A tuple in Python is similar to a [list](https://www.programiz.com/python-programming/list). The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas we can change the elements of a list.

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them.

A tuple can have any number of items and they may be of different types (integer, float, list, [string](https://www.programiz.com/python-programming/string), etc.).

# Different types of tuples

# Empty tuple

my\_tuple = ()print(my\_tuple)

# Tuple having integers

my\_tuple = (1, 2, 3)print(my\_tuple)

# tuple with mixed datatypes

my\_tuple = (1, "Hello", 3.4)print(my\_tuple)

# nested tuple

my\_tuple = ("mouse", [8, 4, 6], (1, 2, 3))print(my\_tuple)

****Output****

()

(1, 2, 3)

(1, 'Hello', 3.4)

('mouse', [8, 4, 6], (1, 2, 3))

A tuple can also be created without using parentheses. This is known as tuple packing.

my\_tuple = 3, 4.6, "dog"print(my\_tuple)

# tuple unpacking is also possible

a, b, c = my\_tuple

print(a) # 3print(b) # 4.6print(c) # dog

****Output****

(3, 4.6, 'dog')

3

4.6

dog

Creating a tuple with one element is a bit tricky.

Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is, in fact, a tuple.

my\_tuple = ("hello")print(type(my\_tuple)) # <class 'str'>

# Creating a tuple having one element

my\_tuple = ("hello",)print(type(my\_tuple)) # <class 'tuple'>

# Parentheses is optional

my\_tuple = "hello",print(type(my\_tuple)) # <class 'tuple'>

****Output****

<class 'str'>

<class 'tuple'>

<class 'tuple'>

## Access Tuple Elements

There are various ways in which we can access the elements of a tuple.

### **1. Indexing**

We can use the index operator [] to access an item in a tuple, where the index starts from 0.

So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an index outside of the tuple index range(6,7,... in this example) will raise an IndexError.

The index must be an integer, so we cannot use float or other types. This will result in TypeError.

Likewise, nested tuples are accessed using nested indexing, as shown in the example below.

# Accessing tuple elements using indexing

my\_tuple = ('p','e','r','m','i','t')

print(my\_tuple[0]) # 'p' print(my\_tuple[5]) # 't'

# IndexError: list index out of range# print(my\_tuple[6])

# Index must be an integer# TypeError: list indices must be integers, not float# my\_tuple[2.0]

# nested tuple

n\_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested indexprint(n\_tuple[0][3]) # 's'print(n\_tuple[1][1]) # 4

****Output****

p

t

s

4

### **2. Negative Indexing**

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on.

# Negative indexing for accessing tuple elements

my\_tuple = ('p', 'e', 'r', 'm', 'i', 't')

# Output: 't'print(my\_tuple[-1])

# Output: 'p'print(my\_tuple[-6])

### **3. Slicing**

We can access a range of items in a tuple by using the slicing operator colon :.

# Accessing tuple elements using slicing

my\_tuple = ('p','r','o','g','r','a','m','i','z')

# elements 2nd to 4th# Output: ('r', 'o', 'g')print(my\_tuple[1:4])

# elements beginning to 2nd# Output: ('p', 'r')print(my\_tuple[:-7])

# elements 8th to end# Output: ('i', 'z')print(my\_tuple[7:])

# elements beginning to end# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')print(my\_tuple[:])

****Output****

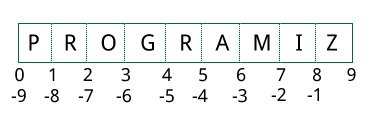
('r', 'o', 'g')

('p', 'r')

('i', 'z')

('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the tuple.

Element Slicing in Python

## Changing a Tuple

Unlike lists, tuples are immutable.

This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like a list, its nested items can be changed.

We can also assign a tuple to different values (reassignment).

# Changing tuple values

my\_tuple = (4, 2, 3, [6, 5])

# TypeError: 'tuple' object does not support item assignment# my\_tuple[1] = 9

# However, item of mutable element can be changed

my\_tuple[3][0] = 9 # Output: (4, 2, 3, [9, 5])print(my\_tuple)

# Tuples can be reassigned

my\_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')print(my\_tuple)

****Output****

(4, 2, 3, [9, 5])

('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

We can use + operator to combine two tuples. This is called ****concatenation****.

We can also ****repeat**** the elements in a tuple for a given number of times using the \* operator.

Both + and \* operations result in a new tuple.

# Concatenation# Output: (1, 2, 3, 4, 5, 6)print((1, 2, 3) + (4, 5, 6))

# Repeat# Output: ('Repeat', 'Repeat', 'Repeat')print(("Repeat",) \* 3)

****Output****

(1, 2, 3, 4, 5, 6)

('Repeat', 'Repeat', 'Repeat')

## Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. It means that we cannot delete or remove items from a tuple.

Deleting a tuple entirely, however, is possible using the keyword [del](https://www.programiz.com/python-programming/keyword-list" \l "del).

# Deleting tuples

my\_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# can't delete items# TypeError: 'tuple' object doesn't support item deletion# del my\_tuple[3]

# Can delete an entire tupledel my\_tuple

# NameError: name 'my\_tuple' is not definedprint(my\_tuple)

****Output****

Traceback (most recent call last):

File "<string>", line 12, in <module>

NameError: name 'my\_tuple' is not defined

## Tuple Methods

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

Some examples of Python tuple methods:

my\_tuple = ('a', 'p', 'p', 'l', 'e',)

print(my\_tuple.count('p')) # Output: 2print(my\_tuple.index('l')) # Output: 3

****Output****

2

3