List vs Tuple vs Dictionary

Python provides different data structures for storing collections of data. The most commonly used ones are **Lists, Tuples, and Dictionaries**. Here's a comparison:

---

1. List

A **list** is an ordered, mutable (changeable) collection that allows duplicate elements.

Characteristics:

- Ordered (items have a defined order).
- Mutable (elements can be modified, added, or removed).
- Allows duplicate values.
- It can store different data types in a single list.
- Uses **square brackets** `[]`.

Example:

```
my_list = [10, 20, 30, 40, 50]  # List of integers
my_list.append(60)  # Adds 60 to the list
my_list[0] = 5  # Modifying an element
print(my_list)  # Output: [5, 20, 30, 40, 50, 60]
```

Use Cases:

- When you need a collection that can change dynamically.
- When ordering of elements matters.

---

2. Tuple

A **tuple** is an ordered, immutable collection that allows duplicate elements.

Characteristics:

- Ordered (elements have a fixed order).
- Immutable (cannot be modified after creation).
- Allows duplicate values.
- Can store different data types.
- Uses **parentheses** `()`.

Example:

```
my_tuple = (10, 20, 30, 40, 50)  # Tuple of integers
# my_tuple[0] = 5  # This will raise an error (tuples are immutable)
print(my_tuple)  # Output: (10, 20, 30, 40, 50)
```

Use Cases:

- When you need a collection that should not be modified.
- When you want to use a sequence as a dictionary key (since tuples are hashable).
- Performance-sensitive scenarios (tuples are faster than lists).

---

3. Dictionary

A **dictionary** is an unordered collection of key-value pairs.

Characteristics:

- **Unordered** (before Python 3.7, but from Python 3.7+, insertion order is maintained).
- **Mutable** (values can be changed, new key-value pairs can be added).
- **Keys must be unique** (values can be duplicated).
- Uses **curly braces** `{}`.

Example:

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
my_dict["age"] = 26  # Modifying an existing key
my_dict["gender"] = "Female"  # Adding a new key-value pair
print(my_dict)  # Output: {'name': 'Alice', 'age': 26, 'city': 'New
York', 'gender': 'Female'}
```

Use Cases:

- When you need a key-value pair structure for fast lookups.
- When data needs to be retrieved based on unique keys.