

# Chapter 5: STRUCTURED TYPES AND MUTABILITY

Python Fundamentals: Data Structures & Operations

CS 101 - Fall 2025

## On For Today

💡 Today's Journey: Python Data Structures & Operations

Let's explore some essential Python concepts!

Topics covered in today's discussion:

- **Tuples** - Immutable ordered collections
- **Ranges and Iterables** - Efficient number sequences
- **List Mutability** - Dynamic, changeable collections
- **List Cloning** - Creating independent copies safely

## Also On For Today

💡 Continued ...

- **List Comprehensions** - Elegant one-line list creation
- **Nested Lists** - Lists within lists for complex data
- **2D Lists** - Grid-based data structures (matrices, game boards)
- **Higher-Order Operations** - `map()` and `filter()` functions
- **Sequence Types** - Strings, Tuples, Ranges, and Lists comparison
- **Sets** - Unique collections with mathematical operations

Ready to master Python's most powerful data structures!

## Tuples: The Unchangeable Twins

### Definition

Tuples are ordered collections of items that are **immutable** (cannot be changed after creation). Think of them as containers that are sealed shut!

**Typical Use Case:** Storing coordinates, RGB color values, or any data that shouldn't change

## Tuples: Code Example

### Example

```
# Creating tuples
coordinates = (10, 20)
rgb_color = (255, 128, 0) # Orange color
student_info = ("Alice", 20, "Computer Science")

# Accessing elements
print(f"X coordinate: {coordinates[0]}") # Output: 10
print(f"Student name: {student_info[0]}") # Output: Alice
```

**Why this works:** Tuples use parentheses and are perfect when you need data that won't change, like a point on a map!

## Tuples: Concatenation, indexing, and slicing

### Example

```
# Like strings, tuples can be concatenated, indexed, and sliced.
t1 = (1, "two", 3)
t2 = (t1, 3.25) # Note, we include t1 here!
print(f" t2 --> {t2}")
print(f" (t1 + t2) --> {(t1 + t2)}")
print(f" (t1 + t2)[3] --> {(t1 + t2)[3]}")
print(f" (t1 + t2)[2:5] --> {(t1 + t2)[2:5]}")
```

```

# Like strings, tuples can be concatenated, indexed, and sliced.
t1 = (1, "two", 3)
t2 = (t1, 3.25) # Note, we include t1 here!
print(f" t2 --> {t2}")
print(f" (t1 + t2) --> {(t1 + t2)}")
print(f" (t1 + t2)[3] --> {(t1 + t2)[3]}")
print(f" (t1 + t2)[2:5] --> {(t1 + t2)[2:5]}")

t2 --> ((1, 'two', 3), 3.25)
(t1 + t2) --> (1, 'two', 3, (1, 'two', 3), 3.25)
(t1 + t2)[3] --> (1, 'two', 3)
(t1 + t2)[2:5] --> (3, (1, 'two', 3), 3.25)

```

## Ranges and Iterables

### Definition

**Ranges** generate sequences of numbers, while **iterables** are objects you can loop through one item at a time.

**Typical Use Case:** Creating number sequences for loops, generating test data, or creating patterns

## Ranges: Code Example

### Example: Different range patterns

```

numbers = range(5)           # 0, 1, 2, 3, 4
evens = range(2, 11, 2)       # 2, 4, 6, 8, 10
countdown = range(10, 0, -1)  # 10, 9, 8, ..., 1

# Using range() in for-loops
for i in range(3):
    print(f"Round {i + 1}!")

for i in countdown: print(i)

```

**Why this works:** Ranges are memory-efficient and perfect for creating predictable number sequences!

## Lists and Mutability

### Definition

**Lists** are ordered, **mutable** collections that can store different types of data and can be modified after creation.

