

Lab Assignment # 03

Course Title : **AI Assistant Coding**
Name of Student : **BG.Sreevani**
Enrollment No. : **2303A54066**
Batch No. : **48**

Lab 3: Prompt Engineering – Improving Prompts and Context Management

Task 1: Zero-Shot Prompting (Palindrome Number Program)

❖ Scenario: 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 zero short prompt python program to generate a Python function that checks whether a given number is a palindrome.

• Screenshot of Generated Code:

The screenshot shows the Microsoft Visual Studio Code interface. On the left, the Explorer sidebar shows a folder named 'AI' containing several Python files: armstrong_checker.py, even_odd_checker.py, factorial_calculator.py, factorial_comparison.py, number_classifier.py, palindrome_checker.py, and perfect_number_checker.py. The 'palindrome_checker.py' file is open in the center editor window. The code defines a function `is_palindrome` that checks if a number is a palindrome by converting it to a string and comparing it to its reverse. It also includes a main program that prompts the user for a number and prints the result. A large callout box on the right side of the screen provides the AI-generated prompt: "write a python programme to check where a given number is plaidrome or not, if yes then prints true otherwise prints false". Below this, the AI suggests creating a Python program to check if a number is a palindrome. The terminal at the bottom shows the command `python palindrome_checker.py` being run and the output of the program running successfully.

- Inputs & Outputs:

Number	Result	Reason
121	✓ True	Reads same forwards and backwards
123	✗ False	Not a palindrome
0	✓ True	Single digit
-121	✓ True	Ignores negative sign
1001	✓ True	Symmetrical
10	✗ False	Not a palindrome
99	✓ True	Repeating digits
12321	✓ True	Symmetrical

- Short Explanation of the Code:

The code checks if a number is the same when read it forwards or backwards. It turns the number into text, flips it, and compares both versions. If they match, it's a palindrome. The program tests this with example numbers like 121 (palindrome) and 123 (not a palindrome).

Task 2: One-Shot Prompting (Factorial Calculation)

❖ Scenario: 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.

- Prompt used:

write a zero short prompt python program to generate a Python function that checks whether a given number is a palindrome.

Screenshot of Generated Code:

```

# Program to calculate the factorial of a given number
def factorial(num):
    """Calculate the factorial of a given number.
    Factorial of n (n!) = n * (n-1) * (n-2) * ... * 1"""
    # Handle edge cases
    if num < 0:
        return "Factorial is not defined for negative numbers"
    elif num == 0 or num == 1:
        return 1
    # Calculate factorial using loop
    result = 1
    for i in range(2, num + 1):
        result *= i
    return result
# Main program
if __name__ == "__main__":
    # Get input from user
    number = int(input("Enter a number: "))
    # Calculate factorial
    fact = factorial(number)
    # Print result
    print(f"Factorial of {number} is: {fact}")

```

- **Comparison of Code:**

Zero-Shot Code	One-Shot Code
<pre>def factorial(n): if n <= 1: return 1 return n * factorial(n - 1)</pre>	<pre>def factorial(n): """Compute the factorial of a given number.""" 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</pre>

- **Comparison Analysis of Code:**

Aspect	Generated	Zero-Shot	Winner
Clarity	Clear, explicit loop-based approach; easy to follow	Concise but less obvious; requires understanding recursion	Generated ✓
Error Handling	Validates negative inputs with descriptive message	No error handling; silently fails on negatives	Generated ✓

- **Key Improvements of Code:**

1. **Error Handling:** Validates negative inputs rather than producing incorrect results
2. **Documentation:** Includes docstring for clarity and IDE support
3. **Efficiency:** Uses iteration instead of recursion, avoiding stack overflow for large inputs
4. **Edge Case Handling:** Explicitly manages 0 and 1 as base cases
5. **Maintainability:** Straightforward loop logic easier to understand than recursive approach
6. **Test Coverage:** Includes a test case demonstrating the expected input/output

Task 3: Few-Shot Prompting (Armstrong Number Check)

❖ *Scenario: 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:

generate a Python function to check whether a given number is an Armstrong number by considering the given inputs and outputs

