

# 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]

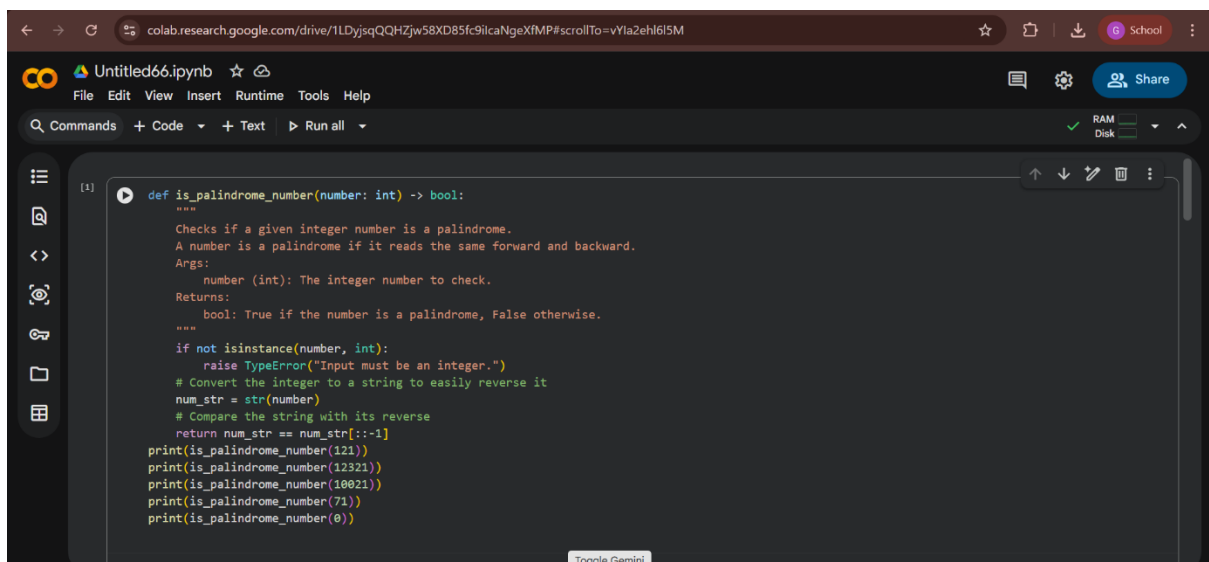
print(is\_palindrome\_number(121))

print(is\_palindrome\_number(12321))

print(is\_palindrome\_number(10021))

print(is\_palindrome\_number(71))

print(is\_palindrome\_number(0))



