Python Programming Bootcamp

Week 2: Data Structures - Lists, Strings, Tuples, Dictionaries & Sets

Hilal Khan

Center for Artificial Intelligence & Emerging Technologies

What We'll Cover This Week

- Data Types vs Data Structures - Understanding the difference
- Lists - Ordered, mutable collections
- Strings - Text processing and manipulation
- Tuples - Immutable sequences
- Dictionaries - Key-value pairs
- Sets - Unique element collections
- Choosing the Right Data Structure - Comparison and use cases
- Practice Exercises - Hands-on challenges

Quick Recap: Week 1

Last week we learned:

- Python basics and syntax
- Variables and data types (int, float, str, bool)
- Operators and expressions
- Input/output operations
- Conditional statements (if/elif/else)
- Loops (for and while)

This week: We'll learn Data Structures - powerful ways to organize and store collections of data!

Data Types vs Data Structures

Understanding the difference:

Data Types (Week 1):

- Single values
- Basic building blocks
- Examples:
  - int: 42
  - float: 3.14
  - str: "Hello"
  - bool: True

Data Structures (This Week):

- Collections of values
- Organize related data
- Examples:
  - Lists: [1, 2, 3]
  - Tuples: (1, 2, 3)
  - Dictionaries: {"a": 1}
  - Sets: {1, 2, 3}
  - Strings: "Hello"

Key Difference

Data types store single values. Data structures store multiple values in organized ways!

Why Do We Need Data Structures?

Solving real-world problems:

Problem: Store Student Grades
Without Data Structures (Bad):

- grade1 = 85
- grade2 = 90
- grade3 = 78
- ... (What if 100 students?)

With Data Structures (Good):

- grades = [85, 90, 78, ...] (List)
- student = {"name": "Alice", "grade": 85} (Dictionary)

This Week's Goal
Learn to choose and use the right data structure for your problem!

Lists in Python

Introduction to Lists

Lists are ordered collections of items:

```
# Creating lists
empty_list = []
numbers = [1, 2, 3, 4, 5]
fruits = ["apple", "banana", "cherry"]
mixed = [1, "hello", 3.14, True]

# Lists can contain anything
nested = [[1, 2], [3, 4], [5, 6]]
```

Key Features

- Ordered - items maintain their position
- Mutable - can be changed after creation
- Allow duplicates - same value can appear multiple times
- Can contain any data type

Accessing List Elements i

Indexing (starts at 0):

```
fruits = ["apple", "banana", "cherry", "date"]

# Positive indexing (from start)
print(fruits[0])     # apple
print(fruits[1])     # banana
print(fruits[3])     # date

# Negative indexing (from end)
print(fruits[-1])    # date (last item)
print(fruits[-2])    # cherry (second from end)
```

Slicing - get multiple elements:

Accessing List Elements ii

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(numbers[2:5])      # [2, 3, 4]
print(numbers[:4])       # [0, 1, 2, 3] (from start)
print(numbers[6:])       # [6, 7, 8, 9] (to end)
print(numbers[::2])      # [0, 2, 4, 6, 8] (every 2nd)
print(numbers[::-1])     # Reverse the list
```

Modifying Lists i

Lists are mutable - we can change them:

```
fruits = ["apple", "banana", "cherry"]

# Change an element
fruits[1] = "blueberry"
print(fruits)  # ["apple", "blueberry", "cherry"]

# Change multiple elements
fruits[0:2] = ["apricot", "avocado"]
print(fruits)  # ["apricot", "avocado", "cherry"]
```

Adding elements:

Modifying Lists ii

```
numbers = [1, 2, 3]

# append() - add to end
numbers.append(4)
print(numbers)  # [1, 2, 3, 4]

# insert() - add at specific position
numbers.insert(0, 0)  # insert 0 at index 0
print(numbers)  # [0, 1, 2, 3, 4]

# extend() - add multiple items
numbers.extend([5, 6, 7])
print(numbers)  # [0, 1, 2, 3, 4, 5, 6, 7]
```

Removing List Elements i

Different ways to remove items:

