Lecture 7 Handout

Data Structures: Tuples, Dictionaries, and Sets

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2025

Required Reading

Textbook: Chapter 7 (Tuples and Sequence Operations), Chapter 8 (Dictionaries and Sets)
Reference Notebooks: `ch07/` (sequence operations), `ch08/` (dictionaries and sets)

Learning Objectives

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

1. **Master sequence operations** including unpacking, slicing, and the del statement
2. **Understand tuple immutability concepts** and appropriate use cases for unchangeable data
3. **Create and manipulate dictionaries** for efficient key-value data relationships
4. **Utilize sets effectively** for unique collections and mathematical operations
5. **Choose appropriate data structures** based on problem requirements and performance
6. **Combine multiple data structures** in sophisticated applications strategically

Prerequisites Review

Building on Your Comprehensive Programming Foundation:

From Lectures 1-2: Variables, data types, arithmetic operations, f-strings, decision structures (if/elif/else), boolean logic, string methods for input validation
From Lectures 3-4: Basic list operations (indexing, slicing, iteration), **list comprehensions** for data processing and transformation
From Lecture 5: Functions (definition, parameters, return values, modules)
From Lecture 6: Exception handling and advanced list operations (try/except statements, defensive programming, .append(), .extend(), .sort(), searching methods)

Transformation Goal: Move from basic collections and defensive programming to **advanced data structure mastery** - building sophisticated applications that manage complex data relationships efficiently.

1 Part 1: Unpacking Sequences and Tuple Immutability

1.1 Understanding Sequence Unpacking

You can unpack any sequence's elements by assigning the sequence to a comma-separated list of variables. This powerful feature makes code more readable and enables elegant handling of structured data.

```python
# Sequence unpacking demonstration
student_tuple = ('Amanda', [98, 85, 87])
print(f"Student tuple: {student_tuple}")

# Unpack into separate variables
first_name, grades = student_tuple
print(f"Name: {first_name}")
print(f"Grades: {grades}")
```

1.2 Unpacking Works with Different Sequence Types

Sequence unpacking is versatile and works with various data types:

```python
# Unpacking works with different sequence types
first, second = 'hi'  # Unpack string
print(f'{first}  {second}')

number1, number2, number3 = [2, 3, 5]  # Unpack list
print(f'{number1}  {number2}  {number3}')

# Even works with ranges
number1, number2, number3 = range(10, 40, 10)
print(f'{number1}  {number2}  {number3}')
```

1.3 Swapping Values Via Packing and Unpacking

Python's tuple packing and unpacking provides elegant value swapping:

```python
# Elegant value swapping using tuple packing/unpacking
number1 = 99
number2 = 22
print(f"Before swap: number1 = {number1}, number2 = {number2}")

# Swap values using tuple packing and unpacking
number1, number2 = (number2, number1)
print(f"After swap: number1 = {number1}, number2 = {number2}")
```

1.4 Accessing Indices and Values with enumerate

The built-in function enumerate provides the preferred way to access both indices and values:

```python
# Using enumerate for index and value access
colors = ['red', 'orange', 'yellow']

# Convert enumerate result to list to see structure
enumerated_list = list(enumerate(colors))
print(f"Enumerate as list: {enumerated_list}")

# Use enumerate in loop with unpacking
```

```python
for index, value in enumerate(colors):
    print(f'{index}: {value}')
```

1.5 Creating a Primitive Bar Chart

Practical example using enumerate to create visual representations:

```python
# Creating a bar chart using enumerate and string multiplication
numbers = [19, 3, 15, 7, 11]

print('\nCreating a bar chart from numbers:')
print(f'Index{"Value":>8}   Bar')

for index, value in enumerate(numbers):
    # Create bar using string multiplication
    bar = "*" * value  # String repetition creates visual bar
    print(f'{index:>5}{value:>8}   {bar}')
```

This demonstrates how sequence unpacking makes code both readable and functional.

