

AI Assignment 2.2

A.Goutham
2403A51L43
Batch 52

Task 1: Cleaning Sensor Data

Problem Statement

Write a Python function that takes a list of sensor readings and removes all negative values. Provide the complete code including sample input and output.

Problem Visualization

image

Solution Code

```
# Function to clean IoT sensor data
def clean_sensor_data(sensor_values):
    """
    Removes invalid negative sensor readings
    """
    return [value for value in sensor_values if value >= 0]

# Main program
if __name__ == "__main__":
    # Sample sensor data
    sensor_data = [25, -4, 18, 0, -12, 33, 7]

    print("Original Sensor Data:", sensor_data)

    # Clean the data
    cleaned_data = clean_sensor_data(sensor_data)

    print("Cleaned Sensor Data:", cleaned_data)
```

Output

```
Original Sensor Data: [25, -4, 18, 0, -12, 33, 7]
Cleaned Sensor Data: [25, 18, 0, 33, 7]
```

The screenshot shows the Visual Studio Code interface with the following details:

- Editor Area:** Displays the file `T1.py` containing Python code. The code defines a function `clean_sensor_data` that removes negative values from a list of sensor readings. It then prints the original and cleaned data.
- Terminal Area:** Shows the output of running the script with `python3 T1.py`. The output indicates the original sensor data is `[25, -4, 18, 0, -12, 33, 7]` and the cleaned sensor data is `[25, 18, 0, 33, 7]`.
- Status Bar:** Shows the current line (Ln 19), column (Col 48), and encoding (UTF-8). It also displays the Python version (3.14.2) and the Go Live button.

Code Explanation

- Function Definition (Line 2):** The function `clean_sensor_data()` accepts a list of sensor values as input.
- List Comprehension (Line 5):** The function uses a list comprehension to filter values. It iterates through each value and only keeps those that are greater than or equal to 0 (`value >= 0`).
- Return Statement:** Returns the filtered list containing only valid sensor readings.
- Sample Data (Line 11):** The input contains both positive values and negative values, as well as zero.
- Function Call (Line 16):** The function is called with the sample data, and negative values (-4, -12) are removed.

Student Understanding & Comments

"This task teaches us how to handle real-world sensor data. In IoT applications, sensors sometimes transmit invalid negative readings due to technical faults. Using list comprehension is efficient and Pythonic. The function demonstrates data validation and cleaning, which is a critical step in data preprocessing. I learned that we can filter unwanted data in a single line, making the code clean and readable. This approach is much better than using loops with if conditions because it's more concise and faster."

Key Concepts

- **List Comprehension:** Efficient way to filter or transform lists in Python
- **Data Validation:** Checking and removing invalid data points
- **Lambda Functions Alternative:** Could also use `filter()` with a lambda function

