Strings, Sequences, and Sets

Bridging Programming Collections with Mathematical Set Theory

CS 5001 & CS 5002 Integrated Course

September 23, 2025

1 Introduction: Collections in Programming and Mathematics

This week we explore the fundamental concept of **collections** from two complementary perspectives: programming sequences (strings, lists, tuples) and mathematical sets. These concepts form the backbone of data organization in both computer science and discrete mathematics.

1.1 Why Study Collections Together?

Collections are everywhere in computing and mathematics:

- **Programming**: Strings store text, lists manage data, tuples group related items
- **Mathematics**: Sets define fundamental relationships and operations
- **Data Science**: Understanding both perspectives enables powerful data analysis
- **Algorithms**: Many efficient algorithms rely on set operations and sequence manipulation
- **Database Systems**: SQL combines sequence operations with set theory

1.2 Learning Objectives

By the end of this class, you will be able to:

1. Manipulate strings and sequences using Python indexing and methods

2. Apply mathematical set theory concepts using Python's set data type
3. Convert between different collection types based on problem requirements
4. Use set operations (union, intersection, difference) in both mathematical and programming contexts
5. Analyze the relationship between sequences (ordered) and sets (unordered, unique)
6. Apply collection concepts to solve real-world data problems

2 Strings as Sequences: The Foundation

2.1 String Fundamentals

A **string** is a sequence of characters, making it our first encounter with ordered collections. In Python, strings are immutable sequences delimited by quotes.

```python
# String creation and basic properties
name = "northeastern"
city = "boston"

# Strings are sequences - we can access individual elements
print(f"First character of {name}: {name[0]}")  # 'n'
print(f"Last character of {city}: {city[-1]}")   # 'n'
print(f"Length of {name}: {len(name)}")          # 12

# String indexing follows zero-based numbering
word = "python"
print("Index positions in 'python':")
for i in range(len(word)):
    print(f"Index {i}: '{word[i]}'")

# Negative indexing counts from the end
print(f"word[-1] = '{word[-1]}'")  # 'n'
print(f"word[-2] = '{word[-2]}'")  # 'o'
```

Listing 1: Filename: string_indexing.py - Basic String Operations and Indexing

2.2 Mathematical Connection: Sequences vs Sets

While strings are sequences (ordered, allow duplicates), they relate to mathematical sets in important ways:

Property	**Sequences (Strings)**	**Sets**
Order	Matters	Doesn't matter
Duplicates	Allowed	Not allowed
Indexing	Yes (by position)	No
Membership	$\in$ (element at position)	$\in$ (element in collection)

```python
# A string with repeated characters
message = "hello world"
print(f"Original string: '{message}'")
print(f"Length: {len(message)}")

# Convert to set to see unique characters
unique_chars = set(message)
print(f"Unique characters: {unique_chars}")
print(f"Number of unique characters: {len(unique_chars)}")

# Convert back to sorted list to see the difference
sorted_unique = sorted(unique_chars)
print(f"Sorted unique characters: {sorted_unique}")

# Demonstrate that order is lost in sets
print(f"Set from 'abc': {set('abc')}")
print(f"Set from 'cba': {set('cba')}")  # Same set!
```

Listing 2: Filename: strings_to_sets.py - Converting Strings to Sets - Removing Duplicates

3 Mathematical Sets: Formal Foundations

3.1 Set Definition and Notation

A **set** is a collection of unique elements. In mathematics, we use curly braces { } to denote sets, and Python adopts the same notation.

3.1.1 Python Sets: A New Data Structure for List Users

Since you've been working with lists in Python, let's understand how sets differ and why they're useful. Think of a set as a special kind of collection with two key rules:

1. **No Duplicates**: Each element can appear only once
2. **No Order**: Elements don't have positions like list indices

```python
# You're familiar with lists - they keep order and allow duplicates
my_list = [1, 2, 2, 3, 1, 4]
print(f"List: {my_list}")
print(f"Length: {len(my_list)}")           # 6 elements
print(f"First element: {my_list[0]}")      # Can access by index

# Sets are different - no duplicates, no order
my_set = {1, 2, 2, 3, 1, 4}  # Same elements as list
print(f"Set: {my_set}")                    # Duplicates automatically removed
print(f"Length: {len(my_set)}")            # Only 4 unique elements

# Can't access set elements by index - this would cause an error:
# print(my_set[0])  # TypeError: 'set' object is not subscriptable

# But you can check if something is in the set (very fast!)
print(f"Is 2 in the set? {2 in my_set}")   # True
print(f"Is 5 in the set? {5 in my_set}")   # False

# Converting between lists and sets
list_with_duplicates = [1, 1, 2, 2, 3, 3, 4, 4]
unique_set = set(list_with_duplicates)     # Remove duplicates
back_to_list = list(unique_set)            # Convert back (order may change)

print(f"Original list: {list_with_duplicates}")
print(f"As set (unique): {unique_set}")
print(f"Back to list: {back_to_list}")
```