```
fruits = ["apple", "banana", "cherry", "date"]

# remove() - remove by value (first occurrence)
fruits.remove("banana")
print(fruits)  # ["apple", "cherry", "date"]

# pop() - remove by index (returns the item)
removed = fruits.pop(1)  # Remove index 1
print(removed)  # cherry
print(fruits)   # ["apple", "date"]

# pop() without index removes last item
last = fruits.pop()
print(last)     # date
```

Removing List Elements ii

```
print(fruits)    # ["apple"]

# del - delete by index or slice
numbers = [1, 2, 3, 4, 5]
del numbers[2]      # Remove index 2
print(numbers)      # [1, 2, 4, 5]

# clear() - remove all items
numbers.clear()
print(numbers)      # []
```

Useful List Methods i

Common list operations:

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

# sort() - sort in place
numbers.sort()
print(numbers)  # [1, 1, 2, 3, 4, 5, 6, 9]

# reverse() - reverse in place
numbers.reverse()
print(numbers)  # [9, 6, 5, 4, 3, 2, 1, 1]

# count() - count occurrences
print(numbers.count(1))  # 2

# index() - find position of value
```

Useful List Methods ii

```
print(numbers.index(5))  # 2 (index of first 5)
```

List information:

```
fruits = ["apple", "banana", "cherry"]

# len() - get length
print(len(fruits))  # 3

# in - check membership
print("apple" in fruits)  # True
print("grape" in fruits)  # False

# min(), max(), sum()
nums = [5, 2, 8, 1, 9]
print(min(nums))  # 1
```

Useful List Methods iii

```
print(max(nums))   # 9
print(sum(nums))   # 25
```

Lists and Loops i

Combining lists with loops:

```
# Iterate over list items
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(f"I like {fruit}")

# Iterate with index using enumerate()
for index, fruit in enumerate(fruits):
    print(f"{index}: {fruit}")
# Output:
# 0: apple
# 1: banana
# 2: cherry
```

Modifying list in loop:

Lists and Loops ii

```
# Square all numbers
numbers = [1, 2, 3, 4, 5]
for i in range(len(numbers)):
    numbers[i] = numbers[i] ** 2
print(numbers)  # [1, 4, 9, 16, 25]

# Create new list from existing
original = [1, 2, 3, 4, 5]
doubled = []
for num in original:
    doubled.append(num * 2)
print(doubled)  # [2, 4, 6, 8, 10]
```

List Comprehensions i

Concise way to create lists:

Basic syntax:

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

# List comprehension (same result, one line)
squares = [x**2 for x in range(10)]
print(squares)  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

With conditions:

List Comprehensions ii

```
# Get only even numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = [x for x in numbers if x % 2 == 0]
print(evens)  # [2, 4, 6, 8, 10]

# Transform and filter
words = ["hello", "world", "python", "code"]
long_words_upper = [w.upper() for w in words if len(w)
    > 4]
print(long_words_upper)  # ['HELLO', 'WORLD', 'PYTHON
   ']
```

More examples:

List Comprehensions iii

```
# Create list of tuples
pairs = [(x, x**2) for x in range(5)]
print(pairs)  # [(0,0), (1,1), (2,4), (3,9), (4,16)]

# Nested comprehension
matrix = [[i*j for j in range(3)] for i in range(3)]
print(matrix)  # [[0,0,0], [0,1,2], [0,2,4]]
```

Strings in Detail

String Basics Recap

Strings are sequences of characters:

```
# Creating strings
single = 'Hello'
double = "World"
multiline = """This is a
multi-line string"""

# Strings are immutable
text = "Python"
# text[0] = "J"  # ERROR! Can't modify
# Must create new string instead
text = "J" + text[1:]  # "Jython"
```

Key Difference from Lists

Strings are immutable - once created, they cannot be changed. You must create a new string for any modification.

