Python Data Types and Data Structures

Complete Reference Guide

Learning Objectives

By the end of this session, students will be able to:

1. Understand Python's built-in data types
2. Create and manipulate different data structures
3. Choose the appropriate data structure for specific tasks
4. Apply methods and operations for each data type
5. Understand mutability vs immutability concepts

PART 1: PYTHON DATA TYPES OVERVIEW

1.1 What are Data Types?

Data types define the kind of data that can be stored and manipulated within a program. Python has several built-in data types that are automatically recognized.

Python's Built-in Data Types

```python
# Numeric Types
integer_num = 42                    # int
float_num = 3.14159                # float
complex_num = 3 + 4j               # complex

# Text Type
text = "Hello, World!"             # str

# Boolean Type
is_true = True                     # bool
is_false = False                   # bool

# Check data type
print(type(integer_num))           # <class 'int'>
print(type(text))                  # <class 'str'>
```

Dynamic Typing in Python

python

```
# Python is dynamically typed
x = 10         # x is an integer
print(type(x))  # <class 'int'>

x = "Hello"    # Now x is a string
print(type(x))  # <class 'str'>

x = [1, 2, 3] # Now x is a list
print(type(x))  # <class 'list'>
```

PART 2: SEQUENTIAL TYPES

Sequential types are ordered collections where items can be accessed by index position.

2.1 Strings (str)

What are Strings?

- Sequence of characters
- Immutable (cannot be changed after creation)
- Enclosed in quotes (single, double, or triple)

Creating Strings

python

```
# Different ways to create strings
single_quote = 'Hello World'
double_quote = "Hello World"
triple_quote = """This is a
multi-line string"""

# Escape characters
escaped = "She said, \"Hello!\""
newline = "Line 1\nLine 2"
tab = "Column1\tColumn2"
```

String Operations

python

```
text = "Python Programming"

# Basic operations
print(len(text))           # 18 (length)
print(text[0])             # 'P' (first character)
print(text[-1])            # 'g' (last character)
print(text[0:6])           # 'Python' (slicing)
print(text[::2])           # 'Pto rgamn' (every 2nd character)

# String methods
print(text.upper())        # 'PYTHON PROGRAMMING'
print(text.lower())        # 'python programming'
print(text.title())        # 'Python Programming'
print(text.replace('Python', 'Java'))  # 'Java Programming'
print(text.split())        # ['Python', 'Programming']
print('---'.join(['a', 'b', 'c']))     # 'a---b---c'
```

String Formatting

python

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

# f-strings (Python 3.6+) - Recommended
message = f"Hello {name}, you are {age} years old and scored {score:.1f}%"

# .format() method
message = "Hello {}, you are {} years old".format(name, age)

# % formatting (older style)
message = "Hello %s, you scored %.1f%%" % (name, score)

print(message)
```

Common String Methods

```python
text = "  Python Programming  "

# Checking methods
print(text.startswith("Python"))   # False (has spaces)
print(text.strip().startswith("Python"))  # True
print(text.endswith("ing"))        # False (has spaces)
print(text.isdigit())              # False
print("123".isdigit())             # True
print(text.isalpha())              # False

# Cleaning methods
print(text.strip())                # Remove whitespace
print(text.lstrip())               # Remove left whitespace
print(text.rstrip())               # Remove right whitespace

# Finding and counting
print(text.find("gram"))           # Returns index of first occurrence
print(text.count("m"))             # Count occurrences
```

🎯 **Practice Exercise**: Create a string with your full name and extract first and last name using string methods.

2.2 Lists

What are Lists?

- Ordered collection of items
- Mutable (can be modified)
- Can contain different data types
- Use square brackets `[]`

Creating Lists

python

```
# Empty list
empty_list = []
empty_list2 = list()

# Lists with data
numbers = [1, 2, 3, 4, 5]
fruits = ['apple', 'banana', 'orange']
mixed = [1, 'hello', 3.14, True, [1, 2, 3]]