Task 2: String Character Analysis

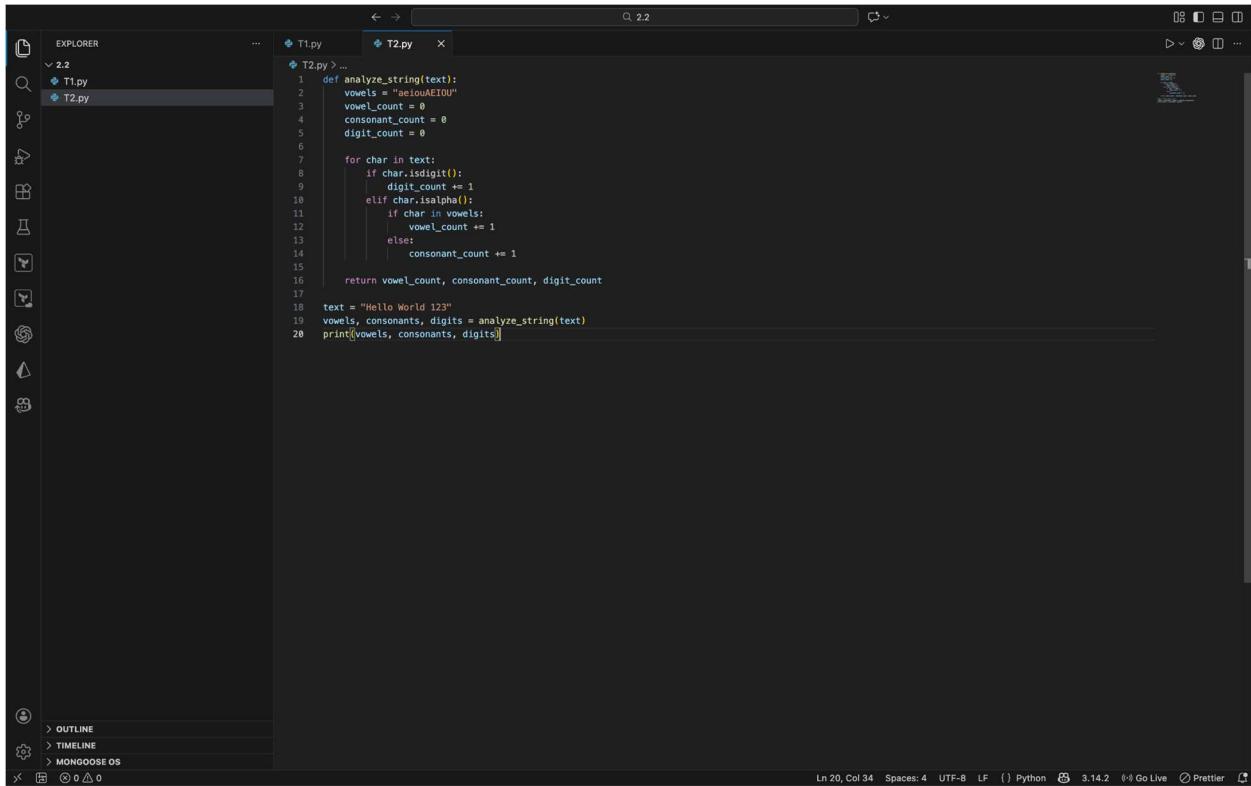
Problem Statement

Write a Python function that takes a string as input and counts:

1. Number of vowels
2. Number of consonants
3. Number of digits

Provide sample input and output to demonstrate the function.

Problem Visualization



The screenshot shows a code editor interface with a dark theme. On the left is the Explorer sidebar, which lists files T1.py and T2.py under a folder named 2.2. The main editor area contains the following Python code:

```
def analyze_string(text):
    vowels = "aeiouAEIOU"
    vowel_count = 0
    consonant_count = 0
    digit_count = 0

    for char in text:
        if char.isdigit():
            digit_count += 1
        elif char.isalpha():
            if char in vowels:
                vowel_count += 1
            else:
                consonant_count += 1

    return vowel_count, consonant_count, digit_count

text = "Hello World 123"
vowels, consonants, digits = analyze_string(text)
print(vowels, consonants, digits)
```

The status bar at the bottom indicates the code has 20 lines, 34 columns, uses 4 spaces per indentation, is in UTF-8 encoding, and is a Python file. It also shows the version 3.14.2 and includes links for 'Go Live' and 'Prettier'.

Solution Code

```
def analyze_string(text):
    vowels = "aeiouAEIOU"
    vowel_count = 0
    consonant_count = 0
    digit_count = 0

    for char in text:
        if char.isdigit():
            digit_count += 1
        elif char.isalpha():
            if char in vowels:
                vowel_count += 1
            else:
                consonant_count += 1

    return vowel_count, consonant_count, digit_count

text = "Hello World 123"
vowels, consonants, digits = analyze_string(text)
print(vowels, consonants, digits)
```