Listing 3: Filename: lists_vs_sets.py - From Lists to Sets - Understanding the Differences

When to use sets instead of lists:

- When you need to eliminate duplicates from data
- When you want to test membership quickly ("Is X in this collection?")
- When you need to perform mathematical operations like union or intersection
- When the order of elements doesn't matter for your problem

When to stick with lists:

- When you need to access elements by position (indexing)
- When the order of elements is important
- When you need to allow duplicate values
- When you need to modify elements in place

3.1.2 Basic Set Concepts

Let $S = \{\text{Mon, Tue, Wed, Thu, Fri, Sat, Sun}\}$ be the set of days of the week.

- **Element membership**: Monday $\in S$ (Monday is an element of S)
- **Non-membership**: January $\notin S$ (January is not an element of S)
- **Cardinality**: $|S| = 7$ (S has 7 elements)

```python
# Creating sets in Python
days_of_week = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"}
print(f"Days of week: {days_of_week}")
print(f"Cardinality: {len(days_of_week)}")

# Element membership testing
print(f"'Mon' in days_of_week: {'Mon' in days_of_week}")      # True
print(f"'January' in days_of_week: {'January' in days_of_week}")  # False

# Sets automatically remove duplicates
duplicate_set = {1, 2, 1, 2, 3}
```

```
print(f"Set with duplicates {1, 2, 1, 2, 3} becomes: {duplicate_set}")

# Cardinality example from the transcript
S1 = {1, 2, 1, 2}  # Appears to have 4 elements
S2 = {1, 2}        # Clearly has 2 elements
print(f"S1 = {S1}, |S1| = {len(S1)}")  # Actually has 2 elements!
print(f"S2 = {S2}, |S2| = {len(S2)}")  # Has 2 elements
print(f"S1 == S2: {S1 == S2}")         # They are the same set!
```

Listing 4: Filename: python_sets_basic_operations.py - Python Sets - Basic Operations

3.2 Set Equality and Order Independence

Sets are equal if they contain exactly the same elements, regardless of order:

```
# Order independence in sets
S2 = {1, 2}
S3 = {2, 1}

print(f"S2 = {S2}")
print(f"S3 = {S3}")
print(f"S2 == S3: {S2 == S3}")  # True - same elements

# Compare with lists (sequences) where order matters
list1 = [1, 2]
list2 = [2, 1]
print(f"List [1, 2] == [2, 1]: {list1 == list2}")  # False - different order

# Converting between sets and lists
my_set = {3, 1, 4, 1, 5}
my_list = list(my_set)
print(f"Set {my_set} as list: {my_list}")  # Order may vary
print(f"Sorted list: {sorted(my_set)}")    # Consistent ordering
```

Listing 5: Filename: set_equality_order.py - Set Equality - Order Doesn't Matter

4 Common Sets and Set Builder Notation

4.1 Standard Mathematical Sets

Mathematics defines several important standard sets:

- **Empty Set**: $\emptyset = \{\}$ or $\varnothing$ (no elements)
- **Natural Numbers**: $\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$
- **Integers**: $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
- **Positive Integers**: $\mathbb{Z}^+ = \{1, 2, 3, 4, \ldots\}$ (excludes 0)
- **Real Numbers**: $\mathbb{R}$ (includes fractions, π, etc.)

```python
# Empty set
empty_set = set()  # Note: {} creates empty dict, not empty set
print(f"Empty set: {empty_set}")
print(f"Cardinality of empty set: {len(empty_set)}")

# Set containing empty set (different from empty set!)
set_with_empty = {frozenset()}  # Contains one element: the empty set
print(f"Set containing empty set: {set_with_empty}")
print(f"Cardinality: {len(set_with_empty)}")  # 1, not 0!

# Natural numbers (finite subset for demonstration)
natural_numbers = set(range(10))  # {0, 1, 2, ..., 9}
print(f"First 10 natural numbers: {natural_numbers}")

# Positive integers (excluding 0)
positive_integers = set(range(1, 11))  # {1, 2, 3, ..., 10}
print(f"First 10 positive integers: {positive_integers}")

# Integers including negatives
integers = set(range(-5, 6))  # {-5, -4, ..., 4, 5}
print(f"Integers from -5 to 5: {integers}")
```

Listing 6: Filename: standard_sets_python.py - Standard Sets in Python

4.2 Set Builder Notation

Set builder notation allows us to define sets using conditions: $S = \{x \mid \text{condition}\}$