# List from range
range_list = list(range(1, 11))  # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Accessing List Elements

python

```
fruits = ['apple', 'banana', 'orange', 'grape', 'kiwi']

# Indexing
print(fruits[0])        # 'apple' (first element)
print(fruits[-1])       # 'kiwi' (last element)
print(fruits[-2])       # 'grape' (second to last)

# Slicing
print(fruits[1:4])      # ['banana', 'orange', 'grape']
print(fruits[:3])       # ['apple', 'banana', 'orange']
print(fruits[2:])       # ['orange', 'grape', 'kiwi']
print(fruits[::2])      # ['apple', 'orange', 'kiwi'] (every 2nd)
print(fruits[::-1])     # Reverse the list
```

Modifying Lists

python

```
fruits = ['apple', 'banana', 'orange']

# Adding elements
fruits.append('grape')              # Add to end
fruits.insert(1, 'kiwi')           # Insert at position
fruits.extend(['mango', 'peach'])  # Add multiple items

# Removing elements
fruits.remove('banana')             # Remove first occurrence
popped = fruits.pop()               # Remove and return last
popped_index = fruits.pop(0)        # Remove and return at index
del fruits[1]                       # Delete by index
fruits.clear()                      # Remove all elements

# Modifying elements
fruits[0] = 'pineapple'            # Change single element
fruits[1:3] = ['cherry', 'plum']   # Change multiple elements
```

List Methods and Operations

python

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

# Sorting and reversing
numbers.sort()                      # Sort in place
numbers.sort(reverse=True)          # Sort descending
numbers.reverse()                   # Reverse in place

# Don't modify original
sorted_nums = sorted(numbers)       # Returns new sorted list
reversed_nums = list(reversed(numbers))  # Returns new reversed list

# Searching and counting
index = numbers.index(4)            # Find index of first occurrence
count = numbers.count(1)            # Count occurrences

# Other operations
print(len(numbers))                 # Length
print(sum(numbers))                 # Sum of elements
print(max(numbers))                 # Maximum value
print(min(numbers))                 # Minimum value
```

List Comprehensions

```python
# Basic list comprehension
squares = [x**2 for x in range(10)]
# Equivalent to:
# squares = []
# for x in range(10):
#     squares.append(x**2)

# With condition
even_squares = [x**2 for x in range(10) if x % 2 == 0]

# With string operations
words = ['python', 'java', 'javascript']
upper_words = [word.upper() for word in words]

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

🎯 **Practice Exercise**: Create a list of student grades and calculate average, find highest/lowest scores.

2.3 Tuples

What are Tuples?

- Ordered collection of items
- Immutable (cannot be changed after creation)
- Can contain different data types
- Use parentheses `()` (optional in many cases)

Creating Tuples

python

```
# Empty tuple
empty_tuple = ()
empty_tuple2 = tuple()

# Tuples with data
coordinates = (10, 20)
rgb_color = (255, 128, 0)
mixed_tuple = (1, 'hello', 3.14, True)

# Single element tuple (comma is required!)
single = (42,)
single2 = 42,

# Tuple without parentheses (tuple packing)
point = 10, 20
print(type(point))  # <class 'tuple'>
```

Accessing Tuple Elements

python

```
coordinates = (10, 20, 30)

# Indexing (same as lists)
print(coordinates[0])    # 10
print(coordinates[-1])   # 30

# Slicing (same as lists)
print(coordinates[1:])   # (20, 30)

# Tuple unpacking
x, y, z = coordinates
print(f"x={x}, y={y}, z={z}")

# Partial unpacking with *
first, *rest = (1, 2, 3, 4, 5)
print(first)  # 1
print(rest)   # [2, 3, 4, 5]
```

Tuple Methods and Operations

python

```python
numbers = (1, 2, 3, 2, 4, 2, 5)

# Limited methods (tuples are immutable)
print(numbers.count(2))    # 3 (count occurrences)
print(numbers.index(3))    # 2 (index of first occurrence)

# General operations
print(len(numbers))        # 7
print(max(numbers))        # 5
print(min(numbers))        # 1
print(sum(numbers))        # 19

# Checking membership
print(3 in numbers)        # True
print(10 not in numbers)   # True
```

When to Use Tuples vs Lists

python

```python
# Use tuples for:
# 1. Fixed data that won't change
coordinates = (latitude, longitude)
rgb_color = (255, 0, 128)
database_record = (id, name, email)

# 2. Multiple return values from functions
def get_name_age():
    return "Alice", 25

name, age = get_name_age()

# 3. Dictionary keys (tuples are hashable)
locations = {
    (0, 0): "Origin",
    (10, 20): "Point A",
    (30, 40): "Point B"
}

# Use lists for:
# Data that changes frequently
shopping_cart = ["apple", "bread", "milk"]
shopping_cart.append("eggs")  # This wouldn't work with tuples
```

🎯 **Practice Exercise**: Create tuples for student records (name, age, grade) and practice unpacking.

