AI ASSISTANT CODING ASSIGNMENT 3.2

NAME :MD SHAHID

ROLL NO : 2303A51421

BATCH NO : 21

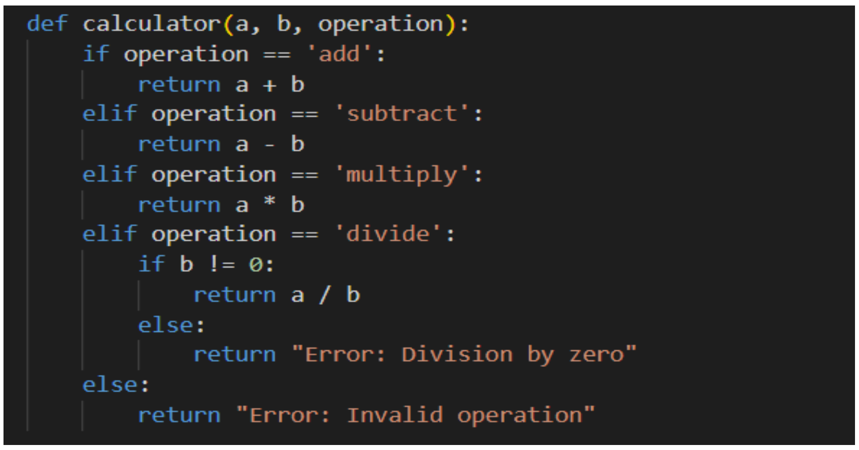
Task Description-1

• 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.

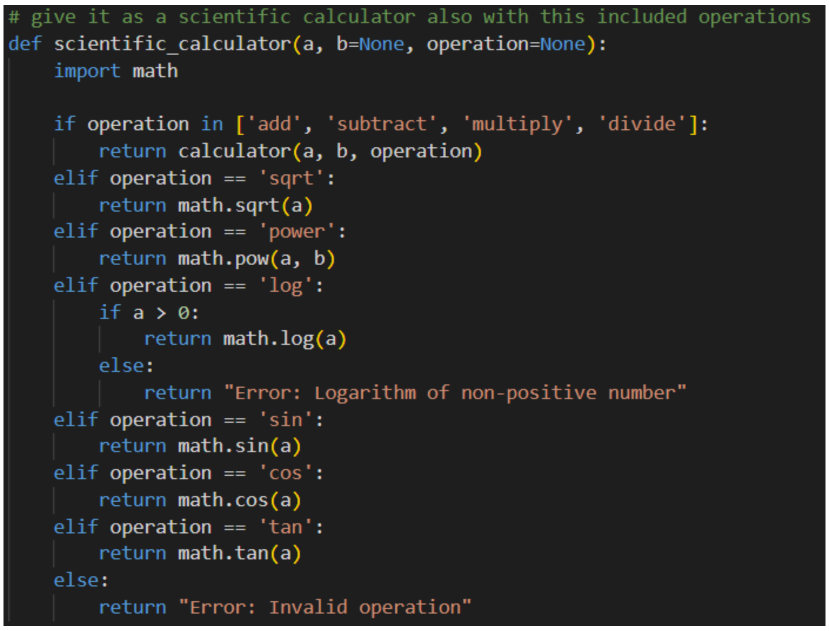
Expected Output-1

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

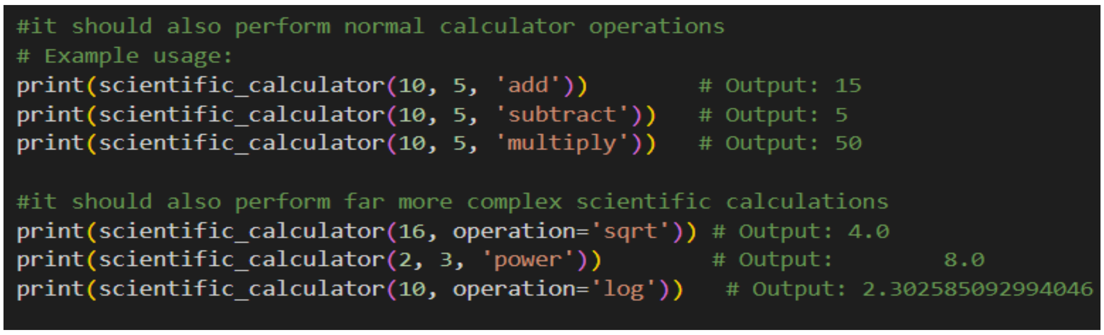
Output :



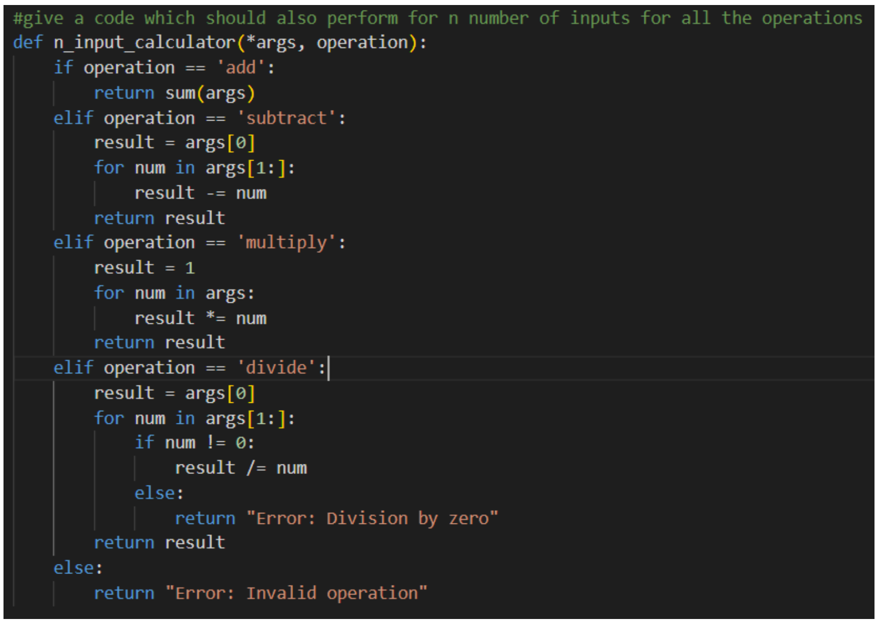
```python
def calculator(a, b, operation):
    if operation == 'add':
        return a + b
    elif operation == 'subtract':
        return a - b
    elif operation == 'multiply':
        return a * b
    elif operation == 'divide':
        if b != 0:
            return a / b
        else:
            return "Error: Division by zero"
    else:
        return "Error: Invalid operation"
```



```python
# give it as a scientific calculator also with this included operations
def scientific_calculator(a, b=None, operation=None):
    import math

    if operation in ['add', 'subtract', 'multiply', 'divide']:
        return calculator(a, b, operation)
    elif operation == 'sqrt':
        return math.sqrt(a)
    elif operation == 'power':
        return math.pow(a, b)
    elif operation == 'log':
        if a > 0:
            return math.log(a)
        else:
            return "Error: Logarithm of non-positive number"
    elif operation == 'sin':
        return math.sin(a)
    elif operation == 'cos':
        return math.cos(a)
    elif operation == 'tan':
        return math.tan(a)
    else:
        return "Error: Invalid operation"
```

