

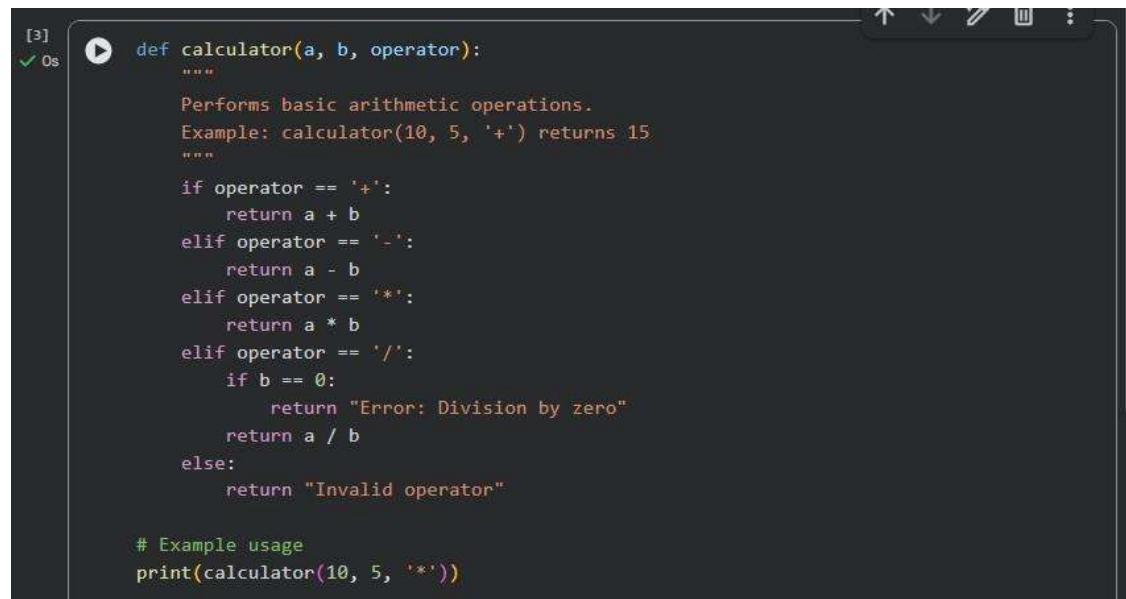
School of Computer Science and Artificial Intelligence**Lab Assignment # 3.2**

Program	: B. Tech (CSE)
Specialization	:
Course Title	: AI Assisted coding
Course Code	:
Semester	: II
Academic Session	: 2025-2026
Name of Student	: Vemula Rikith
Enrollment No.	: 2403A51L35
Batch No.	: 52
Date	: 13-01-2026

Task - 1

Prompt : Create a calculator function that performs addition, subtraction, multiplication, and division.

Include error handling for division by zero.



The screenshot shows a code editor window with a dark theme. The code is a Python function named `calculator` that takes three parameters: `a`, `b`, and `operator`. The function performs basic arithmetic operations based on the operator. It includes error handling for division by zero. The code is annotated with docstrings and comments explaining its purpose and usage.

```
[3] 0s
def calculator(a, b, operator):
    """
    Performs basic arithmetic operations.
    Example: calculator(10, 5, '+') returns 15
    """

    if operator == '+':
        return a + b
    elif operator == '-':
        return a - b
    elif operator == '*':
        return a * b
    elif operator == '/':
        if b == 0:
            return "Error: Division by zero"
        return a / b
    else:
        return "Invalid operator"

    # Example usage
    print(calculator(10, 5, '*'))
```

Explanation

In this task, the goal was to understand how **progressively improving a prompt** affects the quality of AI-generated code.