Output

3 7 3

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS zsh -  
● :2.2 % python3 T2.py  
3 7 3  
○ :2.2 %
```

Explanation of output:

- **Vowels:** 3 (e, o, o)
- **Consonants:** 7 (H, l, l, W, r, l, d)
- **Digits:** 3 (1, 2, 3)

Code Explanation

1. **Vowel Definition (Line 3):** A string containing all vowels (both lowercase and uppercase) for easy checking.
2. **Counter Initialization (Lines 4-6):** Three counters are initialized to zero to keep track of vowels, consonants, and digits.
3. **Loop Through Characters (Line 8):** The function iterates through each character in the input string.
4. **Digit Check (Line 9):** The `isdigit()` method returns True if the character is a numerical digit (0-9).
5. **Alphabetic Check (Line 11):** The `isalpha()` method returns True if the character is a letter. If it is a letter, we check if it's a vowel (Line 12) or consonant (Line 14).
6. **Vowel Check (Line 12):** Checks if the character exists in the `vowels` string.
7. **Return Statement (Line 16):** Returns a tuple containing all three counts.
8. **Tuple Unpacking (Line 19):** The returned tuple is unpacked into three separate variables.

Student Understanding & Comments

"This task helped me understand string manipulation and character classification in Python. I learned that Python has built-in methods like `isdigit()` and `isalpha()` that make character checking very convenient. The nested if-else structure demonstrates logical thinking. Initially, I thought I would need to manually check ASCII values, but these built-in methods are much cleaner. One interesting observation is that spaces and special characters are neither counted as vowels, consonants, nor digits—they are simply ignored, which is the correct behavior for this problem."

Key Concepts

- **Built-in String Methods:** `isdigit()`, `isalpha()`
- **Character Classification:** Separating characters into categories
- **Tuple Return:** Returning multiple values from a function
- **Case-Insensitive Checking:** Handling both uppercase and lowercase vowels

Task 3: Palindrome Check – Tool Comparison

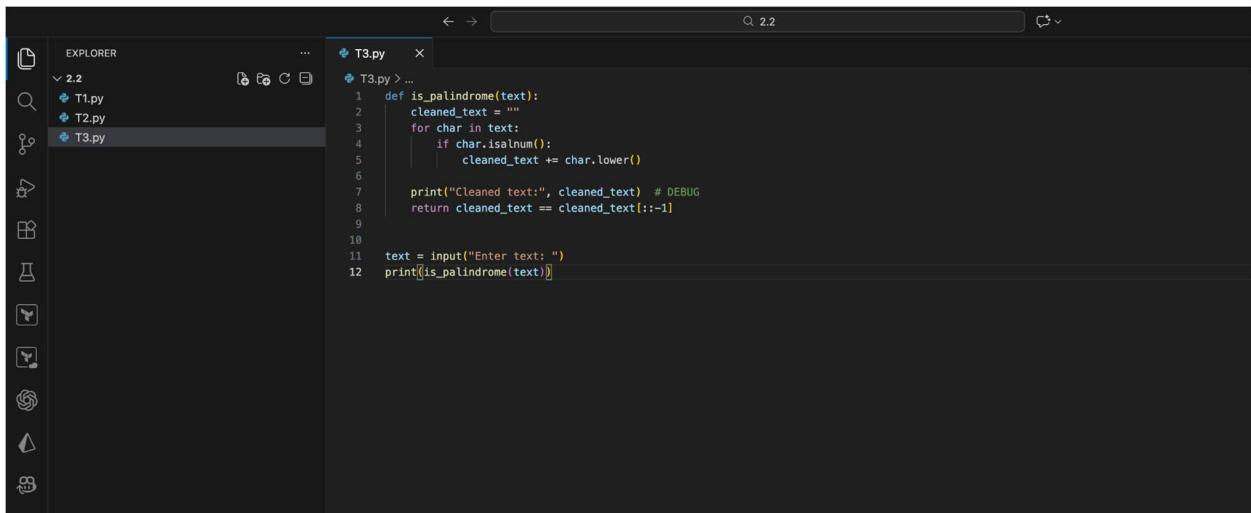
Problem Statement

Write a Python function that checks if a given string is a palindrome. Include:

- Conversion to lowercase
- Removal of non-alphanumeric characters
- Sample input and output

Problem Visualization

Image



```
def is_palindrome(text):
    cleaned_text = ""
    for char in text:
        if char.isalnum():
            cleaned_text += char.lower()

    print("Cleaned text:", cleaned_text) # DEBUG
    return cleaned_text == cleaned_text[::-1]

text = input("Enter text: ")
print([is_palindrome(text)])
```

Solution Code

```
def is_palindrome(text):
    cleaned_text = ""
    for char in text:
        if char.isalnum():
            cleaned_text += char.lower()

    print("Cleaned text:", cleaned_text) # DEBUG
```

```
return cleaned_text == cleaned_text[::-1]