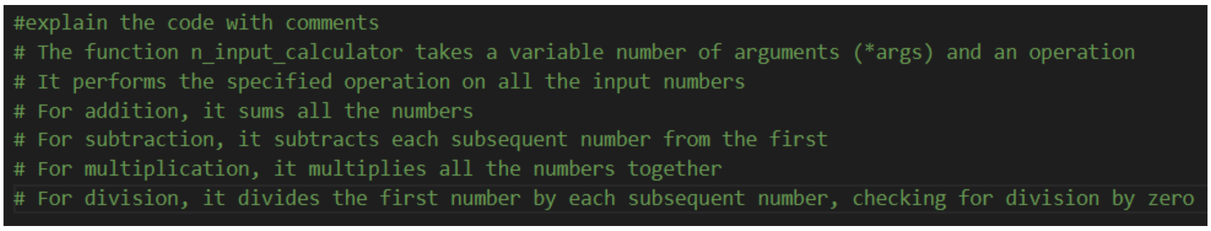


```python
#it should also perform normal calculator operations
# Example usage:
print(scientific_calculator(10, 5, 'add'))        # Output: 15
print(scientific_calculator(10, 5, 'subtract'))   # Output: 5
print(scientific_calculator(10, 5, 'multiply'))   # Output: 50

#it should also perform far more complex scientific calculations
print(scientific_calculator(16, operation='sqrt')) # Output: 4.0
print(scientific_calculator(2, 3, 'power'))        # Output:        8.0
print(scientific_calculator(10, operation='log'))   # Output: 2.302585092994046
```



```python
#give a code which should also perform for n number of inputs for all the operations
def n_input_calculator(*args, operation):
    if operation == 'add':
        return sum(args)
    elif operation == 'subtract':
        result = args[0]
        for num in args[1:]:
            result -= num
        return result
    elif operation == 'multiply':
        result = 1
        for num in args:
            result *= num
        return result
    elif operation == 'divide':
        result = args[0]
        for num in args[1:]:
            if num != 0:
                result /= num
            else:
                return "Error: Division by zero"
        return result
    else:
        return "Error: Invalid operation"
```

EXPLANATION :



```
#explain the code with comments
# The function n_input_calculator takes a variable number of arguments (*args) and an operation
# It performs the specified operation on all the input numbers
# For addition, it sums all the numbers
# For subtraction, it subtracts each subsequent number from the first
# For multiplication, it multiplies all the numbers together
# For division, it divides the first number by each subsequent number, checking for division by zero
```

Final Observation:

Initially, when only the function name was provided, the AI produced a shallow and incomplete calculator with minimal logic and poor structure. After adding descriptive comments, the output improved by introducing parameters and basic arithmetic operations. Finally, when usage examples were included, the AI generated a complete, well-organized calculator with proper conditions, validations, and error handling. This progression demonstrates that better prompts directly lead to better code quality. The more precise and contextual the instructions, the more logical, structured, and reliable the output becomes.

