■ Lists

- **Ordered**: Lists maintain the order of elements. This means that when you add elements to a list, they retain their position, and you can access elements using their index. The order of elements is preserved during iteration.
- **Mutable**: Lists are mutable, meaning you can modify them after creation. You can add, remove, or change elements within a list without creating a new list.
- **Allow duplicates**: Lists can contain duplicate elements. This allows you to have multiple occurrences of the same value within a list.
- **Heterogeneous**: Lists can hold elements of different data types. For example, a list can contain integers, strings, floats, and even other lists.
- **Dynamic size**: Lists in Python are dynamic, meaning their size can change as you add or remove elements.

```
fruits = ["apple", "banana", "cherry"]
print(fruits)
```

■ List Methods

1. `append()`

Adds an element to the end of the list.

```
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")
print(fruits)
```

2. `insert()`

Inserts an element at a specified position.

```
fruits = ["apple", "banana", "cherry"]
fruits.insert(1, "kiwi")
print(fruits)
```

3. `extend()`

Extends the list by adding elements from another list.

```
fruits = ["apple", "banana", "cherry"]
more_fruits = ["grape", "melon"]
fruits.extend(more_fruits)
print(fruits)
```

4. `remove()`

Removes the first occurrence of the specified element.

```
fruits = ["apple", "banana", "cherry"]
fruits.remove("banana")
print(fruits)
```

5. `pop()`

Removes the element at the specified position (default is the last element) and returns it.

```
fruits = ["apple", "banana", "cherry"]
last_fruit = fruits.pop()
print(last_fruit)
print(fruits)
```

6. `clear()`

Removes all elements from the list.

```
fruits = ["apple", "banana", "cherry"]
fruits.clear()
print(fruits)
```

7. `index()`

Returns the index of the first occurrence of the specified element.

```
fruits = ["apple", "banana", "cherry"]
index = fruits.index("banana")
print(index)
```

8. `count()`

Returns the number of occurrences of the specified element.

```
fruits = ["apple", "banana", "cherry"]
count = fruits.count("apple")
print(count)
```

9. `sort()`

Sorts the list in ascending order by default.

```
numbers = [3, 1, 4, 1, 5, 9, 2]
numbers.sort()
print(numbers)
```

10. `reverse()`

Reverses the order of the list.

```
numbers = [3, 1, 4, 1, 5, 9, 2]
numbers.reverse()
print(numbers)
```

12. `len()`

Returns the number of elements in the list.

```
fruits = ["apple", "banana", "cherry"]
length = len(fruits)
print(length)
```

13. List Slicing

You can access a range of elements using slicing.

```
fruits = ["apple", "banana", "cherry", "date", "fig", "grape"]
print(fruits[1:4])  # ['banana', 'cherry', 'date']
print(fruits[:3])   # ['apple', 'banana', 'cherry']
print(fruits[3:])   # ['date', 'fig', 'grape']
print(fruits[-3:])  # ['date', 'fig', 'grape']
```

14. Looping Through a List

```
fruits = ["apple", "banana", "cherry", "date", "fig", "grape"]
for fruit in fruits:
    print(fruit)
```

■ List Use Case

- **Storing User Data**: Keep a list of user names, email addresses, or IDs for easy access and manipulation.
- **Managing To-Do Lists**: Track tasks and their statuses in a to-do list application.
- **Inventory Management**: Maintain a list of product items, quantities, and details in a store inventory system.
- **Processing Orders**: Store and process customer orders in an e-commerce application.
- **Collecting Survey Responses**: Gather and analyze survey responses from multiple participants.
- **Scheduling Events**: Organize and manage a list of events or appointments in a calendar application.
- **Data Analysis**: Store and manipulate datasets for statistical analysis or machine learning.
- **Playlist Management**: Keep track of songs, videos, or other media items in a playlist.
- **Shopping Cart**: Store items added to a shopping cart in an online shopping system.
- **Tracking Scores**: Maintain a list of scores or results for games or competitions.

■ Tuples

- **Ordered**: Like lists, tuples maintain the order of elements. The order in which elements are added is preserved, and they can be accessed using an index.
- **Immutable**: Once a tuple is created, it cannot be modified. You cannot add, remove, or change elements in a tuple after its creation. This immutability makes tuples suitable for use as keys in dictionaries and ensures that the data remains consistent.
- **Allow duplicates**: Tuples can contain duplicate elements. This allows multiple occurrences of the same value within a tuple.
- **Heterogeneous**: Tuples can hold elements of different data types. For example, a tuple can contain integers, strings, floats, and even other tuples.
- **Fixed size**: The size of a tuple is fixed upon creation. Unlike lists, you cannot change the size of a tuple after it is created.

```
fruits = ("apple", "banana", "cherry")
print(fruits)
print(fruits[0])  # Output: apple
print(fruits[1])  # Output: banana
print(fruits[2])  # Output: cherry
```