4.2.1 Mathematical Examples

$$S_1 = \{x \in \mathbb{N} \mid x \leq 3\} = \{0, 1, 2, 3\} \tag{1}$$
$$S_2 = \{x \in \mathbb{N} \mid x \geq 3 \text{ and } x < 5\} = \{3, 4\} \tag{2}$$
$$S_3 = \{x^2 \mid x \in \{1, 2, 3\}\} = \{1, 4, 9\} \tag{3}$$

```python
# S1 = {x in N | x <= 3}
S1 = {x for x in range(10) if x <= 3}
print(f"S1 = {{x in N | x <= 3}} = {S1}")

# Don't forget 0 is in natural numbers!
print(f"Note: 0 is included in natural numbers")

# S2 = {x in N | x >= 3 and x < 5}
S2 = {x for x in range(10) if x >= 3 and x < 5}
print(f"S2 = {{x in N | x >= 3 and x < 5}} = {S2}")

# S3 = {x**2 | x in {1, 2, 3}}
S3 = {x**2 for x in {1, 2, 3}}
print(f"S3 = {{x**2 | x in {{1, 2, 3}}}} = {S3}")

# More complex example: even squares less than 50
even_squares = {x**2 for x in range(1, 10) if (x**2) % 2 == 0 and x**2 < 50}
print(f"Even squares < 50: {even_squares}")

# String example: vowels in a word
word = "northeastern"
vowels_in_word = {char for char in word if char in 'aeiou'}
print(f"Vowels in '{word}': {vowels_in_word}")
```

Listing 7: Filename: set_builder_notation.py - Set Builder Notation in Python - List Comprehensions

5 Sequences and Mutability: Lists vs Strings

5.1 Immutable Sequences: Strings and Tuples

Strings are immutable - once created, their contents cannot be changed. This has important implications for how we work with them.

```
# Strings are immutable
text = "hello"
print(f"Original text: {text}")

# This creates a NEW string, doesn't modify the original
new_text = text.upper()
print(f"After upper(): original = '{text}', new = '{new_text}'")

# Trying to modify a string character raises an error
try:
    text[0] = 'H'  # This will fail!
except TypeError as e:
    print(f"Error: {e}")

# String concatenation creates new strings
greeting = "Hello"
name = "World"
message = greeting + " " + name  # Creates new string
print(f"Concatenated: '{message}'")

# Tuples are also immutable sequences
weekdays = ("Mon", "Tue", "Wed", "Thu", "Fri")
print(f"Weekdays tuple: {weekdays}")
print(f"First weekday: {weekdays[0]}")

# Can't modify tuple elements
try:
    weekdays[2] = "Humpday"  # This will fail!
except TypeError as e:
    print(f"Error: {e}")
```

Listing 8: Filename: string_immutability.py - String Immutability and Its Implications

5.2 Mutable Sequences: Lists

Lists are mutable sequences that can be modified after creation, making them very different from sets in behavior.

```python
# Lists are mutable sequences
data = [13, 15, -4, 31, 5, 6, 19]
print(f"Original list: {data}")

# We can modify individual elements
data[0] = 42
print(f"After modifying index 0: {data}")

# Lists preserve order and allow duplicates
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5]
print(f"List with duplicates: {numbers}")
print(f"Length: {len(numbers)}")

# Convert to set to remove duplicates
unique_numbers = set(numbers)
print(f"As set (unique): {unique_numbers}")
print(f"Unique count: {len(unique_numbers)}")

# Convert back to list (order may change)
back_to_list = list(unique_numbers)
print(f"Back to list: {back_to_list}")

# Sorted version for consistent ordering
sorted_unique = sorted(unique_numbers)
print(f"Sorted unique: {sorted_unique}")
```

Listing 9: Filename: list_mutability_sets.py - List Mutability and Comparison with Sets

6 Subsets and Set Relationships

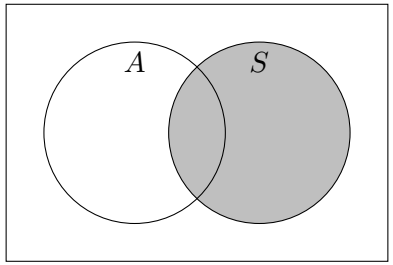
6.1 Subset Definition and Notation

A is a subset of B (written $A \subseteq B$) if and only if every element of A is also an element of B.

6.1.1 Mathematical Examples

Let $S = \{1, 2, 3\}$, $A = \{2\}$, $B = \{1, 2\}$, and $C = \{1, 2, 3\}$.

