AI ASSISTED CODING

ASSIGNMENT-3

B.Sri Laxmi Gayathri

2303a52033

Batch:38

Lab 3: Prompt Engineering – Improving Prompts and Context Management

Task Description-1

Progressive Prompting for Calculator Design: Ask the AI to design a simple calculator program by initially providing only the function name. Gradually enhance the prompt by adding comments and usage examples.

Expected Output-1

Comparison showing improvement in AI-generated calculator logic and structure.

Prompt:

Generate a calculator program when only the function name is provided.

Comments:

1. This program demonstrates progressive prompting, where improving the prompt results in better structured and more reliable calculator code.

2. The calculator function performs basic arithmetic operations and includes error handling for division by zero and invalid operators.

Explanation:

This task explains how improving a prompt step by step helps the AI generate better code. When the prompt is simple, the code is basic. As more details, comments, and examples are added, the code becomes clearer, safer, and easier to understand. This shows that clear prompts lead to better programming results using AI tools.

Code:

```python
def calculator_v1():
    try:
        num1 = float(input("Enter first number: "))
        operator = input("Enter operator (+, -, *, /): ")
        num2 = float(input("Enter second number: "))
        if operator == '+':
            result = num1 + num2
        elif operator == '-':
            result = num1 - num2
        elif operator == '*':
            result = num1 * num2
        elif operator == '/':
            if num2 == 0:
                print("Error: Division by zero is not allowed.")
                return
            result = num1 / num2
        else:
            print("Error: Invalid operator.")
            return
        print(f"The result is: {result}")
    except ValueError:
        print("Error: Invalid input. Please enter numbers.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
print("--- Testing Calculator Version 1 ---")
# Example run: Enter 22, +, 5
calculator_v1()
```

Output:

```
--- Testing Calculator Version 1 ---
Enter first number: 2
Enter operator (+, -, *, /): *
Enter second number: 3
The result is: 6.0
```

Task Description-2

• Refining Prompts for Sorting Logic: Start with a vague prompt for sorting student marks, then refine it to clearly specify sorting order and constraints.

Expected Output-2

• AI-generated sorting function evolves from ambiguous logic to an accurate and efficient implementation.

Prompt:

Write a Python function to sort student marks in ascending order. Show the original list and the sorted list.

Comments:

1. This task shows how refining a vague prompt helps the AI generate a correct and well-defined sorting function.

2. Clear instructions about sorting order and constraints improve the accuracy and efficiency of the code.

Explanation:

This task shows how improving a prompt step by step helps the AI generate better sorting logic. When the prompt is vague, the sorting code is basic and unclear. As the prompt specifies the sorting order and input rules, the code becomes more accurate and efficient. Adding clear constraints helps avoid errors and improves correctness. This proves that clear prompts lead to better AI-generated programs.

Code & Output :

```
def sort_student_marks(marks):
    # This function sorts a list of student marks.
    # It should take a list of numerical marks as input and return a new list
    # with the marks sorted in ascending order.
    # Usage Examples:
    # 1. Basic sorting: Input = [85, 92, 78], Expected Output = [78, 85, 92]
    # 2. Duplicate marks: Input = [70, 70, 60], Expected Output = [60, 70, 70]
    # 3. Empty list: Input = [], Expected Output = []
    # 4. Single mark: Input = [90], Expected Output = [90]
    return sorted(marks)
```

```
sample_marks = [85, 92, 78, 65, 95, 88]
sorted_marks = sort_student_marks(sample_marks)
print(f"Original marks: {sample_marks}")
print(f"Sorted marks: {sorted_marks}")
sample_marks_empty = []
sorted_marks_empty = sort_student_marks(sample_marks_empty)
print(f"Original marks (empty): {sample_marks_empty}")
print(f"Sorted marks (empty): {sorted_marks_empty}")
sample_marks_single = [90]
sorted_marks_single = sort_student_marks(sample_marks_single)
print(f"Original marks (single): {sample_marks_single}")
print(f"Sorted marks (single): {sorted_marks_single}")
```

```
Original marks: [85, 92, 78, 65, 95, 88]
Sorted marks: [65, 78, 85, 88, 92, 95]
Original marks (empty): []
Sorted marks (empty): []
Original marks (single): [90]
Sorted marks (single): [90]
```

Task Description-3

• Few-Shot Prompting for Prime Number Validation: Provide multiple input output examples for a function that checks whether a number is prime. Observe how few-shot prompting improves correctness.

Expected Output-3

• Improved prime-checking function with better edge-case handling.

Prompt:

Write a Python function to check whether a number is prime.

Comments:

1.Few-shot prompting helps the AI understand prime number rules through examples, improving accuracy.

2.Providing edge-case examples leads to better handling of numbers like 0 and 1.

Explanation:

This task explains how few-shot prompting improves prime number validation by giving example inputs and outputs. When examples are provided, the AI clearly understands which numbers are prime and which are not. This helps the function handle edge cases like 0, 1, and even numbers correctly. As a result, the generated code becomes more accurate and reliable.

