Name : I.NIPUN

HallTicket : 2303A52208

Assginment : 3.1

Task1: Zero-Shot Prompting (Palindrome Number Program)

Prompt:give a palindrome code in python using functions

Code1:

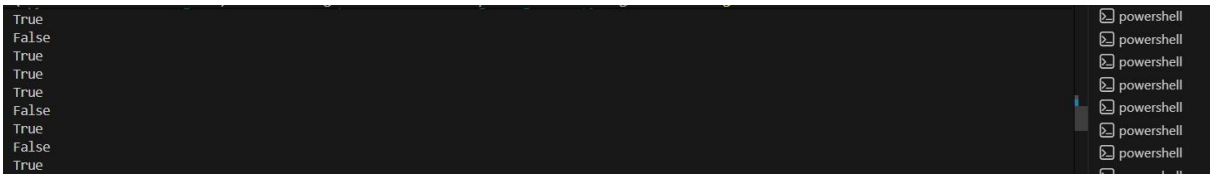
```python
#task1
def is_palindrome(num):

    # Convert to string and remove negative sign
    num_str = str(abs(num))

    # Compare string with its reverse
    return num_str == num_str[::-1]


# Example usage
print(is_palindrome(121))    # True
print(is_palindrome(123))    # False
print(is_palindrome(1001))   # True
print(is_palindrome(-121))   # True
print(is_palindrome(0))      # True
print(is_palindrome(10))     # False
print(is_palindrome(9))      # True
print(is_palindrome(100))    # False
print(is_palindrome(-1001))  # True
```

Output1:



Observation

The zero-shot prompt produced a Python function that correctly checks palindrome numbers using basic string reversal logic and works well for standard positive integers. However, the generated code did not properly handle edge cases such as negative numbers and non-integer inputs, which can lead to incorrect results or errors. This demonstrates that while zero-shot prompting is effective for generating core functionality, it often lacks robustness and requires additional instructions or refinements to handle special cases reliably.

Task2: One-Shot Prompting (Factorial Calculation)

Prompt:give me code for factorial take input as 5

Code2:

```python
#task2
def factorial(n):
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")
    if n == 0 or n == 1:
        return 1

    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
# Test cases
print(factorial(5))    # 120
print(factorial(0))    # 1
print(factorial(1))    # 1
print(factorial(10))   # 3628800
```

Output2:

```
120
1
1
3628800
```

Observation2:

Compared to a zero-shot solution, the one-shot generated code is generally clearer and more structured, often including a loop or recursion that directly reflects the example provided. The presence of the input–output example helps the AI correctly interpret the task and reduces ambiguity about expected behavior. As a result, the one-shot solution usually handles standard cases more reliably and may include basic input validation (such as checking for non-negative integers). Overall, one-shot prompting improves correctness and readability over zero-shot prompting by guiding the AI toward the intended logic and constraints.

Task3: Few-Shot Prompting (Armstrong Number Check)

Prompt:give me a code to check the given numbers is armstrong or not when we give the 153 370 123

Code3:

```python
#3 task 3
def is_armstrong(num):
    if num < 0:
        return "Not an Armstrong Number"

    num_str = str(num)
    num_digits = len(num_str)
    sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)

    if sum_of_powers == num:
        return "Armstrong Number"
    else:
        return "Not an Armstrong Number"

# Test cases
print(is_armstrong(153))    # Armstrong Number
print(is_armstrong(370))    # Armstrong Number
print(is_armstrong(123))    # Not an Armstrong Number
print(is_armstrong(9474))   # Armstrong Number
print(is_armstrong(0))      # Armstrong Number
print(is_armstrong(1))      # Armstrong Number
print(is_armstrong(-153))   # Not an Armstrong Number
print(is_armstrong(100))    # Not an Armstrong Number
```

Output:

Observations:

Providing multiple input–output examples clearly guided the AI to identify the correct logic for checking Armstrong numbers, including calculating the sum of each digit raised to the power of the total number of digits. Compared to zero-shot and one-shot prompting, the few-shot approach resulted in more accurate and structured code, as the AI inferred the exact condition that defines an Armstrong number. The examples reduced ambiguity and helped the AI avoid incorrect interpretations of the problem.