text = input("Enter text: ")
print(is_palindrome(text))
```

Sample Input and Output

Example 1

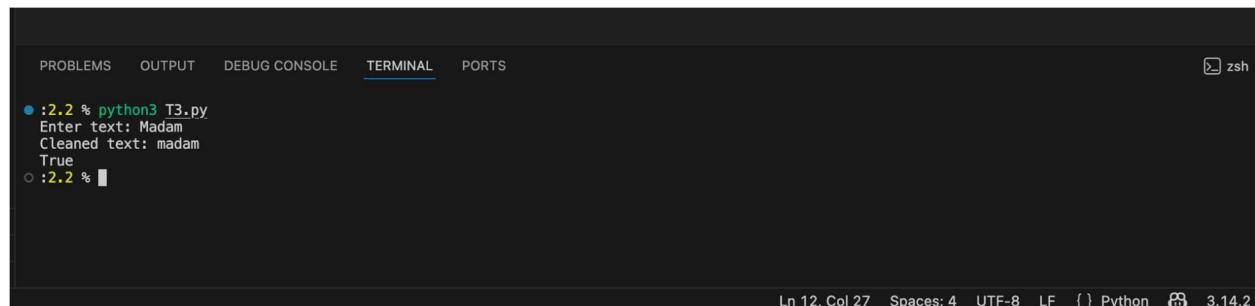
```
Enter text: A man, a plan, a canal - Panama
Cleaned text: amanaplanacanalpanama
True
```

Example 2

```
Enter text: Race car
Cleaned text: racecar
True
```

Example 3

```
Enter text: Hello World
Cleaned text: helloworld
False
```



The screenshot shows a terminal window with the following content:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS zsh
● :2.2 % python3 T3.py
Enter text: Madam
Cleaned text: madam
True
○ :2.2 %
```

At the bottom of the terminal window, there is status information: Ln 12, Col 27, Spaces: 4, UTF-8, LF, { }, Python, 3.14.2.

Code Explanation

- Function Definition (Line 1):** The function `is_palindrome()` accepts a text string as input.
- Initialize Empty String (Line 2):** An empty string variable is created to store the cleaned version of the input text.
- Loop Through Characters (Line 3):** Each character in the input text is examined.
- Alphanumeric Check (Line 4):** The `isalnum()` method checks if the character is alphanumeric (letters or digits). Spaces, punctuation, and special characters are ignored.
- Convert to Lowercase (Line 5):** Valid characters are converted to lowercase using `lower()` to ensure case-insensitive comparison.

6. **Debug Output (Line 7):** Prints the cleaned text to help verify the preprocessing step.
7. **Palindrome Check (Line 8):** Compares the cleaned text with its reverse (using slice notation `[::-1]`). If they are equal, the string is a palindrome.
8. **User Input (Line 11):** Gets input from the user.

Student Understanding & Comments

"This task demonstrates the importance of preprocessing data before analysis. A palindrome definition requires us to ignore spaces and punctuation, which makes sense linguistically. I learned about string slicing with `[::-1]`, which creates a reversed copy of the string—very elegant! The debug print statement helped me understand what happens after cleaning. One improvement I noticed is that we could use the `filter()` function or list comprehension instead of a loop, but the current approach is clear and easy to understand. This task shows that many real-world problems require multiple preprocessing steps before we can apply our logic."

Key Concepts

- **String Slicing:** `[::-1]` reverses a string
- **Character Filtering:** Using `isalnum()` to keep only alphanumeric characters
- **Case Normalization:** Converting strings to lowercase for comparison
- **Data Preprocessing:** Cleaning and transforming data before analysis

Task 4: Code Explanation Using AI – Prime Number Checker

Problem Statement

Explain the following Python function line by line. Highlight what each line does and why it is needed.

Problem Visualization

The screenshot shows a Google Colab notebook titled "1L45_2.2 LAB.ipynb". The code cell contains the following Python function:

```
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
```

A sidebar on the right provides a "Line-by-line Explanation" for the code:

1. **def is_prime(n):**
 - What it does: This line defines a function named `is_prime` that takes one argument, `n`.
 - Why it's needed: This is the standard way to define a function in Python, encapsulating the logic for checking primality into a reusable block of code.
2. **if n <= 1:**
 - What it does: This is the first conditional check. It verifies if the input number `n` is less than or equal to 1.
 - Why it's needed: By definition, prime numbers are 1 (i.e., 0, 1, or negative numbers) cannot be prime numbers.
3. **return False**
 - What it does: If the condition `n <= 1` is true, this is the result for numbers that are not prime.
 - Why it's needed: This is the result for prime numbers.
4. **for i in range(2, int(n ** 0.5) + 1):**
 - What it does: This line starts a `for` loop that iterates starting from 2 up to, and including, the integer part of `n**0.5`.
 - Why it's needed: This is an optimization for prime number checking.