String Indexing and Slicing i

Works just like lists:

```
text = "Python Programming"

# Indexing
print(text[0])       # P
print(text[7])       # P (of Programming)
print(text[-1])      # g (last character)

# Slicing
print(text[0:6])     # Python
print(text[7:])      # Programming
print(text[:6])      # Python
print(text[::2])     # Pto rgamn (every 2nd char)
print(text[::-1])    # gnimmargorP nohtyP (reversed)
```

String Indexing and Slicing ii

Practical examples:

```
email = "user@example.com"

# Extract username
username = email[:email.index("@")]
print(username)  # user

# Extract domain
domain = email[email.index("@")+1:]
print(domain)  # example.com
```

String Methods - Case Conversion i

Changing case:

```
text = "Python Programming"

# upper() - convert to uppercase
print(text.upper())  # PYTHON PROGRAMMING

# lower() - convert to lowercase
print(text.lower())  # python programming

# title() - title case (capitalize each word)
print(text.title())  # Python Programming

# capitalize() - capitalize first letter only
print(text.capitalize())  # Python programming
```

String Methods - Case Conversion ii

```
# swapcase() - swap upper/lower
print(text.swapcase())  # pYTHON pROGRAMMING
```

Practical use:

```
# Case-insensitive comparison
user_input = input("Enter yes or no: ")
if user_input.lower() == "yes":
    print("User said yes!")
```

String Methods - Searching i

Finding substrings:

```
text = "Python is awesome. Python is powerful."

# find() - returns index or -1
print(text.find("Python"))       # 0
print(text.find("awesome"))      # 10
print(text.find("Java"))         # -1 (not found)

# index() - returns index or raises error
print(text.index("is"))          # 7
# print(text.index("Java"))      # ValueError!

# count() - count occurrences
print(text.count("Python"))      # 2
print(text.count("is"))          # 2
```

String Methods - Searching ii

```
# startswith() and endswith()
print(text.startswith("Python")) # True
print(text.endswith("powerful.")) # True
```

String Methods - Splitting and Joining i

split() - string to list:

```
# Split by spaces (default)
sentence = "Python is awesome"
words = sentence.split()
print(words)  # ['Python', 'is', 'awesome']

# Split by custom delimiter
date = "2024-03-15"
parts = date.split("-")
print(parts)  # ['2024', '03', '15']

# Split with limit
text = "a,b,c,d,e"
print(text.split(",", 2))  # ['a', 'b', 'c,d,e']
```

String Methods - Splitting and Joining ii

join() - list to string:

```
words = ["Python", "is", "awesome"]
sentence = " ".join(words)
print(sentence)  # Python is awesome

# Join with different separator
csv = ",".join(words)
print(csv)  # Python,is,awesome

# Join numbers (must be strings)
numbers = ["1", "2", "3"]
result = "-".join(numbers)
print(result)  # 1-2-3
```

String Methods - Cleaning i

Removing whitespace:

```
text = "   Python Programming   "

# strip() - remove from both ends
print(text.strip())          # "Python Programming"

# lstrip() - remove from left
print(text.lstrip())         # "Python Programming   "

# rstrip() - remove from right
print(text.rstrip())         # "   Python Programming"

# Remove specific characters
filename = "###file.txt###"
print(filename.strip("#")) # file.txt
```

String Methods - Cleaning ii

Replacing text:

```
text = "I like Java. Java is cool."

# replace() - replace substring
new_text = text.replace("Java", "Python")
print(new_text)  # I like Python. Python is cool.

# Replace with limit
text = "ha ha ha ha"
print(text.replace("ha", "he", 2))  # he he ha ha
```

String Methods - Checking i

Boolean check methods:

```
# isalpha() - all letters?
print("Python".isalpha())       # True
print("Python3".isalpha())      # False

# isdigit() - all digits?
print("123".isdigit())          # True
print("12.3".isdigit())         # False

# isalnum() - letters or digits?
print("Python3".isalnum())      # True
print("Python 3".isalnum())     # False (space)

# isspace() - all whitespace?
print("   ".isspace())          # True
```