Code & Output :

```
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True
```

```
print(f"Is 0 prime? {is_prime(0)}")
print(f"Is 1 prime? {is_prime(1)}")
print(f"Is 2 prime? {is_prime(2)}")
print(f"Is 7 prime? {is_prime(7)}")
print(f"Is 10 prime? {is_prime(10)}")
print(f"Is 29 prime? {is_prime(29)}")
print(f"Is 33 prime? {is_prime(33)}")
```

```
Is 0 prime? False
Is 1 prime? False
Is 2 prime? True
Is 7 prime? True
Is 10 prime? False
Is 29 prime? True
Is 33 prime? False
```

Task Description-4

Prompt-Guided UI Design for Student Grading System: Create a user interface for a student grading system that calculates total marks, percentage, and grade based on user input.

Expected Output-4

Well-structured UI code with accurate calculations and clear output display.

Prompt:

Design a UI for a student grading system that calculates total marks and percentage. Display the results on the screen.

Comments:

1.Prompt-guided instructions help the AI design a clear and well-structured user interface.

2.Clear requirements ensure accurate calculations and easy-to-understand output display.

Explanation:

This task explains how clear prompts help in designing a proper user interface using AI. When the prompt clearly mentions inputs, calculations, and output display, the AI creates a structured and user-friendly grading system. The UI correctly calculates total marks and percentage and assigns grades based on the result. Clear prompts also help in displaying results neatly. This shows that prompt guidance improves UI design quality.

Code & Output:

```python
import tkinter as tk
def create_grading_ui():
    root = tk.Tk()
    root.title("Student Grading System")
    # Student Name Input
    tk.Label(root, text="Student Name:").grid(row=0, column=0, padx=5, pady=5, sticky="w")
    name_entry = tk.Entry(root, width=30)
    name_entry.grid(row=0, column=1, padx=5, pady=5)
    # Subject Marks Input
    tk.Label(root, text="Subject 1 Marks:").grid(row=1, column=0, padx=5, pady=5, sticky="w")
    subject1_entry = tk.Entry(root, width=10)
    subject1_entry.grid(row=1, column=1, padx=5, pady=5)
    tk.Label(root, text="Subject 2 Marks:").grid(row=2, column=0, padx=5, pady=5, sticky="w")
    subject2_entry = tk.Entry(root, width=10)
    subject2_entry.grid(row=2, column=1, padx=5, pady=5)
    tk.Label(root, text="Subject 3 Marks:").grid(row=3, column=0, padx=5, pady=5, sticky="w")
    subject3_entry = tk.Entry(root, width=10)
    subject3_entry.grid(row=3, column=1, padx=5, pady=5)
    # Display Labels for Results
    total_marks_label = tk.Label(root, text="Total Marks: ")
    total_marks_label.grid(row=4, column=0, columnspan=2, padx=5, pady=5, sticky="w")
    percentage_label = tk.Label(root, text="Percentage: ")
    percentage_label.grid(row=5, column=0, columnspan=2, padx=5, pady=5, sticky="w")
    grade_label = tk.Label(root, text="Grade: ")
    grade_label.grid(row=6, column=0, columnspan=2, padx=5, pady=5, sticky="w")
    # For now, just create the UI without calculation logic
    # In a real app, you'd add buttons and functions to interact with these widgets
    # root.mainloop() # Commented out to prevent TclError in non-graphical environments
# Call the function to create and run the UI
# create_grading_ui() # Commented out for the same reason
print("Tkinter UI code is defined, but the UI cannot be displayed in this environment (no display server).")
print("To see the UI, please run this code in a local Python environment with a display server.")
```

```
Tkinter UI code is defined, but the UI cannot be displayed in this environment (no display server).
To see the UI, please run this code in a local Python environment with a display server.
```

Task Description-5

Analyzing Prompt Specificity in Unit Conversion Functions: Improving a Unit Conversion Function (Kilometers to Miles and Miles to Kilometers) Using Clear Instructions.

Expected Output-5

Analysis of code quality and accuracy differences across multiple prompt variations.

Prompt:

Write a Python function that converts kilometers to miles and miles to kilometers. Show clear output for both conversions.

Comments:

1.Clear and specific prompts help the AI generate accurate and reliable unit conversion logic.

2.Adding conversion direction and input rules improves code correctness and readability.

Explanation:

This task explains how prompt specificity affects the quality of AI-generated unit conversion functions. When the prompt is vague, the conversion logic may be incomplete or inaccurate. By clearly specifying conversion types and input rules, the AI produces correct and reliable code. Clear instructions also help in

handling invalid inputs properly. This shows that detailed prompts lead to better and more accurate programs.

