AI Assistant Coding
Assignment-2

Name of Student	: Nikhil N
Enrollment No.	: 2303A52477
Batch	: 38

Task 1: Cleaning Sensor Data

❖ **Scenario:**

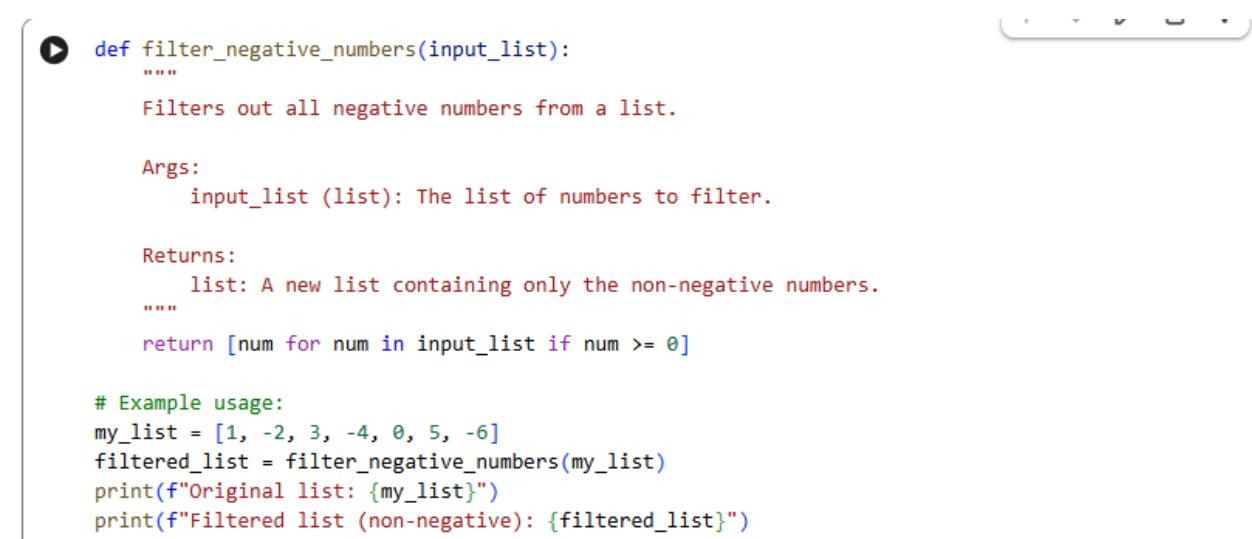❖ **You are cleaning IoT sensor data where negative values are invalid.**

❖ **Task:**

Use Gemini in Colab to generate a function that filters out all negative numbers from a list.

Prompt:

give me a function that filters out all negative numbers from a list

Code:

```python
def filter_negative_numbers(input_list):
    """
    Filters out all negative numbers from a list.

    Args:
        input_list (list): The list of numbers to filter.

    Returns:
        list: A new list containing only the non-negative numbers.
    """
    return [num for num in input_list if num >= 0]

# Example usage:
my_list = [1, -2, 3, -4, 0, 5, -6]
filtered_list = filter_negative_numbers(my_list)
print(f"Original list: {my_list}")
print(f"Filtered list (non-negative): {filtered_list}")
```

Output:

```
Original list: [1, -2, 3, -4, 0, 5, -6]
Filtered list (non-negative): [1, 3, 0, 5]
```

Justification:

It provides a well-defined function to filter negative numbers, includes docstrings for readability, and demonstrates its usage with an example, making it immediately useful and easy to understand.

Task 2: String Character Analysis

❖ **Scenario:**

You are building a text-analysis feature.

❖ **Task:**

Use Gemini to generate a Python function that counts vowels, consonants, and digits in a string.