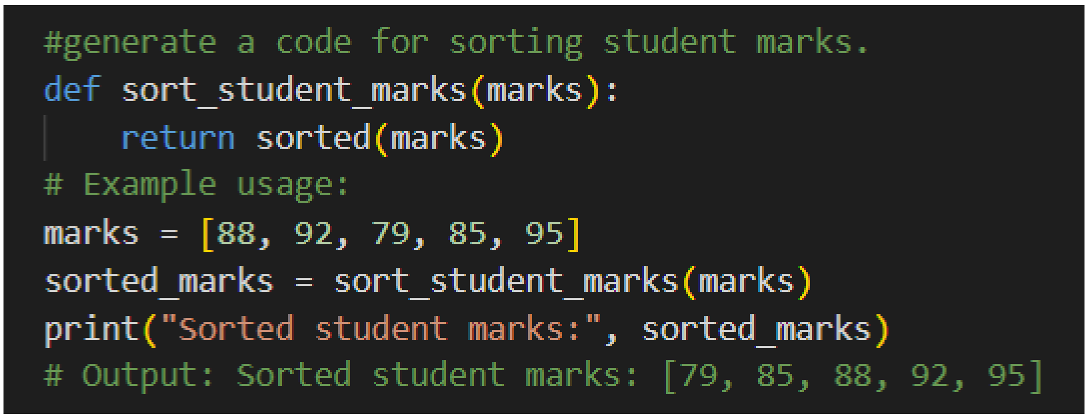
Task Description-2

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

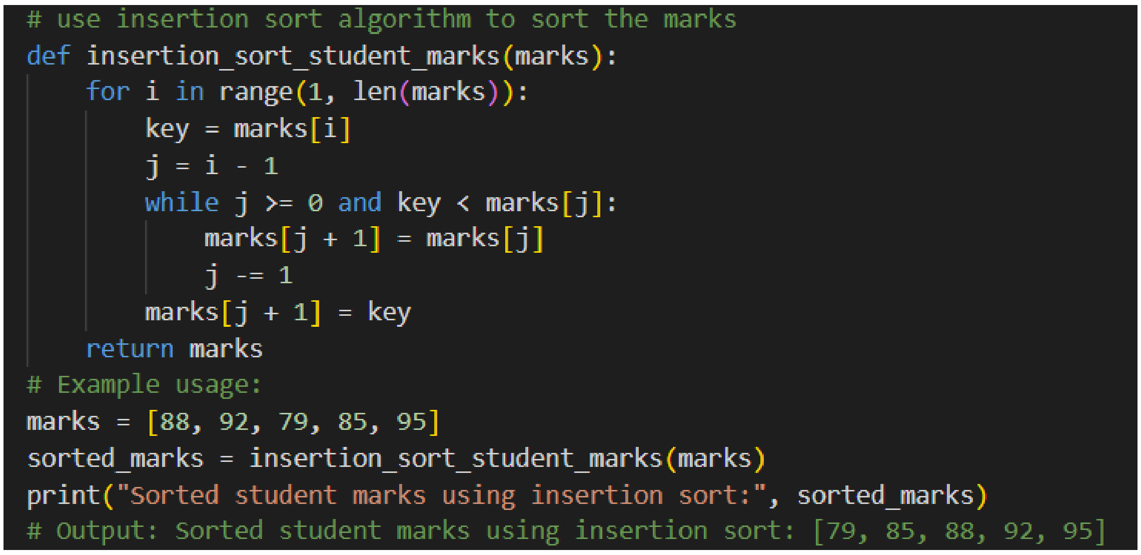
Expected Output-2

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

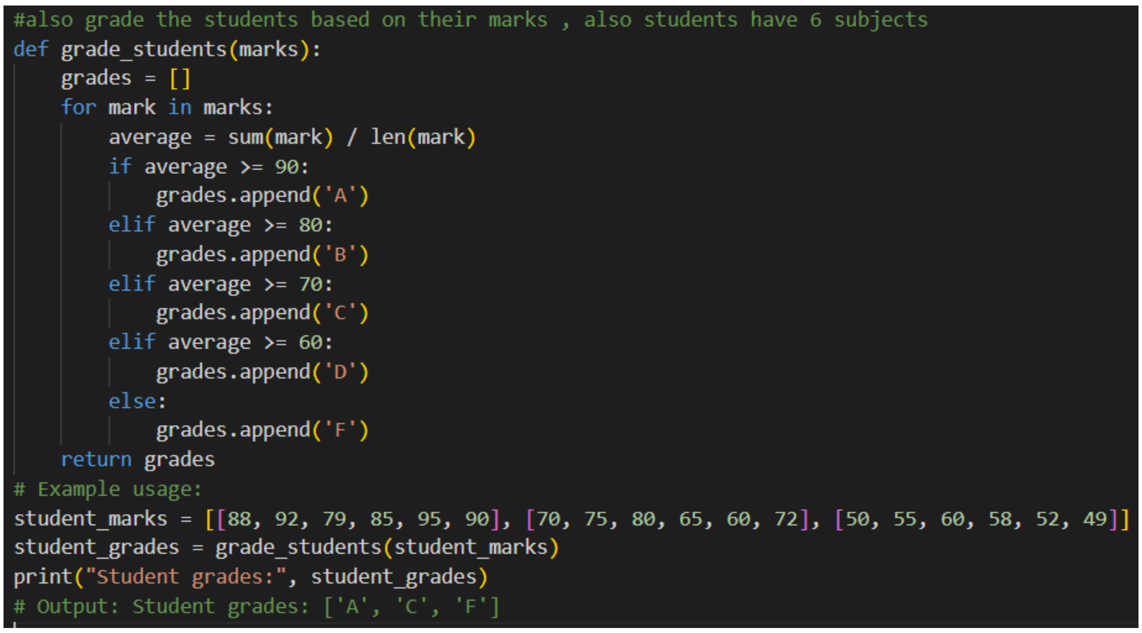
OUTPUT:



```python
#generate a code for sorting student marks.
def sort_student_marks(marks):
    return sorted(marks)
# Example usage:
marks = [88, 92, 79, 85, 95]
sorted_marks = sort_student_marks(marks)
print("Sorted student marks:", sorted_marks)
# Output: Sorted student marks: [79, 85, 88, 92, 95]
```

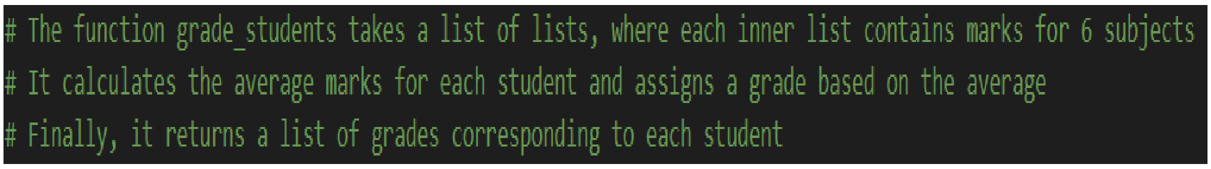


```python
# use insertion sort algorithm to sort the marks
def insertion_sort_student_marks(marks):
    for i in range(1, len(marks)):
        key = marks[i]
        j = i - 1
        while j >= 0 and key < marks[j]:
            marks[j + 1] = marks[j]
            j -= 1
        marks[j + 1] = key
    return marks
# Example usage:
marks = [88, 92, 79, 85, 95]
sorted_marks = insertion_sort_student_marks(marks)
print("Sorted student marks using insertion sort:", sorted_marks)
# Output: Sorted student marks using insertion sort: [79, 85, 88, 92, 95]
```



```python
#also grade the students based on their marks , also students have 6 subjects
def grade_students(marks):
    grades = []
    for mark in marks:
        average = sum(mark) / len(mark)
        if average >= 90:
            grades.append('A')
        elif average >= 80:
            grades.append('B')
        elif average >= 70:
            grades.append('C')
        elif average >= 60:
            grades.append('D')
        else:
            grades.append('F')
    return grades
# Example usage:
student_marks = [[88, 92, 79, 85, 95, 90], [70, 75, 80, 65, 60, 72], [50, 55, 60, 58, 52, 49]]
student_grades = grade_students(student_marks)
print("Student grades:", student_grades)
# Output: Student grades: ['A', 'C', 'F']
```