- **Screenshot of Generated Code:**

```

1 armstrong_checker.py > ...
5 def is_armstrong(num):
12     num_str = str(num)
13     num_digits = len(num_str)
14     # Calculate sum of each digit raised to power of number of digits
15     sum_of_powers = 0
16     for digit in num_str:
17         sum_of_powers += int(digit) ** num_digits
18     # Check if sum equals original number
19     return sum_of_powers == num
20
21 # Main program
22 if __name__ == "__main__":
23     # Get input from user
24     number = int(input("Enter a number: "))
25     # Check if it's an Armstrong number
26     result = is_armstrong(number)
27     # Print result
28     print(result)
29     # Optional: Print detailed explanation
30     if result:
31         print(f"\n{number} is an Armstrong number!")
32         num_str = str(number)
33         num_digits = len(num_str)

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS + ... < X & "C:/Users/Sreevani B G/OneDrive/Desktop/AI/armstrong_checker.py"

Enter a number: 153
True

Ln 27, Col 18 Spaces: 4 UTF-8 CRLF {} Python 3.13.9 (Microsoft Store) Go Live

- **Short Explanation of the Code:**

Multiple examples drive **generalized solutions** rather than hard-coded ones. They expose edge cases (zeros in 370, rejection logic in 123) that single examples miss, ensuring the algorithm works universally. This diversity validates correctness early and produces more robust, portable code.

- **Function with boundary values and invalid inputs of the Code:**

- ✓ Zero (edge case): 0 → True
- ✓ Single digit: 1 → True
- ✓ Single digit boundary: 9 → True
- ✓ Two digits: 10 → False
- ✓ Valid Armstrong: 153 → True
- ✓ Valid Armstrong with 0: 370 → True
- ✓ Not Armstrong: 123 → False
- ✓ 4-digit Armstrong: 9474 → True
- ✓ Negative number: -153 → None
- ✓ Float input: 3.14 → None
- ✓ String input: 153 → None

Task 4: Context-Managed Prompting (Optimized Number Classification)

❖ *Scenario: 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:**

Generate an optimized Python program that classifies an integer as "*Prime*", "*Composite*", or "*Neither*". Constraints: check divisibility only up to \sqrt{n} , skip even numbers after 2, handle edge cases ($n \leq 1 \rightarrow$ Neither, $n=2 \rightarrow$ Prime). Input via `input ()`, output must be one of the three strings.

- **Screenshot of Generated Code:**

```

1 # Program to classify a number as Prime, Composite, or Neither
2 def classify_number(num):
3     """Classify a given number as:
4         - Prime: Number divisible only by 1 and itself (greater than 1)
5         - Composite: Number with more than 2 factors
6         - Neither: Numbers like 0, 1, and negative numbers
7     Returns a string classification."""
8     # Handle special cases (Neither)
9     if num <= 1:
10         return "Neither"
11     # Check if number is prime
12     # A prime number is only divisible by 1 and itself
13     if num == 2:
14         return "Prime"
15     # Even numbers (except 2) are composite
16     if num % 2 == 0:
17         return "Composite"
18     # Check odd divisors up to square root of num
19     # If num has a divisor, it must be <= sqrt(num)
20     for i in range(3, int(num ** 0.5) + 1, 2):
21         if num % i == 0:
22             return "Composite"
    
