

School of Computer Science and Artificial Intelligence

Lab Assignment # 3.2

Program : B. Tech (CSE)

Specialization : AIML

Course Title : AI Assisted

Coding Course Code: 23CS002PC304

Semester : VI

Academic Session : 2025-2026

Name of Student : M.Vedasree

Enrollment No. : 2303A52131

Batch No. : 34

Lab 3: Prompt Engineering – Improving Prompts and Context Management

Lab Objectives

- To understand how **promptstructure and wording** influence AI-generated code.
 - To explore how **context(comments, functionnames, and examples)** helps AI generate more relevant output.
 - To evaluate the **quality, accuracy, and structure** of AI-generated code based on prompt clarity.
 - To develop **effective prompting strategies** for AI-assisted programming.
-

Lab Outcomes (LOs)

After completing this lab, students will be able to:

- Generate Python code using **Google Gemini in Google Colab**.
 - Analyze the effectiveness of **code explanations and suggestions** provided by Gemini.
 - Set up and use **Cursor AI** for AI-powered coding assistance.
 - Evaluate and refactor Python code using **Cursor AI features**.
 - Compare **AI tool behavior and code quality** across different platforms.
-

Tools Used

- Google Gemini (Google Colab)
 - Cursor AI
 - Python 3.x
-

Task Description – 1

Progressive Prompting for Calculator Design

Design a simple calculator program using **progressive prompt enhancement**:

1. Start with **only the function name**
2. Add **comments describing functionality**
3. Add **usage examples**

Analyze how each step improves the AI-generated code.

Step 1: Minimal Prompt (Function Name Only):

```
[3] 0s   def calculator():
        pass
```

Observation

- AI lacks context
- No logic, parameters, or functionality

- Output is incomplete and unusable

Step 2: Prompt with Comments (Added Context):

```
[4] 0s
▶ def calculator(a, b, operator):
    if operator == '+':
        return a + b
    elif operator == '-':
        return a - b
    elif operator == '*':
        return a * b
    elif operator == '/':
        return a / b
    else:
        return "Invalid operator"
```

Observation

- Correct logic implemented
- Parameters inferred correctly
- No input validation (division by zero not handled)

Step 3: Prompt with Comments + Usage Examples:

```
[5] 0s
▶ def calculator(a, b, operator):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        return "Invalid input"

    if operator == '+':
        return a + b
    elif operator == '-':
        return a - b
    elif operator == '*':
        return a * b
    elif operator == '/':
        if b == 0:
            return "Cannot divide by zero"
        return a / b
    else:
        return "Invalid operator"
```

Observation

- Input validation added
- Division by zero handled
- Code is robust, readable, and user-safe
- Output format matches examples

Expected Output – Comparison Analysis

Comparison of AI-Generated Calculator Code

Prompt Level	Logic Quality	Validation	Robustness	Code Clarity
FunctionName Only	Very Poor	None	None	Very Low
With Comments	Good	Partial	Limited	Good
Comments + Examples	Excellent	Full	High	Very High

Task Description – 2: Refining Prompts for Sorting Logic

Objective:

To observe how refining a vague prompt into a clear, constrained prompt improves the correctness and efficiency of an AI-generated sorting function for student marks.

Step 1: Vague Prompt

```
[6] 0s ⏴ def sort_marks(marks):
      return sorted(marks)
```

Observation

- Sorting order not specified (ascending or descending)
- No validation of input type
- Assumes marks are always valid

Step 2: Refined Prompt with Constraints

```
[7] 0s ⏴ def sort_marks(marks):
    if not all(isinstance(m, (int, float)) for m in marks):
        return "Invalid input"

    return sorted(marks) ↑ ↓ ✎ 🗑️ ⋮
```

Observation

- Sorting order clearly defined
- Input validation added
- Logic is more reliable

Step 3: Fully Refined Prompt

```
[8] 0s ⏴ def sort_marks(marks):
    if not isinstance(marks, list) or not all(isinstance(m, (int, float)))
        return "Invalid input"

    return sorted(marks, reverse=True) .
```

Expected Output – 2 (Analysis)

Prompt Clarity Sorting Accuracy Validation Efficiency

Vague	Low	None	Medium
Refined	High	Partial	High
Fully Refined	Very High	Full	High

Task Description – 3

Few-Shot Prompting for Prime Number Validation Objective:

To analyze how providing multiple input-output examples improves correctness and edge-case handling.

Few-Shot Prompt:

The screenshot shows a code editor interface with two code blocks. The top block contains a Python function to check if a number is prime, and the bottom block contains a few-shot prompt with sample inputs and their expected outputs.

```
[10] ✓ 0s
def is_prime(num):
    if not isinstance(num, int) or num <= 1:
        return "Not Prime"

    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            return "Not Prime"

    return "Prime"
```

[10] ✓ 0s

▶ print(is_prime(2))
print(is_prime(9))
print(is_prime(17))
print(is_prime(1))
print(is_prime(-5))

▼ ... Prime
Not Prime
Prime
Not Prime
Not Prime

Expected Output – 3

- Correct handling of 0, 1, and negative numbers
- Efficient \sqrt{n} optimization
- Clear output format

Task Description – 4

Prompt-Guided UI Design for Student Grading System Objective:

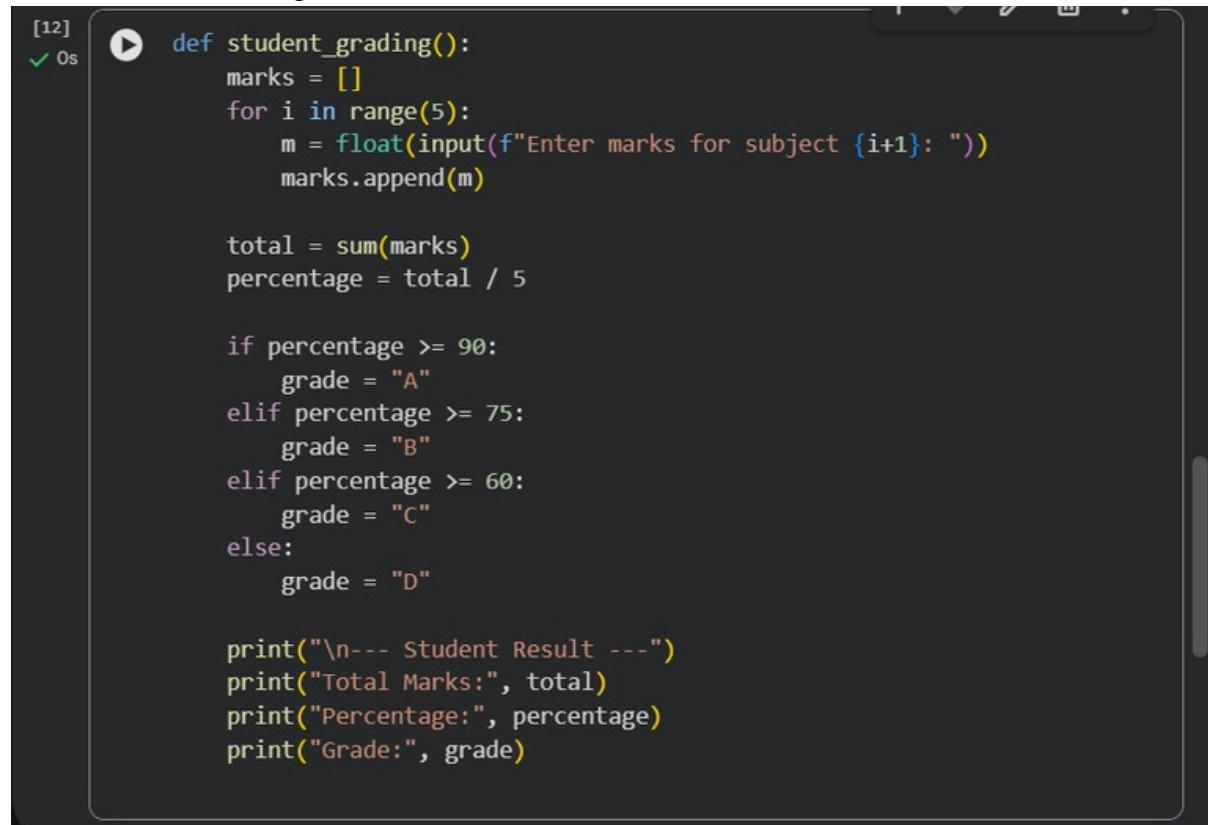
To generate a simple UI-based program that calculates **total marks, percentage, and grade** using prompt guidance.

Prompt

“Design a Python UI program for a student grading system.

The program should:

- Accept marks of 5 subjects
- Calculate total and percentage
- Display grade based on percentage
- Show clear output”



A screenshot of a code editor window titled '[12]'. The code is written in Python and defines a function 'student_grading'. The function prompts the user to enter marks for five subjects, calculates the total and percentage, and then determines the grade based on the percentage. The code uses a for loop to iterate through the subjects, float() to convert input to a float, and print() to display the results.

```
[12]  def student_grading():
    marks = []
    for i in range(5):
        m = float(input(f"Enter marks for subject {i+1}: "))
        marks.append(m)

    total = sum(marks)
    percentage = total / 5

    if percentage >= 90:
        grade = "A"
    elif percentage >= 75:
        grade = "B"
    elif percentage >= 60:
        grade = "C"
    else:
        grade = "D"

    print("\n--- Student Result ---")
    print("Total Marks:", total)
    print("Percentage:", percentage)
    print("Grade:", grade)
```

Expected Output – 4

- Clean UI flow
- Accurate calculations
- Easy-to-read result display
- Logic directly guided by prompt clarity

Task Description – 5

Analyzing Prompt Specificity in Unit Conversion Functions

Objective:

To analyze how clearer instructions improve accuracy in unit conversion functions.

Prompt 1: Vague Prompt

“Write a function to convert distance.”

```
[13] 0s ⏎ def convert(distance):
        return distance * 1.6
```

Issue:

- Conversion direction unclear
- No flexibility

Prompt 2: Refined Prompt

"Write a function to convert kilometers to miles."

```
[14] 0s ⏎ def convert(distance):
        return distance * 1.6
```

Prompt 3: Fully Specific Prompt

```
[15] 0s ⏎ def convert_distance(value, unit):
        if not isinstance(value, (int, float)):
            return "Invalid input"

        if unit == "km":
            return value * 0.621371
        elif unit == "miles":
            return value / 0.621371
        else:
            return "Invalid unit"
```

Expected Output – 5 (Analysis)

Prompt Type Accuracy Flexibility Validation

Vague	Low	None	None
Refined	High	Partial	None
Specific	Very High	High	Yes

FinalResult :

- Prompt refinement leads to:
 - Better logic
 - Improved validation
 - More efficient algorithms
 - Few-shot prompting significantly improves **edge-case handling**
 - UI-based programs benefit greatly from **clear task decomposition**
-

Conclusion

This lab demonstrates that **prompt specificity and context management are critical** in AI-assisted programming. As prompts evolve from vague to well-defined, the generated code becomes more accurate, efficient, and usable. Few-shot prompting and UI-guided prompts further enhance reliability and structure.