EXPLANATION:



```
# The function grade_students takes a list of lists, where each inner list contains marks for 6 subjects
# It calculates the average marks for each student and assigns a grade based on the average
# Finally, it returns a list of grades corresponding to each student
```

Final Observation:

When the prompt was unclear, the AI returned a generic sorting approach that lacked direction and ignored important constraints. As the instructions became more specific, such as defining the sorting order, the results grew more precise and relevant. After adding clear requirements and examples, the AI produced a well-structured, efficient, and reliable sorting function. This progression shows that the quality of the output directly depends on the clarity of the input—better prompts reduce ambiguity and lead to more accurate and dependable solutions.

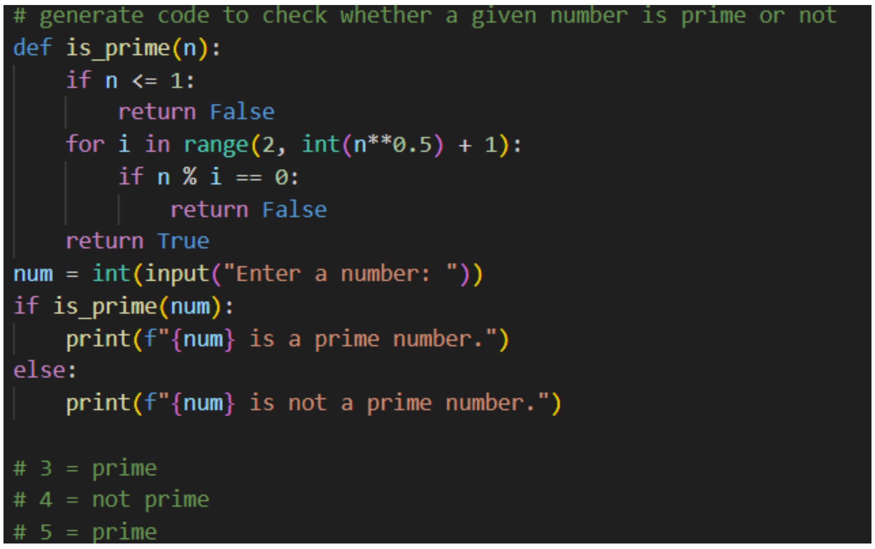
Task Description-3

• **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.**

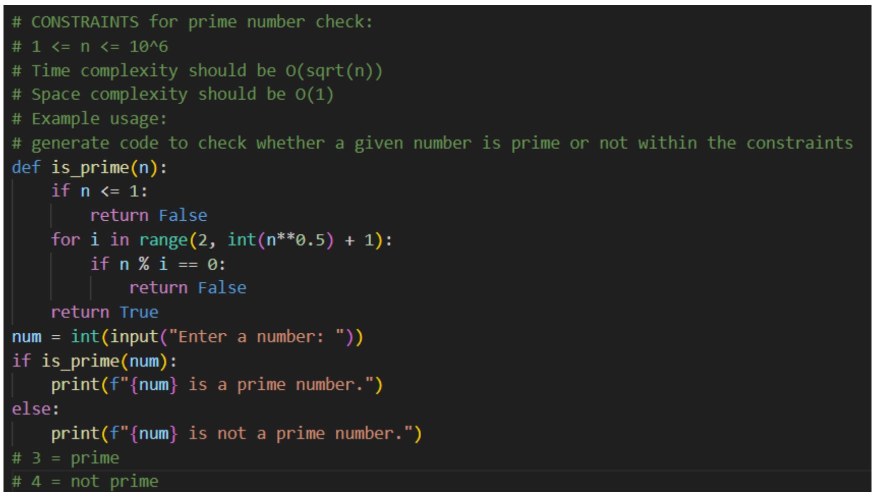
Expected Output-3

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

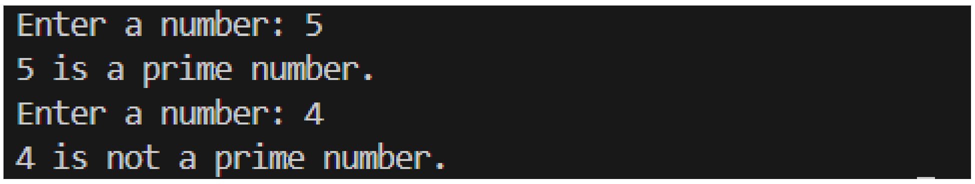
OUTPUT:



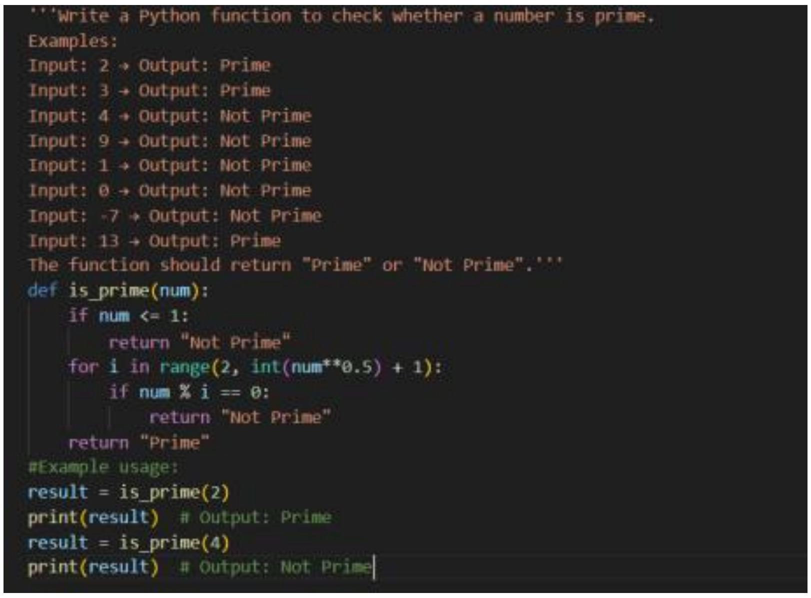
```python
# generate code to check whether a given number is prime or not
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
num = int(input("Enter a number: "))
if is_prime(num):
    print(f"{num} is a prime number.")
else:
    print(f"{num} is not a prime number.")

# 3 = prime
# 4 = not prime
# 5 = prime
```



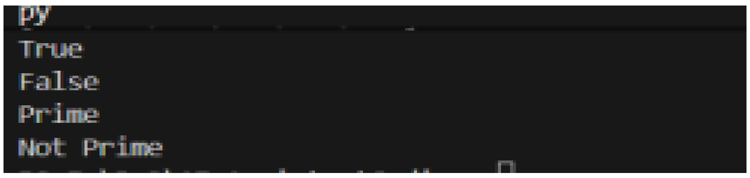
```python
# CONSTRAINTS for prime number check:
# 1 <= n <= 10^6
# Time complexity should be O(sqrt(n))
# Space complexity should be O(1)
# Example usage:
# generate code to check whether a given number is prime or not within the constraints
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
num = int(input("Enter a number: "))
if is_prime(num):
    print(f"{num} is a prime number.")
else:
    print(f"{num} is not a prime number.")
# 3 = prime
# 4 = not prime
```



```
Enter a number: 5
5 is a prime number.
Enter a number: 4
4 is not a prime number.
```

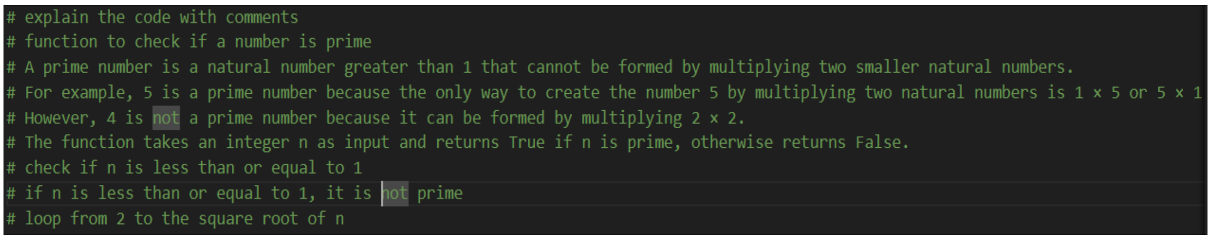


```python
'''Write a Python function to check whether a number is prime.
Examples:
Input: 2 → Output: Prime
Input: 3 → Output: Prime
Input: 4 → Output: Not Prime
Input: 9 → Output: Not Prime
Input: 1 → Output: Not Prime
Input: 0 → Output: Not Prime
Input: -7 → Output: Not Prime
Input: 13 → Output: Prime
The function should return "Prime" or "Not Prime".'''
def is_prime(num):
    if num <= 1:
        return "Not Prime"
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return "Not Prime"
    return "Prime"
#Example usage:
result = is_prime(2)
print(result)  # Output: Prime
result = is_prime(4)
print(result)  # Output: Not Prime
```



```
py
True
False
Prime
Not Prime
```

EXPLANATION :



```
# explain the code with comments
# function to check if a number is prime
# A prime number is a natural number greater than 1 that cannot be formed by multiplying two smaller natural numbers.
# For example, 5 is a prime number because the only way to create the number 5 by multiplying two natural numbers is 1 × 5 or 5 × 1
# However, 4 is not a prime number because it can be formed by multiplying 2 × 2.
# The function takes an integer n as input and returns True if n is prime, otherwise returns False.
# check if n is less than or equal to 1
# if n is less than or equal to 1, it is not prime
# loop from 2 to the square root of n
```

Final Observation:

With only a minimal prompt, the AI produced a simple prime-checking function that overlooked several edge cases. Adding a single example led to slight improvements, but the logic was still incomplete. When multiple input–output examples were provided, the AI handled special cases such as 0, 1, and negative numbers correctly and generated a more accurate and robust solution. This makes it clear that few-shot prompting significantly improves correctness, reliability, and edge-case handling.

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

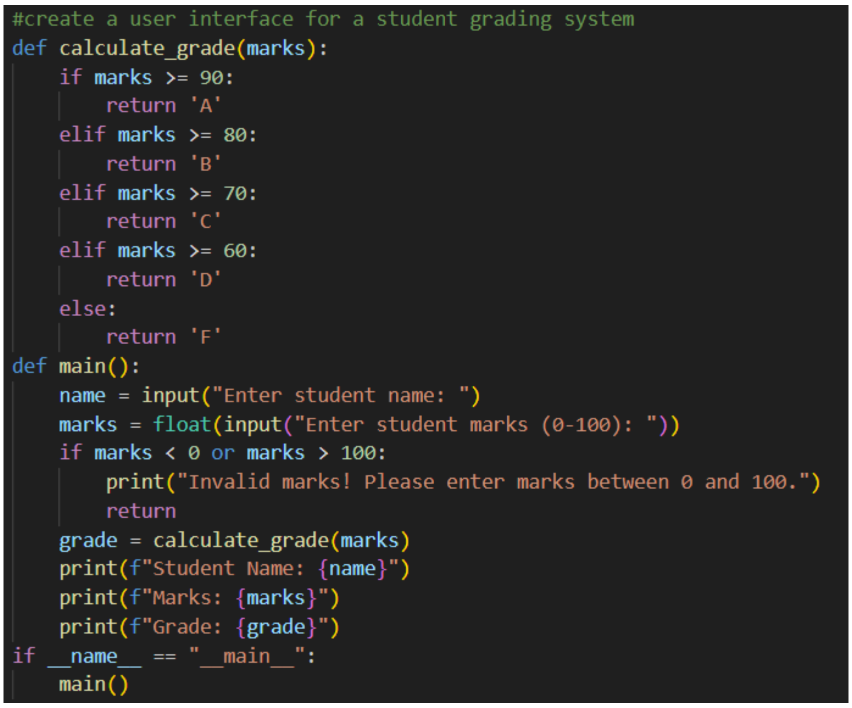
input.

