**Q1. What is a tuple in Python?**

A **tuple** in Python is a built-in data type that represents an ordered, immutable collection of elements. Tuples can contain duplicate values.

**Q2. How do you create a tuple in Python?**

**Ans:** We can create a tuple in Python by enclosing elements in parentheses () and separating them with commas.

Example:

my\_tuple = (1, 'apple', 3.14, True)

print(my\_tuple)

# Output: (1, 'apple', 3.14, True)

**Q3. What is the difference between a tuple and a list in Python?**

**Ans:-** The main differences between a tuple and a list in Python are based on their characteristics, usage, and behaviour. Here’s a comparison:

### 1. Mutability

Tuple: Immutable. Once a tuple is created, you cannot change, add, or remove elements.

List: Mutable. We can modify a list by adding, removing, or changing elements.

### 2. Syntax

Tuple: Created using parentheses () or the tuple() constructor.

Ex: for tuple

my\_tuple = (1, 2, 3)

Ex: for list

my\_list = [1, 2, 3]

### **3. Performance**

**Tuple**: Generally faster than lists for iteration and access due to their immutability.

**List**: Slightly slower for certain operations, as they need to manage dynamic memory allocation.

### **4. Usage**

**Tuple**: Often used for fixed collections of items, such as coordinates, database records, or when you want to ensure that data remains unchanged.

**List**: Used for collections of items that may need to be modified, like a list of shopping items or a collection of user inputs.

### **5. Methods**

**Tuple**: Limited built-in methods (e.g., count(), index()) due to their immutability.

**List**: More built-in methods available (e.g., append(), remove(), extend(), pop()) to facilitate modification.

### **6. Nested Structures**

Both tuples and lists can contain nested structures, but the immutability of tuples can be a constraint in some cases.

**Q4. Can a tuple be changed in Python?**

Ans: No, a tuple in Python cannot be changed once it is created. This characteristic is known as **immutability**.

We cannot add, remove, or modify elements in a tuple after it is created. We can create new tuples based on existing ones.

**Q5. How do you access elements in a tuple?**

**Ans:** We can access elements in a tuple using indexing and slicing.

### **1. Accessing Elements by Index**

Tuples are zero-indexed, meaning the first element is at index 0, the second at index 1, and so on.

my\_tuple = (10, 20, 30, 40, 50)

# Accessing the first element

first\_element = my\_tuple[0]

print(first\_element)

# Output: 10

# Accessing the third element

third\_element = my\_tuple[2]

print(third\_element) # Output: 30

### **2. Negative Indexing**

we can also use negative indexing to access elements from the end of the tuple. The last element is at index -1, the second to last at -2, and so on.

Ex:

# Accessing the last element

last\_element = my\_tuple[-1]

print(last\_element) # Output: 50

# Accessing the second to last element

second\_to\_last = my\_tuple[-2]

print(second\_to\_last) # Output: 40

### **3. Slicing**

We can access a range of elements (a slice) using the colon : operator.

Ex:

# Slicing to get the first three elements

slice\_tuple = my\_tuple[0:3]

print(slice\_tuple) # Output: (10, 20, 30)

# Slicing with negative indices

slice\_tuple\_neg = my\_tuple[-3:]

print(slice\_tuple\_neg) # Output: (30, 40, 50)

**Q6. How do you unpack a tuple in Python?**

**Ans:** We can unpack a tuple in Python by assigning its elements to individual variables in a single assignment statement.

### **Basic Tuple Unpacking**

When we have a tuple with a known number of elements, we can unpack it directly:

Ex: my\_tuple = (1, 2, 3)

# Unpacking the tuple

a, b, c = my\_tuple

print(a) # Output: 1

print(b) # Output: 2

print(c) # Output: 3

### **Unpacking with Fewer Variables**

If we only want to unpack some elements of the tuple, we can use an underscore \_ as a placeholder for the elements we want to ignore:

Ex: my\_tuple = (1, 2, 3)

# Unpacking and ignoring the second element

a, \_, c = my\_tuple

print(a) # Output: 1

print(c) # Output: 3

### **Unpacking with a Nested Tuple**

We can also unpack nested tuples:

Ex: nested\_tuple = (1, (2, 3), 4)

# Unpacking the nested tuple

a, (b, c), d = nested\_tuple

print(a) # Output: 1

print(b) # Output: 2

print(c) # Output: 3

print(d) # Output: 4

### **Using \* for Extended Unpacking**

We can use the \* operator to capture multiple elements in a list:

my\_tuple = (1, 2, 3, 4, 5)

# Unpacking with extended unpacking

a, \*b, c = my\_tuple

print(a) # Output: 1

print(b) # Output: [2, 3, 4]

print(c) # Output: 5