PART 3: NON-SEQUENTIAL TYPES

3.1 Dictionaries

What are Dictionaries?

- Collection of key-value pairs
- Unordered (ordered as of Python 3.7+)
- Mutable
- Keys must be unique and immutable
- Use curly braces `{}`

Creating Dictionaries

python

```python
# Empty dictionary
empty_dict = {}
empty_dict2 = dict()

# Dictionary with data
student = {
    'name': 'Alice Johnson',
    'age': 20,
    'grade': 'A',
    'courses': ['Math', 'Physics', 'Chemistry']
}

# Using dict() constructor
student2 = dict(name='Bob Smith', age=22, grade='B')

# From list of tuples
pairs = [('a', 1), ('b', 2), ('c', 3)]
dict_from_pairs = dict(pairs)
```

Accessing Dictionary Values

python

```python
student = {'name': 'Alice', 'age': 20, 'grade': 'A'}

# Direct access
print(student['name'])          # 'Alice'
# print(student['phone'])       # KeyError!

# Safe access with get()
print(student.get('name'))      # 'Alice'
print(student.get('phone'))     # None
print(student.get('phone', 'Not provided'))  # 'Not provided'

# Check if key exists
if 'age' in student:
    print(f"Age: {student['age']}")
```

Modifying Dictionaries

python

```python
student = {'name': 'Alice', 'age': 20}

# Adding/updating values
student['grade'] = 'A'          # Add new key-value
student['age'] = 21             # Update existing value
student.update({'phone': '123-456-7890', 'email': 'alice@email.com'})

# Removing items
del student['phone']            # Remove specific key
grade = student.pop('grade')    # Remove and return value
phone = student.pop('phone', 'N/A')  # Remove with default if not found

# Remove and return arbitrary item
item = student.popitem()        # Returns (key, value) tuple

# Clear all items
student.clear()
```

Dictionary Methods and Operations

python

```python
student = {
    'name': 'Alice',
    'age': 20,
    'courses': ['Math', 'Physics'],
    'grade': 'A'
}

# Getting keys, values, items
print(student.keys())           # dict_keys(['name', 'age', 'courses', 'grade'])
print(student.values())         # dict_values(['Alice', 20, ['Math', 'Physics'], 'A'])
print(student.items())          # dict_items([('name', 'Alice'), ...])

# Converting to lists
keys_list = list(student.keys())
values_list = list(student.values())

# Iterating
for key in student:
    print(f"{key}: {student[key]}")

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

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

Nested Dictionaries

```python
# Dictionary of dictionaries
students = {
    'student1': {
        'name': 'Alice',
        'grades': {'math': 95, 'science': 87},
        'info': {'age': 20, 'city': 'New York'}
    },
    'student2': {
        'name': 'Bob',
        'grades': {'math': 78, 'science': 92},
        'info': {'age': 21, 'city': 'London'}
    }
}

# Accessing nested data
print(students['student1']['name'])                    # 'Alice'
print(students['student1']['grades']['math'])          # 95
print(students['student2']['info']['city'])            # 'London'

# Safely accessing nested data
alice_age = students.get('student1', {}).get('info', {}).get('age', 'Unknown')
```

🎯 **Practice Exercise**: Create a dictionary to store information about movies (title, year, genre, rating) and practice various operations.

3.2 Sets

What are Sets?

- Collection of unique elements
- Unordered
- Mutable (the set itself, but elements must be immutable)
- No duplicate values
- Use curly braces `{}` or `set()` function

Creating Sets

python

```python
# Empty set (must use set(), not {})
empty_set = set()

# Set with data
fruits = {'apple', 'banana', 'orange'}
numbers = {1, 2, 3, 4, 5}

# From list (removes duplicates)
list_with_duplicates = [1, 2, 2, 3, 3, 3, 4]
unique_numbers = set(list_with_duplicates)  # {1, 2, 3, 4}

