

(AAC) Lab-Assignment-3.1

P.Sai Bunny

2303A51911

Batch-12

Task 1:

- Record the AI-generated code.
- Test the code with multiple inputs.
- Identify any logical errors or missing edge-case handling.

Prompt:

#One shot

#Factorial Calculator #Example:

#input: 5

#output: 120

Code:

```
main.py > ...
1 #Zero shot
2 #Palindrom Number Checker
3 def is_palindrome(n):
4     s = str(n)
5     return s == s[::-1]
6
7 num = int(input("Enter a number: "))
8 if is_palindrome(num):
9     print(f"{num} is a palindrome.")
10 else:
11     print(f"{num} is not a palindrome.")
```

Output:

```
Enter a number: 123
123 is not a palindrome number.
PS C:\Users\admin\OneDrive\Desktop\AI ASSISTED> 121
121
PS C:\User> & C:/Users/admin/AppData/Local/Programs/Python/Python311/python.exe
D/3.1.1.py"
```

Explanation:

Zero-shot prompting relies entirely on the AI's prior knowledge without examples, which may lead to correct but minimally robust solutions. The generated palindrome code typically works

for standard positive integers but may ignore edge cases like negative numbers or non-integer inputs. Testing with multiple values helps reveal logical gaps. This highlights the limitations of zero-shot prompting in handling validations.

Task 2:

- Compare the generated code with a zero-shot solution.
- Examine improvements in clarity and correctness.

Prompt:

#few shot

#Armstrong Numbers

#Examples: #Input:

153

#Output: True

#Input: 123

#Output: False

Code:

```
main.py > ...
1 #One shot
2 #Factorial Calculator
3 #Example:
4 #input: 5
5 #output: 120
6
7 def factorial(n):
8     if n == 0 or n == 1:
9         return 1
10    else:
11        return n * factorial(n - 1)
12
13 num = int(input("Enter a number: "))
14 print(f"The factorial of {num} is {factorial(num)}")
```

Output:

```
Enter a number: 5
Factorial of 5 is 120
& C:/Users/admin/AppData/Local/Programs/Python/Python311/python.exe
```

Explanation:

One-shot prompting provides a single example that helps the AI infer the required logic more accurately. Compared to zero-shot solutions, the generated factorial function is usually clearer and more reliable. The example improves correctness by guiding base case handling (e.g., factorial of 0). This demonstrates how minimal guidance enhances code quality.

Task 3:

- **Analyze how multiple examples influence code structure and accuracy.**
- **Test the function with boundary values and invalid inputs.** Prompt- #write a code using functions to reverse a given string without using built-in functions

Code:

```

main.py > ...
11 def is_armstrong_number(num):
12     digits = str(num)
13     num_digits = len(digits)
14
15     # Calculate the sum of each digit raised to the power of num_digits
16     sum_of_powers = sum(int(digit) ** num_digits for digit in digits)
17
18     # An Armstrong number is equal to the sum of its own digits each raised to the power of the number of digits
19     return sum_of_powers == num
20
21 # Test cases
22 print(is_armstrong_number(153)) # Output: True
23 print(is_armstrong_number(123)) # Output: False
24 print(is_armstrong_number(9474)) # Output: True

```

Output:

```

rs/admin/OneDrive/Desktop/AI ASSISTED/3.1.3.py"
True
False
True
PS C:\Users\admin\OneDrive\Desktop\AI ASSISTED>

```

Explanation:

Few-shot prompting uses multiple examples, allowing the AI to better understand patterns and constraints. The generated Armstrong number function is generally more accurate and structured. Providing varied examples improves handling of digit-based calculations. Testing boundary and invalid inputs exposes remaining weaknesses in input validation.

Task 4:

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

Prompt:

#optimised number classifier(prime/composite/neither)

Code-

```

main.py > ...
1 #optimised number classifier(prime/composite/neither)
2 def classify_number(n):
3     if n <= 1:
4         return "neither"
5     if n == 2:
6         return "prime"
7     if n % 2 == 0:
8         return "composite"
9
10    for i in range(3, int(n**0.5) + 1, 2):
11        if n % i == 0:
12            return "composite"
13
14    return "prime"
15
16 # Example usage
17 number = int(input("Enter a number: "))
18 result = classify_number(number)
19 print(f"The number {number} is classified as: {result}")

```

Output:

```

rs/admin/OneDrive/Desktop/AI ASSISTED/3.1.4.py"
Enter a number: 2
2 is prime.
PS C:\User> & C:/Users/admin/AppData/Local/Programs/Python/Python311/python.exe

```

Explanation:

Context-managed prompting clearly defines rules, constraints, and expectations, resulting in more efficient and optimized code. The generated program correctly classifies numbers while validating inputs such as negative values and non-integers. Compared to earlier prompting strategies, the logic is more optimized and readable. This approach produces the most reliable output.

Task 5:

- Record the AI-generated code.
- Test the program with multiple inputs.
- Identify any missing conditions or inefficiencies in the logic.

Prompt:

#Zero shot prompting example

#perfect number checker

Code-

```
main.py > ...
1 #Zero shot prompting example
2 #perfect number checker
3 def is_perfect_number(n):
4     if n < 1:
5         return False
6     divisors_sum = sum(i for i in range(1, n) if n % i == 0)
7     return divisors_sum == n
8 # Example usage
9 number = int(input("Enter a number to check if it's a perfect number: "))
10 if is_perfect_number(number):
11     print(f"{number} is a perfect number.")
12 else:
13     print(f"{number} is not a perfect number.")
14 #Output: 28 is a perfect number.
```

Output:

```
rs/admin/OneDrive/Desktop/AI ASSISTED/3.1.5.py"
Enter a number: 222
222 is not a perfect number.
PS C:\Users\admin\OneDrive\Desktop\AI ASSISTED> █
```

Explanation:

In zero-shot prompting, the AI generates logic based on general understanding, which may be functionally correct but inefficient. The perfect number check often includes unnecessary iterations and lacks input validation. Testing reveals performance issues for larger numbers. This shows the trade-off between simplicity and efficiency.

Task 6:

- Analyze how examples improve input handling and output clarity.

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

Prompt:

#few shot prompting example

#Even/Odd classification with validation #Examples:

#Input: 8

#Output: Even

#Input: 15

#Output: Odd

#Input: 0

#Output: Even

Code:

```
main.py > ...
1 #few shot prompting example
2 #Even/Odd classification with validation
3 #Examples:
4 #Input: 8
5 #Output: Even
6
7 #Input: 15
8 #Output: Odd
9
10 #Input: 0
11 #Output: Even
12
13 def classify_number(num):
14     if not isinstance(num, int):
15         return "Invalid input"
16     return "Even" if num % 2 == 0 else "Odd"
17 # Test cases
18 print(classify_number(8)) # Output: Even
19 print(classify_number(15)) # Output: Odd
20 print(classify_number(0)) # Output: Even
21 print(classify_number(-3)) # Output: Odd
22 print(classify_number(-22)) # Output: Even
```

Output:

```
PS C:\User> & C:/Users/admin/AppData/Local/Programs/Python/Python316/Python.exe C:/Users/admin/AppData/Local/Programs/Python/Python316/Python.exe D/3.1.6.py
even
None
PS C:\Users\admin\OneDrive\Desktop\AI ASSISTED>
```

Explanation:

Few-shot examples help the AI include proper input validation and clear output formatting. The generated program correctly handles zero and negative numbers. Compared to zero-shot output, the logic is more user-friendly and robust. Examples significantly improve both correctness and clarity.