String Methods - Checking ii

```
print(" a ".isspace())          # False
```

Case checking:

```
# isupper() - all uppercase?
print("PYTHON".isupper())       # True
print("Python".isupper())       # False

# islower() - all lowercase?
print("python".islower())       # True

# istitle() - title case?
print("Python Programming".istitle())  # True
print("Python programming".istitle())  # False
```

String Formatting Review

Different formatting methods:

```
name = "Alice"
age = 25
score = 95.5

# f-strings (recommended)
print(f"{name} is {age} years old")
print(f"Score: {score:.1f}%")

# format() method
print("{} is {} years old".format(name, age))
print("Score: {:.1f}%".format(score))

# % operator (old style)
print("%s is %d years old" % (name, age))

# String concatenation
print(name + " is " + str(age) + " years old")
```

Tuples

Introduction to Tuples i

Tuples are immutable sequences:

```
# Creating tuples
empty_tuple = ()
single_item = (5,)  # Note the comma!
coordinates = (10, 20)
person = ("Alice", 25, "Engineer")
mixed = (1, "hello", 3.14, True)

# Tuples can be created without parentheses
point = 3, 4
print(point)  # (3, 4)
```

Introduction to Tuples ii

Key Features

- Ordered - items maintain their position
- Immutable - cannot be changed after creation
- Allow duplicates - same value can appear multiple times
- Faster than lists for fixed data

Tuples vs Lists i

Main difference: Mutability

```
# Lists are mutable
my_list = [1, 2, 3]
my_list[0] = 10   # Works fine
print(my_list)    # [10, 2, 3]

# Tuples are immutable
my_tuple = (1, 2, 3)
# my_tuple[0] = 10   # ERROR! TypeError
```

When to use tuples:

- Data that shouldn't change (constants)
- Function return values

Tuples vs Lists ii

- Dictionary keys (lists can't be keys)
- Better performance for fixed data

Accessing Tuple Elements i

Works just like lists:

```
colors = ("red", "green", "blue", "yellow")

# Indexing
print(colors[0])     # red
print(colors[-1])    # yellow (last item)

# Slicing
print(colors[1:3])   # ('green', 'blue')
print(colors[:2])    # ('red', 'green')
print(colors[::2])   # ('red', 'blue')

# Length
print(len(colors))   # 4
```

Accessing Tuple Elements ii

```
# Membership
print("red" in colors)     # True
print("purple" in colors) # False
```

Finding elements:

```
numbers = (5, 2, 8, 2, 9, 2)

# index() - find position
print(numbers.index(8))  # 2

# count() - count occurrences
print(numbers.count(2))  # 3
```

Tuple Packing and Unpacking i

Packing - creating tuples:

```
# Automatic packing
coordinates = 10, 20, 30
print(coordinates)  # (10, 20, 30)

person = "Alice", 25, "Engineer"
print(person)  # ('Alice', 25, 'Engineer')
```

Unpacking - extracting values:

Tuple Packing and Unpacking ii

```
# Basic unpacking
x, y, z = (10, 20, 30)
print(x)  # 10
print(y)  # 20
print(z)  # 30

# Unpacking from function return
def get_coordinates():
    return 100, 200

x, y = get_coordinates()
print(f"x={x}, y={y}")  # x=100, y=200
```

Advanced unpacking:

Tuple Packing and Unpacking iii

```
# Using * to capture remaining items
first, *rest = (1, 2, 3, 4, 5)
print(first)  # 1
print(rest)   # [2, 3, 4, 5]

# Ignoring values with _
name, _, job = ("Alice", 25, "Engineer")
print(f"{name} is a {job}")  # Alice is a Engineer

# Swapping variables
a = 5
b = 10
a, b = b, a
print(f"a={a}, b={b}")  # a=10, b=5
```