- $A \subseteq S$ because $2 \in S$
- $B \subseteq S$ because $1 \in S$ and $2 \in S$
- $C \subseteq S$ because $1, 2, 3 \in S$ (in fact, $C = S$)



Venn Diagram Explanation: The smaller circle $A = \{2\}$ is completely contained within the larger circle $S = \{1, 2, 3\}$, illustrating that $A \subseteq S$.

```
# Define our sets
S = {1, 2, 3}
A = {2}
B = {1, 2}
C = {1, 2, 3}

print(f"S = {S}")
print(f"A = {A}")
print(f"B = {B}")
print(f"C = {C}")

# Test subset relationships
    print(f"A subset S: {A.issubset(S)}")  # True
    print(f"B subset S: {B.issubset(S)}")  # True
    print(f"C subset S: {C.issubset(S)}")  # True

    # Proper vs improper subsets
```

```python
    print(f"A proper subset S: {A < S}")    # True (proper subset)
    print(f"C proper subset S: {C < S}")    # False (equal sets)
    print(f"C = S: {C == S}")            # True (same set)

    # Every set is a subset of itself
    print(f"S subset S: {S.issubset(S)}")    # True

    # Empty set is subset of every set
    empty = set()
    print(f"empty subset S: {empty.issubset(S)}")  # True
    print(f"empty subset A: {empty.issubset(A)}")  # True
```

Listing 10: Filename: subset_relationships.py - Subset Relationships in Python

6.2 Comparing Subset Notation with Numerical Inequalities

The subset relationships parallel numerical inequalities:

Numbers	Sets	Meaning
$a < b$	$A \subset B$	Proper (strict) relationship
$a \leq b$	$A \subseteq B$	Allows equality
$a = b$	$A = B$	Equality

7 Power Sets: All Possible Subsets

7.1 The Ice Cream Shop Example

Imagine visiting an ice cream shop with flavors $\{C, V, M\}$ (Chocolate, Vanilla, Mocha). What are your options?

- Choose no flavors: $\{\}$
- Choose one flavor: $\{C\}$, $\{V\}$, $\{M\}$
- Choose two flavors: $\{C, V\}$, $\{C, M\}$, $\{V, M\}$
- Choose all flavors: $\{C, V, M\}$

The **power set** $P(S)$ is the set of all possible subsets of S.

$$P(\{C, V, M\}) = \{\{\}, \{C\}, \{V\}, \{M\}, \{C, V\}, \{C, M\}, \{V, M\}, \{C, V, M\}\}$$

7.2 Power Set Cardinality

For a set with n elements, its power set has 2^n elements: $|P(S)| = 2^{|S|}$

Let's explore this concept through two complementary examples.

7.2.1 Basic Power Set Generation

```python
from itertools import combinations

def power_set(s):
    """Generate all subsets of a set"""
    s = list(s)  # Convert to list for indexing
    power_set_list = []

    # Generate all possible combinations of all lengths
    for r in range(len(s) + 1):
        for combo in combinations(s, r):
            power_set_list.append(set(combo))

    return power_set_list

# Ice cream flavors example
flavors = {'Chocolate', 'Vanilla', 'Mocha'}
power_flavors = power_set(flavors)

print(f"Original set of flavors: {flavors}")
print(f"Power set P(flavors) - all possible ice cream combinations:")

for i, subset in enumerate(power_flavors):
    if len(subset) == 0:
        print(f"  {i}: {subset} (no ice cream)")
    else:
        print(f"  {i}: {subset}")
```

```
27
28 print(f"\nCardinality verification:")
29 print(f"  |S| = {len(flavors)} flavors")
30 print(f"  |P(S)| = {len(power_flavors)} total combinations")
31 print(f"  Formula: 2^|S| = 2^{len(flavors)} = {2**len(flavors)}")
32 print(f"  Formula verified: {len(power_flavors) == 2**len(flavors)}")
```