■ Tuple Methods

1. `count()`

Returns the number of times a specified value appears in the tuple.

```
numbers = (1, 2, 3, 2, 4, 2)
count_of_twos = numbers.count(2)
print(count_of_twos)  # Output: 3
```

2. `index()`

Returns the index of the first occurrence of the specified value.

```
numbers = (1, 2, 3, 2, 4, 2)
index_of_three = numbers.index(3)
print(index_of_three)  # Output: 2
```

3. Looping Through a Tuple

You can loop through the elements of a tuple using a `for` loop.

```
fruits = ("apple", "banana", "cherry")
for fruit in fruits:
    print(fruit)
```

4. Tuple Slicing

You can access a range of elements in a tuple using slicing.

```
fruits = ("apple", "banana", "cherry")
print(fruits[1:3])  # Output: ('banana', 'cherry')
print(fruits[:2])   # Output: ('apple', 'banana')
print(fruits[1:])   # Output: ('banana', 'cherry')
print(fruits[-2:])  # Output: ('banana', 'cherry')
```

5. Converting Between Lists and Tuples

You can convert lists to tuples and vice versa using the `tuple()` and `list()` functions.

```
# List to tuple
my_list = [1, 2, 3]
my_tuple = tuple(my_list)
print(my_tuple)  # Output: (1, 2, 3)

# Tuple to list
my_tuple = (4, 5, 6)
my_list = list(my_tuple)
print(my_list)  # Output: [4, 5, 6]
```

■ Tuples Use case

- **Immutable Data Storage**: Store read-only configuration settings that should not be altered during program execution.
- **Return Multiple Values**: Return multiple values from a function efficiently.
- **Fixed Collections**: Store a fixed collection of related data, such as coordinates (x, y, z) or RGB color values.
- **Dictionary Keys**: Use tuples as keys in dictionaries for composite key lookups.

- **Database Records**: Represent rows of database records as tuples for easy and consistent access.
- **Data Integrity**: Ensure data integrity by using tuples for sequences of data that should remain constant.
- **Grouping Data**: Group heterogeneous data types together logically, such as an employee record with a name, ID, and position.
- **Efficient Iteration**: Iterate over a fixed set of elements without the overhead of mutable data structures.
- **Named Tuples**: Use named tuples to create self-documenting code and improve code readability.
- **Function Arguments**: Pass a fixed set of parameters to functions, ensuring the parameters remain unchanged.

■ Sets in Python

- **Unordered**: The elements in a set do not have a defined order. Unlike lists or tuples, when you iterate over a set, the items may appear in a different order each time, and they cannot be accessed using an index.
- **Mutable**: Sets are mutable, meaning you can add or remove elements after the set is created. However, the elements within the set must be immutable types, such as strings, numbers, or tuples.
- **No duplicates**: Sets do not allow duplicate elements. If you try to add a duplicate element to a set, it will be ignored, ensuring that all elements in the set are unique.
- **Heterogeneous**: Like lists and tuples, sets can hold elements of different data types. A set can contain integers, strings, floats, and even other immutable types.
- **Dynamic size**: The size of a set can change as you add or remove elements. There is no fixed limit on the number of elements a set can hold.
- **Efficient membership testing**: Sets are optimized for checking whether a specific element is part of the set, making them ideal for operations that require fast membership testing.

```
fruits = {"apple", "banana", "cherry"}
print(fruits)
```

■ Set Methods

1. `add()`

Adds an element to the set.

```
fruits = {"apple", "banana", "cherry"}
fruits.add("orange")
print(fruits)
```

2. `update()`

Adds multiple elements to the set.

```
fruits = {"apple", "banana", "cherry"}
fruits.update(["grape", "melon"])
print(fruits)
```

3. `remove()`

Removes the specified element from the set. Raises a `KeyError` if the element is not found.

```
fruits = {"apple", "banana", "cherry"}
fruits.remove("banana")
print(fruits)
```

4. `discard()`

Removes the specified element from the set if it is present.

```
fruits = {'apple', 'cherry', 'grape', 'melon', 'orange'}
fruits.discard("melon")
print(fruits)
```

5. `pop()`

Removes and returns an arbitrary element from the set. Raises a `KeyError` if the set is empty.

```
fruits = {'apple', 'cherry', 'grape', 'melon', 'orange'}
element = fruits.pop()
print(element)
print(fruits)
```

6. `clear()`

Removes all elements from the set.

```
fruits = {'apple', 'cherry', 'grape', 'melon', 'orange'}
fruits.clear()
print(fruits)
```

7. `union()`

Returns a new set with all elements from both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result = set1.union(set2)
print(result)  # Output: {1, 2, 3, 4, 5}
```

8. `intersection()`

Returns a new set with only the elements that are common to both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result = set1.intersection(set2)
print(result)
```

9. `difference()`

Returns a new set with elements in the first set that are not in the second set.