2 Part 2: Sequence Slicing and the del Statement

2.1 Understanding Advanced Slicing Operations

You can slice sequences to create new sequences of the same type containing subsets of the original elements. Slice operations work identically for lists, tuples and strings.

```python
# Comprehensive slicing demonstration
numbers = [2, 3, 5, 7, 11, 13, 17, 19]
print(f"Original list: {numbers}")
print("Index positions: 0  1  2  3   4   5   6   7")

# Basic slicing with start and end
subset1 = numbers[2:6]  # Index 6 not included
print(f"numbers[2:6]: {subset1}")

# Slicing with only ending index (start assumed 0)
subset2 = numbers[:6]
print(f"numbers[:6]: {subset2}")

# Slicing with only starting index (end assumed length)
subset3 = numbers[6:]
print(f"numbers[6:]: {subset3}")

# Slicing entire sequence (creates copy)
subset4 = numbers[:]
print(f"numbers[:] (copy): {subset4}")
```

2.2 Advanced Slicing with Steps

Slicing supports step parameters for more sophisticated operations:

```python
# Advanced slicing with steps
numbers = [2, 3, 5, 7, 11, 13, 17, 19]

# Slicing with step of 2
every_other = numbers[::2]
```

```python
print(f"Every other element: {every_other}")

# Reverse the entire list
reversed_list = numbers[::-1]
print(f"Reversed list: {reversed_list}")
```

2.3 Modifying Lists Via Slices

You can modify lists by assigning to slices, providing flexible list modification:

```python
# Modifying lists through slice assignment
numbers = [2, 3, 5, 7, 11, 13, 17, 19]
print(f"Original: {numbers}")

# Replace multiple elements with slice assignment
numbers[0:3] = ['two', 'three', 'five']
print(f"After replacing first 3: {numbers}")

# Delete elements using slice assignment to empty list
numbers[0:3] = []
print(f"After deleting first 3: {numbers}")
```

2.4 Advanced Slice Modification with Steps

Slice assignment works with step parameters for complex modifications:

```python
# Advanced slice modification with steps
numbers = [2, 3, 5, 7, 11, 13, 17, 19]
print(f"Original: {numbers}")

# Modify every other element
numbers[::2] = [100, 100, 100, 100]
print(f"After modifying every other element: {numbers}")

# Clear entire list using slice
numbers[:] = []
print(f"After clearing with slice: {numbers}")
```

2.5 del Statement for Element Removal

The del statement provides flexible ways to remove elements from lists:

```python
# Using del statement for various removal operations
numbers = list(range(0, 10))
print(f"Original list: {numbers}")

# Delete element at specific index
del numbers[-1]  # Delete last element
print(f"After deleting last element: {numbers}")

# Delete a slice
del numbers[0:2]  # Delete first two elements
print(f"After deleting first two: {numbers}")

# Delete every other element
del numbers[::2]
print(f"After deleting every other: {numbers}")
```

```python

# Delete entire contents
del numbers[:]
print(f"After deleting all contents: {numbers}")
```

3 Part 3: Dictionary Fundamentals

3.1 Understanding Dictionary Key-Value Relationships

A dictionary associates keys with values, enabling efficient data lookup based on meaningful identifiers rather than numeric positions. This key-value relationship provides O(1) average access time for professional applications.

```python
# Creating dictionaries with different data types
# Country names and Internet country codes
country_codes = {'Finland': 'fi', 'South Africa': 'za', 'Nepal': 'np'}
print(f"Country codes: {country_codes}")

# Roman numerals dictionary (string keys, int values)
roman_numerals = {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 10}
print(f"Roman numerals: {roman_numerals}")

# Dictionary length and emptiness checking
print(f"Country codes length: {len(country_codes)}")

# Using dictionary as boolean condition
if country_codes:
    print('country_codes is not empty')
else:
    print('country_codes is empty')
```

3.2 Basic Dictionary Operations

Dictionaries support efficient access, modification, and removal operations:

```python
# Basic dictionary operations demonstration
roman_numerals = {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 100}  # X
    intentionally wrong
print(f"Initial dictionary: {roman_numerals}")

# Accessing values by key
x_value = roman_numerals['X']
print(f"Value of 'X': {x_value}")

# Updating existing key-value pair
roman_numerals['X'] = 10  # Correct the value
print(f"After correcting X: {roman_numerals}")

# Adding new key-value pair
roman_numerals['L'] = 50
print(f"After adding L: {roman_numerals}")
```

3.3 Removing Key-Value Pairs

Various methods for removing dictionary entries:

```python
# Removing key-value pairs
print(f"Before removal: {roman_numerals}")

# Remove using del statement
del roman_numerals['III']
print(f"After deleting III: {roman_numerals}")

# Remove using .pop() method (returns removed value)
removed_value = roman_numerals.pop('X')
print(f"Removed value for X: {removed_value}")
print(f"After popping X: {roman_numerals}")
```