Listing 11: Filename: basic_power_sets.py - Basic Power Set Generation

7.2.2 Binary Representation Connection

The connection between power sets and binary numbers reveals why $|P(S)| = 2^{|S|}$:

```
1  def binary_to_subset(binary_str, elements):
2      """Convert a binary string to a subset based on element positions"""
3      subset = set()
4      for i, bit in enumerate(binary_str):
5          if bit == '1' and i < len(elements):
6              subset.add(elements[i])
7      return subset
8
9  # Binary representation connection with 3 elements
10 elements = ['C', 'V', 'M']  # Chocolate, Vanilla, Mocha (abbreviated)
11 n = len(elements)
12
13 print(f"Binary Representation Connection")
14 print(f"Elements: {elements}")
15 print(f"Each subset corresponds to a {n}-bit binary number:")
16 print()
17
18 for i in range(2**n):  # 2^n possibilities
19     binary = format(i, f'0{n}b')  # n-bit binary representation
20     subset = binary_to_subset(binary, elements)
21     print(f"  {binary} -> {subset}")
22
23 print(f"\nTotal subsets: {2**n} = 2^{n}")
24
25 # Verify the formula with different sized sets
```

```python
print(f"\nFormula Verification Across Different Set Sizes:")
print(f"{'Set Size':<10} {'Actual |P(S)|':<15} {'Expected 2^n':<15} {'Verified':<10}")
print("-" * 55)

for size in range(1, 6):
    actual_count = 2 ** size  # We know this mathematically
    expected_count = 2 ** size
    verified = actual_count == expected_count

    print(f"{size:<10} {actual_count:<15} {expected_count:<15} {'YES' if verified else 'NO':<10}")

print(f"\nKey Insight: Every subset can be represented as a binary number!")
print(f"   - Bit position i: include element i if bit = 1, exclude if bit = 0")
print(f"   - This is why |P(S)| = 2^|S| for any finite set S")
```

Listing 12: Filename: binary_power_sets.py - Binary Representation and Power Set Formula

8 Set Operations: Building Complex Relationships

8.1 Fundamental Set Operations

Set operations allow us to combine and manipulate sets in powerful ways. Each operation has both mathematical notation and Python implementation.

8.1.1 Union ($A \cup B$)

The union contains all elements that are in either set (or both).

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$

8.1.2 Intersection ($A \cap B$)

The intersection contains only elements that are in both sets.

$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$$

8.1.3 Difference ($A - B$ or $A \setminus B$)

The difference contains elements in the first set but not the second.

$$A - B = \{x \mid x \in A \text{ and } x \notin B\}$$

8.1.4 Complement ($\overline{A}$ or A^c)

The complement contains all elements in the universe that are not in the set.

$$\overline{A} = \{x \in U \mid x \notin A\}$$

```python
# Define example sets
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}
U = {1, 2, 3, 4, 5, 6, 7, 8}  # Universe set

print(f"A = {A}")
print(f"B = {B}")
print(f"Universe U = {U}")
print()

    # Union: A union B
    union = A | B  # or A.union(B)
    print(f"A union B = {union}")

    # Intersection: A intersect B
    intersection = A & B  # or A.intersection(B)
    print(f"A intersect B = {intersection}")

    # Difference: A - B
    difference = A - B  # or A.difference(B)
    print(f"A - B = {difference}")

    # Symmetric difference: A XOR B (elements in A or B, but not both)
    sym_diff = A ^ B  # or A.symmetric_difference(B)
    print(f"A XOR B = {sym_diff}")

```

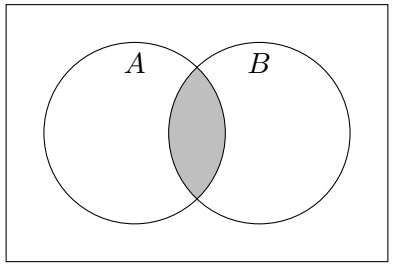
```
    # Complement: complement of A (with respect to universe U)
    complement_A = U - A
    print(f"complement of A in U = {complement_A}")

    # Verify De Morgan's laws
    # not(A union B) = (not A) intersect (not B)
    left_side = U - (A | B)
    right_side = (U - A) & (U - B)
    print(f"\nDe Morgan's Law verification:")
    print(f"not(A union B) = {left_side}")
    print(f"(not A) intersect (not B) = {right_side}")
print(f"Equal? {left_side == right_side}")
```

Listing 13: Filename: set_operations.py - Set Operations in Python

8.2 Venn Diagrams and Visual Representation

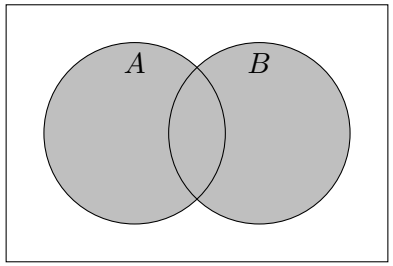
Venn diagrams help visualize set relationships and operations.

8.2.1 Intersection ($A \cap B$)



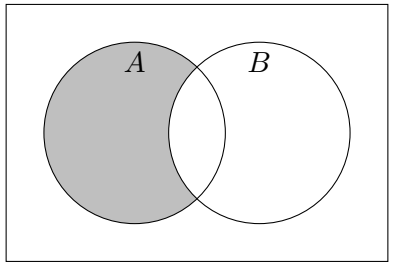
Intersection: The shaded region shows elements that belong to both A and B.