Code:

```
def convert_units(value, from_unit, to_unit):
    # This function converts a numerical value between kilometers and miles.
    # Parameters:
    #   value (float or int): The numerical amount to be converted.
    #   from_unit (str): The unit of the input value. Expected values: 'km' (kilometers) or 'miles'.
    #   to_unit (str): The desired unit for the output. Expected values: 'km' or 'miles'.
    # Conversion Factor:
    #   1 mile = 1.60934 kilometers
    #   1 kilometer = 0.621371 miles
    # Error Handling Considerations:
    # - If 'from_unit' or 'to_unit' is not 'km' or 'miles', raise a ValueError for invalid unit.
    # - If 'value' is not a number, raise a TypeError.
    # - If 'from_unit' and 'to_unit' are the same, return the original value without conversion.
    # Usage Examples:
    # 1. Kilometers to Miles:
    #    Input: value=10, from_unit='km', to_unit='miles'
    #    Expected Output: 6.21371
    # 2. Miles to Kilometers:
    #    Input: value=5, from_unit='miles', to_unit='km'
    #    Expected Output: 8.0467
    # 3. Invalid 'from_unit':
    #    Input: value=10, from_unit='meters', to_unit='miles'
    #    Expected Error: ValueError("Invalid 'from_unit'. Must be 'km' or 'miles'.")
    # 4. Invalid 'to_unit':
    #    Input: value=10, from_unit='km', to_unit='centimeters'
    #    Expected Error: ValueError("Invalid 'to_unit'. Must be 'km' or 'miles'.")
    # 5. Non-numeric 'value':
    #    Input: value='abc', from_unit='km', to_unit='miles'
    #    Expected Error: TypeError("Input 'value' must be a number.")
    # 6. Same units:
    #    Input: value=10, from_unit='km', to_unit='km'
    #    Expected Output: 10
    if not isinstance(value, (int, float)):
        raise TypeError("Input 'value' must be a number.")
    valid_units = {'km', 'miles'}
    if from_unit not in valid_units:
        raise ValueError(f"Invalid 'from_unit': {from_unit}. Must be 'km' or 'miles'.")
    if to_unit not in valid_units:
        raise ValueError(f"Invalid 'to_unit': {to_unit}. Must be 'km' or 'miles'.")
    if from_unit == to_unit:
        return value
    # Conversion factors
    km_to_miles_factor = 0.621371
    miles_to_km_factor = 1.60934
    if from_unit == 'km' and to_unit == 'miles':
        return value * km_to_miles_factor
    elif from_unit == 'miles' and to_unit == 'km':
        return value * miles_to_km_factor
```

```python
print("--- Testing convert_units function (Version with Comments, Specific Instructions, and Usage Examples) ---")
# 1. Kilometers to Miles
try:
    result1 = convert_units(value=10, from_unit='km', to_unit='miles')
    print(f"10 km to miles: {result1:.5f} (Expected: 6.21371)")
except Exception as e:
    print(f"Error for 10 km to miles: {e}")
# 2. Miles to Kilometers
try:
    result2 = convert_units(value=5, from_unit='miles', to_unit='km')
    print(f"5 miles to km: {result2:.4f} (Expected: 8.0467)")
except Exception as e:
    print(f"Error for 5 miles to km: {e}")
# 3. Invalid 'from_unit'
try:
    convert_units(value=10, from_unit='meters', to_unit='miles')
except ValueError as e:
    print(f"Caught expected error for invalid from_unit: {e}")
except Exception as e:
    print(f"Unexpected error for invalid from_unit: {e}")
# 4. Invalid 'to_unit'
try:
    convert_units(value=10, from_unit='km', to_unit='centimeters')
except ValueError as e:
    print(f"Caught expected error for invalid to_unit: {e}")
except Exception as e:
    print(f"Unexpected error for invalid to_unit: {e}")
# 5. Non-numeric 'value'
try:
    convert_units(value='abc', from_unit='km', to_unit='miles')
except TypeError as e:
    print(f"Caught expected error for non-numeric value: {e}")
except Exception as e:
    print(f"Unexpected error for non-numeric value: {e}")
# 6. Same units
try:
    result6 = convert_units(value=10, from_unit='km', to_unit='km')
    print(f"10 km to km: {result6} (Expected: 10)")
except Exception as e:
    print(f"Error for same units: {e}")
# Additional test: negative value
try:
    result7 = convert_units(value=-10, from_unit='km', to_unit='miles')
    print(f"-10 km to miles: {result7:.5f} (Expected: -6.21371)")
except Exception as e:
    print(f"Error for negative value: {e}")
```

Output:

```
--- Testing convert_units function (Version with Comments, Specific Instructions, and Usage Examples) ---
10 km to miles: 6.21371 (Expected: 6.21371)
5 miles to km: 8.0467 (Expected: 8.0467)
Caught expected error for invalid from_unit: Invalid 'from_unit': meters. Must be 'km' or 'miles'.
Caught expected error for invalid to_unit: Invalid 'to_unit': centimeters. Must be 'km' or 'miles'.
Caught expected error for non-numeric value: Input 'value' must be a number.
10 km to km: 10 (Expected: 10)
-10 km to miles: -6.21371 (Expected: -6.21371)
```