3.4 Safe Dictionary Access with get() Method

The .get() method provides safe access by returning None or a default value when keys don't exist:

```python
# Safe dictionary access demonstration
roman_numerals = {'I': 1, 'II': 2, 'V': 5, 'L': 50}

# Safe access with .get() (returns None if key not found)
missing_value = roman_numerals.get('III')
print(f"Missing key 'III': {missing_value}")

# Safe access with custom default
missing_with_default = roman_numerals.get('III', 'III not in dictionary
    ')
print(f"Missing key with default: {missing_with_default}")

# Safe access for existing key
existing_value = roman_numerals.get('V')
print(f"Existing key 'V': {existing_value}")
```

3.5 Testing Dictionary Membership and Iteration

Membership testing and dictionary iteration methods:

```python
# Dictionary membership testing
print(f"Dictionary contents: {roman_numerals}")

# Test key membership
print(f"'V' in roman_numerals: {'V' in roman_numerals}")
print(f"'III' in roman_numerals: {'III' in roman_numerals}")

# Dictionary iteration methods
months = {'January': 1, 'February': 2, 'March': 3}

# Iterate over key-value pairs
print("Key-value pairs:")
for month_name, month_number in months.items():
    print(f"  {month_name}: {month_number}")

# Iterate over keys only
print("\nMonth names:")
for month_name in months.keys():
    print(f"  {month_name}")
```

```python

# Iterate over values only
print("\nMonth numbers:")
for month_number in months.values():
    print(f"  {month_number}")
```

3.6 Dictionary Views and Dynamic Updates

Dictionary methods return views that reflect the current state:

```python
# Demonstrating dynamic dictionary views
months_view = months.keys()
print("Original keys view:")
for key in months_view:
    print(f"  {key}")

# Add new item to dictionary
months['December'] = 12
print(f"\nAfter adding December: {months}")

# View automatically reflects changes
print("Updated keys view:")
for key in months_view:
    print(f"  {key}")

# Processing in sorted order
print("\nMonths in alphabetical order:")
for month_name in sorted(months.keys()):
    print(f"  {month_name}: {months[month_name]}")
```

4 Part 4: Sets and Mathematical Operations

4.1 Understanding Set Uniqueness and Creation

A set is an unordered collection of unique values that automatically eliminates duplicates. This uniqueness property makes sets perfect for data deduplication and membership testing.

```python
# Creating sets and demonstrating uniqueness
colors = {'red', 'orange', 'yellow', 'green', 'red', 'blue'}
print(f"Set with duplicates removed: {colors}")
print(f"Number of unique colors: {len(colors)}")

# Membership testing
print(f"'red' in colors: {'red' in colors}")
print(f"'purple' in colors: {'purple' in colors}")

# Iterating through sets (order not guaranteed)
print("Colors in uppercase:")
for color in colors:
    print(f"  {color.upper()}")
```

4.2 Creating Sets with the set() Function

The built-in set() function creates sets from any iterable:

```python
# Creating sets from lists with duplicates
numbers = list(range(10)) + list(range(5))  # Create list with
    duplicates
print(f"List with duplicates: {numbers}")

# Convert to set for deduplication
unique_numbers = set(numbers)
print(f"Set with unique values: {unique_numbers}")

# Empty set creation
empty_dict = {}  # This creates a dictionary
empty_set = set()  # This creates a set
print(f"Empty dictionary: {empty_dict} (type: {type(empty_dict)})")
print(f"Empty set: {empty_set} (type: {type(empty_set)})")
```

4.3 Mathematical Set Operations

Sets support mathematical operations for powerful data analysis:

```python
# Mathematical set operations demonstration
set1 = {1, 3, 5}
set2 = {2, 3, 4}
print(f"Set 1: {set1}")
print(f"Set 2: {set2}")

# Union - all unique elements from both sets
union_result = set1 | set2
print(f"Union (set1 | set2): {union_result}")

# Union method with iterable argument
union_method = set1.union([20, 20, 3, 40, 40])
print(f"Union method: {union_method}")

# Intersection - elements common to both sets
intersection_result = set1 & set2
print(f"Intersection (set1 & set2): {intersection_result}")

# Difference - elements in left set but not in right set
difference_result1 = set1 - set2
print(f"Difference (set1 - set2): {difference_result1}")

difference_result2 = set2 - set1
print(f"Difference (set2 - set1): {difference_result2}")
```

4.4 Advanced Set Operations and Comparisons

More sophisticated set operations for data analysis:

```python
# Symmetric difference and disjoint testing
set1 = {1, 3, 5}
set2 = {2, 3, 4}

# Symmetric difference - elements in either set but not both
sym_diff_result = set1 ^ set2
print(f"Symmetric difference (set1 ^ set2): {sym_diff_result}")