8.2.2 Union ($A \cup B$)



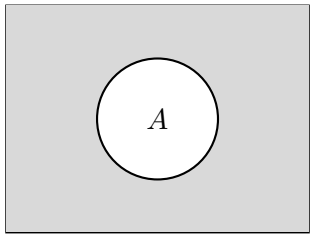
Union: The shaded region shows all elements that belong to either A or B (or both).

8.2.3 Difference ($A - B$)



Difference: The shaded region shows elements that are in A but not in B.

8.2.4 Complement ($\overline{A}$)



Complement: The shaded region shows all elements in the universe U that are not in A. The rectangle represents the universe U.

9 Advanced Applications: Combining Sequences and Sets

9.1 Text Analysis with Sets and Sequences

Let's apply both sequence and set concepts to analyze text data.

```
def analyze_text(text):
    """Comprehensive text analysis using both sequences and sets"""

    # Sequence analysis
    print(f"Text: '{text}'")
    print(f"Length (with spaces): {len(text)}")
    print(f"Character at position 0: '{text[0]}'")
    print(f"Character at position -1: '{text[-1]}'")

    # Convert to lowercase for analysis
    text_lower = text.lower()

    # Set analysis - unique characters
    unique_chars = set(text_lower)
    print(f"Unique characters: {sorted(unique_chars)}")
    print(f"Number of unique characters: {len(unique_chars)}")

    # Vowels and consonants
    vowels = set('aeiou')
    consonants = set('bcdfghjklmnpqrstvwxyz')

    text_vowels = unique_chars & vowels
    text_consonants = unique_chars & consonants

    print(f"Vowels in text: {sorted(text_vowels)}")
    print(f"Consonants in text: {sorted(text_consonants)}")

    # Character frequency (using sequences)
    char_freq = {}
    for char in text_lower:
        if char.isalpha():  # Only count letters
            char_freq[char] = char_freq.get(char, 0) + 1

```

```
    print(f"Character frequencies: {dict(sorted(char_freq.items()))}")

    # Most common characters
    if char_freq:
        max_freq = max(char_freq.values())
        most_common = {char for char, freq in char_freq.items() if freq == max_freq}
        print(f"Most frequent character(s): {most_common} (appears {max_freq} times)")

# Analyze different texts
texts = [
    "Hello World",
    "Northeastern University",
    "Computer Science and Mathematics"
]

for text in texts:
    print("=" * 50)
    analyze_text(text)
    print()
```

Listing 14: Filename: text_analysis.py - Text Analysis - Combining Strings and Sets

9.2 Data Deduplication and Filtering

A common real-world application combining sequences and sets:

```
def process_student_data():
    """Demonstrate data processing using sequences and sets"""

    # Student enrollment data (sequences with possible duplicates)
    cs5001_students = ["Alice", "Bob", "Charlie", "Diana", "Alice", "Eve"]
    cs5002_students = ["Bob", "Charlie", "Frank", "Grace", "Alice"]

    print("Original enrollment lists (sequences):")
    print(f"CS 5001: {cs5001_students}")
    print(f"CS 5002: {cs5002_students}")

    # Convert to sets for analysis
```

```
    cs5001_set = set(cs5001_students)
    cs5002_set = set(cs5002_students)

    print(f"\nUnique students per class (sets):")
    print(f"CS 5001: {cs5001_set}")
    print(f"CS 5002: {cs5002_set}")

    # Set operations for analysis
    both_classes = cs5001_set & cs5002_set
    only_5001 = cs5001_set - cs5002_set
    only_5002 = cs5002_set - cs5001_set
    all_students = cs5001_set | cs5002_set

    print(f"\nEnrollment analysis:")
    print(f"Taking both classes: {both_classes}")
    print(f"Only CS 5001: {only_5001}")
    print(f"Only CS 5002: {only_5002}")
    print(f"All students: {all_students}")

    # Statistics
    print(f"\nStatistics:")
    print(f"Total unique students: {len(all_students)}")
    print(f"Students in both classes: {len(both_classes)}")
    print(f"Percentage taking both: {len(both_classes)/len(all_students)*100:.1f}%")

    # Convert back to sorted lists for reporting
    print(f"\nSorted lists for reports:")
    print(f"All students (alphabetical): {sorted(all_students)}")
    print(f"Both classes (alphabetical): {sorted(both_classes)}")

process_student_data()
```

Listing 15: Filename: data_processing.py - Data Processing - Sequences to Sets and Back

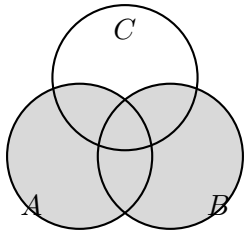
10 Complex Set Operations: Step-by-Step Analysis