**Typical Use Case:** Storing shopping lists, student grades, or any collection that needs to grow or change

### Lists: Code Example

#### Example

```
# Creating and modifying lists
fruits = ["apple", "banana", "orange"]
print(f"Original: {fruits}")

# Adding items
fruits.append("grape")
fruits.insert(1, "mango")
print(f"After adding: {fruits}")
```

**Why this works:** Lists are flexible containers that can grow, shrink, and change - perfect for dynamic data!

## Cloning Lists

### Definition

**Cloning lists** means creating independent copies so changes to one don't affect the other.

**Typical Use Case:** Backing up data before modifications, creating templates, or parallel processing

## List Cloning: The Wrong Way

Dangerous Example

```
original = [1, 2, 3, 4, 5]

# Wrong way (creates reference, not copy)
not_a_copy = original
not_a_copy.append(6)
print(f"Original changed too! {original}")
# Output: [1, 2, 3, 4, 5, 6] - Oops!
```

## List Cloning: The Right Way

Safe Examples

```
original = [1, 2, 3, 4, 5]

# Right ways to clone
copy1 = original.copy()
copy2 = original[:]
copy3 = list(original)

copy1.append(7)
print(f"Original safe: {original}")      # [1, 2, 3, 4, 5]
print(f"Copy modified: {copy1}")          # [1, 2, 3, 4, 5, 7]
```

**Why this works:** Proper cloning creates independent lists, preventing unwanted side effects!

## Lists: Another Cloning Example

### Clones

```
L1 = [1,2,3]
print(f"L1 --> {L1}")
L2 = L1
print(f" L2 is copy of L1 --> {L2}")

L1.append("100") # Modify L1
print(f"L1 with appended value--> {L1}")
print(f" L2 is copy of L1 --> {L2}")
L2.append("2000") # Modify L2
print(f"Appending to L2 modifies L1 = {L2}")
```

```
L1 = [1,2,3]
print(f"L1 --> {L1}")
L2 = L1
print(f" L2 is copy of L1 --> {L2}")

L1.append("100") # Modify L1
print(f"L1 with appended value--> {L1}")
print(f" L2 is copy of L1 --> {L2}")
L2.append("2000") # Modify L2
print(f"Appending to L2 modifies L1 = {L2}")
```

```
L1 --> [1, 2, 3]
L2 is copy of L1 --> [1, 2, 3]
L1 with appended value--> [1, 2, 3, '100']
L2 is copy of L1 --> [1, 2, 3, '100']
Appending to L2 modifies L1 = [1, 2, 3, '100', '2000']
```

## List Comprehensions

### Definition

**List comprehensions** provide a concise way to create lists using a single line of code with optional conditions.

**Typical Use Case:** Transforming data, filtering lists, or creating mathematical sequences efficiently

## List Comprehensions: Traditional vs Modern

### Comparison

```
# Traditional way
squares = []
for x in range(10):
    squares.append(x**2)

# List comprehension way - much cleaner!
squares = [x**2 for x in range(10)]
print(squares) # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

## List Comprehensions: With Conditions

### Advanced Examples

```
# With conditions
even_squares = [x**2 for x in range(10) if x % 2 == 0]
print(even_squares) # [0, 4, 16, 36, 64]

# String processing
words = ["hello", "world", "python", "rocks"]
caps = [word.upper() for word in words if len(word) > 4]
print(caps) # ['HELLO', 'WORLD', 'PYTHON', 'ROCKS']
```

**Why this works:** List comprehensions are Pythonic, readable, and often faster than traditional loops!

## Lists to Make Tuples

### Cartesian Formatting

```
list1 = [1, 2, 3]
list2 = ['a', 'b']

# Create a list of tuples combining elements from list1 and list2
combined_list = [(x, y) for x in list1 for y in list2]
print(combined_list)
```

```
list1 = [1, 2, 3]
list2 = ['a', 'b']

# Create a list of tuples combining elements from list1 and list2
combined_list = [(x, y) for x in list1 for y in list2]
print(combined_list)
```

```
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')]
```

## Nested Lists

### Definition

**Nested lists** are lists that contain other lists as elements, creating multi-dimensional data structures.

**Typical Use Case:** Storing hierarchical data, representing matrices, or organizing complex information

## Nested Lists: Code Example

Example

```
# Creating nested lists
shopping_lists = [
    ["apples", "bananas", "oranges"],      # Fruits
    ["carrots", "broccoli", "spinach"],     # Vegetables
    ["chicken", "beef", "fish"]            # Proteins
]