```

```python
# Disjoint test - sets with no common elements
disjoint1 = {1, 3, 5}.isdisjoint({2, 4, 6})
print(f"Are {1, 3, 5} and {2, 4, 6} disjoint? {disjoint1}")

disjoint2 = {1, 3, 5}.isdisjoint({4, 6, 1})
print(f"Are {1, 3, 5} and {4, 6, 1} disjoint? {disjoint2}")
```

4.5 Mutable Set Operations and Methods

Sets support methods that modify the set in place:

```python
# Adding and removing individual elements
numbers = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
print(f"Initial set: {numbers}")

# Add element (no effect if already present)
numbers.add(17)
print(f"After adding 17: {numbers}")

numbers.add(3)  # Adding existing element has no effect
print(f"After adding 3 again: {numbers}")

# Remove element (raises KeyError if not present)
numbers.remove(3)
print(f"After removing 3: {numbers}")

# Discard element (no error if not present)
numbers.discard(999)  # Safe removal of non-existent element
print(f"After discarding 999: {numbers}")

# Update set with union assignment
numbers |= {2, 3, 4}
print(f"After union assignment: {numbers}")
```

5 Part 5: Integrated Data Structure Applications

5.1 Professional Academic Management System

This comprehensive example demonstrates strategic use of all data structures, leveraging each structure's strengths for optimal application design:

```python
# Comprehensive academic system using all data structures strategically

# System configuration (tuple for immutability)
SYSTEM_CONFIG = ("Academic Management System v2.0", "2024", "Quinnipiac
    University")
system_name, version_year, institution = SYSTEM_CONFIG
print(f"{system_name} - {institution} ({version_year})")

# Course catalog (dictionary for efficient lookup)
course_catalog = {
    "CS101": {"name": "Introduction to Programming", "credits": 3, "
        prerequisites": set()},
    "CS201": {"name": "Data Structures", "credits": 3, "prerequisites":
        {"CS101"}},
    "MATH201": {"name": "Calculus I", "credits": 4, "prerequisites":
        set()},
    "ENG101": {"name": "English Composition", "credits": 3, "prerequisites": set()}
}

print(f"\nCourse Catalog: {len(course_catalog)} courses available")
```

5.2 Student Database with Integrated Data Structures

Complete student management using all four data structures strategically:

```
# Student database with integrated data structures
student_database = {
    "S001": {
        "personal_info": ("Alice Johnson", "alice@qu.edu", "123-456-7890"),  # Immutable contact
        "academic_record": {
            "major": "Computer Science",
            "enrolled_courses": {"CS101", "MATH201", "ENG101"},  # Set for unique enrollments
            "completed_courses": {"CS101"},  # Set for completed courses
            "grade_history": [  # List for ordered grade history
                ("CS101", "A", 95),
                ("MATH201", "B+", 87),
                ("ENG101", "A-", 92)
            ],
            "semester_gpas": [3.8, 3.6, 3.9]  # List for GPA progression
        }
    },
    "S002": {
        "personal_info": ("Bob Smith", "bob@qu.edu", "123-456-7891"),
        "academic_record": {
            "major": "Mathematics",
            "enrolled_courses": {"MATH201", "ENG101"},
            "completed_courses": {"MATH201"},
            "grade_history": [
                ("MATH201", "A", 94),
                ("ENG101", "B", 88)
            ],
            "semester_gpas": [3.9, 3.7]
        }
    }
}

print(f"Student Database: {len(student_database)} students enrolled")
```

5.3 Comprehensive Analysis Functions

Analysis functions using integrated data structures:

```
# Comprehensive analysis functions using all data structures

def analyze_student_progress(student_id):
    """Analyze individual student progress using integrated data structures."""
    if student_id not in student_database:
```

```python
        return f"Student {student_id} not found"

    student = student_database[student_id]

    # Unpack immutable personal info (tuple)
    name, email, phone = student["personal_info"]

    # Access academic record (dictionary)
    academic = student["academic_record"]

    # Calculate current GPA from ordered history (list)
    if academic["semester_gpas"]:
        current_gpa = academic["semester_gpas"][-1]  # Most recent GPA
        avg_gpa = sum(academic["semester_gpas"]) / len(academic["
            semester_gpas"])
    else:
        current_gpa = avg_gpa = 0.0

    # Set operations for course analysis
    enrolled = academic["enrolled_courses"]
    completed = academic["completed_courses"]
    in_progress = enrolled - completed  # Set difference

    return {
        "name": name,
        "email": email,
        "major": academic["major"],
        "current_gpa": current_gpa,
        "average_gpa": avg_gpa,
        "courses_completed": len(completed),
        "courses_in_progress": len(in_progress),
        "in_progress_courses": in_progress
    }

# Test the analysis function
analysis = analyze_student_progress("S001")
print("\nStudent Progress Analysis:")
for key, value in analysis.items():
    print(f"  {key}: {value}")
```