The screenshot shows a Google Colab notebook interface. The browser address bar displays a URL from colab.research.google.com. The notebook is titled 'Untitled66.ipynb'. The code editor contains a Python function `is_palindrome_number` that takes an integer `number` and returns a boolean. The function includes a docstring, a type check using `isinstance`, conversion of the number to a string, and a comparison with its reverse. Below the function definition, there are five print statements testing the function with the values 121, 12321, 10021, 71, and 0. The left sidebar shows standard Colab navigation icons. The bottom status bar indicates 'Toggle Gemini'.

```
[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))
```

```

        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):

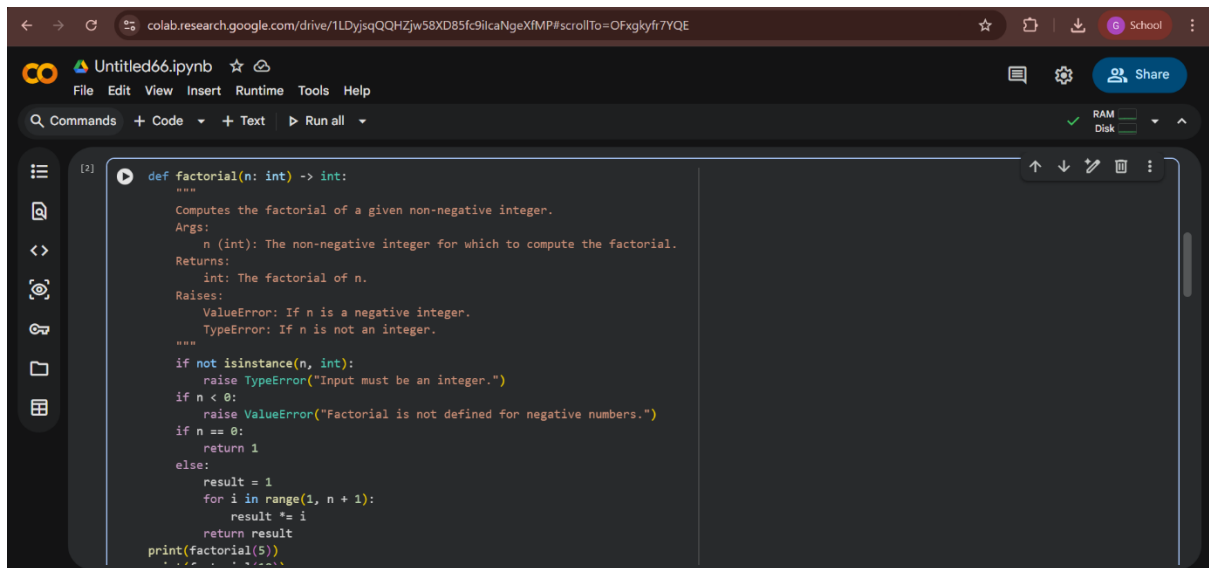
result \*= i

return result

print(factorial(5))

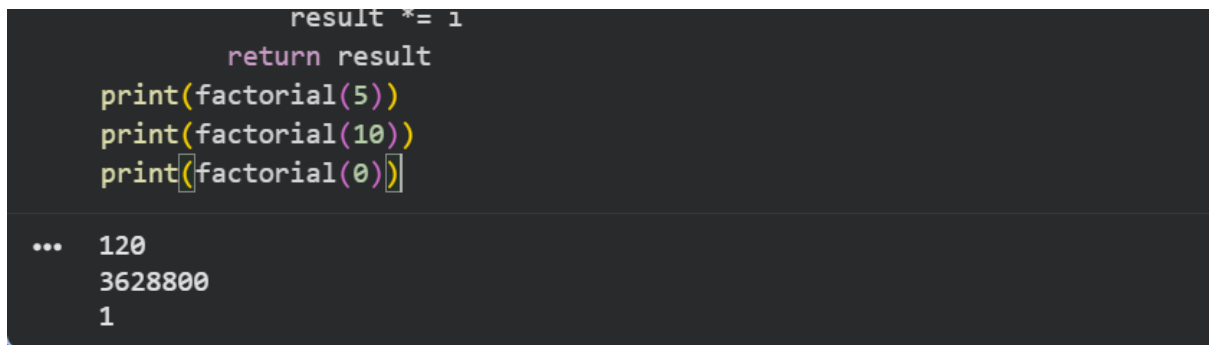
print(factorial(10))

print(factorial(0))



The screenshot shows a Google Colab notebook titled 'Untitled66.ipynb'. The code defines a function `factorial(n: int) -> int:` with a docstring that describes its purpose, arguments, returns, and raises. The function includes type checking and error handling for non-integer and negative inputs. It uses a loop to calculate the factorial for positive integers. The code is as follows:

```
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))
```



This block shows the continuation of the code from the previous block, including the function calls and their outputs. The code is as follows:

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

The output of the code is displayed below the code cells:

```
... 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))
```

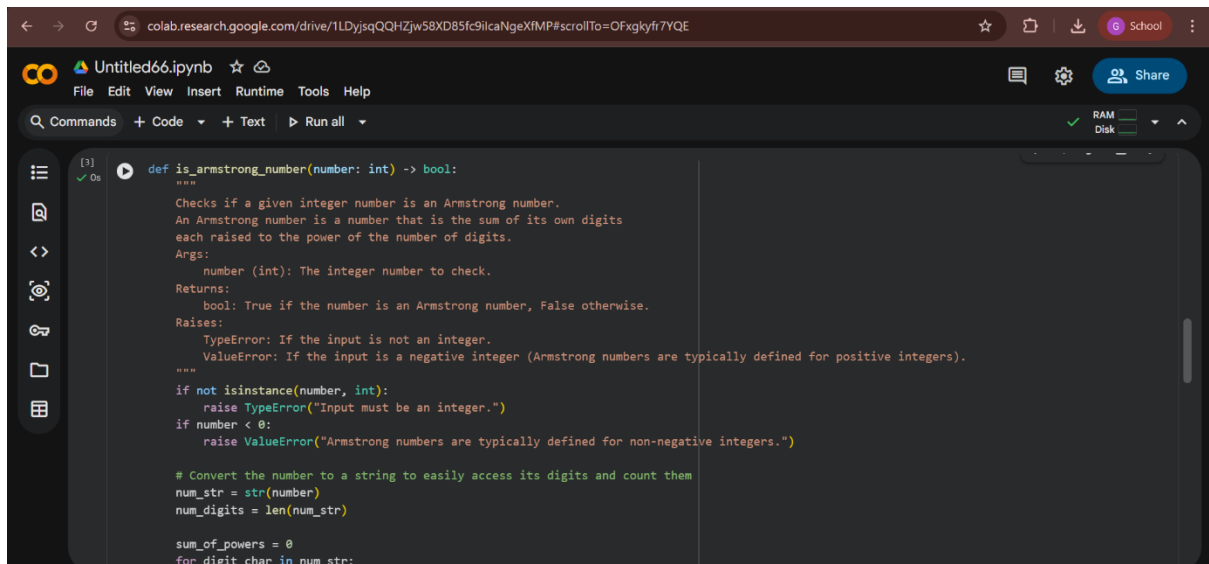
```
print(is_armstrong_number(371))
```

```
print(is_armstrong_number(407))
```

```
print(is_armstrong_number(1634))
```

```
print(is_armstrong_number(8208))
```

```
print(is_armstrong_number(9474))
```

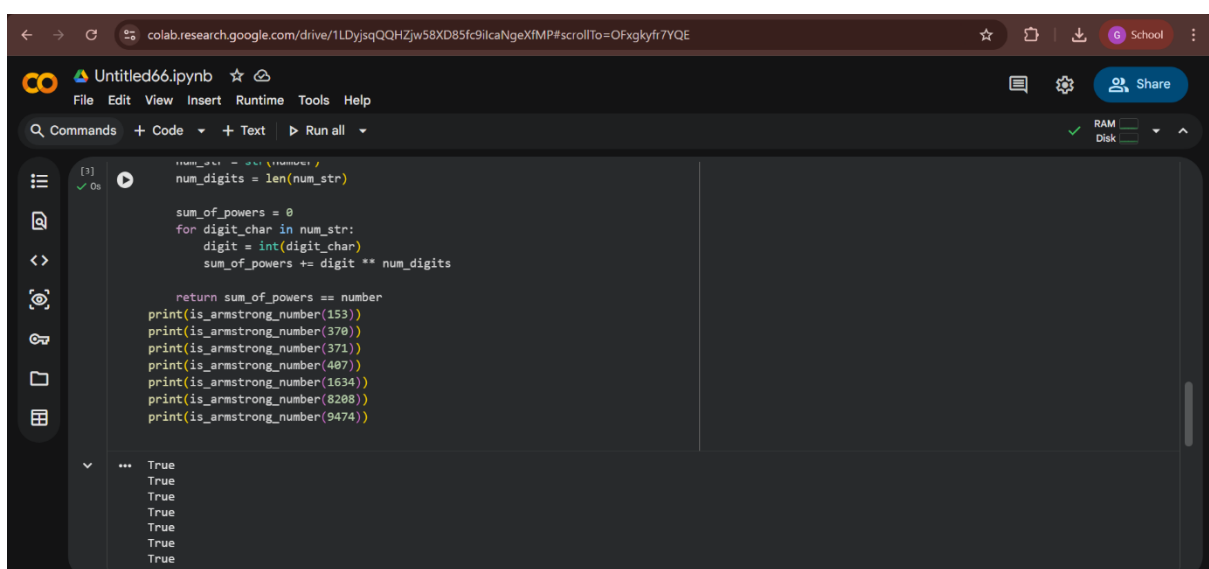


The screenshot shows a Google Colab notebook interface. The top bar includes the Google Colab logo, the file name 'Untitled66.ipynb', and various menu options like File, Edit, View, Insert, Runtime, Tools, and Help. Below the menu bar, there's a search bar and a 'Run all' button. The main code area contains a function definition for `is_armstrong_number`. The function takes an integer `number` and returns a boolean. It includes docstrings, type hints, and error handling for non-integer and negative inputs. The function logic converts the number to a string, iterates over its digits, and calculates the sum of each digit raised to the power of the total number of digits.