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

```
result = set1.difference(set2)
print(result)
```

10. `symmetric_difference()`

Returns a new set with elements in either set but not in both.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result = set1.symmetric_difference(set2)
print(result)
```

11. `issubset()`

Checks if all elements of the set are present in another set.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1.issubset(set2))
```

12. `issuperset()`

Checks if the set contains all elements of another set.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1.issuperset({1, 2}))
```

13. `isdisjoint()`

Checks if two sets have no elements in common.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1.isdisjoint(set2))
```

■ Sets Use case

- **Removing Duplicates**: Automatically eliminate duplicate entries from a collection of data.
- **Membership Testing**: Quickly check if an item is part of a collection.
- **Mathematical Set Operations**: Perform union, intersection, difference, and symmetric difference operations on collections.
- **Unique Elements**: Maintain a collection of unique items, such as unique user IDs or product codes.
- **Filtering Data**: Filter out non-unique items from a list or other iterable.
- **Tagging System**: Implement a system to tag items where each tag is unique.
- **Graph Algorithms**: Use sets to manage adjacency lists for nodes in graph algorithms.
- **Tracking Seen Items**: Keep track of items that have already been processed or visited in loops or algorithms.
- **Sparse Data Structures**: Represent sparse datasets efficiently where only a few elements out of a large range are present.
- **Event Handling**: Manage a set of unique event handlers or callbacks in event-driven programming.

■ Dictionary

- **Unordered**: Dictionaries in Python store key-value pairs without a specific order. While the insertion order is preserved in Python 3.7 and later, dictionaries do not support indexing like lists or tuples.
- **Mutable**: Dictionaries are mutable, meaning you can change them after they are created. You can add, remove, or modify key-value pairs in a dictionary at any time.
- **Key uniqueness**: Each key in a dictionary must be unique. If you try to insert a duplicate key, the existing value associated with that key will be overwritten with the new value.
- **Heterogeneous**: Both keys and values in a dictionary can be of any data type. Keys are typically immutable types (like strings, numbers, or tuples), while values can be of any type, including other dictionaries.
- **Dynamic size**: Dictionaries in Python can grow or shrink as you add or remove key-value pairs. There is no fixed limit on the number of elements a dictionary can hold.
- **Efficient key-based access**: Dictionaries provide fast access to values based on their keys, making them ideal for use cases where you need to quickly retrieve, add, or modify data associated with specific keys.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
print(student)
print(student["name"])  # Output: Alice
print(student.get("age"))  # Output: 23
```

■ Dictionary Methods

1. `keys()`

Returns a view object containing the dictionary's keys.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
print(student.keys())
```

2. `values()`

Returns a view object containing the dictionary's values.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
print(student.values())
```

3. `items()`

Returns a view object containing the dictionary's key-value pairs.

```
student = {
    "name": "Alice",
```

```
    "age": 23,
    "major": "Computer Science"
}
print(student.items())
```

4. `update()`

Updates the dictionary with the specified key-value pairs.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
student.update({"age": 24, "graduated": True})
print(student)
```

5. `pop()`

Removes the specified key and returns its value. Raises a `KeyError` if the key is not found.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
age = student.pop("age")
print(age)  # Output: 24
print(student)
```

6. `popitem()`

Removes and returns the last inserted key-value pair as a tuple.

```
student = {'name': 'Alice', 'age': 24, 'major': 'Computer Science', 'graduated':
True}
item = student.popitem()
print(item)  # Output: ('graduated', True)
print(student)
```

7. `clear()`

Removes all elements from the dictionary.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}
student.clear()
print(student)  # Output: {}
```

8. `get()`

Returns the value for the specified key if it exists, otherwise returns the default value.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}

age = student.get("age", "Not Available")
print(age)
```

10. `copy()`

Returns a shallow copy of the dictionary.

```
student = {
    "name": "Alice",
    "age": 23,
    "major": "Computer Science"
}

student_copy = student.copy()
print(student_copy)
```

■ Dictionary Use case

- **Storing Configuration Settings**: Keep configuration settings for applications where each setting has a unique key.
- **Fast Lookups**: Quickly retrieve values based on unique keys, such as looking up a user's profile by user ID.
- **Counting Occurrences**: Count the frequency of items in a dataset, such as word counts in a document.
- **Caching Results**: Store and retrieve the results of expensive computations to avoid redundant calculations.
- **Mapping Relationships**: Represent relationships between items, such as mapping employee IDs to employee names.
- **Storing JSON Data**: Work with JSON data structures, which are naturally represented as dictionaries in Python.
- **Organizing Data**: Organize and group related pieces of information together, such as storing contact information.
- **Database Records**: Represent records from a database where each record is a dictionary with column names as keys.
- **User Preferences**: Store user preferences and settings for applications.
- **Nested Data Structures**: Create complex data structures by nesting dictionaries within dictionaries to represent hierarchical data.