5.4 System-Wide Analytics

Generate comprehensive reports using integrated data analysis:

```python
# System-wide analytics using all data structures

def generate_system_report():
    """Generate comprehensive system report using integrated data
       analysis."""

    # Collect all unique students across courses (set operations)
    all_enrolled_students = set()
    course_enrollment_stats = {}

    # Analyze course enrollments
    for course_code in course_catalog.keys():
        enrolled_in_course = set()

        # Check each student's enrollment (dictionary iteration)
        for student_id, student_data in student_database.items():
            enrolled_courses = student_data["academic_record"]["enrolled_courses"]
            if course_code in enrolled_courses:  # Set membership
                enrolled_in_course.add(student_id)
                all_enrolled_students.add(student_id)

        course_enrollment_stats[course_code] = len(enrolled_in_course)

    # Calculate major distribution
    major_counts = {}
    gpa_data = []  # List for GPA analysis

    for student_data in student_database.values():
        major = student_data["academic_record"]["major"]
        major_counts[major] = major_counts.get(major, 0) + 1

        # Collect GPA data (list processing)
        gpas = student_data["academic_record"]["semester_gpas"]
        if gpas:
            gpa_data.extend(gpas)

    # Calculate system-wide statistics
    avg_system_gpa = sum(gpa_data) / len(gpa_data) if gpa_data else 0.0

    return {
        "total_students": len(student_database),
        "total_courses": len(course_catalog),
        "course_enrollments": course_enrollment_stats,
        "major_distribution": major_counts,
        "system_average_gpa": round(avg_system_gpa, 2),
        "total_unique_enrollments": len(all_enrolled_students)
    }

# Generate and display system report
report = generate_system_report()
print("\n=== SYSTEM REPORT ===")
print(f"Total Students: {report['total_students']}")
print(f"Total Courses: {report['total_courses']}")
print(f"System Average GPA: {report['system_average_gpa']}")

print("\nCourse Enrollment Statistics:")
for course, count in report['course_enrollments'].items():
    course_name = course_catalog[course]['name']
    print(f"  {course} ({course_name}): {count} students")

print("\nMajor Distribution:")
for major, count in report['major_distribution'].items():
    print(f"  {major}: {count} students")
```

6 Key Concepts Summary

Advanced Data Structure Mastery:

Technical Achievements:

1. **Sequence Operations Mastery**: Unpacking patterns, advanced slicing, and del state-

ment usage

2. **Tuple Immutability Mastery**: Data integrity, multiple returns, unpacking patterns, and configuration management
3. **Dictionary Efficiency**: Key-value relationships, safe access with .get(), view operations, and nested structures
4. **Set Mathematical Operations**: Uniqueness enforcement, membership testing, union, intersection, difference operations
5. **Strategic Integration**: Combining structures based on performance requirements and data characteristics

Professional Problem-Solving Capabilities:
Performance Optimization:

- Selecting appropriate structures for O(1) lookups and efficient membership testing
- Understanding memory implications of mutable vs immutable structures
- Implementing data structure selection strategies based on requirements analysis

System Architecture Design:

- Designing integrated applications that leverage each structure's strengths
- Building scalable systems with appropriate data organization
- Implementing robust error handling and defensive programming techniques

Real-World Application Readiness:
Your comprehensive data structure understanding, combined with previous knowledge of functions, exception handling, and control structures, provides the foundation for:

- Database application development and API integration
- Web application backend systems and data processing pipelines
- Complex business logic implementation and system architecture design
- Advanced programming topics like object-oriented programming and framework development

Connection to Future Learning

Your Programming Evolution:

1. **Lectures 1-2**: Variables, operations, and basic control structures
2. **Lectures 3-4**: Lists, loops, and data processing with comprehensions
3. **Lecture 5**: Function-oriented programming with modular design
4. **Lecture 6**: Defensive programming with exception handling + Advanced list operations
5. **Lecture 7**: Advanced data structure mastery for complex problem solving

This progression has prepared you for professional software development where sophisticated data management is essential. Continue building projects that demonstrate integrated data structure usage, practicing design decisions based on requirements analysis rather than convenience.

You are now equipped for professional-level Python programming! Your solid foundation in data structures will support advanced topics like object-oriented programming, file processing, database integration, and framework development. Practice designing systems where structure selection is driven by systematic requirements analysis and performance considerations.