Expected Output-4

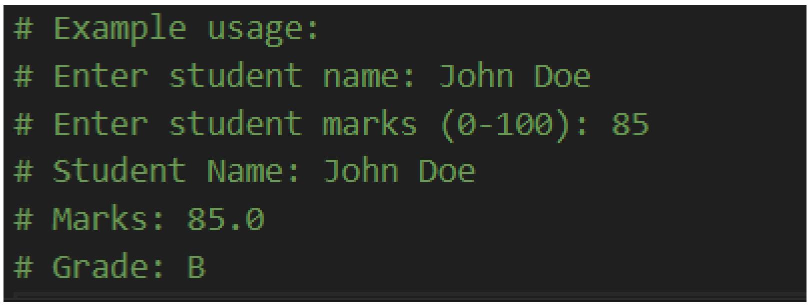
• Well-structured UI code with accurate calculations and clear output display.

Output :

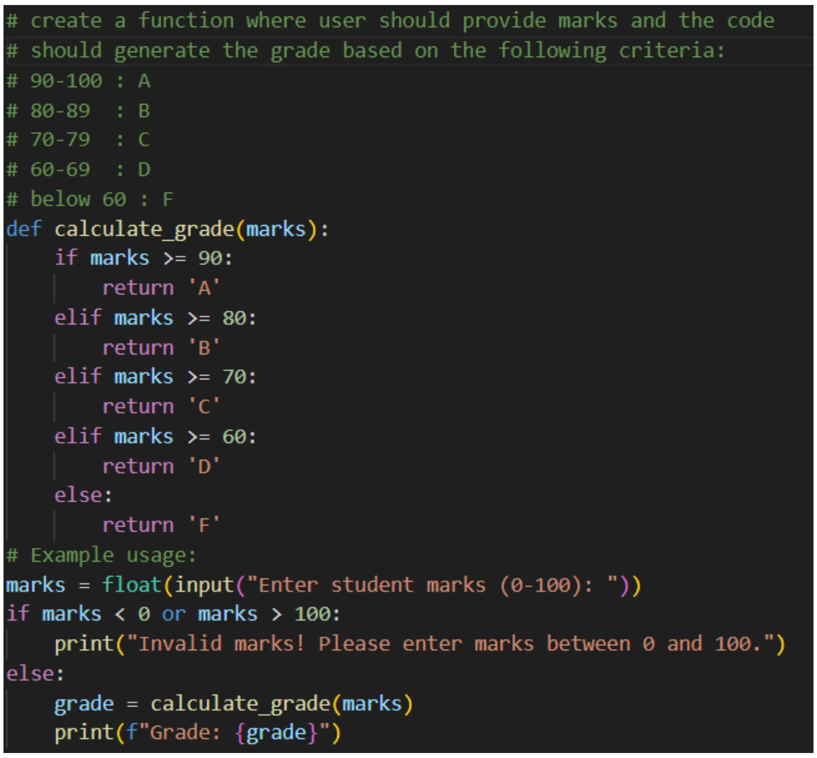
Code :



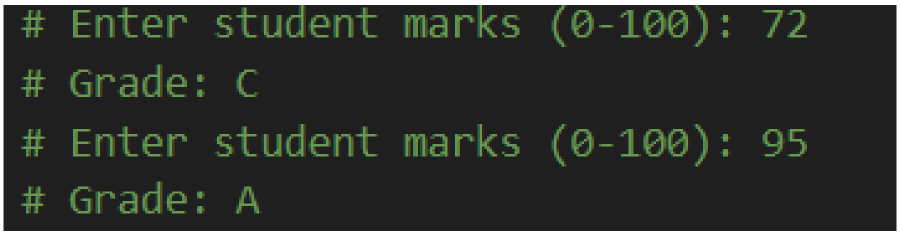
```python
#create a user interface for a student grading system
def calculate_grade(marks):
    if marks >= 90:
        return 'A'
    elif marks >= 80:
        return 'B'
    elif marks >= 70:
        return 'C'
    elif marks >= 60:
        return 'D'
    else:
        return 'F'
def main():
    name = input("Enter student name: ")
    marks = float(input("Enter student marks (0-100): "))
    if marks < 0 or marks > 100:
        print("Invalid marks! Please enter marks between 0 and 100.")
        return
    grade = calculate_grade(marks)
    print(f"Student Name: {name}")
    print(f"Marks: {marks}")
    print(f"Grade: {grade}")
if __name__ == "__main__":
    main()
```



```
# Example usage:
# Enter student name: John Doe
# Enter student marks (0-100): 85
# Student Name: John Doe
# Marks: 85.0
# Grade: B
```