Tuple Operations i

Combining and repeating:

```
# Concatenation
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
combined = tuple1 + tuple2
print(combined)  # (1, 2, 3, 4, 5, 6)

# Repetition
colors = ("red", "blue")
repeated = colors * 3
print(repeated)  # ('red', 'blue', 'red', 'blue', 'red
    ', 'blue')

# min, max, sum (for numeric tuples)
numbers = (5, 2, 8, 1, 9)
```

Tuple Operations ii

```
print(min(numbers))   # 1
print(max(numbers))   # 9
print(sum(numbers))   # 25
```

Tuples as Return Values i

Return multiple values from functions:

```
def get_student_info():
    name = "Alice"
    age = 20
    grade = 'A'
    return name, age, grade  # Returns a tuple

# Unpack the return values
student_name, student_age, student_grade =
    get_student_info()
print(f"{student_name}, {student_age}, Grade: {
    student_grade}")

# Or use as tuple
info = get_student_info()
```

Tuples as Return Values ii

```
print(info)  # ('Alice', 20, 'A')
```

Practical Example

```
def calculate_stats(numbers):
    total = sum(numbers)
    avg = total / len(numbers)
    return total, avg, min(numbers), max(numbers)

total, average, minimum, maximum = calculate_stats([1,
    2, 3, 4, 5])
print(f"Total: {total}, Avg: {average}")
```

Dictionaries

Introduction to Dictionaries i

Dictionaries store key-value pairs:

```
# Creating dictionaries
empty_dict = {}
person = {
    "name": "Alice",
    "age": 25,
    "job": "Engineer"
}

# Using dict() constructor
student = dict(name="Bob", age=20, grade="A")

# Mixed data types
mixed = {
    "name": "Alice",
```

Introduction to Dictionaries ii

```
    "scores": [95, 87, 92],
    "active": True,
    123: "numeric key"
}
```

Key Features

- Unordered (before Python 3.7) / Ordered (Python 3.7+)
- Mutable - can be changed after creation
- Keys must be unique and immutable
- Very fast lookup by key

Accessing Dictionary Values i

Multiple ways to access values:

```
person = {
    "name": "Alice",
    "age": 25,
    "job": "Engineer"
}

# Using square brackets
print(person["name"])  # Alice
# print(person["salary"])  # KeyError!

# Using get() - safer
print(person.get("name"))       # Alice
print(person.get("salary"))     # None (no error)
print(person.get("salary", 0)) # 0 (default value)
```

Accessing Dictionary Values ii

Checking for keys:

```
# in operator
if "age" in person:
    print(f"Age: {person['age']}")

# Check if key doesn't exist
if "salary" not in person:
    print("Salary information not available")

# Get all keys, values, items
print(person.keys())    # dict_keys(['name', 'age', '
    job'])
print(person.values())  # dict_values(['Alice', 25, '
    Engineer'])
print(person.items())   # dict_items([('name', 'Alice
    '), ...])
```

Accessing Dictionary Values iii

Modifying Dictionaries i

Adding and updating:

```
person = {"name": "Alice", "age": 25}

# Add new key-value pair
person["job"] = "Engineer"
print(person)  # {'name': 'Alice', 'age': 25, 'job': '
    Engineer'}

# Update existing value
person["age"] = 26
print(person)  # {'name': 'Alice', 'age': 26, 'job': '
    Engineer'}

# Update multiple items
person.update({"age": 27, "city": "New York"})
```

Modifying Dictionaries ii

```
print(person)
```

Removing items:

```
person = {"name": "Alice", "age": 25, "job": "Engineer
    "}

# pop() - remove and return value
job = person.pop("job")
print(job)      # Engineer
print(person)   # {'name': 'Alice', 'age': 25}

# popitem() - remove last item (Python 3.7+)
item = person.popitem()
print(item)     # ('age', 25)
```

Modifying Dictionaries iii

```
# del - remove specific key
del person["name"]

# clear() - remove all items
person.clear()
print(person)  # {}
```

