print("The number is not a palindrome.")
```

Output:

```
PS D:\AI assist> python -u "d:\AI assist\code3.py"
• Enter a number: 121
The number is a palindrome.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
• Enter a number: 2
The number is a palindrome.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
• Enter a number: -5
The number is a palindrome.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
• Enter a number: -121
The number is a palindrome.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
• Enter a number: 123
The number is not a palindrome.
❖ PS D:\AI assist> 
```

Justification:

A zero-shot prompt was used to generate a Python program that checks whether a number is a palindrome without providing any examples.

The AI-generated code was tested with multiple inputs, and it correctly identified positive palindromes such as 121 and 2.

However, it failed to handle negative numbers properly, revealing a missing edge-case check in the logic.

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-shots solution.
- Examine improvements in clarity and correctness.

Prompt used:

write a python code to calculate the factorial of a number using functions, the number input should be taken from the user. example: input=5, output=120

Code:

```
# write a python code to calculate the factorial of a number using functions, the number input should be taken from the user.
#example: input=5, output=120
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
user_input = int(input("Enter a number: "))
result = factorial(user_input)
print(f"The factorial of {user_input} is {result}.")
```

Output:

```
PS D:\AI assist> python -u "d:\AI assist\code3.py"
• Enter a number: 5
The factorial of 5 is 120.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
• Enter a number: 0
The factorial of 0 is 1.
• PS D:\AI assist> python -u "d:\AI assist\code3.py"
Enter a number: 6
The factorial of 6 is 720.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
• Enter a number: 1
The factorial of 1 is 1.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
• Enter a number: 10
The factorial of 10 is 3628800.
❖ PS D:\AI assist> 
```

Justification:

The one-shot prompting approach generates clearer and more accurate factorial code by providing a sample input-output example.

This guidance helps the AI correctly implement base cases such as 0 and 1, improving correctness.

Compared to zero-shot prompting, the one-shot solution is more reliable and easier to understand.

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.

Prompt used:

Write a Python code using functions, that checks whether a given number is an Armstrong number or not.

example Test cases:

Input:153→Output:ArmstrongNumber

Input:370→Output:ArmstrongNumber

Input:123→Output:NotanArmstrongNumber

Code:

```
# Write a Python code using functions, that checks whether a given number is an Armstrong number or not.
#example Test cases:
# Input: 153 → Output: Armstrong Number
# Input: 370 → Output: Armstrong Number
# Input: 123 → Output: Not an Armstrong Number
def is_armstrong(n):
    num_str = str(n)
    num_digits = len(num_str)
    sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)
    return sum_of_powers == n
user_input = int(input("Enter a number: "))
if is_armstrong(user_input):
    print(f"{user_input} is an Armstrong Number.")
else:
    print(f"{user_input} is Not an Armstrong Number.")
```

Output:

```

PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: 153
153 is an Armstrong Number.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: 270
270 is Not an Armstrong Number.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: 370
370 is an Armstrong Number.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: 123
123 is Not an Armstrong Number.
⊗ PS D:\AI assist> python -u "d:\AI assist\code3.py"
Enter a number: -153
Traceback (most recent call last):
  File "d:\AI assist\code3.py", line 34, in <module>
    if is_armstrong(user_input):
  File "d:\AI assist\code3.py", line 31, in is_armstrong
    sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)
  File "d:\AI assist\code3.py", line 31, in <genexpr>
    sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)
ValueError: invalid literal for int() with base 10: '-'
PS D:\AI assist> python -u "d:\AI assist\code3.py"
⊗ Enter a number: 12.5
Traceback (most recent call last):
  File "d:\AI assist\code3.py", line 33, in <module>
    user_input = int(input("Enter a number: "))
ValueError: invalid literal for int() with base 10: '12.5'
❖ PS D:\AI assist>

```

Justification:

Multiple examples in the few-shot prompt help the AI clearly understand the Armstrong number pattern, resulting in well-structured and accurate logic for valid inputs like 153 and 370.

The code correctly calculates digit powers based on the number of digits, improving accuracy compared to zero-shot or one-shot prompts.

However, the program still fails for negative and non-integer inputs, showing that few-shot prompting improves logic but does not guarantee complete edge-case handling.

Question4:Context-ManagedPrompting(OptimizedNumber 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.

Task:

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

Prompt used:

write an optimized python program that takes a input number from the user and classifies it as prime number, Composite number and neither prime nor composite

Constraints:

1. should handle negative number, Zero, one, and non-integer inputs
2. use an efficient prime-checking logic
3. keep the code readable with meaningful variable names and comments.
4. Display the clear results

Code:


```

# write an optimized python program that takes a input number from the user and classifies it as
# prime number, Composite number and neither prime nor composite
# Constarints:
# shuold handle negative number, Zero, one, and non-integer inputs
# use an efficcient prime-checking logic
# keep the code readable with meaningful variable names and comments.
# Display the clear results
def classify_number(input_value):
    try:
        number = int(input_value)
        if number < 0:
            return "Negative numbers are neither prime nor composite."
        elif number == 0:
            return "Zero is neither prime nor composite."
        elif number == 1:
            return "One is neither prime nor composite."
        else:
            for i in range(2, int(number**0.5) + 1):
                if number % i == 0:
                    return f"{number} is a Composite number."
            return f"{number} is a Prime number."
    except ValueError:
        return "Invalid input. Please enter an integer."
user_input = input("Enter a number: ")
result = classify_number(user_input)
print(result)

```

Output:

```

PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: 2
2 is a Prime number.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: 4
4 is a Composite number.
● PS D:\AI assist> python -u "d:\AI assist\code3.py"
Enter a number: 1
One is neither prime nor composite.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: -7
Negative numbers are neither prime nor composite.
● PS D:\AI assist> python -u "d:\AI assist\code3.py"
Enter a number: 2.5
Invalid input. Please enter an integer.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: abc
Invalid input. Please enter an integer.

```

Justification:

Compared to zero-shot and one-shot prompting, context-managed prompting produces more structured and optimized code by clearly defining constraints and expectations.

It improves input validation and efficiency by explicitly instructing the AI to handle edge cases and use optimized logic.

Overall, context-managed prompting results in more reliable and production-ready code than earlier prompting strategies.

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.

Prompt used:

write a python code that take number input from the user to check if the number is perfect number or not using functions.

Code:

```
# write a python code that take number input from the user to check if the number is perfect number or not using functions.

def is_perfect_number(n):
    if n <= 1:
        return False
    divisors_sum = sum(i for i in range(1, n) if n % i == 0)
    return divisors_sum == n

user_input = int(input("Enter a number: "))
if is_perfect_number(user_input):
    print(f"{user_input} is a Perfect Number.")
else:
    print(f"{user_input} is not a Perfect Number.")
```

Output:

```
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: 6
6 is a Perfect Number.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: 496
496 is a Perfect Number.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: 28
28 is a Perfect Number.
● PS D:\AI assist> python -u "d:\AI assist\code3.py"
Enter a number: 1
1 is not a Perfect Number.
● PS D:\AI assist> python -u "d:\AI assist\code3.py"
Enter a number: 0
0 is not a Perfect Number.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: -6
-6 is not a Perfect Number.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
⊗ Enter a number: 2.5
Traceback (most recent call last):
  File "d:\AI assist\code3.py", line 76, in <module>
    user_input = int(input("Enter a number: "))
ValueError: invalid literal for int() with base 10: '2.5'
```

Justification:

The zero-shot AI-generated perfect number program lacks proper input validation for negative and non-integer values.

Its logic is inefficient due to unnecessary full-range iteration and missing square-root optimization.

These limitations highlight the need for explicit constraints when using zero-shot prompting.

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.

Prompt used:

Write a Python code using functions, that checks whether a given number is Even or odd. handle negative numbers and Reject non-integer inputs

example Testcases: Input: 8

→ Output: Even Input: 15

→ Output: Odd Input: 0

→ Output: Even

Code:

```
# Write a Python code using functions, that checks whether a given number is Even or odd. handle negative numbers and Reject non-in
#example Test cases:
# Input: 8 → Output: Even
# Input: 15 → Output: Odd
# Input: 0 → Output: Even
def check_even_odd(input_value):
    try:
        number = int(input_value)
        if number % 2 == 0:
            return f"{number} is Even."
        else:
            return f"{number} is Odd."
    except ValueError:
        return "Invalid input. Please enter an integer."
user_input = input("Enter a number: ")
result = check_even_odd(user_input)
print(result)
```

Output:

```
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: 8
8 is Even.
● PS D:\AI assist> python -u "d:\AI assist\code3.py"
Enter a number: 15
15 is Odd.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: 0
0 is Even.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: -2
-2 is Even.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: -5
-5 is Odd.
PS D:\AI assist> python -u "d:\AI assist\code3.py"
● Enter a number: abc
Invalid input. Please enter an integer.
❖ PS D:\AI assist> █
```

Justification:

Multiple examples in the few-shot prompt help the AI clearly understand how to classify numbers as even or odd, including boundary cases like zero.

The examples guide the AI to improve input handling by considering negative values and validating integer inputs.

This results in clearer output messages and more accurate classification compared to zero-shot prompting.