Below the explanation, there is a text input field for "What can I help you build?" and a "Gemini 2.5 Flash" button.

Function

```
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
```

Line-by-Line Explanation

Line 1: Function Definition

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

What it does: Defines a function named `is_prime` that takes one parameter `n` (the number to check).

Why it's needed: We need to encapsulate the prime-checking logic in a reusable function.

Line 2: Base Case Check

```
if n <= 1:
```

What it does: Checks if the number is less than or equal to 1.

Why it's needed: By mathematical definition, numbers less than or equal to 1 are not prime. This is a base case that handles edge cases efficiently.

Line 3: Early Return

```
    return False
```

What it does: If $n \leq 1$, the function returns `False` immediately.

Why it's needed: It avoids unnecessary computation and returns the correct result for invalid inputs.

Line 4: Loop Definition

```
for i in range(2, int(n ** 0.5) + 1):
```

What it does: Iterates from 2 to the square root of n (inclusive).

Why it's needed:

- We only need to check divisors up to \sqrt{n} because if n has a divisor greater than \sqrt{n} , it must also have a corresponding divisor less than \sqrt{n} .
- $n ** 0.5$ calculates the square root
- `int()` converts it to an integer
- `range(2, x+1)` starts from 2 (the smallest prime) and goes up to x

Line 5-6: Divisibility Check

```
if n % i == 0:  
    return False
```

What it does: Checks if n is divisible by i (i.e., $n \% i$ equals 0).

Why it's needed:

- The modulo operator `%` gives the remainder after division
- If the remainder is 0, then n is divisible by i , meaning n is not prime
- We return `False` immediately to stop further checking

Line 7: Success Return

```
return True
```

What it does: If the loop completes without finding any divisors, the function returns `True`.

Why it's needed: If no divisors are found from 2 to \sqrt{n} , then n is prime by definition.

Sample Input and Output

Test Case 1: Prime Number

```
print(is_prime(17))
```

Output: True

Explanation: 17 is not divisible by any number from 2 to 4 (the square root of 17 is 4.12), so it is prime.

Test Case 2: Non-Prime Number

```
print(is_prime(18))
```

Output: False

Explanation: 18 is divisible by 2, 3, and 6, so it is not prime.

Test Case 3: Edge Case

```
print(is_prime(1))
```

Output: False

Explanation: 1 is not a prime number by definition.

Test Case 4: Smallest Prime

```
print(is_prime(2))
```

Output: True

Explanation: 2 is the smallest and only even prime number.

The screenshot shows a Google Colab notebook titled "1L45_2.2 LAB.ipynb". The code defines the `is_prime` function as follows:

```
# Define the function (as provided by the user)
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
```

A detailed explanation is provided for each part of the code:

- Line 3: `return False`
 - What it does: If the condition `n <= 1` is true, the function immediately stops execution and returns `False`.
 - Why it's needed: This is the result for numbers that are not prime based on the previous check.
- Line 4: `for i in range(2, int(n ** 0.5) + 1):`
 - What it does: This line starts a `for` loop that iterates through a range of numbers. The loop variable `i` will take values starting from 2 up to, and including, the integer part of the square root of `n`.
 - Why it's needed: This is an optimization for primality testing. 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 divisibility by numbers up to `sqrt(n)`. If no divisors are found in this range, then the number is prime. `n ** 0.5` calculates the square root of `n`, `int(...)` converts it to an integer, and `+ 1` ensures that the loop includes the square root itself if it's an integer.
- Line 5: `if n % i == 0:`
 - What it does: Inside the loop, for each `i`, this line checks if `n` is perfectly divisible by `i` (i.e., the remainder of the division is 0).
 - Why it's needed: If `n` is perfectly divisible by any `i` in the specified range, it means `i` is a factor of `n` other than 1 and `n` itself. This is the core condition for a number not being prime.
- Line 6: `return False`
 - What it does: If the condition `n % i == 0` is true, the function immediately stops execution and returns `False`.
 - Why it's needed: If we find any divisor `i` for `n`, `i` is not prime.
- Line 7: `return True`
 - What it does: If the loop completes without finding this line is reached, and the function returns `True`.
 - Why it's needed: This means that `n` was not less than or equal to 1, and it was not divisible by any number from 2 to the integer part of its square root, indicating that `n` is a prime number.