# Accessing nested elements
print(f"First fruit: {shopping_lists[0][0]}")      # apples
print(f"Second vegetable: {shopping_lists[1][1]}")  # broccoli
```

**Why this works:** Nested lists let us organize related data in logical groups, like folders in a filing cabinet!

## 2D Lists

Definition

**2D lists** are special nested lists arranged in rows and columns, like a spreadsheet or game board.

**Typical Use Case:** Representing game boards, matrices, pixel data, or any grid-based information

## 2D Lists: Tic-Tac-Toe Example

### Example

```
# Creating a 3x3 tic-tac-toe board
board = [
    [' ', ' ', ' '],
    [' ', ' ', ' '],
    [' ', ' ', ' ']
]
# Making moves
board[0][0] = 'X' # Top-left
board[1][1] = 'O' # Center
board[2][2] = 'X' # Bottom-right
```

**Why this works:** 2D lists give us row[column] access, perfect for grid-based data and games!

## Higher-Order Functions

### Definition

**Higher-order operations** are built-in functions like `map()`, `filter()`, and `reduce()` that work on entire collections.

**Typical Use Case:** Processing large datasets, functional programming patterns, and others

### Higher-Order: `map()` and `filter()`

### Example

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# map() - apply function to all elements
squares = list(map(lambda x: x**2, numbers))
print(f"Squares: {squares}")
# [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# filter() - keep elements that meet condition
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(f"Evens: {evens}") # [2, 4, 6, 8, 10]
```

**Why this works:** Higher-order functions let us process entire collections efficiently with functional programming!

## Higher Order Functions

### Function As Parameters

Add print statements to see what is happening in the code.

```
def apply_to_each(L,f):
    """ Assume L is a list, F is a function
    Mutates L by replacing each element, e, of L by f(e)"""
    for i in range(len(L)):
        print(f" position : {i}L[i] is now : {L[i]}")
        L[i] = f(L[i])

L = [1, -2, 3.33, -5]
print(f"1. L = {L}")
print("Apply ABS() to each element of L")
apply_to_each(L,abs)
print(f"2. L = {L}")
apply_to_each(L,int)
print(f"3. L = {L}")
apply_to_each(L,float)
print(f"4. L = {L}")
print(f"Apply squaring to each element of L : {L}")
apply_to_each(L, lambda x: x**2)
print(f"5. L = {L}")
```

```

def apply_to_each(L,f):
    """ Assume L is a list, F is a function
    Mutates L by replacing each element, e, of L by f(e)"""
    for i in range(len(L)):
        print(f" position : {i}L[i] is now : {L[i]}")
        L[i] = f(L[i])

L = [1, -2, 3.33, -5]
print(f"1. L = {L}")
print("Apply ABS() to each element of L")
apply_to_each(L,abs)
print(f"2. L = {L}")
apply_to_each(L,int)
print(f"3. L = {L}")
apply_to_each(L,float)
print(f"4. L = {L}")
print(f"Apply squaring to each element of L : {L}")
apply_to_each(L, lambda x: x**2)
print(f"5. L = {L}")

```

```

1. L = [1, -2, 3.33, -5]
Apply ABS() to each element of L
position : 0L[i] is now : 1
position : 1L[i] is now : -2
position : 2L[i] is now : 3.33
position : 3L[i] is now : -5
2. L = [1, 2, 3.33, 5]
position : 0L[i] is now : 1
position : 1L[i] is now : 2
position : 2L[i] is now : 3.33
position : 3L[i] is now : 5
3. L = [1, 2, 3, 5]
position : 0L[i] is now : 1
position : 1L[i] is now : 2
position : 2L[i] is now : 3
position : 3L[i] is now : 5
4. L = [1.0, 2.0, 3.0, 5.0]
Apply squaring to each element of L : [1.0, 2.0, 3.0, 5.0]
position : 0L[i] is now : 1.0
position : 1L[i] is now : 2.0
position : 2L[i] is now : 3.0

```

```
position : 3L[i] is now : 5.0
5. L = [1.0, 4.0, 9.0, 25.0]
```

The function `apply_to_each()` is called higher-order because it has an argument that is itself a function.

## Sequence Types

### Definition

**Strings, Tuples, Ranges, and Lists** are all **sequence types** that share common operations but have different characteristics and use cases.