```

The screenshot shows the VS Code interface with the code editor open to the `number_classifier.py` file. The code implements a classification algorithm based on the constraints provided. The terminal below shows the program running and correctly identifying the number 5 as prime. A sidebar on the right contains a summary of the program's behavior and a section titled "Example outputs" showing the classification of various numbers.

- **Short Explanation of the Code:**

The program takes a number and checks if it's Prime, Composite, or Neither. Numbers 1 or less are "Neither", 2 is "Prime", and even numbers are "Composite". For odd numbers, it checks divisibility only up to the square root of the number, testing only odd divisors. If it finds any divisor, it's "Composite"; otherwise, it's "Prime". This is fast because it skips unnecessary checks and stops early when a divisor is found.

- **Prompting Strategies - Comparison:**

Strategy	Result	Efficiency
Zero-Shot (constraints only)	Optimized, all constraints applied	Best - 19 lines
One-Shot (1 example)	Over-simplified, missing optimizations	Poor - 30+ lines
Few-Shot (multiple examples)	Verbose, defensive coding	Okay - 40+ lines

Task 5: Zero-Shot Prompting (Perfect Number Check)

❖ Scenario: Write a zero-shot prompt (without providing any examples) to generate a Python function that checks whether a given number is 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 zero-shot prompt to generate a Python function that checks whether a given number is a perfect number and test the program with multiple inputs. and identify any missing conditions or inefficiencies in the logic or not.

• Screenshot of Generated Code:

The screenshot shows the Microsoft Visual Studio Code interface. The Explorer sidebar on the left lists several Python files: armstrong_checker.py, even_odd_checker.py, factorial_calculator.py, factorial_comparison.py, number_classifier.py, palindrome_checker.py, and perfect_number_checker.py. The perfect_number_checker.py file is currently open in the editor. The code defines two functions: `is_perfect_number` and `get_divisors`. The `is_perfect_number` function calculates the sum of proper divisors of a number and returns True if it equals the number, False otherwise. The `get_divisors` function generates a list of proper divisors for a given number. The terminal at the bottom shows the output of running the script with the input 6, which prints "6 is a Perfect Number!", "Proper divisors of 6: [1, 2, 3]", and "Sum: 1 + 2 + 3 = 6". A Chat window on the right contains a user prompt asking for a Python program to check if a number is perfect, followed by a detailed description of what the program does, including handling edge cases and printing all proper divisors. The status bar at the bottom indicates the code is in Line 41, Column 36, with 4 spaces, using UTF-8 encoding, and is running in Python 3.13.9.

• Short Explanation of the Code:

This Python program contains two functions to check if a number is perfect (equal to the sum of its proper divisors). The `is_perfect_number()` function uses a brute-force approach, checking all numbers from 1 to $n-1$, while `is_perfect_number_optimized()` improves efficiency by only checking divisors up to \sqrt{n} using divisor pairs, achieving $\sim 32x$ speedup. Both return a tuple with a boolean result and the list of divisors. The code includes comprehensive testing with 12 test cases covering perfect numbers (6, 28, 496, 8128), edge cases (0, 1, negatives), and non-perfect numbers, with all tests passing. Input validation ensures type checking, and the analysis section identifies that all critical conditions are properly handled, with the main inefficiency being the $O(n)$ complexity of the brute-force version versus the $O(\sqrt{n})$ optimized version.

Task 6: Few-Shot Prompting (Even or Odd Classification with Validation)

❖ Scenario: 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 program that checks if a number is even or odd with input validation. Test with positive, negative, zero, and non-integer inputs.

• Screenshot of Generated Code:

The screenshot shows the Microsoft Visual Studio Code interface with the following details:

- File Explorer:** Shows files including armstrong_checker.py, even_odd_checker.py, factorial_calculator.py, factorial_comparison.py, number_classifier.py, palindrome_checker.py, and perfect_number_checker.py. even_odd_checker.py is the active file.
- Code Editor:** Displays the Python code for even-odd classification. The code includes a function to check if a number is even or odd and a function to get valid user input. A callout box highlights the AI-generated prompt: "write a python programme to check whether the given number is even or odd, including proper input validation".
- Terminal:** Shows the output of the program running. It prompts the user for a number, receives "6", and outputs "Result: 6 is Even", "Explanation: 6 is divisible by 2 (remainder = 0)", and "Formula: 6 / 2 = 3 with remainder 0".
- Output:** Shows the results of the run_tests() function, indicating success for all tests.
- Status Bar:** Shows the current file is even_odd_checker.py, the line is 34, column is 63, and the encoding is UTF-8.

• Short Explanation of the Code:

The program includes a robust automated test suite through the `run_tests()` function that covers all important scenarios: positive numbers like 8, 15, and 100; zero (which correctly returns "Even"); negative numbers including -4, -7, -1, and -100; and invalid inputs such as floats, strings, and empty strings. Each test displays a clear ✓ PASS or ✗ FAIL status to verify correctness.