Iterating Over Dictionaries i

Different ways to loop through dictionaries:

```
person = {"name": "Alice", "age": 25, "job": "Engineer
    "}

# Iterate over keys (default)
for key in person:
    print(key)  # name, age, job

# Iterate over keys explicitly
for key in person.keys():
    print(f"{key}: {person[key]}")

# Iterate over values
for value in person.values():
    print(value)  # Alice, 25, Engineer
```

Iterating Over Dictionaries ii

```
# Iterate over key-value pairs
for key, value in person.items():
    print(f"{key} = {value}")
# Output:
# name = Alice
# age = 25
# job = Engineer
```

Practical example:

Iterating Over Dictionaries iii

```
# Count word frequencies
text = "hello world hello python python python"
words = text.split()

frequency = {}
for word in words:
    if word in frequency:
        frequency[word] += 1
    else:
        frequency[word] = 1

for word, count in frequency.items():
    print(f"{word}: {count}")
# Output:
# hello: 2
```

Iterating Over Dictionaries iv

```
# world: 1
# python: 3
```

Dictionary Comprehensions i

Create dictionaries concisely:

```
# Basic comprehension
squares = {x: x**2 for x in range(6)}
print(squares)  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5:
    25}

# With condition
even_squares = {x: x**2 for x in range(10) if x % 2 ==
    0}
print(even_squares)  # {0: 0, 2: 4, 4: 16, 6: 36, 8:
    64}

# From two lists
keys = ["name", "age", "job"]
values = ["Alice", 25, "Engineer"]
```

Dictionary Comprehensions ii

```
person = {k: v for k, v in zip(keys, values)}
print(person)  # {'name': 'Alice', 'age': 25, 'job': '
    Engineer'}

# Transform existing dictionary
person = {"name": "alice", "job": "engineer"}
uppercase = {k: v.upper() for k, v in person.items()}
print(uppercase)  # {'name': 'ALICE', 'job': 'ENGINEER
    '}
```

Nested Dictionaries i

Dictionaries can contain other dictionaries:

```
# Student database
students = {
    "S001": {
        "name": "Alice",
        "age": 20,
        "grades": [95, 87, 92]
    },
    "S002": {
        "name": "Bob",
        "age": 21,
        "grades": [88, 90, 85]
    }
}
```

Nested Dictionaries ii

```
# Accessing nested data
print(students["S001"]["name"])  # Alice
print(students["S002"]["grades"][0])  # 88

# Adding nested data
students["S003"] = {
    "name": "Charlie",
    "age": 19,
    "grades": [91, 93, 89]
}
```

Iterating nested dictionaries:

Nested Dictionaries iii

```
# Print all student information
for student_id, info in students.items():
    print(f"\nStudent ID: {student_id}")
    print(f"Name: {info['name']}")
    print(f"Age: {info['age']}")
    print(f"Average Grade: {sum(info['grades'])/len(
    info['grades']):.2f}")
```

Common Dictionary Methods i

Useful dictionary operations:

```
person = {"name": "Alice", "age": 25}

# setdefault() - get or set default
job = person.setdefault("job", "Unknown")
print(job)      # Unknown
print(person)   # {'name': 'Alice', 'age': 25, 'job': '
    Unknown'}

# fromkeys() - create dict from sequence
keys = ["a", "b", "c"]
default_dict = dict.fromkeys(keys, 0)
print(default_dict)  # {'a': 0, 'b': 0, 'c': 0}

# copy() - shallow copy
```

Common Dictionary Methods ii

```
original = {"name": "Alice", "age": 25}
copy_dict = original.copy()
copy_dict["age"] = 30
print(original["age"])  # 25 (unchanged)

# len() - number of key-value pairs
print(len(person))  # 3
```

