

Harika Jupaka

2403A51L31

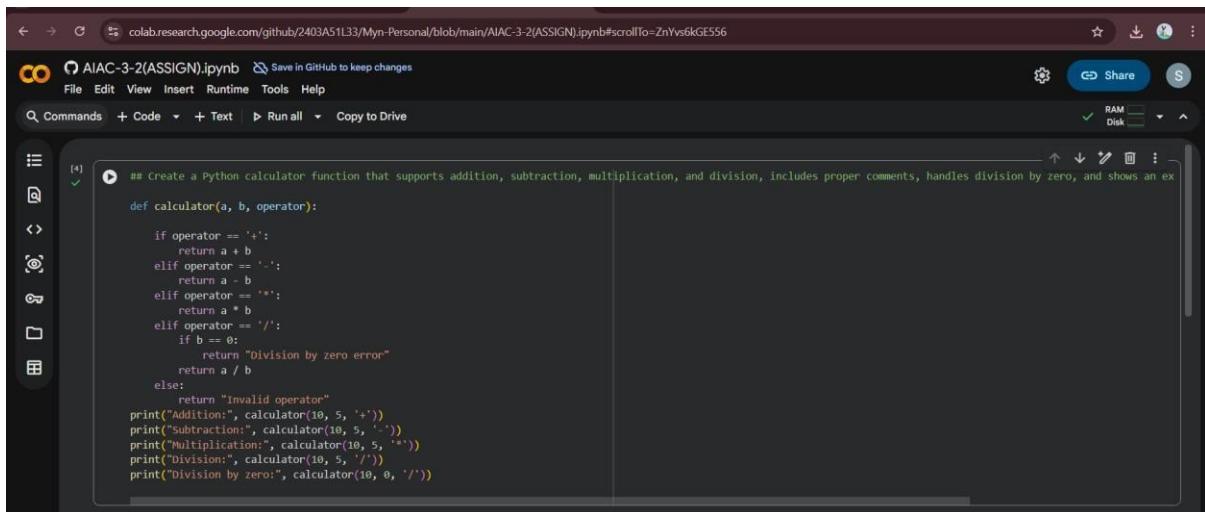
B-52

ASSIGNMENT – 3.2

Lab 3: Prompt Engineering – Improving Prompts and Context Management

Task– 1: Progressive Prompting for Calculator Design

Prompt: Create a Python calculator function that supports addition, subtraction, multiplication, and division, includes proper comments, handles division by zero, and shows an example of how the function is used.



The screenshot shows a Google Colab notebook titled "AIAC-3-2(ASSIGN).ipynb". The code cell contains the following Python code:

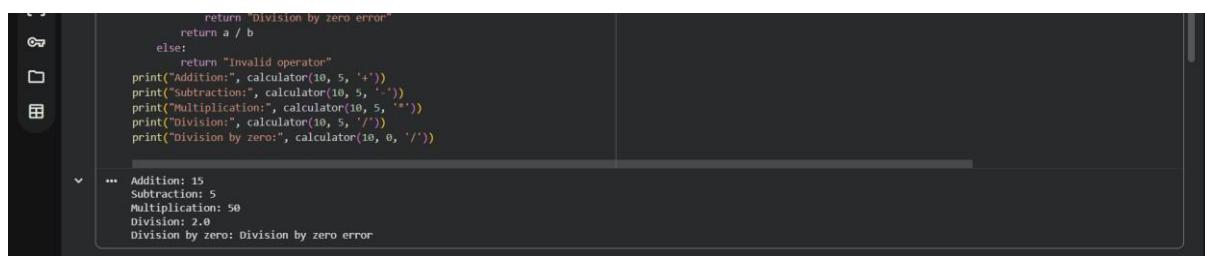
```
## Create a Python calculator function that supports addition, subtraction, multiplication, and division, includes proper comments, handles division by zero, and shows an example of how the function is used.

def calculator(a, b, operator):

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

print("Addition:", calculator(10, 5, '+'))
print("Subtraction:", calculator(10, 5, '-'))
print("Multiplication:", calculator(10, 5, '*'))
print("Division:", calculator(10, 5, '/'))
print("Division by zero:", calculator(10, 0, '/'))
```

OUTPUT:



The screenshot shows the output of the code execution in Google Colab. The output is:

```
... Addition: 15
Subtraction: 5
Multiplication: 50
Division: 2.0
Division by zero: Division by zero error
```

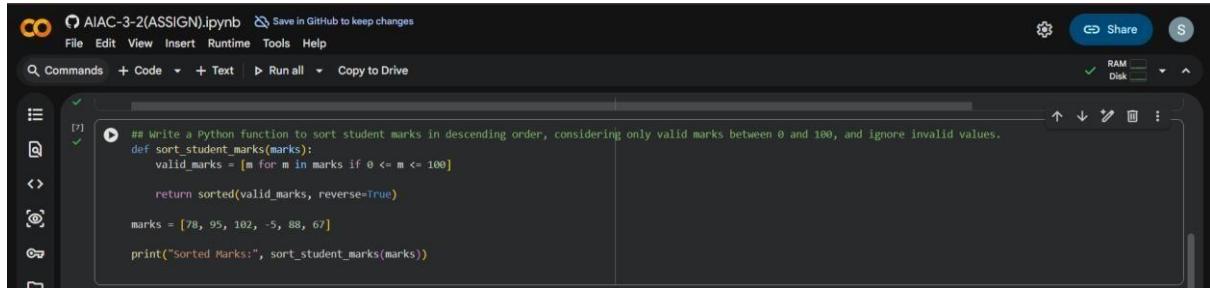
Explanation:

Initially, limited prompt information leads to a simple implementation. Adding comments, examples, and constraints helps the AI generate structured logic with proper error handling.

