🐍 Definition of Sets in Python

A **set** in Python is an **unordered collection of unique elements**.
Sets are **mutable**, meaning you can add or remove elements, but they do **not allow duplicate values**.
Commonly used for **union**, **intersection**, and **difference** operations.

📋 Key Characteristics of Sets

1️⃣ **Unordered** — Elements have no specific order.
2️⃣ **Unique Elements** — Duplicates are automatically removed.
3️⃣ **Mutable** — You can add or remove elements.
4️⃣ **Iterable** — Supports iteration in loops.

🛠️ How Python Implements Sets

Python sets are built using a **hash table**.

1. Hash Table

- Elements stored as **keys**.
- Fast lookup: **O(1)** average time.

2. Hashing

- Uses `hash()` to compute element positions.

3. No Duplicates

- Elements with the same hash aren't added again.

4. Unordered Nature

- Element order is not preserved.

🧪 Example of Set Behavior

```
# Set with duplicates
my_set = {1, 2, 2, 3, 4}
print(my_set)  # Output: {1, 2, 3, 4}

# Fast membership test
print(2 in my_set)  # Output: True
```

📊 Comparison with Other Data Structures

Feature	Set	List	Dictionary
Order	Unordered	Ordered	Insertion Order (Python 3.7+)
Duplicates	Not allowed	Allowed	Keys: Not allowed, Values: Allowed
Lookup Time	O(1)	O(n)	O(1)
Use Case	Membership testing	Sequential data storage	Key-value pairs

---

🎯 When to Use Sets

- To **store unique elements**.
- For **set operations**: union, intersection, difference.
- For **fast membership tests**.

📝 Definition of Lists in Python

A **list** in Python is an **ordered, mutable, and iterable collection of elements**.

Lists can store elements of **any data type** (e.g., integers, strings, objects) and **allow duplicate values**.

📋 Key Characteristics of Lists

1️⃣ **Ordered**

- Elements maintain their insertion order.

2️⃣ **Mutable**

- You can modify a list by adding, removing, or changing elements.

3️⃣ **Allows Duplicates**

- Lists can contain duplicate elements.

4️⃣ **Heterogeneous**

- Lists can store elements of different data types.

🛠️ How Python Implements Lists

Python implements lists as **dynamic arrays**.

1. Dynamic Array

- A list is a contiguous block of memory storing **references (pointers)** to its elements.
- When the list exceeds its current capacity, Python allocates a **larger block of memory** and copies existing elements.

2. Amortized Time Complexity

- **Appending** to the end: O(1) on average.
- **Resizing** when needed: O(n) due to copying elements.

3. Indexing and Slicing

- Indexing is **O(1)** (fast, direct access).
- Slicing creates a new list: O(k), where **k is the slice size**.

4. Memory Overhead

- Lists store **references**, not objects themselves, leading to some memory overhead.

🧪 Example of List Behavior

```
# Create a list
my_list = [1, 2, 3, 4, 5]

# Access by index
print(my_list[0])   # Output: 1
print(my_list[-1])  # Output: 5

# Append element
my_list.append(6)
print(my_list)      # Output: [1, 2, 3, 4, 5, 6]

# Slicing
print(my_list[1:4]) # Output: [2, 3, 4]
```

📊 Comparison with Other Data Structures

Feature	List	Tuple	Set
Order	Ordered	Ordered	Unordered
Mutability	Mutable	Immutable	Mutable
Duplicates	Allowed	Allowed	Not allowed
Use Case	Sequential data storage	Fixed collections	Unique elements

---

🎯 When to Use Lists

✔️ When you need to store a **collection of elements in a specific order**.
✔️ When you need to **frequently modify** the collection (adding, removing, updating).
✔️ When you need to **allow duplicate elements**.

📖 Definition of Dictionaries in Python

A **dictionary** in Python is an **unordered, mutable collection of key-value pairs**.

- Keys must be **unique and immutable** (e.g., strings, numbers, tuples).
- Values can be of **any data type** and can be duplicated.

📋 Key Characteristics of Dictionaries

1️⃣ **Key-Value Pairs**

- 🔹 Each element is a pair of a key and its corresponding value.

2️⃣ **Unordered**

- 🔹 Dictionaries do not maintain order of elements (before Python 3.7).
- 🔹 From Python 3.7+, they maintain **insertion order**.

3️⃣ **Mutable**

- 🔹 You can add, remove, or update key-value pairs.