10.1 Compound Operations with Venn Diagrams

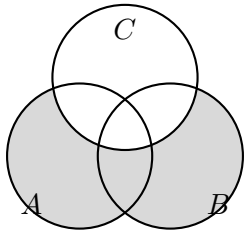
Let's analyze complex expressions step by step: $(A \cup B) - C$ and $(A \cap C) \cup \overline{B}$

10.1.1 Expression 1: $(A \cup B) - C$

Step 1: First, find $A \cup B$ (union of A and B)



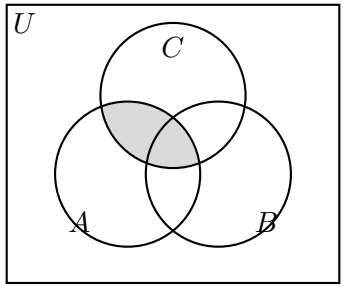
Step 2: Then subtract C from $(A \cup B)$ - elements in A or B but not in C



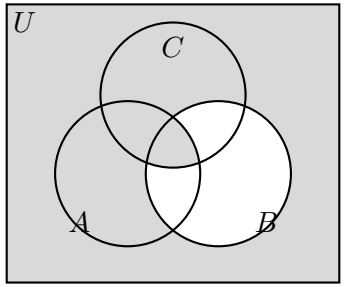
Result: Elements in A or B but not in C.

10.1.2 Expression 2: $(A \cap C) \cup \overline{B}$

Step 1: Find $A \cap C$ (intersection of A and C)



Step 2: Find $\overline{B}$ (complement of B) and take union with $(A \cap C)$



Result: Elements in both A and C, plus all elements not in B.

```python
def analyze_complex_operations():
    """Step-by-step analysis of complex set operations"""

    # Define our sets
    A = {1, 2, 3, 4}
    B = {3, 4, 5, 6}
    C = {2, 4, 6, 8}
    U = set(range(1, 11))  # Universe: {1, 2, 3, ..., 10}

    print("Given sets:")
    print(f"A = {A}")
    print(f"B = {B}")
    print(f"C = {C}")
    print(f"U = {U}")
    print()

```

```
    # Expression 1: (A union B) - C
    print("Expression 1: (A union B) - C")
    print("Step 1: Calculate A union B")
    union_AB = A | B
    print(f"  A union B = {union_AB}")

    print("Step 2: Calculate (A union B) - C")
    result1 = union_AB - C
    print(f"  (A union B) - C = {result1}")
    print()

    # Expression 2: (A intersect C) union complement(B)
    print("Expression 2: (A intersect C) union complement(B)")
    print("Step 1: Calculate A intersect C")
    intersection_AC = A & C
    print(f"  A intersect C = {intersection_AC}")

    print("Step 2: Calculate complement(B)")
    complement_B = U - B
    print(f"  complement(B) = {complement_B}")

    print("Step 3: Calculate (A intersect C) union complement(B)")
    result2 = intersection_AC | complement_B
    print(f"  (A intersect C) union complement(B) = {result2}")
    print()

    # Verify using direct Python evaluation
    print("Verification using direct Python evaluation:")
    direct1 = (A | B) - C
    direct2 = (A & C) | (U - B)

    print(f"(A | B) - C = {direct1}")
    print(f"(A & C) | (U - B) = {direct2}")
    print(f"Results match: {result1 == direct1 and result2 == direct2}")

analyze_complex_operations()
```

Listing 16: Filename: complex_set_operations.py - Complex Set Operations - Step by Step

11 Practical Applications: Real-World Problem Solving

11.1 Database-Style Queries Using Sets

Sets are fundamental to database operations and data analysis:

```
def database_operations():
    """Simulate database operations using sets"""

    # Employee database simulation
    employees = {
        "engineering": {"Alice", "Bob", "Charlie", "Diana"},
        "marketing": {"Eve", "Frank", "Alice", "Grace"},
        "sales": {"Bob", "Grace", "Henry", "Iris"},
        "management": {"Alice", "Frank", "Henry"}
    }

    print("Employee Database:")
    for dept, people in employees.items():
        print(f"  {dept}: {people}")
    print()

    # Query 1: Who works in multiple departments?
    all_employees = set()
    for dept_employees in employees.values():
        all_employees |= dept_employees

    multi_dept = set()
    for employee in all_employees:
        dept_count = sum(1 for dept_employees in employees.values()
                         if employee in dept_employees)
        if dept_count > 1:
            multi_dept.add(employee)

    print(f"Employees in multiple departments: {multi_dept}")

    # Query 2: Engineering OR Marketing (union)
    eng_or_marketing = employees["engineering"] | employees["marketing"]
    print(f"Engineering OR Marketing: {eng_or_marketing}")
```

```
34
35      # Query 3: Engineering AND Marketing (intersection)
36      eng_and_marketing = employees["engineering"] & employees["marketing"]
37      print(f"Engineering AND Marketing: {eng_and_marketing}")
38
39      # Query 4: In Engineering but NOT in Management
40      eng_not_mgmt = employees["engineering"] - employees["management"]
41      print(f"Engineering but not Management: {eng_not_mgmt}")
42
43      # Query 5: Department-specific analysis
44      print(f"\nDepartment sizes:")
45      for dept, people in employees.items():
46          print(f"  {dept}: {len(people)} employees")
47
48      # Query 6: Find employees unique to each department
49      print(f"\nEmployees unique to each department:")
50      for dept, people in employees.items():
51          others = set()
52          for other_dept, other_people in employees.items():
53              if other_dept != dept:
54                  others |= other_people
55          unique_to_dept = people - others
56          print(f"  {dept} only: {unique_to_dept}")
57
58  database_operations()
```

