Data Types and Collections

Python's Built-in Data Structures

Table of contents

1 Data Types & Collections

1.1 Python's Core Data Structures

- **Lists** - Ordered, mutable collections
- **Tuples** - Ordered, immutable collections

- **Dictionaries** - Key-value mappings
- **Sets** - Unique elements, fast lookup

1.2 Lists - Dynamic Arrays

```
# Creating and manipulating lists
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]

# Adding elements
fruits.append("orange")
fruits.insert(1, "grape")

# Accessing elements
first_fruit = fruits[0]      # "apple"
last_fruit = fruits[-1]      # "orange"

# Slicing
first_three = fruits[:3]
```

Key Features: Ordered, mutable, allow duplicates

1.3 List Operations & Methods

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

# Common operations
print(len(numbers))          # Length: 8
print(sum(numbers))          # Sum: 31
print(max(numbers))          # Maximum: 9

# Sorting and reversing
numbers.sort()               # [1, 1, 2, 3, 4, 5, 6, 9]
numbers.reverse()            # [9, 6, 5, 4, 3, 2, 1, 1]

# List comprehensions
squares = [x**2 for x in range(1, 6)]
# Result: [1, 4, 9, 16, 25]
```

1.4 Tuples - Immutable Sequences

```python
# Creating tuples
point = (10, 20)
rgb_color = (255, 128, 0)
person = ("Alice", 25, "Engineer")

# Tuple unpacking
name, age, job = person
print(f"{name} is {age} years old")

# Tuples are immutable
# point[0] = 15  # This would cause an error!

# But you can create new tuples
new_point = (point[0] + 5, point[1] + 3)
```

Perfect for: Coordinates, RGB values, database records

1.5 Dictionaries - Key-Value Power

```python
# Creating dictionaries
student = {
    "name": "Alice",
    "age": 20,
    "major": "Computer Science",
    "gpa": 3.8
}

# Accessing and modifying
print(student["name"])              # "Alice"
student["graduation_year"] = 2025   # Add new key
student["age"] = 21                 # Update existing

# Safe access with get()
email = student.get("email", "Not provided")

# Iterating
for key, value in student.items():
    print(f"{key}: {value}")
```

1.6 Dictionary Methods & Operations

```python
inventory = {
    "apples": 50,
    "bananas": 30,
    "oranges": 25
}

# Useful methods
print(inventory.keys())    # dict_keys(['apples', 'bananas', 'oranges'])
print(inventory.values())  # dict_values([50, 30, 25])

# Membership testing
if "apples" in inventory:
    print(f"We have {inventory['apples']} apples")

# Dictionary comprehension
doubled = {k: v*2 for k, v in inventory.items()}
```

1.7 Sets - Unique Collections

```
# Creating sets
fruits = {"apple", "banana", "cherry"}
numbers = {1, 2, 3, 4, 5}

# Adding elements
fruits.add("orange")

# Sets automatically handle duplicates
mixed = {1, 2, 2, 3, 3, 3}
print(mixed)  # {1, 2, 3}

# Set operations
set_a = {1, 2, 3, 4, 5}
set_b = {4, 5, 6, 7, 8}

union = set_a | set_b           # {1, 2, 3, 4, 5, 6, 7, 8}
intersection = set_a & set_b    # {4, 5}
difference = set_a - set_b      # {1, 2, 3}
```

1.8 Choosing the Right Data Structure

Structure	When to Use	Example
List	Ordered, mutable collection	Shopping list, scores
Tuple	Ordered, immutable data	Coordinates, RGB
Dictionary	Key-value lookup	User profiles, settings
Set	Unique items, fast lookup	Tags, unique IDs

1.9 Real-World Example: Student System

```
# Student management using different data structures
students = []  # List of student records

def add_student(name, age, grades):
    student = {
```

```
        "name": name,           # Dictionary for structured data
        "age": age,
        "grades": tuple(grades), # Tuple for immutable grade history
        "subjects": set(),       # Set for unique subjects
        "average": sum(grades) / len(grades)
    }
    students.append(student)

# Usage
add_student("Alice", 20, [95, 87, 92])
add_student("Bob", 19, [78, 84, 88])

# Find top student
top_student = max(students, key=lambda s: s["average"])
print(f"Top student: {top_student['name']}")
```

1.10 List Comprehensions - Powerful & Pythonic

```
# Traditional approach
squares = []
for x in range(1, 6):
    squares.append(x**2)

# List comprehension (Pythonic!)
squares = [x**2 for x in range(1, 6)]

# With conditions
even_squares = [x**2 for x in range(1, 11) if x % 2 == 0]
# Result: [4, 16, 36, 64, 100]

# Working with strings
words = ["hello", "world", "python"]
uppercase = [word.upper() for word in words]
lengths = [len(word) for word in words]
```

1.11 Dictionary & Set Comprehensions

```
# Dictionary comprehension
squares_dict = {x: x**2 for x in range(1, 6)}
# Result: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# Filter existing dictionary
prices = {"apple": 1.2, "banana": 0.8, "orange": 1.5}
expensive = {k: v for k, v in prices.items() if v > 1.0}

# Set comprehension
vowels_in_words = {char for word in ["hello", "world"]
                   for char in word if char in "aeiou"}
# Result: {'e', 'o'}
```

1.12 Performance Considerations

Lists - Append: O(1) - Insert: O(n) - Search: O(n) - Good for: Sequential access

Sets & Dicts - Add/Remove: O(1) - Lookup: O(1) - Good for: Fast membership testing

```
# Fast membership testing
large_set = set(range(10000))
large_list = list(range(10000))

# Set lookup is much faster!
print(9999 in large_set)    # Very fast
print(9999 in large_list)   # Slower for large lists
```

1.13 Common Patterns & Best Practices

1. **Use tuples for immutable data** (coordinates, configurations)
2. **Use sets for membership testing** and eliminating duplicates
3. **Use dict.get()** for safe key access
4. **List comprehensions** over loops when possible
5. **Choose the right tool** for the job

1.14 Practical Exercise

Create a word frequency analyzer:

```
def word_frequency(text):
    words = text.lower().split()
    frequency = {}

    for word in words:
        frequency[word] = frequency.get(word, 0) + 1

    return frequency

# Test it
text = "python is great python is powerful"
result = word_frequency(text)
print(result)  # {'python': 2, 'is': 2, 'great': 1, 'powerful': 1}

# Find most common word
most_common = max(result.items(), key=lambda x: x[1])
```

1.15 Key Takeaways

1. **Lists**: Ordered, mutable, perfect for sequences
2. **Tuples**: Ordered, immutable, great for fixed data
3. **Dictionaries**: Key-value mapping, fast lookup
4. **Sets**: Unique elements, excellent for membership testing
5. **Comprehensions**: Pythonic way to create collections
6. **Performance matters**: Choose the right data structure

1.16 What's Next?

- **Control Flow** - Making decisions with if/else
- **Loops** - Repeating actions efficiently
- **Functions** - Organizing code into reusable blocks
- **Real Projects** - Building applications with data structures

---

1.17 Navigation

Previous:
Introduction & Setup

Home:
Slides Index | Book

Next:
Control Flow

1.18 Thank You!

1.18.1 Questions?

Ready to continue with Control Flow?

Practice more: - Build a contact book with dictionaries - Create a shopping cart with lists - Implement a tag system with sets