

School of Computer Science and Artificial Intelligence

Lab Assignment # 3.1

Program : B. Tech (CSE)

Specialization : AIML

Course Title : AI Assisted

Coding Course Code: 23CS002PC304

Semester : VI

Academic Session : 2025-2026

Name of Student : A. Prashanth

Enrollment No. : 2303A52092

Batch No. : 33

Date : 13/01/26

Title

Experiment on Prompt Engineering Techniques for Python Program Generation Using AI-Assisted Tools

Lab Objectives

1. To understand and apply different **prompt engineering techniques** for generating Python programs using AI-assisted tools.
 2. To analyze the **impact of context, constraints, and examples** on the accuracy and efficiency of AI-generated code.
 3. To develop and refine **real-world Python applications** through iterative prompt improvement and testing.
-

Lab Outcomes

1. Students will be able to **design effective prompts** to generate correct and optimized Python code.
 2. Students will be able to **compare and evaluate AI-generated solutions** produced using different prompting strategies.
 3. Students will be able to **implement, test, and document real-world Python applications** using AI-assisted coding tools.
-

Tools Used

- AI-assisted coding tool (ChatGPT)
- Python 3.x
- Standard Python IDE / Interpreter

Experiment 1: Zero-Shot Prompting – Palindrome Number

Prompt Type: Zero-Shot

Objective: Check whether a number is a palindrome.

Observations

- AI generated correct logic for positive integers.
- Failed to explicitly handle negative numbers.
- Required manual refinement for edge cases.

CODE:

A screenshot of a Jupyter Notebook interface. The top menu bar includes File, Edit, View, Insert, Runtime, Tools, Help, Commands, + Code, + Text, and Run all. The status bar shows RAM and Disk usage. The code cell [1] contains a function to check if a number is a palindrome. The code cell [2] contains a series of print statements testing the function with various inputs. The output cell shows the results: True, False, True, True, False.

```
#Check Palindrome:  
def is_palindrome(num):  
    original = num  
    reverse = 0  
  
    while num > 0:  
        digit = num % 10  
        reverse = reverse * 10 + digit  
        num //= 10  
  
    return original == reverse  
  
[2]:  
print(is_palindrome(121))  
print(is_palindrome(123))  
print(is_palindrome(0))  
print(is_palindrome(1221))  
print(is_palindrome(-121))  
  
True  
False  
True  
True  
False
```

Experiment 2: One-Shot Prompting – Factorial Calculation

Prompt Type: One-Shot

Example Given: Input: 5 → Output: 120

Observations

- Code clarity improved compared to zero-shot.
- Handled 0! correctly.
- Included basic validation for negative numbers.

CODE:

A screenshot of a Jupyter Notebook interface. The top menu bar includes File, Edit, View, Insert, Runtime, Tools, Help, Commands, + Code, + Text, and Run all. The status bar shows RAM and Disk usage. The code cell [3] contains a function to calculate the factorial of a number. The code cell [4] contains a series of print statements testing the function with various inputs. The output cell shows the results: 120, 1, 1, 5040, None.

```
#Find Factorial:  
def factorial(n):  
    if n < 0:  
        return None  
    result = 1  
    for i in range(1, n + 1):  
        result *= i  
    return result  
  
[4]:  
print(factorial(5))  
print(factorial(0))  
print(factorial(1))  
print(factorial(7))  
print(factorial(-3))  
  
120  
1  
1  
5040  
None
```

Experiment 3: Few-Shot Prompting – Armstrong Number Check

Prompt Type: Few-Shot

Examples Provided:

- 153 → Armstrong Number
- 370 → Armstrong Number
- 123 → Not an Armstrong Number

Observations

- Correct mathematical logic inferred.
- Output format matched examples exactly.
- Input validation required additional refinement.

CODE:

The screenshot shows a Jupyter Notebook interface with a dark theme. In the code cell [5], the following Python function is defined:

```
#Check Armstrong Number:  
def is_armstrong(num):  
    temp = num  
    digits = len(str(num))  
    total = 0  
  
    while temp > 0:  
        digit = temp % 10  
        total += digit ** digits  
        temp //= 10  
  
    if total == num:  
        return "Armstrong Number"  
    else:  
        return "Not an Armstrong Number"
```

In the output cell [6], the function is called with several arguments:

```
print(is_armstrong(153))  
print(is_armstrong(370))  
print(is_armstrong(123))  
print(is_armstrong(0))  
print(is_armstrong(9))  
print(is_armstrong(9474))
```

The resulting output is:

```
Armstrong Number  
Armstrong Number  
Not an Armstrong Number  
Armstrong Number  
Armstrong Number
```

Experiment 4: Context-Managed Prompting – Number Classification

Prompt Type: Context-Managed

Task: Classify a number as **Prime**, **Composite**, or **Neither**.