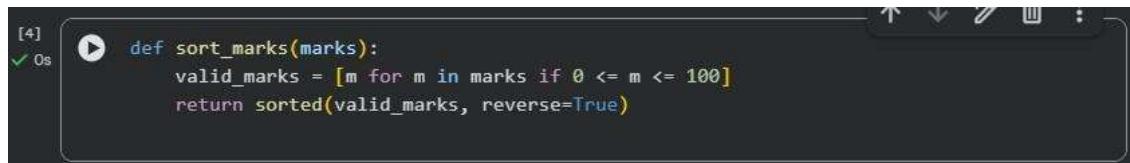
- When a **minimal prompt** was used, the AI generated a very basic calculator function with limited operations and no error handling.
- Adding **comments and context** helped the AI understand the expected functionality, resulting in support for more arithmetic operations.
- Including **constraints and usage examples** further improved the code by adding proper error handling, documentation, and clearer structure.

Output:

```
# Example usage
print(calculator(10, 5, '*'))
...
50
```

Task – 2

Prompt: Write a Python function to sort student marks in descending order. Marks should be integers between 0 and 100.



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

```
[4] 0s  def sort_marks(marks):
    valid_marks = [m for m in marks if 0 <= m <= 100]
    return sorted(valid_marks, reverse=True)
```

Explanation

This task focused on improving AI output by **refining vague prompts into specific ones.**

- A vague prompt resulted in a basic sorting function without a defined order or validation.
- When sorting order and constraints (valid mark range) were explicitly mentioned, the AI produced a more accurate and meaningful solution.
- The refined prompt led to better logic, including filtering invalid values and sorting in the correct order.

Output:

```
[5] ✓ 0s
    print(sort_marks([78, 95, 120, 67, -5]))
[95, 78, 67]
```

Task–

3

Prompt:

Create a Python function `is_prime(n)` that checks whether a given number is prime.

Examples:

`is_prime(2)` → True

`is_prime(3)` → True

`is_prime(4)` → False

`is_prime(17)` → True

`is_prime(1)` → False

`is_prime(0)` → False

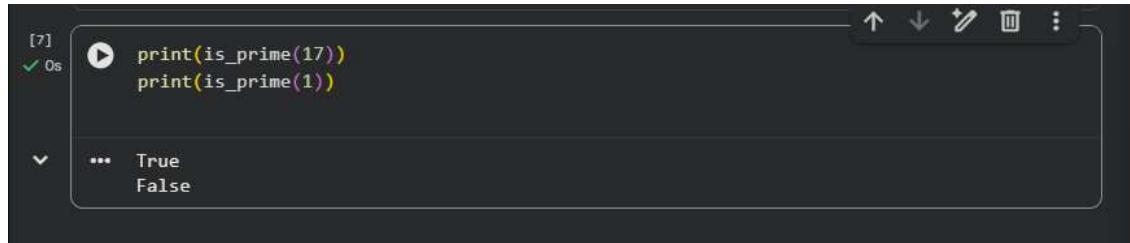
```
[6] ✓ 0s
    def is_prime(n):
        if n <= 1:
            return False
        for i in range(2, int(n ** 0.5) + 1):
            if n % i == 0:
                return False
        return True
```

Explanation

Few-shot prompting involves providing **example inputs and expected outputs** along with the prompt.

- By including multiple examples, the AI clearly understood how to handle edge cases such as 0 and 1.
- The generated code correctly implemented an optimized prime-checking algorithm.
- Compared to zero-shot prompting, few-shot prompting significantly improved correctness and efficiency

Output:



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

```
[7] ✓ 0s
    print(is_prime(17))
    print(is_prime(1))

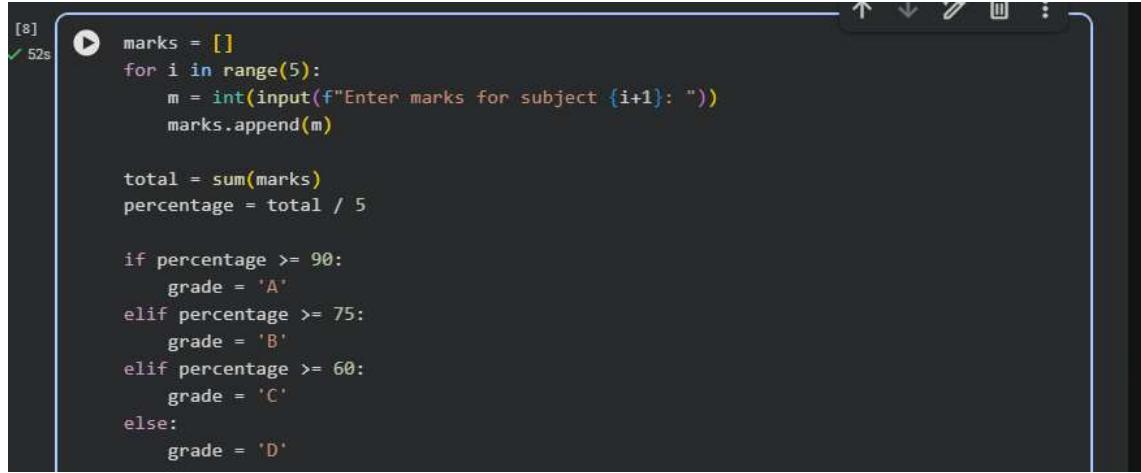
    ...
    ... True
    ... False
```

The cell has a status bar indicating it took 0 seconds to run. The output shows two lines of text: "True" and "False".

Task-4 Prompt:

Create a simple Python-based user interface for a student grading system. The program should:

- Ask the user to enter marks for 5 subjects
- Calculate total marks
- Calculate percentage
- Display grade based on percentage



```
[8] 52s
marks = []
for i in range(5):
    m = int(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'
```

Explanation

This task demonstrated how a **structured and detailed prompt** can guide AI to create a complete user-interface-based program.

The prompt clearly specified user input, calculations, and output requirements.

As a result, the AI generated a well-structured program that calculates total marks, percentage, and grade.

The code followed a logical flow, making it easy to understand and user-friendly.

Output:

```
print("Total Marks:", total)
print("Percentage:", percentage)
print("Grade:", grade)

...
*** Enter marks for subject 1: 10
Enter marks for subject 2: 20
Enter marks for subject 3: 30
Enter marks for subject 4: 40
Enter marks for subject 5: 50
Total Marks: 150
Percentage: 30.0
Grade: D
```

Task – 5:

Prompt:

Create two Python functions:

1. Convert kilometers to miles
2. Convert miles to kilometers

Use accurate conversion formulas and return the result.

```
[9]
✓ 0s
def km_to_miles(km):
    return km * 0.621371

def miles_to_km(miles):
    return miles / 0.621371
```

Explanation

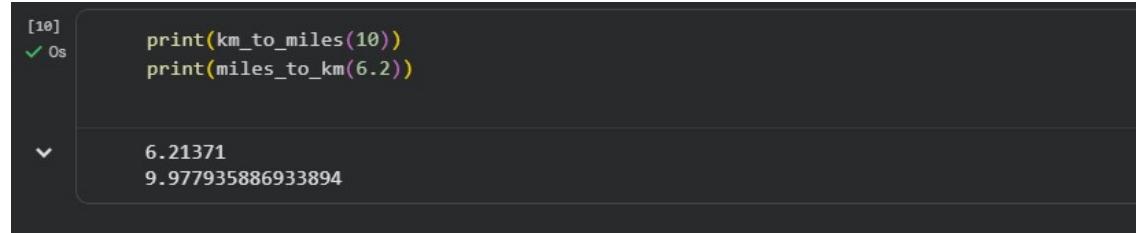
This task analyzed how **prompt specificity affects accuracy and code quality**.

A vague prompt produced an unclear and inaccurate conversion function.

A more specific prompt resulted in correct and separate functions for each unit conversion.

Explicit instructions ensured the use of correct formulas and meaningful function names.

Output:



```
[10] ✓ 0s
print(km_to_miles(10))
print(miles_to_km(6.2))

6.21371
9.977935886933894
```

Conclusion

Across all tasks, it was observed that:

- AI performance improves with **clear, structured, and specific prompts**.
- Adding comments, constraints, and examples significantly enhances output quality.
- Prompt engineering is a critical skill for effective AI-assisted programming.