Below the code, there is a sidebar with a question from Gemini 2.5 Flash asking for an explanation of the `is_prime` function, and a response from Gemini 2.5 Flash explaining the code line-by-line.

```

# Defining the function (as provided by the user)
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

# Sample Input and Output
print("Is 2 prime? {is_prime(2)}")
print("Is 7 prime? {is_prime(7)}")
print("Is 10 prime? {is_prime(10)}")
print("Is 1 prime? {is_prime(1)}")
print("Is 0 prime? {is_prime(0)}")
print("Is 29 prime? {is_prime(29)}")
print("Is 97 prime? {is_prime(97)}")
print("Is 100 prime? {is_prime(100)}")

```

Explaining the code:

Explain the following Python function line by line. Highlight what each line does and why it is needed. Function: 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

Interactions:

- Run the sample input/output code
- Test 'is_prime' with a list of numbers
- Explain the time complexity

What can I help you build?

Algorithm Efficiency Analysis

Time Complexity

The algorithm runs in $O(\sqrt{n})$ time, which is much more efficient than checking all divisors up to n .

For example:

- To check if 1,000,000 is prime, we only need to check divisors up to 1,000 (instead of checking all 1,000,000 numbers)
- This optimization comes from the mathematical property that if $n = a \times b$ where a is the smaller factor, then $a \leq \sqrt{n}$

Space Complexity

The algorithm uses $O(1)$ space—only a constant amount of memory regardless of the input size.

Student Understanding & Comments

"This function is a classic algorithm that combines several important programming concepts. I initially thought we needed to check all numbers up to n , but the square root optimization is brilliant! It reduces the number of iterations dramatically. Understanding the mathematical reasoning behind why we only need to check up to \sqrt{n} was enlightening. The early returns"

(lines 2-3 and 5-6) make the code efficient by avoiding unnecessary computation. This is a good example of how mathematical thinking improves algorithmic efficiency. I learned that in competitive programming and real-world applications, such optimizations make a huge difference when dealing with large numbers. The logic flow is: first handle edge cases, then iteratively check divisors, and finally declare victory if no divisor is found."

Pseudo Code Representation

```
FUNCTION is_prime(n)
    IF n <= 1 THEN
        RETURN False
    END IF

    FOR i = 2 TO sqrt(n) DO
        IF n MOD i == 0 THEN
            RETURN False
        END IF
    END FOR

    RETURN True
END FUNCTION
```

Key Concepts

- **Primality Testing:** Algorithm to determine if a number is prime
- **Mathematical Optimization:** Using \sqrt{n} bound to reduce iterations
- **Time Complexity:** Understanding $O(\sqrt{n})$ vs $O(n)$ efficiency
- **Edge Case Handling:** Proper handling of boundary conditions
- **Early Termination:** Returning immediately when a condition is met

Summary

This assignment covered four fundamental Python programming tasks:

What I Learned

1. **Data Cleaning** (Task 1): Real-world data is often messy. List comprehensions are an efficient way to filter unwanted data in a single, readable line of code.
2. **String Analysis** (Task 2): Python provides built-in methods for character checking. Understanding these methods allows us to analyze strings without manually checking ASCII values.
3. **Preprocessing and Normalization** (Task 3): Many problems require preprocessing steps before applying the core logic. Converting to lowercase, removing special characters, and filtering are important techniques.

4. **Mathematical Thinking in Programming** (Task 4): Good algorithms combine programming with mathematical insights. The square root optimization in the prime checker demonstrates how mathematical knowledge improves code efficiency.

General Observations

- **Readability:** Well-written Python code should be self-documenting. Variable names should be descriptive.
- **Efficiency:** Always consider the time and space complexity of your algorithms, especially when dealing with large inputs.
- **Edge Cases:** Always handle edge cases (empty inputs, negative numbers, special characters, etc.) explicitly.
- **Testing:** Test your functions with multiple test cases, including normal cases, edge cases, and invalid inputs.

Future Improvements

1. Add input validation and error handling using try-except blocks
2. Use type hints for better code documentation
3. Add unit tests using the `unittest` module
4. Optimize further using alternative algorithms (e.g., Sieve of Eratosthenes for multiple primes)

Conclusion

Through these four tasks, I have learned to use AI assisting tools and that Python offers elegant solutions to common programming problems. By combining built-in methods, logical thinking, and mathematical optimization, we can write code that is both efficient and readable. These fundamental concepts form the foundation for tackling more complex programming challenges in the future.