

# AI ASSISTANT CODING

## LAB-3.1

Name : GUMMADAVELLI SHRAVANI

Roll No : 2303A52361

Batch No : 34

### **Experiment – Prompt Engineering Techniques**

#### **Task Description**

Design and refine prompts using different prompting strategies to generate Python programs for basic computational problems.

#### **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 function that checks whether a given integer number is a palindrome. The function should return True if the number reads the same forward and backward, otherwise return False.

#### **AI-Refactored Code :**

```
def is_palindrome_number(number: int) -> bool:
```

```
    """
```

Checks if a given integer number is a palindrome.

A number is a palindrome if it reads the same forward and backward.

Args:

number (int): The integer number to check.

Returns:

bool: True if the number is a palindrome, False otherwise.

....

```
if not isinstance(number, int):
```

```
    raise TypeError("Input must be an integer.")
```

```
# Convert the integer to a string to easily reverse it
```

```
num_str = str(number)
```

```
# Compare the string with its reverse
```

```
return num_str == num_str[::-1]
```

```
print(is_palindrome_number(121))
```

```
print(is_palindrome_number(12321))
```

```
print(is_palindrome_number(10021))
```

```
print(is_palindrome_number(71))
```

```
print(is_palindrome_number(0))
```

The screenshot shows a Jupyter Notebook cell with the following code:

```
[1] def is_palindrome_number(number: int) -> bool:
    """
    Checks if a given integer number is a palindrome.
    A number is a palindrome if it reads the same forward and backward.
    Args:
        number (int): The integer number to check.
    Returns:
        bool: True if the number is a palindrome, False otherwise.
    """
    if not isinstance(number, int):
        raise TypeError("Input must be an integer.")
    # Convert the integer to a string to easily reverse it
    num_str = str(number)
    # Compare the string with its reverse
    return num_str == num_str[::-1]
print(is_palindrome_number(121))
print(is_palindrome_number(12321))
print(is_palindrome_number(10021))
print(is_palindrome_number(71))
print(is_palindrome_number(0))
```

The cell output is shown at the bottom left as "True".

```
    raise TypeError("Input must be an integer.")
    # Convert the integer to a string to easily reverse it
    num_str = str(number)
    # Compare the string with its reverse
    return num_str == num_str[::-1]
print(is_palindrome_number(121))
print(is_palindrome_number(12321))
print(is_palindrome_number(10021))
print(is_palindrome_number(71))
print(is_palindrome_number(0))
```

```
... True  
True  
False  
False  
True
```

## 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-shot solution.
- Examine improvements in clarity and correctness.

### Prompt Used :

Write a Python function to compute the factorial of a given non-negative integer.

#### CODE:

```
def factorial(n: int) -> int:
```

```
    ...
```

Computes the factorial of a given non-negative integer.

Args:

n (int): The non-negative integer for which to compute the factorial.

Returns:

int: The factorial of n.

Raises:

ValueError: If n is a negative integer.

TypeError: If n is not an integer.

....

```
if not isinstance(n, int):
    raise TypeError("Input must be an integer.")

if n < 0:
    raise ValueError("Factorial is not defined for negative numbers.")

if n == 0:
    return 1

else:
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

print(factorial(5))
print(factorial(10))
print(factorial(0))
```

A screenshot of a Google Colab notebook titled "Untitled66.ipynb". The code cell contains a Python function named `factorial`:

```
def factorial(n: int) -> int:
    """
    Computes the factorial of a given non-negative integer.

    Args:
        n (int): The non-negative integer for which to compute the factorial.

    Returns:
        int: The factorial of n.

    Raises:
        ValueError: If n is a negative integer.
        TypeError: If n is not an integer.
    """
    if not isinstance(n, int):
        raise TypeError("Input must be an integer.")
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    if n == 0:
        return 1
    else:
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result
print(factorial(5))
```

```
result *= i
return result
print(factorial(5))
print(factorial(10))
print(factorial(0))

...
120
3628800
1
```

### 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.

## Prompts Used :

Write a Python function to check whether a given integer is an Armstrong number.

### CODE:

```
def is_armstrong_number(number: int) -> bool:
```

```
    """
```

Checks if a given integer number is an Armstrong number.

An Armstrong number is a number that is the sum of its own digits

each raised to the power of the number of digits.

Args:

    number (int): The integer number to check.

Returns:

    bool: True if the number is an Armstrong number, False otherwise.

Raises:

    TypeError: If the input is not an integer.

    ValueError: If the input is a negative integer (Armstrong numbers are typically defined for positive integers).

```
    """
```

```
if not isinstance(number, int):
```

```
    raise TypeError("Input must be an integer.")
```

```
if number < 0:
```

```
    raise ValueError("Armstrong numbers are typically defined for non-negative integers.")
```

```
# Convert the number to a string to easily access its digits and count them
```

```
num_str = str(number)
```

```
num_digits = len(num_str)
```

```
sum_of_powers = 0
```

```
for digit_char in num_str:
```

```

digit = int(digit_char)

sum_of_powers += digit ** num_digits


return sum_of_powers == number

print(is_armstrong_number(153))

print(is_armstrong_number(370))

print(is_armstrong_number(371))

print(is_armstrong_number(407))

print(is_armstrong_number(1634))

print(is_armstrong_number(8208))

print(is_armstrong_number(9474))