Listing 17: Filename: database_operations.py - Database Operations Using Set Theory

11.2 Text Processing: Word Analysis

Combining string manipulation with set operations for natural language processing:

```
1   def analyze_documents():
2       """Analyze text documents using sets and sequences"""
3
4       documents = [
5           "Python is a powerful programming language for data science",
6           "Data science requires strong programming and mathematical skills",
```

```
        "Mathematical foundations are essential for computer science"
    ]

    print("Document Analysis:")
    print("=================")

    # Process each document
    doc_words = []
    for i, doc in enumerate(documents):
        # Convert to lowercase and split into words
        words = doc.lower().replace(",", "").replace(".", "").split()
        doc_words.append(set(words))  # Convert to set for analysis

        print(f"Document {i+1}: '{doc}'")
        print(f"  Words: {sorted(words)}")
        print(f"  Unique words: {len(doc_words[i])}")
        print()

    # Set operations on documents
    print("Cross-Document Analysis:")
    print("========================")

    # Words in all documents (intersection)
    common_words = doc_words[0]
    for word_set in doc_words[1:]:
        common_words &= word_set
    print(f"Words in ALL documents: {sorted(common_words)}")

    # Words in any document (union)
    all_words = set()
    for word_set in doc_words:
        all_words |= word_set
    print(f"ALL unique words: {sorted(all_words)}")
    print(f"Total vocabulary size: {len(all_words)}")

    # Words unique to each document
    print(f"\nWords unique to each document:")
    for i, word_set in enumerate(doc_words):
        others = set()
```

```
        for j, other_set in enumerate(doc_words):
            if i != j:
                others |= other_set
        unique = word_set - others
        print(f"  Document {i+1} only: {sorted(unique)}")

    # Find documents sharing specific words
    target_words = {"programming", "science", "data"}
    print(f"\nDocuments containing each target word:")
    for word in target_words:
        containing_docs = []
        for i, word_set in enumerate(doc_words):
            if word in word_set:
                containing_docs.append(i + 1)
        print(f"  '{word}': Documents {containing_docs}")

analyze_documents()
```

Listing 18: Filename: nlp_analysis.py - Natural Language Processing with Sets and Sequences

12 Summary and Key Takeaways

12.1 Connections Between Domains

Concept	**Programming (CS 5001)**	**Mathematics (CS 5002)**
Collections	Strings, Lists, Tuples	Sets, Sequences
Order	Sequences preserve order	Sets ignore order
Uniqueness	Lists allow duplicates	Sets enforce uniqueness
Operations	String methods, List methods	Union, Intersection, Complement
Membership	`in` operator	$\in$ notation
Size	`len()` function	Cardinality $\|S\|$
Empty	`[]` or `set()`	$\emptyset$ or $\varnothing$

12.2 Best Practices

1. **Choose the right data structure**:

- Use strings for text that won't change
- Use lists when order matters and duplicates are allowed
- Use sets when uniqueness is important and order doesn't matter
- Use tuples for immutable sequences

2. **Leverage set operations**:
   - Use intersection (`&`) to find common elements
   - Use union (`|`) to combine collections
   - Use difference (`-`) to find unique elements
   - Use symmetric difference (`^`) for "either but not both"
3. **Convert between types strategically**:
   - Convert to set to remove duplicates: `list(set(my_list))`
   - Convert to list to sort: `sorted(my_set)`
   - Use list comprehensions for filtering: `[x for x in data if condition]`

12.3 Looking Ahead

These fundamental concepts will appear throughout both courses:

- **CS 5001**: Advanced data structures, algorithms, file processing, web scraping
- **CS 5002**: Relations, functions, graph theory, combinatorics
- **Integration**: Database design, algorithm analysis, data science applications

Understanding both the programming implementation and mathematical foundation of collections gives you powerful tools for solving complex problems in computer science and beyond.