TOPSTechnologies

Tuple

Presented for :

TOPs Technologies

Presented by :

Sumit B Yadav

Que 1

A tuple is a built-in data structure in Python that is used to store an ordered collection of elements. Like lists, tuples can hold items of different data types (e.g., integers, strings, floats, other tuples, or even lists). However, unlike lists, tuples are immutable, meaning their content cannot be changed after they are created.

Key Characteristics of Tuples:

1. Ordered: Tuples maintain the order of elements. This means the first item in a tuple stays in the same position unless the tuple is replaced.
2. Immutable: Once a tuple is created, its contents cannot be modified (no adding, removing, or changing elements).
3. Heterogeneous: Tuples can contain elements of different data types.
4. Indexable: Elements in a tuple can be accessed using an index, starting at 0.
5. Allow Duplicates: Tuples can have duplicate elements, as they do not enforce uniqueness.

syntax-

```
# Using parentheses
my_tuple = (1, 2, 3)

# Using tuple() constructor
another_tuple = tuple([4, 5, 6])
```

Why Immutability Matters

1. Safety: Immutability ensures the integrity of the data, as tuples cannot be accidentally modified.
2. Hashability: Because tuples are immutable, they can be used as keys in dictionaries or elements in sets, unlike lists.
3. Performance: Tuples can be faster than lists for certain operations, as they require less memory and fewer operations for creation.

example -

```
# A tuple
example_tuple = (1, 2, 3)

# Trying to modify a tuple (will raise an error)
example_tuple[0] = 10  # TypeError: 'tuple' object does not support item assignment
```

Que. 2

Creating Tuples

Tuples can be created in several ways:

- Using Parentheses ():

syntax-

```
# Tuple with integers
my_tuple = (1, 2, 3)
# Tuple with mixed data types
mixed_tuple = (42, "Hello", 3.14)
# Nested tuple
nested_tuple = (1, (2, 3), [4, 5])
```

- Without Parentheses (Implicit Tuple): Tuples can be created without parentheses by separating values with commas.

syntax-

```
implicit_tuple = 1, 2, 3
print(type(implicit_tuple))  # Output: <class 'tuple'>
```

- Using the tuple() Constructor:

syntax-

```
# From a list
list_to_tuple = tuple([1, 2, 3])
# From a string (creates a tuple of characters)
string_to_tuple = tuple("hello")
```

Accessing Elements in a Tuple

1. Indexing: You can access elements in a tuple using indices (starting from 0 for the first element).

syntax-

```
my_tuple = (10, 20, 30, 40)
print(my_tuple[0])  # Output: 10
print(my_tuple[2])  # Output: 30
```

2. Negative Indexing: Use negative indices to access elements from the end of the tuple.

syntax -

```
print(my_tuple[-1])  # Output: 40 (last element)
print(my_tuple[-3])  # Output: 20
```

3. Slicing: Use slicing to access a subset of elements from the tuple.

syntax-

```
print(my_tuple[1:3])  # Output: (20, 30)
print(my_tuple[:2])   # Output: (10, 20)
print(my_tuple[::2])  # Output: (10, 30) (step of 2)
```

Que. 3

Tuples support several useful operations that make them versatile when handling ordered collections of items

1. Concatenation

Tuples can be concatenated using the + operator, which combines two or more tuples into a new tuple.

Example:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

# Concatenate two tuples
result = tuple1 + tuple2
print(result)  # Output: (1, 2, 3, 4, 5, 6)
```

2. Repetition

You can repeat a tuple using the * operator. This creates a new tuple by repeating the elements of the original tuple a specified number of times.

Example:

```
tuple1 = (1, 2)

# Repeat the tuple 3 times
repeated = tuple1 * 3
print(repeated)  # Output: (1, 2, 1, 2, 1, 2)
```

3. Membership Testing

You can check if an element exists in a tuple using the in keyword, which returns True if the element is present and False otherwise.

Example:

```
tuple1 = (1, 2, 3, 4, 5)

# Check if an element exists in the tuple
print(3 in tuple1)  # Output: True
print(6 in tuple1)  # Output: False
```