```

Untitled66.ipynb

```

def is_armstrong_number(number: int) -> bool:
    """
    Checks if a given integer number is an Armstrong number.
    An Armstrong number is a number that is the sum of its own digits
    each raised to the power of the number of digits.
    Args:
        number (int): The integer number to check.
    Returns:
        bool: True if the number is an Armstrong number, False otherwise.
    Raises:
        TypeError: If the input is not an integer.
        ValueError: If the input is a negative integer (Armstrong numbers are typically defined for positive integers).
    """
    if not isinstance(number, int):
        raise TypeError("Input must be an integer.")
    if number < 0:
        raise ValueError("Armstrong numbers are typically defined for non-negative integers.")

    # Convert the number to a string to easily access its digits and count them
    num_str = str(number)
    num_digits = len(num_str)

    sum_of_powers = 0
    for digit_char in num_str:

```

Untitled66.ipynb

```

num_str = str(number)
num_digits = len(num_str)

sum_of_powers = 0
for digit_char in num_str:
    digit = int(digit_char)
    sum_of_powers += digit ** num_digits

return sum_of_powers == number
print(is_armstrong_number(153))
print(is_armstrong_number(370))
print(is_armstrong_number(371))
print(is_armstrong_number(407))
print(is_armstrong_number(1634))
print(is_armstrong_number(8208))
print(is_armstrong_number(9474))

```

... True  
True  
True  
True  
True  
True  
True

## Question 4: Context-Managed Prompting (Optimized Number 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.

### Prompt:

*You are an expert Python developer writing optimized and readable code.*

*Task:* Write a Python program that classifies a given integer number as **Prime**, **Composite**, or **Neither**.

*Rules & Constraints:*

- If the number is less than or equal to 1, classify it as **Neither**.
- A **Prime** number has exactly two distinct positive divisors.
- A **Composite** number has more than two positive divisors.
- Use an optimized approach by checking divisibility only up to  $\sqrt{n}$ .
- The program must handle invalid inputs gracefully.
- Output must be one of the following strings only: "Prime", "Composite", or "Neither".
- Write clean, efficient, and well-commented Python code.

### CODE:

```
def classify_number(n):
    if not isinstance(n, int):
        return "Neither"

    if n <= 1:
        return "Neither"

    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return "Composite"
```

```

return "Prime"

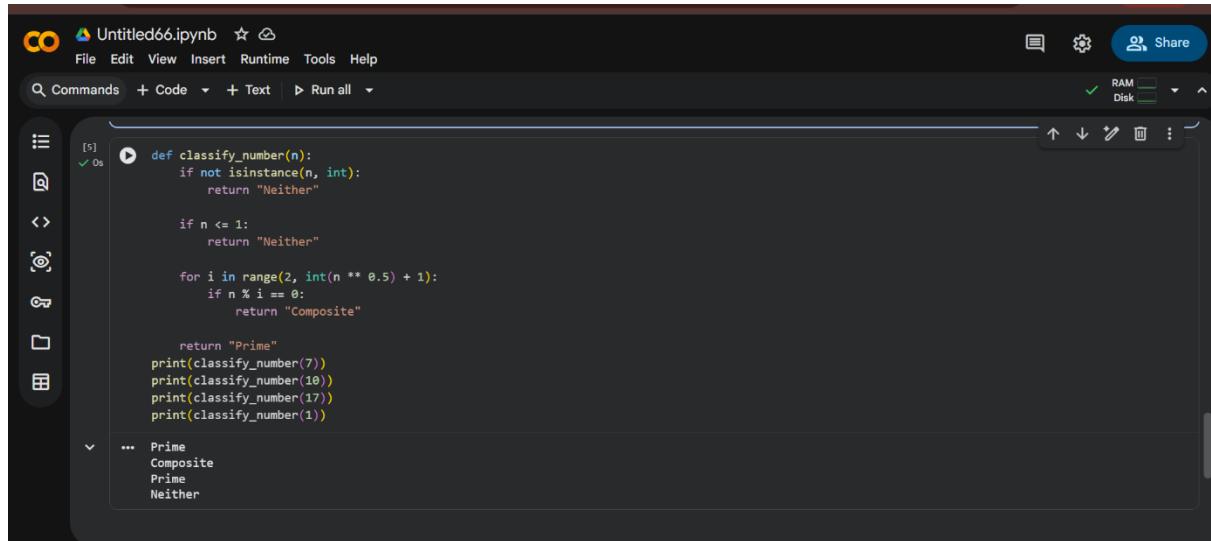
print(classify_number(7))

print(classify_number(10))

print(classify_number(17))

print(classify_number(1))

```



The screenshot shows a Jupyter Notebook interface with the following details:

- Title:** Untitled66.ipynb
- Toolbar:** File, Edit, View, Insert, Runtime, Tools, Help.
- Cell Type:** Code (indicated by a play button icon).
- Code Content:**

```

def classify_number(n):
    if not isinstance(n, int):
        return "Neither"

    if n <= 1:
        return "Neither"

    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return "Composite"

    return "Prime"
print(classify_number(7))
print(classify_number(10))
print(classify_number(17))
print(classify_number(1))

```
- Output:**

```

...
Prime
Composite
Prime
Neither

```
- Runtime Status:** RAM and Disk usage indicators.

## Conclusion :

This experiment demonstrated how different prompting techniques influence AI-generated code quality. Zero-shot prompting produced basic solutions, while one-shot and few-shot prompting improved clarity and accuracy through examples. Context-managed prompting delivered the most optimized and reliable results by clearly defining constraints and expectations. Overall, effective prompt design significantly enhances the correctness, efficiency, and robustness of AI-assisted coding.