Prompt:

generate a function that counts vowels, consonants and digits in a string

Code:

```python
def count_chars(input_string):
    """
    Counts the number of vowels, consonants, and digits in a string.

    Args:
        input_string (str): The string to analyze.

    Returns:
        dict: A dictionary containing the counts of vowels, consonants, and digits.
    """
    vowels = 0
    consonants = 0
    digits = 0

    # Define sets for quicker lookup
    vowel_set = {'a', 'e', 'i', 'o', 'u'}

    for char in input_string:
        char_lower = char.lower()
        if char_lower.isalpha():  # Check if it's an alphabet
            if char_lower in vowel_set:
                vowels += 1
            else:
                consonants += 1
        elif char_lower.isdigit():  # Check if it's a digit
            digits += 1

    return {
        "vowels": vowels,
        "consonants": consonants,
        "digits": digits
    }

# Example usage:
my_string = "Hello World 123!"
counts = count_chars(my_string)

print(f"Original string: '{my_string}'")
print(f"Vowels: {counts['vowels']}")
print(f"Consonants: {counts['consonants']}")
print(f"Digits: {counts['digits']}")
```

Output:

```
Original string: 'Hello World 123!'
Vowels: 3
Consonants: 7
Digits: 3
```

Justification:

the **count_chars** function is essential for string analysis, offering a straightforward method to categorize and count characters within a given text. This capability is particularly useful for tasks like data validation, linguistic analysis, or basic text processing where understanding the composition of a string (how many vowels, consonants, or digits it contains) is crucial. It provides clear, structured output, making it easy to integrate into larger applications or data pipelines..

Task 3: Palindrome Check – Tool Comparison

❖ **Scenario:**

You must decide which AI tool is clearer for string logic.

❖ **Task:**

Generate a palindrome-checking function using Gemini and Copilot, then compare the results.

Prompt:

generate a palindrome-checking function

Code:

Google Colab:

```python
import re

def is_palindrome(s):
    """
    Checks if a given string is a palindrome, handling edge cases.

    A palindrome is a word, phrase, number, or other sequence of characters
    which reads the same backward as forward, ignoring punctuation, case,
    and spacing.

    Args:
        s (str): The input string to check.

    Returns:
        bool: True if the string is a palindrome, False otherwise.
    """
    # Handle None or non-string input gracefully
    if not isinstance(s, str):
        return False

    # 1. Convert to lowercase
    s_lower = s.lower()

    # 2. Remove non-alphanumeric characters using a regular expression
    #    (only keep letters and digits)
    cleaned_s = re.sub(r'[^a-z0-9]', '', s_lower)

    # 3. Check if the cleaned string is equal to its reverse
    return cleaned_s == cleaned_s[::-1]

# Example usage with various edge cases:
print(f"'racecar' is a palindrome: {is_palindrome('racecar')}")                    # Basic palindrome
print(f"'RaceCar' is a palindrome: {is_palindrome('RaceCar')}")                     # Case-insensitive
print(f"'A man, a plan, a canal: Panama' is a palindrome: {is_palindrome('A man, a plan, a canal: Panama')}") #
print(f"'hello' is a palindrome: {is_palindrome('hello')}")                   # Not a palindrome
print(f"'' is a palindrome: {is_palindrome('')}")                   # Empty string
print(f"'a' is a palindrome: {is_palindrome('a')}")                    # Single character
print(f"'madam i\'m adam' is a palindrome: {is_palindrome('madam i\'m adam')}") # More complex phrase
print(f"'12321' is a palindrome: {is_palindrome('12321')}")                 # Digits
print(f"None is a palindrome: {is_palindrome(None)}")                 # Non-string input
print(f"12321 (int) is a palindrome: {is_palindrome(12321)}")                 # Non-string input
```