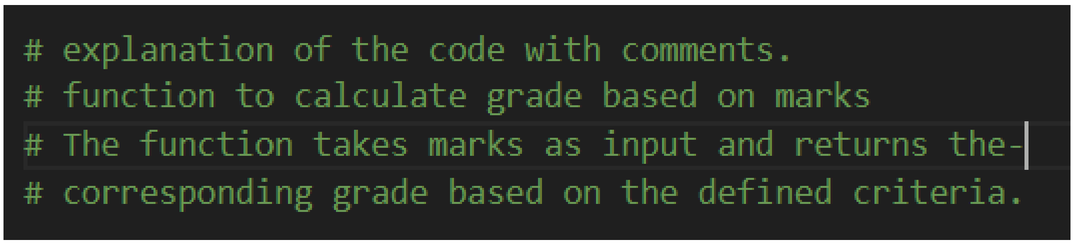


```python
# create a function where user should provide marks and the code
# should generate the grade based on the following criteria:
# 90-100 : A
# 80-89  : B
# 70-79  : C
# 60-69  : D
# below 60 : F
def calculate_grade(marks):
    if marks >= 90:
        return 'A'
    elif marks >= 80:
        return 'B'
    elif marks >= 70:
        return 'C'
    elif marks >= 60:
        return 'D'
    else:
        return 'F'
# Example usage:
marks = float(input("Enter student marks (0-100): "))
if marks < 0 or marks > 100:
    print("Invalid marks! Please enter marks between 0 and 100.")
else:
    grade = calculate_grade(marks)
    print(f"Grade: {grade}")
```



```
# Enter student marks (0-100): 72
# Grade: C
# Enter student marks (0-100): 95
# Grade: A
```

Explanation :


```
# explanation of the code with comments.
# function to calculate grade based on marks
# The function takes marks as input and returns the-
# corresponding grade based on the defined criteria.
```

Final Observation:

With an unclear prompt, the AI generated a vague and poorly defined interface with little practical value. As the requirements became more specific, the design grew more purposeful and aligned with the intended functionality. Once complete instructions were provided—including inputs, calculations, grading rules, and output display—the AI produced a structured and functional user interface that correctly computed totals, percentages, and grades while presenting the results clearly. This demonstrates that precise guidance directly improves both the structure and usability of the UI.

Task Description-5

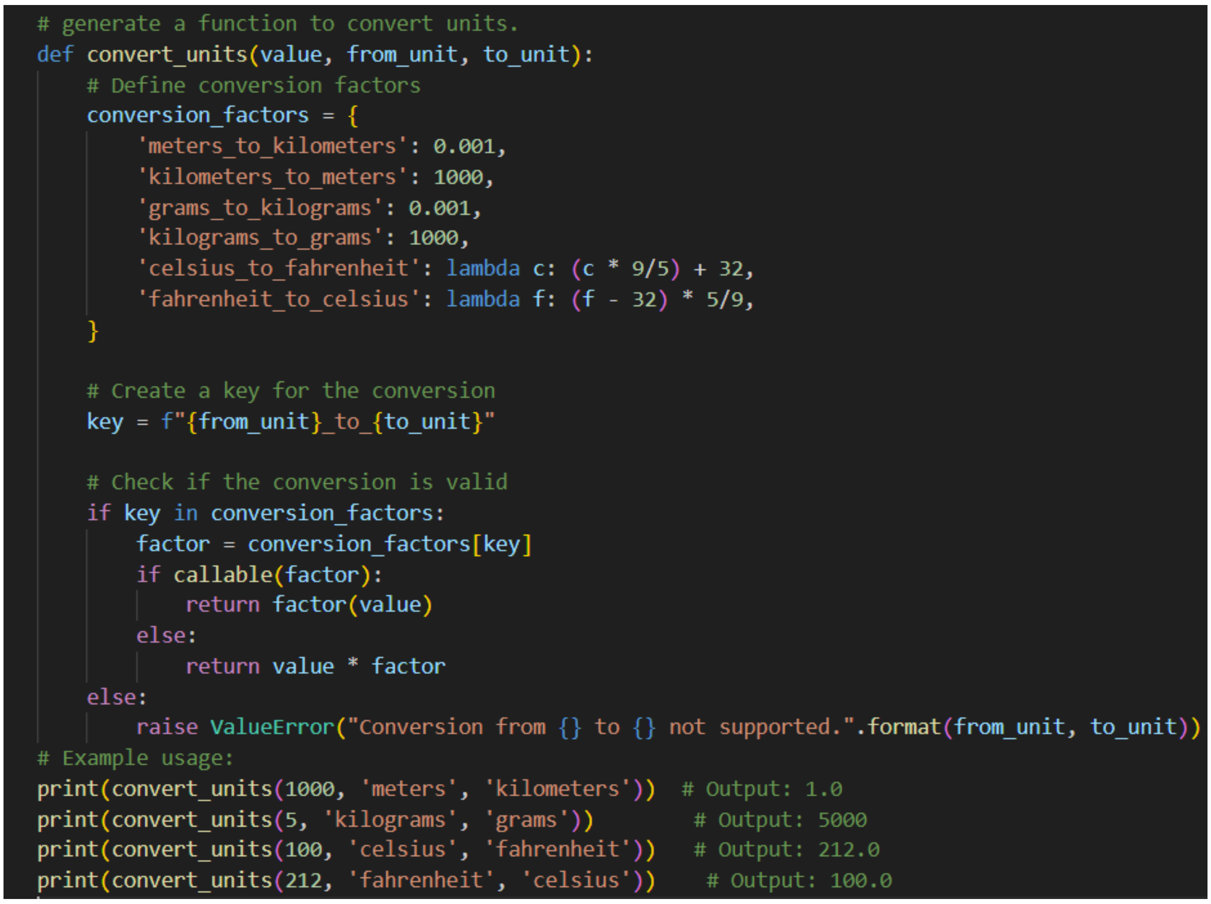
• Analyzing Prompt Specificity in Unit Conversion Functions: Improving a Unit Conversion Function (Kilometers to Miles and Miles to Kilometers) Using Clear Instructions.

Expected Output-5

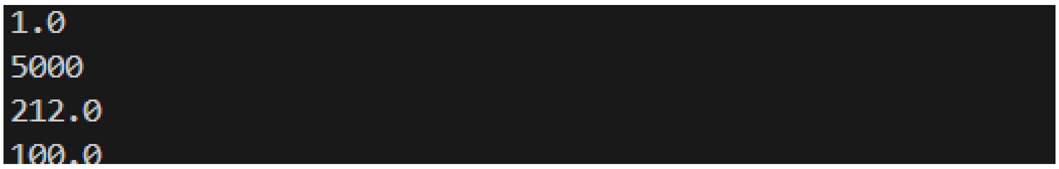
• Analysis of code quality and accuracy differences across multiple prompt variations.