Sets

Introduction to Sets i

Sets are unordered collections of unique elements:

```
# Creating sets
empty_set = set()  # Note: {} creates empty dict!
numbers = {1, 2, 3, 4, 5}
mixed = {1, "hello", 3.14, True}

# From list (removes duplicates)
my_list = [1, 2, 2, 3, 3, 3, 4]
my_set = set(my_list)
print(my_set)  # {1, 2, 3, 4}

# From string (unique characters)
letters = set("hello")
print(letters)  # {'h', 'e', 'l', 'o'}
```

Introduction to Sets ii

Key Features

- Unordered - no indexing or slicing
- Unique elements - duplicates automatically removed
- Mutable - can add/remove items
- Fast membership testing

Basic Set Operations i

Adding and removing elements:

```
fruits = {"apple", "banana"}

# add() - add one element
fruits.add("cherry")
print(fruits)  # {'apple', 'banana', 'cherry'}

# update() - add multiple elements
fruits.update(["orange", "grape"])
print(fruits)  # {'apple', 'banana', 'cherry', 'orange
    ', 'grape'}

# remove() - remove element (error if not found)
fruits.remove("banana")
# fruits.remove("kiwi")  # KeyError!
```

Basic Set Operations ii

```
# discard() - remove element (no error if not found)
fruits.discard("kiwi")  # No error

# pop() - remove and return arbitrary element
item = fruits.pop()
print(f"Removed: {item}")

# clear() - remove all elements
fruits.clear()
print(fruits)  # set()
```

Set Mathematical Operations i

Union - all elements from both sets:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Using | operator
union = set1 | set2
print(union)  # {1, 2, 3, 4, 5}

# Using union() method
union = set1.union(set2)
print(union)  # {1, 2, 3, 4, 5}
```

Intersection - common elements:

Set Mathematical Operations ii

```
# Using & operator
intersection = set1 & set2
print(intersection)  # {3}

# Using intersection() method
intersection = set1.intersection(set2)
print(intersection)  # {3}
```

Difference - elements in first but not second:

Set Mathematical Operations iii

```
# Using - operator
difference = set1 - set2
print(difference)  # {1, 2}

# Using difference() method
difference = set1.difference(set2)
print(difference)  # {1, 2}
```

Symmetric difference - elements in either but not both:

Set Mathematical Operations iv

```
# Using ^ operator
sym_diff = set1 ^ set2
print(sym_diff)  # {1, 2, 4, 5}

# Using symmetric_difference() method
sym_diff = set1.symmetric_difference(set2)
print(sym_diff)  # {1, 2, 4, 5}
```

Set Comparison Operations i

Checking relationships between sets:

```
set1 = {1, 2, 3}
set2 = {1, 2, 3, 4, 5}
set3 = {6, 7, 8}

# Subset - all elements in another set
print(set1.issubset(set2))    # True
print(set1 <= set2)            # True

# Superset - contains all elements of another set
print(set2.issuperset(set1))  # True
print(set2 >= set1)            # True

# Disjoint - no common elements
print(set1.isdisjoint(set3))  # True
```

Set Comparison Operations ii

```
print(set1.isdisjoint(set2))  # False
```

Set Comprehensions i

Create sets concisely:

```
# Basic comprehension
squares = {x**2 for x in range(10)}
print(squares)  # {0, 1, 64, 4, 36, 9, 16, 49, 81, 25}

# With condition
even_squares = {x**2 for x in range(10) if x % 2 == 0}
print(even_squares)  # {0, 64, 4, 16, 36}

# From string - unique characters
text = "hello world"
unique_chars = {char for char in text if char != ' '}
print(unique_chars)  # {'h', 'e', 'l', 'o', 'w', 'r',
    'd'}
```

Set Comprehensions ii

```
# Multiple conditions
filtered = {x for x in range(20) if x % 2 == 0 if x %
    3 == 0}
print(filtered)  # {0, 6, 12, 18}
```

Practical Uses of Sets i

Remove duplicates:

```
# Remove duplicates from list
numbers = [1, 2, 2, 3, 3, 3, 4, 4, 5]
unique = list(set(numbers))
print(unique)  # [1, 2, 3, 4, 5]
```

Fast membership testing:

```
# Sets are much faster for 'in' operations
large_list = list(range(1000000))
large_set = set(large_list)

# This is much faster with set
999999 in large_set  # Very fast
999999 in large_list  # Slower for large lists
```

Practical Uses of Sets ii

Find common elements:

```
students_math = {"Alice", "Bob", "Charlie"}
students_physics = {"Bob", "Diana", "Charlie"}

both_classes = students_math & students_physics
print(both_classes)  # {'Bob', 'Charlie'}
```

Choosing the Right Data Structure

Data Structure Comparison

When to use each data structure:

Lists

- Ordered collection that can change
- Need to access by index
- Allow duplicates
- Example: Shopping list, sequence of events

Tuples

- Ordered collection that shouldn't change
- Return multiple values from functions
- Use as dictionary keys
- Example: Coordinates, RGB colors, fixed configurations

Data Structure Comparison (cont.)

Dictionaries

- Key-value pairs
- Fast lookup by key
- Descriptive data access
- Example: Student records, configuration settings, JSON data

Sets

- Unique elements only
- Fast membership testing
- Mathematical set operations
- Example: Unique tags, removing duplicates, set theory problems

Performance Comparison i

Time complexity for common operations:

Operation	List	Tuple	Dict	Set
Access by index	O(1)	O(1)	-	-
Search (in)	O(n)	O(n)	O(1)	O(1)
Add/Insert	O(n)	-	O(1)	O(1)
Delete	O(n)	-	O(1)	O(1)

Key Takeaway

Use dictionaries and sets for fast lookups. Use lists when order matters and you need to modify the collection. Use tuples for fixed data.

Week 2 Summary

What We Learned This Week

Data Structures Covered:

- **Lists:** Ordered, mutable collections - [1, 2, 3]
- **Strings:** Immutable text sequences - "Hello"
- **Tuples:** Ordered, immutable sequences - (1, 2, 3)
- **Dictionaries:** Key-value pairs - "name": "Alice"
- **Sets:** Unordered, unique elements - 1, 2, 3

Key Takeaway
Each data structure has its strengths. Choose the right one based on your needs: mutability, ordering, uniqueness, and access patterns.

Practice Exercises

Exercise 1: Fibonacci Sequence i

Generate the first n Fibonacci numbers:

Task
Write a program that generates the first n numbers in the Fibonacci sequence, where each number is the sum of the two preceding ones.

Hint:

```
# Start with [0, 1]
# Each new number = sum of last two
# Use a loop and append to list
```

Example output:

```
First 10 Fibonacci numbers:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Exercise 2: Palindrome Checker i

Check if a string is a palindrome:

Task
Write a function that checks if a given string reads the same forwards and backwards (ignoring spaces and case).

Hint:

```
# Remove spaces and convert to lowercase
# Compare with reversed string
# Use slicing [::-1] to reverse
```

Example:

Exercise 2: Palindrome Checker ii

```
"racecar" -> True
"A man a plan a canal Panama" -> True
"hello" -> False
```

Week 2 Skills Checklist

You should now be able to:

- Create and manipulate lists
- Work with strings and string methods
- Use tuples for immutable data
- Store data in dictionaries with key-value pairs
- Perform set operations for unique collections
- Choose the appropriate data structure
- Apply data structures to solve real problems

Master the Fundamentals
Understanding data structures is crucial for writing efficient Python code!

Next Week Preview

Week 3: Functions & Modules

- Defining and calling functions
- Function parameters and return values
- Scope and namespaces
- Docstrings and documentation
- Modules and imports
- Built-in functions
- Creating reusable code

Homework

- Complete all practice exercises
- Build a mini-project using different data structures
- Experiment with dictionary and set operations
- Review all data structure comparisons

Thank You!

Thank you for your attention!

Keep Practicing!

Access Course Materials:

Download Course Materials

Center for Artificial Intelligence & Emerging Technologies