Cursor AI:

```python
"""
Palindrome checking function with comprehensive edge case handling.
"""


def is_palindrome(s):
    """
    Check if a string is a palindrome, handling all edge cases.

    Edge cases handled:
    - None/empty strings
    - Single character strings
    - Whitespace-only strings
    - Case-insensitive comparison
    - Non-alphanumeric characters (punctuation, spaces)
    - Unicode characters
    - Numbers (as strings)
    - Mixed content

    Args:
        s: Input to check (string, int, or None)

    Returns:
        bool: True if palindrome, False otherwise

    Examples:
        >>> is_palindrome("racecar")
        True
        >>> is_palindrome("A man a plan a canal Panama")
        True
        >>> is_palindrome("")
        True
        >>> is_palindrome("a")
        True
        >>> is_palindrome(None)
        False
    """
    # Handle None
    if s is None:
        return False

    # Convert to string if not already (handles integers, floats, etc.)
    s = str(s)

    # Empty string is considered a palindrome
    if len(s) == 0:
        return True

    # Single character is always a palindrome
    if len(s) == 1:
        return True

    # Normalize: convert to lowercase and remove non-alphanumeric characters
    # This handles:
    # - Case differences (RaceCar vs racecar)
    # - Punctuation and spaces (A man, a plan, a canal: Panama)
    # - Special characters
    normalized = ''.join(char.lower() for char in s if char.isalnum())

    # After normalization, check if empty (e.g., only punctuation/spaces)
    if len(normalized) == 0:
        return True  # String with only non-alphanumeric chars is palindrome

    # Check if normalized string is palindrome
    left = 0
    right = len(normalized) - 1

    while left < right:
        if normalized[left] != normalized[right]:
            return False
        left += 1
        right -= 1

    return True


# Example usage and test cases
if __name__ == "__main__":
    # Test cases
    test_cases = [
        ("racecar", True),
        ("hello", False),
        ("", True),
        ("a", True),
        ("RaceCar", True),
        ("A man a plan a canal Panama", True),
        ("Was it a car or a cat I saw?", True),
        ("12321", True),
        ("12345", False),
        (12321, True),
        (None, False),
    ]

    print("Testing is_palindrome() function:")
    print("=" * 60)

    for test_input, expected in test_cases:
        result = is_palindrome(test_input)
        status = "PASS" if result == expected else "FAIL"
        print(f"{status}: is_palindrome({repr(test_input)}) = {result} (expected {expected})")

    print("=" * 60)
```

Output:

Google Colab:

```
'racecar' is a palindrome: True
'RaceCar' is a palindrome: True
'A man, a plan, a canal: Panama' is a palindrome: True
'hello' is a palindrome: False
'' is a palindrome: True
'a' is a palindrome: True
'madam i'm adam' is a palindrome: True
'12321' is a palindrome: True
None is a palindrome: False
12321 (int) is a palindrome: False
```

Cursor AI:

```
PS C:\Users\nikhi\Projects\AI Assistant Coding> & C:/Users/nikhi/AppData/Local/Programs/Python/Python314/python.exe "c:/Users/nikhi/Projects/AI Assistant Coding/p
alindrome.py"
Testing is_palindrome() function:
================================================================
PASS: is_palindrome('racecar') = True (expected True)
PASS: is_palindrome('hello') = False (expected False)
PASS: is_palindrome('') = True (expected True)
PASS: is_palindrome('a') = True (expected True)
PASS: is_palindrome('RaceCar') = True (expected True)
PASS: is_palindrome('A man a plan a canal Panama') = True (expected True)
PASS: is_palindrome('Was it a car or a cat I saw?') = True (expected True)
PASS: is_palindrome('12321') = True (expected True)
PASS: is_palindrome('12345') = False (expected False)
PASS: is_palindrome(12321) = True (expected True)
PASS: is_palindrome(None) = False (expected False)
================================================================
```

Comaparison:

Feature	Google Colab (Gemini AI) (Regex + Slice)	Cursor AI (Two Pointers)
Code size	Very short	Slightly longer
Readability	Clean & compact	Clear, step-by-step
Palindrome check	s == s[::-1]	Left–right pointers
Edge cases	Basic	Handles more cases
Unicode support	Limited	Better (isalnum)
Best for	Interviews, quick tasks	Learning, production

Justification:

- Google Colab(Gemini AI) feels great when you want something quick and elegant—it's easy to read and perfect for interviews or short problems.

- Cursor AI takes a more careful, real-world approach, clearly showing each step and handling more edge cases.
  If speed and simplicity matter, go with Gemini AI.
  If clarity, learning, and robustness matter more, Cursor AI is the better choice.

Task 4: Code Explanation Using AI

❖ Scenario:

You are reviewing unfamiliar code written by another developer.

❖ Task:

Ask Gemini to explain a Python function (prime check OR palindrome check) line by line.