```
[3] ✓ 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:
```



The screenshot shows the same Google Colab notebook interface, but now the function is being called for several numbers. The code area shows the function calls and the output of the function, which is `True` for all the numbers listed. The output is displayed in a scrollable area at the bottom of the notebook.

```
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:*

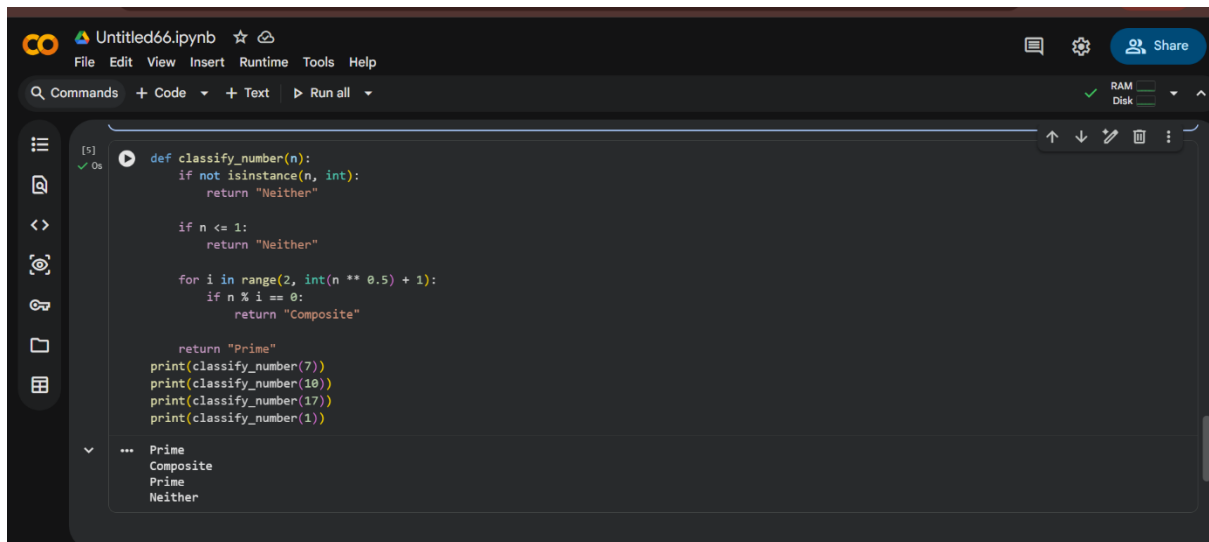
- If the number is less than or equal to 1, classify it as **Neither**.
- A **Prime** number has exactly two distinct positive divisors.
- A **Composite** number has more than two positive divisors.
- Use an optimized approach by checking divisibility only up to  $\sqrt{n}$ .
- The program must handle 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 window titled 'Untitled66.ipynb'. The code cell contains a function 'classify\_number(n)' that checks if a number is prime, composite, or neither. The function returns 'Prime' for 7, 'Composite' for 10, 'Prime' for 17, and 'Neither' for 1. The output cell shows the results of these calls: Prime, Composite, Prime, and Neither.

```
[5]: 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))

      *** Prime
      Composite
      Prime
      Neither
```

## 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.