# From string
unique_chars = set("hello")  # {'h', 'e', 'l', 'o'}
```

Set Operations

python

```python
fruits = {'apple', 'banana', 'orange'}

# Adding elements
fruits.add('grape')             # Add single element
fruits.update(['kiwi', 'mango']) # Add multiple elements

# Removing elements
fruits.remove('banana')         # Remove element (raises error if not found)
fruits.discard('banana')        # Remove element (no error if not found)
popped = fruits.pop()           # Remove and return arbitrary element
fruits.clear()                  # Remove all elements

# Membership testing (very fast!)
print('apple' in fruits)        # True/False
```

Mathematical Set Operations

python

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

# Union (elements in either set)
union = set1 | set2             # {1, 2, 3, 4, 5, 6, 7, 8}
union = set1.union(set2)        # Same result

# Intersection (elements in both sets)
intersection = set1 & set2      # {4, 5}
intersection = set1.intersection(set2)  # Same result

# Difference (elements in first set but not second)
difference = set1 - set2        # {1, 2, 3}
difference = set1.difference(set2)  # Same result

# Symmetric difference (elements in either set, but not both)
sym_diff = set1 ^ set2          # {1, 2, 3, 6, 7, 8}
sym_diff = set1.symmetric_difference(set2)  # Same result
```

Set Relationships

python

```
set_a = {1, 2, 3}
set_b = {1, 2, 3, 4, 5}
set_c = {6, 7, 8}

# Subset and superset
print(set_a.issubset(set_b))    # True (A is subset of B)
print(set_b.issuperset(set_a))  # True (B is superset of A)

# Disjoint sets (no common elements)
print(set_a.isdisjoint(set_c))  # True (no common elements)
print(set_a.isdisjoint(set_b))  # False (has common elements)
```

Practical Uses of Sets

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

# Find common elements between lists
list1 = ['apple', 'banana', 'orange']
list2 = ['banana', 'grape', 'apple']
common = list(set(list1) & set(list2))  # ['apple', 'banana']

# Find unique elements in each list
unique_to_list1 = list(set(list1) - set(list2))  # ['orange']
unique_to_list2 = list(set(list2) - set(list1))  # ['grape']

# Check if lists have any common elements
have_common = bool(set(list1) & set(list2))  # True
```

🎯 **Practice Exercise**: Use sets to find common courses between students and unique courses for each student.

PART 4: CHOOSING THE RIGHT DATA STRUCTURE

4.1 Decision Guide

Use Strings When:

- Working with text data
- Need immutable sequence of characters
- Processing text files, user input, or display messages

Use Lists When:

- Need ordered, mutable collection
- Allow duplicate values
- Frequent insertion/deletion at end
- Need to maintain sequence/order

Use Tuples When:

- Need ordered, immutable collection

- Representing fixed data (coordinates, RGB values)
- Multiple return values from functions
- Dictionary keys (must be hashable)

Use Dictionaries When:

- Need key-value relationships
- Fast lookups by key
- Representing structured data with named fields
- Counting occurrences

Use Sets When:

- Need unique elements only
- Fast membership testing
- Mathematical set operations
- Removing duplicates

4.2 Performance Comparison

```python
import time

# Membership testing performance
big_list = list(range(10000))
big_set = set(range(10000))

# Testing if 9999 is in collection
# List: O(n) - slow for large collections
start = time.time()
9999 in big_list
list_time = time.time() - start

# Set: O(1) - very fast regardless of size
start = time.time()
9999 in big_set
set_time = time.time() - start

print(f"List lookup: {list_time:.6f}s")
print(f"Set lookup: {set_time:.6f}s")
```

4.3 Mutability Summary

python

```python
# Immutable types (cannot be changed)
immutable_types = [
    42,           # int
    3.14,         # float
    "hello",      # str
    (1, 2, 3),    # tuple
    frozenset([1, 2, 3])  # frozenset
]

# Mutable types (can be changed)
mutable_types = [
    [1, 2, 3],    # list
    {'a': 1},     # dict
    {1, 2, 3}     # set
]