This shows how prompt refinement improves code quality.

Task – 2: Refining Prompts for Sorting Logic

Prompt: Write a Python function to sort student marks in descending order, considering only valid marks between 0 and 100, and ignore invalid values.



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

```
## Write a Python function to sort student marks in descending order, considering only valid marks between 0 and 100, and ignore invalid values.
def sort_student_marks(marks):
    valid_marks = [m for m in marks if 0 <= m <= 100]
    return sorted(valid_marks, reverse=True)

marks = [78, 95, 102, -5, 88, 67]
print("Sorted Marks:", sort_student_marks(marks))
```

OUTPUT:



The screenshot shows the execution output of the code cell, which prints the sorted marks:

```
marks = [78, 95, 102, -5, 88, 67]
print("Sorted Marks:", sort_student_marks(marks))
...
... Sorted Marks: [95, 88, 78, 67]
```

Explanation:

A vague prompt results in generic sorting without validation.

Providing clear constraints such as order and valid range enables the AI to produce accurate and meaningful logic.

Prompt clarity removes ambiguity in implementation.

Task– 3: Few-Shot Prompting for Prime Number Validation

Prompt: Using the examples (2 → True, 4 → False, 1 → False), write a Python function that checks whether a given number is prime and correctly handles edge cases.

The screenshot shows a Google Colab interface. In the top bar, there's a link to colab.research.google.com and a GitHub integration button. The menu bar includes File, Edit, View, Insert, Runtime, Tools, and Help. On the right, there are sharing and storage options (Share, RAM, Disk). The main area has a code cell containing Python code. The code defines a function `is_prime` that checks if a number is prime. It includes examples for numbers 2, 4, 1, and 13. The output of the code cell shows the results of these examples.

```
print("Is 2 prime?", is_prime(2))
print("Is 4 prime?", is_prime(4))
print("Is 1 prime?", is_prime(1))
print("Is 13 prime?", is_prime(13))
```

OUTPUT:

This screenshot shows the execution results of the code in the previous image. The terminal output pane displays the results of the `is_prime` function calls for the numbers 2, 4, 1, and 13. The output shows that 2 and 13 are prime (True), while 4 and 1 are not (False).

```
Is 2 prime? True
Is 4 prime? False
Is 1 prime? False
Is 13 prime? True
```

Explanation:

Few-shot prompting provides clear expectations through examples. This helps the AI understand edge cases and generate correct primechecking logic.

Accuracy improves compared to zero-example prompts.

Task– 4: Prompt-Guided UI Design for Student Grading System

Prompt: Create a Python program that takes student marks as input, calculates total marks, percentage, and grade, and displays the results clearly to the user.

```

## Create a Python program that takes student marks as input, calculates total marks, percentage, and grade, and displays the results clearly to the user.
marks1 = float(input("Enter marks for Subject 1: "))
marks2 = float(input("Enter marks for Subject 2: "))
marks3 = float(input("Enter marks for Subject 3: "))
marks4 = float(input("Enter marks for Subject 4: "))
marks5 = float(input("Enter marks for Subject 5: "))

total_marks = marks1 + marks2 + marks3 + marks4 + marks5

percentage = (total_marks / 500) * 100

if percentage >= 90:
    grade = "A+"
elif percentage >= 80:
    grade = "A"
elif percentage >= 70:
    grade = "B"
elif percentage >= 60:
    grade = "C"
elif percentage >= 50:
    grade = "D"
else:
    grade = "Fail"

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

```

Variables Terminal ✓ 12:24 PM Python 3

OUTPUT:

```

print("Percentage: ", percentage, "%")
print("Grade: ", grade)

... Enter marks for Subject 1: 24
Enter marks for Subject 2: 58
Enter marks for Subject 3: 48
Enter marks for Subject 4: 68
Enter marks for Subject 5: 78

--- Student Result ---
Total Marks: 268.0
Percentage: 52.0 %
Grade: D

```

Explanation:

Clear prompt instructions guide the AI to generate a structured and interactive program.

The code correctly handles user input, calculations, and result display. Prompt guidance improves usability and readability.

Task– 5: Analysing Prompt Specificity in Unit Conversion Functions

Prompt: Write two Python functions to accurately convert kilometers to miles and miles to kilo-meters using standard conversion values and clear function names.

```

File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all Copy to Drive RAM Disk

... Percentage: 52.0 %
Grade: D

[?] ✓ Os
## Write two Python functions to accurately convert kilometers to miles and miles to kilo-meters using standard conversion values and clear function names.
def kilometers_to_miles(kilometers):
    return kilometers * 0.621371

def miles_to_kilometers(miles):
    return miles / 0.621371
print(kilometers_to_miles(10))
print(miles_to_kilometers(5))

```

OUTPUT:

The screenshot shows a Google Colab interface with a dark theme. At the top, it displays the file name "AIAC-3-2(ASSIGN).ipynb" and a "Save in GitHub to keep changes" button. The menu bar includes File, Edit, View, Insert, Runtime, Tools, and Help. Below the menu is a toolbar with icons for Commands, Code, Text, Run all, and Copy to Drive. On the right side, there are status indicators for RAM and Disk usage. The main area contains a code cell with the following Python code:

```
def miles_to_kilometers(miles):
    return miles / 0.621371
print(kilometers_to_miles(10))
print(miles_to_kilometers(5))
```

The output of the code is displayed below the cell, showing the results of the two print statements.

Explanation:

Specific prompts lead to accurate and well-defined conversion functions. Clear instructions ensure correct formulas and readable function names. This demonstrates how prompt specificity improves program correctness.