Lab-Assignment-3.1

Name: Balaji

Hall ticket: 2303A51572

Batch-11

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:

assignment3.1.py > ...

```python
#zero shot
#palindrome checker
def is_palindrome(s):
    s = s.lower().replace(" ", "")
    return s == s[::-1]
string = input("Enter a string to check if it's a palindrome: ")
if is_palindrome(string):
    print(f'"{string}" is a palindrome.')
else:
    print(f'"{string}" is not a palindrome.')
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\Rishik\OneDrive\Desktop\AI> p^C
PS C:\Users\Rishik\OneDrive\Desktop\AI> & C:/Users/Rishik/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/Rishik/OneDrive/Desktop/AI/assignment3.1.py
Enter a string to check if it's a palindrome: 321
"321" is not a palindrome.
PS C:\Users\Rishik\OneDrive\Desktop\AI>
```

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:

assignment3.1.py > ...

```python
#zero shot
#palindrome checker
#example
#input:5
#output:120
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
num = int(input("Enter a number to find its factorial: "))
result = factorial(num)
print(f"The factorial of {num} is {result}")
```

Output:

```
p/AI/assignment3.1.py
Enter a string to check if it's a palindrome: 321
"321" is not a palindrome.
PS C:\Users\Rishik\OneDrive\Desktop\AI> & C:/Users/Rishik/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/Rishik/OneDrive/Desktop/AI/assignment3.1.py
Enter a number to find its factorial: 5
The factorial of 5 is 120
PS C:\Users\Rishik\OneDrive\Desktop\AI>
```

Explanation:

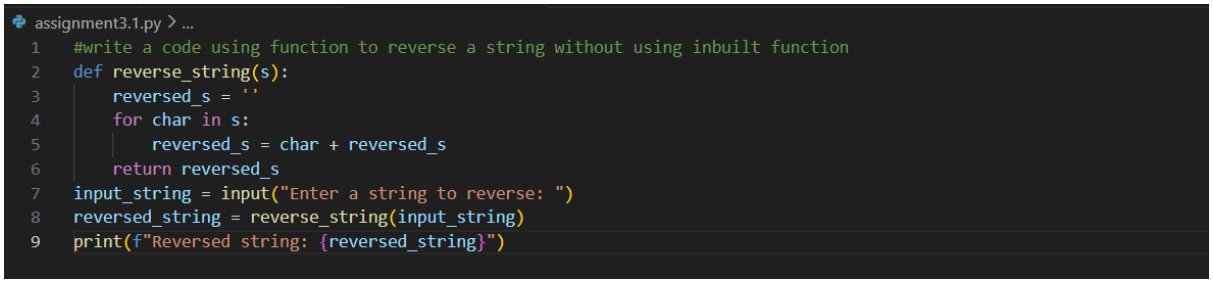
One-shot prompting provides a single example that helps the AI infer the required logic more accurately. Compared to zero-shot solutions, the generated factorial function is usually clearer and more reliable. The example improves correctness by guiding base case handling (e.g., factorial of 0). This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

Task 3:

• Analyze how multiple examples influence code structure and accuracy.

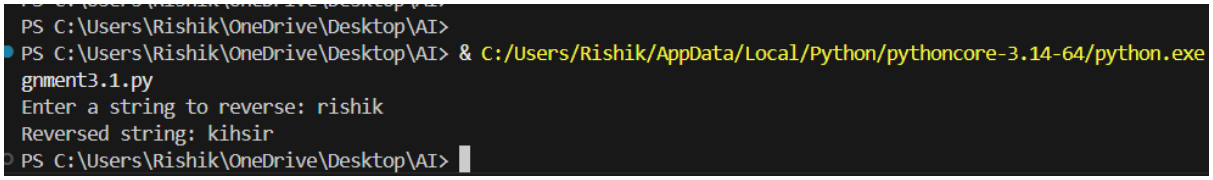
• Test the function with boundary values and invalid inputs.Prompt- #write a code using functions to reverse a given string without using built-in functions

Code:



```
assignment3.1.py > ...
1   #write a code using function to reverse a string without using inbuilt function
2   def reverse_string(s):
3       reversed_s = ''
4       for char in s:
5           reversed_s = char + reversed_s
6       return reversed_s
7   input_string = input("Enter a string to reverse: ")
8   reversed_string = reverse_string(input_string)
9   print(f"Reversed string: {reversed_string}")
```

Output:



```
PS C:\Users\Rishik\OneDrive\Desktop\AI>
PS C:\Users\Rishik\OneDrive\Desktop\AI> & C:/Users/Rishik/AppData/Local/Python/pythoncore-3.14-64/python.exe
gnment3.1.py
Enter a string to reverse: rishik
Reversed string: kihsir
PS C:\Users\Rishik\OneDrive\Desktop\AI>
```

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-

assignment3.1.py > ...

```python
#optimised number classifier(prime/composite/neither)
def classify_number(num):
    if num <= 1:
        return "neither prime nor composite"
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return "composite"
    return "prime"
number = int(input("Enter a number to classify: "))
result = classify_number(number)
print(f"{number} is {result}.")
```

Output:

```
Enter a number to classify: 7
7 is prime.
PS C:\Users\Rishik\OneDrive\Desktop\AI> & C:/Users/Rishik/AppData/Local/Python/pythoncore-3.14-64/python.exe
gnment3.1.py
Enter a number to classify: 10
10 is composite.
PS C:\Users\Rishik\OneDrive\Desktop\AI>
```

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:

```python
#zero shot prompting example
#perfect number checker with function
def is_perfect(num):
    if num < 1:
        return False
    sum_of_divisors = 0
    for i in range(1, num):
        if num % i == 0:
            sum_of_divisors += i
    return sum_of_divisors == num
number = int(input("Enter a number to check if it's a perfect number: "))
if is_perfect(number):
    print(f"{number} is a perfect number.")
else:
    print(f"{number} is not a perfect number.")
```

Output:

```
Enter a number to check if it's a perfect number: 28
28 is a perfect number.
PS C:\Users\Rishik\OneDrive\Desktop\AI> & C:/Users/Rishik/AppData/Local/Python/pythoncore-3.14-64/python.exe
gnment3.1.py
Enter a number to check if it's a perfect number: 153
153 is not a perfect number.
PS C:\Users\Rishik\OneDrive\Desktop\AI>
```

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:

```python
#zero shot prompting example
#even/odd classification with validation
#example input: 4
#output: 4 is even
#example input: 7
#output: 7 is odd
def classify_even_odd(num):
    if not isinstance(num, int):
        return "Invalid input. Please enter an integer."
    if num % 2 == 0:
        print(f"{num} is even.")
    else:
        print(f"{num} is odd.")
# Example usage
classify_even_odd(4)  # Output: "4 is even."
classify_even_odd(7)  # Output: "7 is odd."
classify_even_odd("a")  # Output: "Invalid input. Please enter an integer."
classify_even_odd(3.5)  # Output: "Invalid input. Please enter an integer."
classify_even_odd(-2)  # Output: "-2 is even."
```

Output:

```
PS C:\Users\Rishik\OneDrive\Desktop\AI>
PS C:\Users\Rishik\OneDrive\Desktop\AI> & C:/Users/Rishik/AppData/Local/Python/pythoncore-3.14-64/python.exe
4 is even.
7 is odd.
-2 is even.
PS C:\Users\Rishik\OneDrive\Desktop\AI>
```

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.