❖ Expected Output:

➢ Code snippet

➢ AI explanation

➢ Student comments on understanding

Prompt:

generate prime check function and explain the code line by line

Code:

```python
def is_prime(n):
    """
    Checks if a given number is a prime number.

    A prime number is a natural number greater than 1 that has no positive
    divisors other than 1 and itself.

    Args:
        n (int): The integer to check for primality.

    Returns:
        bool: True if the number is prime, False otherwise.
    """
    # Prime numbers are natural numbers greater than 1.
    # Numbers less than or equal to 1 are not prime.
    if n <= 1:
        return False
    # 2 is the only even prime number.
    elif n == 2:
        return True
    # All other even numbers are not prime.
    elif n % 2 == 0:
        return False
    # Check for divisors from 3 up to the square root of n,
    # incrementing by 2 (only checking odd numbers).
    # If n has a divisor, it must have one less than or equal to its square root.
    i = 3
    while i * i <= n:
        if n % i == 0:
            return False
        i += 2
    # If no divisors are found, the number is prime.
    return True


# Example usage:
print(f"Is 1 prime? {is_prime(1)}")    # Expected: False
print(f"Is 2 prime? {is_prime(2)}")    # Expected: True
print(f"Is 7 prime? {is_prime(7)}")    # Expected: True
print(f"Is 10 prime? {is_prime(10)}") # Expected: False
print(f"Is 13 prime? {is_prime(13)}") # Expected: True
print(f"Is 25 prime? {is_prime(25)}") # Expected: False
print(f"Is 97 prime? {is_prime(97)}") # Expected: True
print(f"Is 0 prime? {is_prime(0)}")    # Expected: False
```

Output:

```
...  Is 1 prime? False
     Is 2 prime? True
     Is 7 prime? True
     Is 10 prime? False
     Is 13 prime? True
     Is 25 prime? False
     Is 97 prime? True
     Is 0 prime? False
```

AI Explanation(Line-By-Line):

Let's break down the `is_prime` function line by line:

```
def is_prime(n):
```

- **`def is_prime(n):`:** This line defines a function named `is_prime` that takes one argument, `n`, which is the number we want to check for primality.

```
"""
Checks if a given number is a prime number.

A prime number is a natural number greater than 1 that has no
```

```
positive
    divisors other than 1 and itself.

Args:
    n (int): The integer to check for primality.

Returns:
    bool: True if the number is prime, False otherwise.
"""
```

- This is a **docstring**, providing a brief explanation of what the function does, what arguments it takes, and what it returns. It's good practice for code documentation.

```
if n <= 1:
    return False
```

- **`if n <= 1:`:** Prime numbers are defined as natural numbers greater than 1. This condition checks if the input `n` is less than or equal to 1. If it is, it's not prime.
- **`return False`:** If `n` is 1 or less, the function immediately returns `False` because such numbers are not prime.

```
elif n == 2:
    return True
```

- **`elif n == 2:`:** The number 2 is a special case. It's the smallest and only even prime number.
- **`return True`:** If `n` is 2, the function immediately returns `True` because 2 is prime.

```
elif n % 2 == 0:
    return False
```

- **`elif n % 2 == 0:`:** This condition checks if `n` is an even number (by checking if it has a remainder of 0 when divided by 2).
- **`return False`:** If `n` is an even number greater than 2 (since 2 was handled above), it cannot be prime, so the function returns `False`.

```
i = 3
while i * i <= n:
    if n % i == 0:
        return False
    i += 2
```

- **`i = 3`:** We initialize a variable `i` to 3. This `i` will be used to check for odd divisors.
- **`while i * i <= n:`:** This is an optimization. If a number `n` has a divisor greater than

its square root, it must also have a divisor smaller than its square root. So, we only need to check for divisors up to the square root of `n`. Checking `i * i <= n` is mathematically equivalent and avoids calculating `sqrt(n)`.

- **if n % i == 0::** Inside the loop, this checks if `n` is perfectly divisible by `i` (i.e., `i` is a divisor of `n`).

- **return False**: If a divisor is found, `n` is not prime, and the function returns `False`.
- **i += 2**: We increment `i` by 2 in each iteration. This is another optimization: since we've already handled even numbers, we only need to check odd potential divisors (3, 5, 7, etc.).

```
return True
```

- **return True**: If the loop completes without finding any divisors (meaning `n` was not divisible by any odd number up to its square root), then the number `n` is prime, and the function returns `True`.

Understanding/Outcome:

I now understand the is_prime function much better! The explanation clearly broke down the logic, especially why numbers less than or equal to 1 are not prime, why 2 is a special case, and the optimization of checking divisors only up to the square root and incrementing by 2. This makes the code efficient and easy to follow.