4️⃣ **Fast Lookups**

- 🔹 Provides fast access to values using keys.

🛠️ How Python Implements Dictionaries

Dictionaries use a **hash table** for efficient storage and lookup.

1. Hash Table

- A dictionary is backed by a hash table (fixed size array).
- Each key's hash value is computed using Python's `hash()` function.
- Hash determines the index in the hash table for storage.

2. Collision Handling

- When two keys hash to the same index, Python uses **open addressing** or **chaining** to resolve collisions and store the key-value pairs correctly.

3. Dynamic Resizing

- When the dictionary gets too full, Python **automatically resizes** the hash table.
- Resizing involves creating a new, larger hash table and **rehashing** existing keys.

4. Time Complexity

- **Average Case:** O(1) for access, insertion, deletion.
- **Worst Case:** O(n) in rare cases (e.g., many hash collisions).

🧪 Example of Dictionary Behavior

```
# Create a dictionary
my_dict = {"a": 1, "b": 2, "c": 3}

# Access a value by key
print(my_dict["a"])  # Output: 1

# Add a new key-value pair
my_dict["d"] = 4
print(my_dict)  # Output: {'a': 1, 'b': 2, 'c': 3, 'd': 4}

# Update an existing key
my_dict["a"] = 10
print(my_dict)  # Output: {'a': 10, 'b': 2, 'c': 3, 'd': 4}

# Check if a key exists
print("b" in my_dict)  # Output: True
```

📊 Comparison with Other Data Structures

Feature	Dictionary	List	Set
Order	Maintains insertion order (Python 3.7+)	Ordered	Unordered
Mutability	Mutable	Mutable	Mutable
Duplicates	Keys: Not allowed, Values: Allowed	Allowed	Not allowed
Use Case	Key-value mapping	Sequential data storage	Unique elements

---

🎯 When to Use Dictionaries

✔️ When you need to store **data as key-value pairs**.
✔️ When you need **fast lookups** for specific keys.
✔️ When you need to dynamically **add, remove, or update elements**.

📦 Definition of Tuples in Python

A **tuple** in Python is an **ordered, immutable collection of elements**.

- Similar to lists, but **cannot be modified after creation**.
- Useful for storing **related data that should not change**.

📋 Key Characteristics of Tuples

1️⃣ **Ordered**

- 🔹 Tuples maintain the order of elements.

2️⃣ **Immutable**

- 🔹 Once created, elements **cannot be changed, added, or removed**.

3️⃣ **Allows Duplicates**

- 🔹 Tuples can contain **duplicate elements**.

4️⃣ **Heterogeneous**

- 🔹 Can store **elements of different data types**.

🛠️ How Python Implements Tuples

Python implements tuples as **immutable sequences**.

1. Fixed Size

- Stored as a contiguous block of memory (like lists).
- Size is fixed at creation.

2. Efficient Storage

- More **memory-efficient** than lists (no dynamic resizing overhead).

3. Immutable Nature

- Elements cannot change after creation.
- Allows tuples to be **hashable**, usable as dictionary keys or set elements.

4. Time Complexity

- **Indexing:** O(1) — direct access via index.
- **Iteration:** O(n) — linear with number of elements.

🧪 Example of Tuple Behavior

```
# Create a tuple

my_tuple = (1, 2, 3, 4, 5)



# Access elements by index

print(my_tuple[0])  # Output: 1

print(my_tuple[-1]) # Output: 5



# Tuples are immutable

# my_tuple[0] = 10  # This will raise a TypeError



# Duplicates are allowed

duplicate_tuple = (1, 2, 2, 3)

print(duplicate_tuple)  # Output: (1, 2, 2, 3)



# Mixed data types

mixed_tuple = (1, "hello", 3.14)

print(mixed_tuple)  # Output: (1, 'hello', 3.14)
```

📊 Comparison with Other Data Structures

Feature	Tuple	List	Set
Order	Ordered	Ordered	Unordered
Mutability	Immutable	Mutable	Mutable
Duplicates	Allowed	Allowed	Not allowed
Use Case	Fixed collections	Sequential data storage	Unique elements

---

🎯 When to Use Tuples

✔️ When storing **elements that should not change**.

✔️ When using as a **key in dictionaries** or **elements in sets**.

✔️ When needing a **lightweight, memory-efficient** alternative to lists for fixed-size data.