**Typical Use Case:** Understanding when to use each type for optimal performance and code clarity

## Sequence Types: Common Operations

### Example

```
# Common operations across sequence types
text = "Hello"
numbers_tuple = (1, 2, 3, 4, 5)
numbers_range = range(1, 6)
numbers_list = [1, 2, 3, 4, 5]

# Indexing works on all
print(f"String[0]: {text[0]}")           # H
print(f"Tuple[1]: {numbers_tuple[1]}")    # 2
print(f"List[2]: {numbers_list[2]}")      # 3
```

## Sequence Types: Key Differences

Mutability Matters!

```
# Slicing works on all
print(f"String slice: {text[1:4]}")      # ell
print(f"List slice: {numbers_list[1:4]}") # [2, 3, 4]

# But mutability differs!
# text[0] = 'h'          # Error! Strings are immutable
# numbers_tuple[0] = 0   # Error! Tuples are immutable
numbers_list[0] = 0      # Works! Lists are mutable
```

**Why this works:** All sequences share similar operations, but mutability determines which operations are allowed!

## Sets

Definition

**Sets** are unordered collections of unique elements. No duplicates allowed!

**Typical Use Case:** Removing duplicates, finding common elements, or checking membership quickly

## Sets: Creating and Basic Operations

### Example

```
# Creating sets
colors1 = {"red", "green", "blue", "red"} # Duplicate "red" ignored
colors2 = {"blue", "yellow", "purple"}
numbers = set([1, 2, 2, 3, 3, 3, 4])      # From list

print(f"Colors1: {colors1}") # {'red', 'green', 'blue'}
print(f"Numbers: {numbers}") # {1, 2, 3, 4}

colors1.add("purple")
print(f"Add purple to Colors1: {colors1}") # {'red', 'green', 'blue', 'purple'}
colors1.add("purple")
print(f"Add purple to Colors1 (again): {colors1}") # {'red', 'green', 'blue', 'purple'}
```

Output:

```
# Creating sets
colors1 = {"red", "green", "blue", "red"} # Duplicate "red" ignored
colors2 = {"blue", "yellow", "purple"}
numbers = set([1, 2, 2, 3, 3, 3, 4])      # From list

print(f"Colors1: {colors1}") # {'red', 'green', 'blue'}
print(f"Numbers: {numbers}") # {1, 2, 3, 4}

colors1.add("purple")
print(f"Add purple to Colors1: {colors1}") # {'red', 'green', 'blue', 'purple'}
colors1.add("purple")
print(f"Add purple to Colors1 (again): {colors1}") # {'red', 'green', 'blue', 'purple'}
```

Colors1: {'red', 'green', 'blue'}  
Numbers: {1, 2, 3, 4}  
Add purple to Colors1: {'red', 'purple', 'green', 'blue'}  
Add purple to Colors1 (again): {'red', 'purple', 'green', 'blue'}

## Sets: Mathematical Operations

### Set Operations

```
# Creating sets
colors1 = {"red", "green", "blue", "red"} # Duplicate "red" ignored
colors2 = {"blue", "yellow", "purple"}
# Set operations
common = colors1 & colors2           # Intersection
all_colors = colors1 | colors2        # Union
unique_to_1 = colors1 - colors2       # Difference

print(f"Common colors: {common}")      # {'blue'}
print(f"All colors: {all_colors}")
# {'red', 'green', 'blue', 'yellow', 'purple'}

# Fast membership testing
print("red" in colors1) # True - very fast!
```

Output:

```
# Creating sets
colors1 = {"red", "green", "blue", "red"} # Duplicate "red" ignored
colors2 = {"blue", "yellow", "purple"}
# Set operations
common = colors1 & colors2           # Intersection
all_colors = colors1 | colors2        # Union
unique_to_1 = colors1 - colors2       # Difference

print(f"Common colors: {common}")      # {'blue'}
print(f"All colors: {all_colors}")
# {'red', 'green', 'blue', 'yellow', 'purple'}

# Fast membership testing
print("red" in colors1) # True - very fast!
```

```
Common colors: {'blue'}
All colors: {'yellow', 'purple', 'blue', 'red', 'green'}
True
```

**Why this works:** Sets automatically handle uniqueness and provide super-fast lookups and mathematical operations!