OUTPUT :

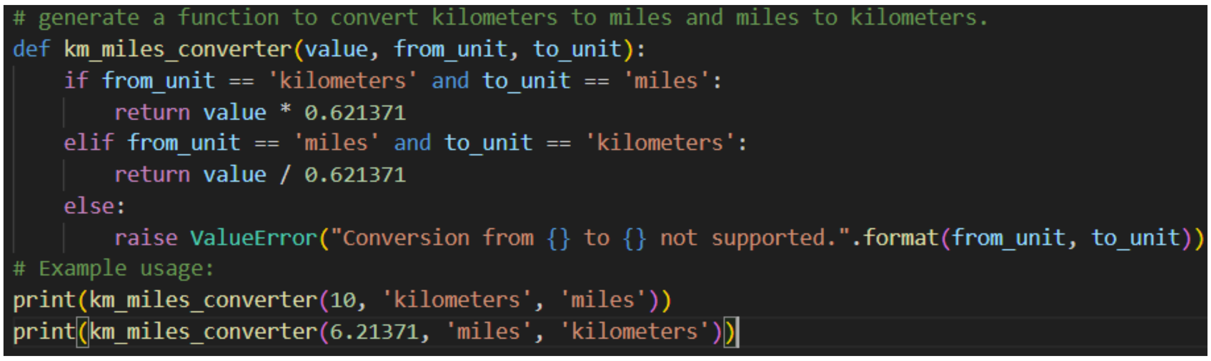
Code :



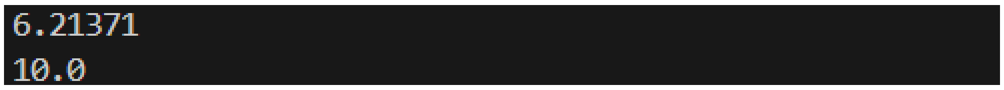
```python
# generate a function to convert units.
def convert_units(value, from_unit, to_unit):
    # Define conversion factors
    conversion_factors = {
        'meters_to_kilometers': 0.001,
        'kilometers_to_meters': 1000,
        'grams_to_kilograms': 0.001,
        'kilograms_to_grams': 1000,
        'celsius_to_fahrenheit': lambda c: (c * 9/5) + 32,
        'fahrenheit_to_celsius': lambda f: (f - 32) * 5/9,
    }

    # Create a key for the conversion
    key = f"{from_unit}_to_{to_unit}"

    # Check if the conversion is valid
    if key in conversion_factors:
        factor = conversion_factors[key]
        if callable(factor):
            return factor(value)
        else:
            return value * factor
    else:
        raise ValueError("Conversion from {} to {} not supported.".format(from_unit, to_unit))
# Example usage:
print(convert_units(1000, 'meters', 'kilometers'))  # Output: 1.0
print(convert_units(5, 'kilograms', 'grams'))        # Output: 5000
print(convert_units(100, 'celsius', 'fahrenheit'))   # Output: 212.0
print(convert_units(212, 'fahrenheit', 'celsius'))    # Output: 100.0
```



```
1.0
5000
212.0
100.0
```

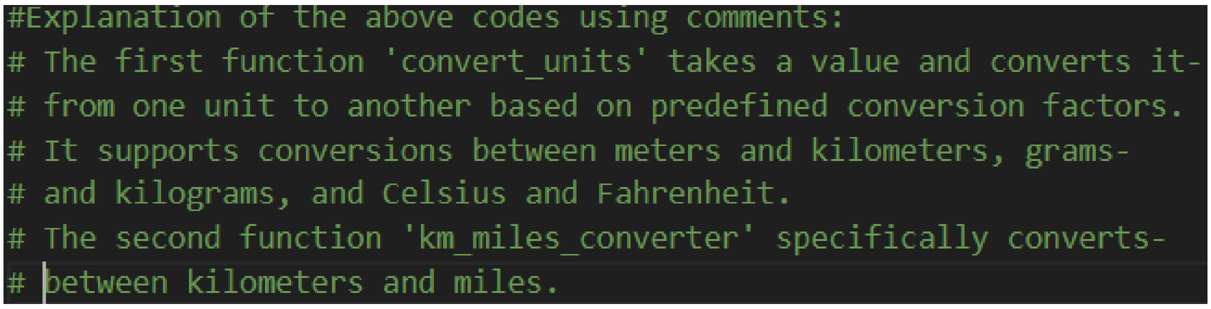


```python
# generate a function to convert kilometers to miles and miles to kilometers.
def km_miles_converter(value, from_unit, to_unit):
    if from_unit == 'kilometers' and to_unit == 'miles':
        return value * 0.621371
    elif from_unit == 'miles' and to_unit == 'kilometers':
        return value / 0.621371
    else:
        raise ValueError("Conversion from {} to {} not supported.".format(from_unit, to_unit))
# Example usage:
print(km_miles_converter(10, 'kilometers', 'miles'))
print(km_miles_converter(6.21371, 'miles', 'kilometers'))
```



```
6.21371
10.0
```

Explanation :



```
#Explanation of the above codes using comments:
# The first function 'convert_units' takes a value and converts it-
# from one unit to another based on predefined conversion factors.
# It supports conversions between meters and kilometers, grams-
# and kilograms, and Celsius and Fahrenheit.
# The second function 'km_miles_converter' specifically converts-
# between kilometers and miles.
```

Final Observation:

With a vague prompt, the AI produced generic and unfocused conversion code that lacked clear purpose. Once the specific type of conversion was defined, the output improved to a simple one-way converter, but the logic remained limited. After adding detailed instructions, formulas, and validation rules, the AI generated an accurate, well-structured, and reusable unit conversion function. This clearly shows that greater prompt specificity leads to higher code quality, accuracy, and reliability.