# Demonstration
my_list = [1, 2, 3]
my_tuple = (1, 2, 3)

my_list[0] = 10      # This works - lists are mutable
# my_tuple[0] = 10   # This would cause an error - tuples are immutable
```

PART 5: PRACTICAL EXAMPLES AND EXERCISES

5.1 Real-World Example: Student Management System

python

```python
# Using different data structures together
students_database = {
    'CS101': {
        'course_name': 'Introduction to Computer Science',
        'instructor': 'Dr. Smith',
        'students': [
            {
                'id': 'S001',
                'name': 'Alice Johnson',
                'grades': [85, 92, 78, 96],
                'contact': ('alice@email.com', '123-456-7890')
            },
            {
                'id': 'S002',
                'name': 'Bob Wilson',
                'grades': [76, 82, 90, 85],
                'contact': ('bob@email.com', '098-765-4321')
            }
        ],
        'topics_covered': {'variables', 'functions', 'loops', 'data_structures'}
    }
}

# Accessing and manipulating data
course = students_database['CS101']
first_student = course['students'][0]
print(f"Student: {first_student['name']}")
print(f"Average grade: {sum(first_student['grades']) / len(first_student['grades'])}")
print(f"Email: {first_student['contact'][0]}")

# Add new topic
course['topics_covered'].add('algorithms')

# Add new grade
first_student['grades'].append(88)
```

5.2 Practice Exercises

Exercise 1: Text Processing

python

```python
# Given text, count word frequency using dictionary
text = "python is great python is powerful python is easy"
# Expected output: {'python': 3, 'is': 3, 'great': 1, 'powerful': 1, 'easy': 1}

# Your solution here:
words = text.split()
word_count = {}
for word in words:
    word_count[word] = word_count.get(word, 0) + 1
```

Exercise 2: Data Cleaning

python

```python
# Remove duplicates and sort a list of mixed data
messy_data = [1, 'apple', 2, 'banana', 1, 'apple', 3, 'cherry', 2]
# Clean numeric and string data separately

numbers = []
strings = []
for item in messy_data:
    if isinstance(item, int):
        numbers.append(item)
    elif isinstance(item, str):
        strings.append(item)

unique_numbers = sorted(list(set(numbers)))
unique_strings = sorted(list(set(strings)))
```

Exercise 3: Coordinate System

```python
# Store and manipulate 2D points using tuples
points = [(0, 0), (3, 4), (1, 1), (5, 0)]

# Calculate distance from origin for each point
import math
distances = []
for point in points:
    x, y = point
    distance = math.sqrt(x**2 + y**2)
    distances.append((point, distance))

# Sort points by distance from origin
distances.sort(key=lambda item: item[1])
```

SUMMARY AND KEY TAKEAWAYS

Quick Reference Table

Data Structure	Ordered	Mutable	Duplicates	Syntax	Use Case
String (str)	Yes	No	Yes	"text"	Text processing
List	Yes	Yes	Yes	[1,2,3]	Ordered collections
Tuple	Yes	No	Yes	(1,2,3)	Fixed data
Dictionary	Yes*	Yes	No (keys)	{"a":1}	Key-value pairs
Set	No	Yes	No	{1,2,3}	Unique elements

*Ordered as of Python 3.7+

Best Practices

1. **Choose the right data structure** for your specific use case
2. **Use list comprehensions** when appropriate for cleaner code
3. **Leverage set operations** for mathematical operations and uniqueness
4. **Use dictionaries** for fast lookups and structured data
5. **Consider immutability** when data shouldn't change
6. **Test membership** with sets for better performance on large datasets

Common Patterns

python

```python
# Pattern 1: Counting occurrences
count_dict = {}
for item in data:
    count_dict[item] = count_dict.get(item, 0) + 1

# Pattern 2: Grouping data
groups = {}
for item in data:
    key = get_group_key(item)
    if key not in groups:
        groups[key] = []
    groups[key].append(item)

# Pattern 3: Filtering and transforming
filtered_data = [transform(x) for x in data if condition(x)]

# Pattern 4: Set operations for data analysis
common_elements = set(list1) & set(list2)
unique_elements = set(list1) ^ set(list2)
```

This comprehensive guide covers all the essential Python data types and structures. Practice with real data and gradually build complexity in your projects!