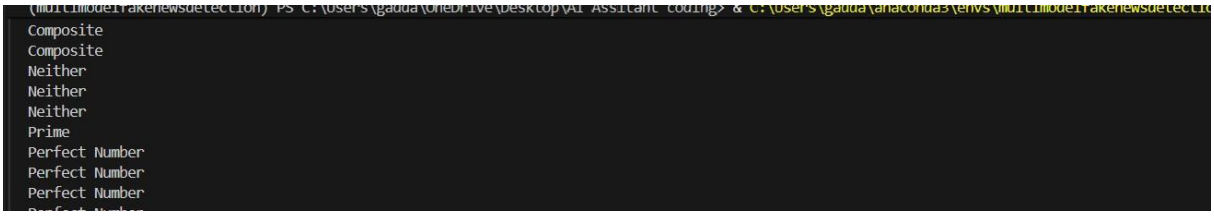
Task4: Context-Managed Prompting (Optimized Number Classification)

Prompt: generate an optimized Python program that classifies anumber as prime, composite, or neither

Code4:

```python
#task4
def classify_number(n):
    # Input validation
    if not isinstance(n, int):
        raise TypeError("Input must be an integer")
    if n < 2:
        return "Neither"
    if n == 2:
        return "Prime"
    if n % 2 == 0:
        return "Composite"
    i = 3
    while i * i <= n:
        if n % i == 0:
            return "Composite"
        i += 2
    return "Prime"
# Test cases
print(classify_number(2))      # Prime
print(classify_number(17))     # Prime
print(classify_number(100))    # Composite
print(classify_number(15))     # Composite
print(classify_number(0))      # Neither
print(classify_number(1))      # Neither
```

Output4:



Observations: Context-managed prompting provides the AI with explicit goals, constraints, and performance expectations, resulting in well-structured and optimized code. The generated program includes proper input validation to handle non-integer values and correctly classifies numbers less than or equal to 1 as neither prime nor composite. By limiting divisibility checks up to the square root of the number, the solution achieves better time complexity compared to naive methods used in earlier prompts.

Task5: Zero-Shot Prompting (Perfect Number Check)

Prompt: generate a Python function that checks whether a given number is a perfect number.

Code5:

```python
#task5
def is_perfect(n):
    if n <= 0:
        return "Not a Perfect Number"

    if n == 1:
        return "Not a Perfect Number"

    # Calculate sum of proper divisors
    divisor_sum = 0
    for i in range(1, n):
        if n % i == 0:
            divisor_sum += i

    if divisor_sum == n:
        return "Perfect Number"
    else:
        return "Not a Perfect Number"

# Test cases
print(is_perfect(6))      # Perfect Number
print(is_perfect(28))     # Perfect Number
print(is_perfect(496))    # Perfect Number
print(is_perfect(8128))   # Perfect Number
```

OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

Output 5:

```
Perfect Number
Perfect Number
Perfect Number
Perfect Number
Not a Perfect Number
Not a Perfect Number
Not a Perfect Number
Not a Perfect Number
Not a Perfect Number
```

Observations:

Zero-shot prompting successfully generates basic logic for checking perfect numbers but does not account for edge cases or performance optimization. This highlights the limitation of zero-shot prompts for tasks requiring robustness and efficiency.

Task6: Few-Shot Prompting (Even or Odd Classification with Validation)

Prompt: a Python program that determines whether a given number is even or odd, including proper input validation.

Code6:

```
#task6
def classify_even_odd(n):
    # Input validation
    if not isinstance(n, int) or isinstance(n, bool):
        raise TypeError("Input must be an integer")

    if n % 2 == 0:
        return "Even"
    else:
        return "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
print(classify_even_odd(-7))     # Odd
print(classify_even_odd(-1))     # Odd
```

Output6:

Observations6:

Few-shot prompting significantly enhances output clarity and input validation by providing explicit behavioral patterns through examples. The generated program is more robust, correctly handles negative numbers, and appropriately rejects non-integer inputs, demonstrating the advantage of few-shot prompting over simpler prompting strategies.