Observations

- Efficient \sqrt{n} optimization applied.
- Proper handling of 0, 1, and invalid inputs.
- Most robust and production-ready solution.

CODE:

A screenshot of a Jupyter Notebook interface titled "Untitled12.ipynb". The code cell [8] contains a function to classify numbers as Prime, Composite, or Neither. The output cell [8] shows the results for various inputs: 2 (Prime Number), 7 (Neither Prime Nor Composite), 10 (Composite Number), 1 (Neither Prime Nor Composite), 0 (Neither Prime Nor Composite), -5 (Composite Number), 15 (Neither Prime Nor Composite), and 11 (Prime Number). The notebook also shows memory usage at the top right.

```
#Armstrong Number
def classify_number(num):
    if not isinstance(num, int) or num < 0:
        return "Invalid Input"
    if num == 0 or num == 1:
        return "Neither Prime Nor Composite"
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            return "Composite Number"
    return "Prime Number"

print(classify_number(2))
print(classify_number(7))
print(classify_number(10))
print(classify_number(1))
print(classify_number(0))
print(classify_number(-5))
print(classify_number(15))
print(classify_number("11"))
```

[8]

```
Prime Number
Neither Prime Nor Composite
Neither Prime Nor Composite
Neither Prime Nor Composite
Invalid Input
Composite Number
Neither Prime Nor Composite
Prime Number
```

Experiment 5: Zero-Shot Prompting – Perfect Number Check

Prompt Type: Zero-Shot

Observations

- Basic logic generated correctly.
- Logical error for input 0.
- Inefficient O(n) loop required optimization.

CODE:

A screenshot of a Jupyter Notebook interface titled "Untitled12.ipynb". The code cell [9] contains a function to check if a number is perfect. The output cell [9] shows the results for various inputs: 6 (True), 28 (True), 12 (False), 1 (False), and 0 (True). The notebook also shows memory usage at the top right.

```
#Perfect Number
def is_perfect(num):
    sum_divisors = 0
    for i in range(1, num):
        if num % i == 0:
            sum_divisors += i
    return sum_divisors == num

print(is_perfect(6))
print(is_perfect(28))
print(is_perfect(12))
print(is_perfect(1))
print(is_perfect(0))
```

[9]

[10] 0s

[11] 0s

True
True
False
False
True

Variables Terminal 09:44 Python 3

Experiment 6: Few-Shot Prompting – Even or Odd with Validation

Prompt Type: Few-Shot

Examples Provided:

- 8 → Even
- 15 → Odd
- 0 → Even

Observations

- Proper input validation inferred.
- Clear and consistent output.
- Negative numbers handled correctly.

CODE:

The screenshot shows a Jupyter Notebook interface with a single code cell containing Python code. The code defines a function `even_or_odd` that checks if a number is even or odd. It also includes a series of print statements calling this function with various inputs. The output pane below the cell shows the results: it prints "Even" for 8, "Odd" for 15, "Even" for 0, "Even" for -4, and "Invalid Input" for both 3.5 and 10. The notebook has tabs for "Untitled12.ipynb" and "Untitled13.ipynb". The status bar at the bottom right indicates "RAM Disk".

```
#Even-Odd Number Check:  
def even_or_odd(num):  
    if not isinstance(num, int):  
        return "Invalid Input"  
  
    if num % 2 == 0:  
        return "Even"  
    else:  
        return "Odd"  
  
print(even_or_odd(8))  
print(even_or_odd(15))  
print(even_or_odd(0))  
print(even_or_odd(-4))  
print(even_or_odd(3.5))  
print(even_or_odd("10"))
```

Even
Odd
Even
Even
Invalid Input
Invalid Input

Comparative Analysis

Prompting Technique	Accuracy	Validation	Efficiency	Clarity
Zero-Shot	Medium	Low	Low	Average
One-Shot	Good	Medium	Medium	Good
Few-Shot	High	High	Medium	Very Good
Context-Managed	Very High	Very High	High	Excellent

Result

- The quality of AI-generated Python code **improves significantly** with better prompt design.

- Few-shot and context-managed prompting produced **more accurate, optimized, and reliable programs**.
 - Zero-shot prompting is suitable only for **simple tasks** and requires manual verification.
-

Conclusion:

This lab successfully demonstrated the effectiveness of various **prompt engineering techniques** in generating Python programs using AI-assisted tools. As the level of guidance in prompts increased—from zero-shot to context-managed—the **accuracy, efficiency, and robustness** of the generated code also improved. Proper prompt design plays a critical role in producing reliable AI-generated software solutions.

Future Scope:

1. Applying prompt engineering techniques to **larger real-world applications** such as web development and data analysis.
2. Exploring advanced prompting methods like **chain-of-thought and self-consistency prompting**.
3. Integrating AI-assisted coding tools into **software engineering workflows** for improved productivity.