 TUPLE

A tuple is a collection of objects which are ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Difference between tuple and list

### **Mutability**

This is the most important difference that distinguishes tuples from lists. Mutability means the ability to change. In the context of Python objects, a mutable object can be changed by adding or removing items.

[***Lists***](https://techvidvan.com/tutorials/python-lists/) are mutable in nature which means we can add or remove items from the list. [***Tuples***](https://techvidvan.com/tutorials/python-tuples/) are immutable so we cannot change, add or remove any item once a tuple is created.

### **Methods**

There are differences in the available methods on lists and tuples. Some methods are common in both the data structures. Other methods like append(), insert(), pop(), remove(), reverse() and sort() are available in lists while tuples have less methods because of its nature of immutability.

#### **Python List Methods**

There are 46 available methods on lists.

**Code:**

len(dir( list))

dir( list )

**Output:**

46  
[‘\_\_add\_\_’, ‘\_\_class\_\_’, ‘\_\_contains\_\_’, ‘\_\_delattr\_\_’, ‘\_\_delitem\_\_’, ‘\_\_dir\_\_’, ‘\_\_doc\_\_’, ‘\_\_eq\_\_’, ‘\_\_format\_\_’, ‘\_\_ge\_\_’, ‘\_\_getattribute\_\_’, ‘\_\_getitem\_\_’, ‘\_\_gt\_\_’, ‘\_\_hash\_\_’, ‘\_\_iadd\_\_’, ‘\_\_imul\_\_’, ‘\_\_init\_\_’, ‘\_\_init\_subclass\_\_’, ‘\_\_iter\_\_’, ‘\_\_le\_\_’, ‘\_\_len\_\_’, ‘\_\_lt\_\_’, ‘\_\_mul\_\_’, ‘\_\_ne\_\_’, ‘\_\_new\_\_’, ‘\_\_reduce\_\_’, ‘\_\_reduce\_ex\_\_’, ‘\_\_repr\_\_’, ‘\_\_reversed\_\_’, ‘\_\_rmul\_\_’, ‘\_\_setattr\_\_’, ‘\_\_setitem\_\_’, ‘\_\_sizeof\_\_’, ‘\_\_str\_\_’, ‘\_\_subclasshook\_\_’, ‘append’, ‘clear’, ‘copy’, ‘count’, ‘extend’, ‘index’, ‘insert’, ‘pop’, ‘remove’, ‘reverse’, ‘sort’]

#### **Python Tuple Methods**

There are 33 available methods on tuples.

**Code:**

len(dir( tuple))

print( dir( tuple ) )

**Output:**

33  
[‘\_\_add\_\_’, ‘\_\_class\_\_’, ‘\_\_contains\_\_’, ‘\_\_delattr\_\_’, ‘\_\_dir\_\_’, ‘\_\_doc\_\_’, ‘\_\_eq\_\_’, ‘\_\_format\_\_’, ‘\_\_ge\_\_’, ‘\_\_getattribute\_\_’, ‘\_\_getitem\_\_’, ‘\_\_getnewargs\_\_’, ‘\_\_gt\_\_’, ‘\_\_hash\_\_’, ‘\_\_init\_\_’, ‘\_\_init\_subclass\_\_’, ‘\_\_iter\_\_’, ‘\_\_le\_\_’, ‘\_\_len\_\_’, ‘\_\_lt\_\_’, ‘\_\_mul\_\_’, ‘\_\_ne\_\_’, ‘\_\_new\_\_’, ‘\_\_reduce\_\_’, ‘\_\_reduce\_ex\_\_’, ‘\_\_repr\_\_’, ‘\_\_rmul\_\_’, ‘\_\_setattr\_\_’, ‘\_\_sizeof\_\_’, ‘\_\_str\_\_’, ‘\_\_subclasshook\_\_’, ‘count’, ‘index’]

### Size Difference

Python allocates memory to tuples in terms of larger blocks with a low overhead because they are immutable. On the other hand, for lists, Pythons allocates small memory blocks. At the end of it, the tuple will have a smaller memory compared to the list. This makes tuples a bit faster than lists when you have a large number of elements.

For example:

tuple\_names = ('Nicholas', 'Michelle', 'Alex')

list\_names = ['Nicholas', 'Michelle', 'Alex']

print(tuple\_names.\_\_sizeof\_\_())

print(list\_names.\_\_sizeof\_\_())

**Output:**

48

64

### Homogeneous vs. Heterogeneous

Tuples are used to store heterogeneous elements, which are elements belonging to different data types. Lists, on the other hand, are used to store homogenous elements, which are elements that belong to the same type.

However, note that this is only a semantic difference. You can store elements of the same type in a tuple and elements of different types in a list. For example:

list\_elements = ['Nicholas', 10, 'Alex']

tuple\_elements = ('Nicholas', "Michelle", 'Alex')

Syntax

Tuples are initiated with rounded brackets while lists with square brackets

Execution of tuple is faster than lists

Size occupied with tuples is less the the list

Tuples are used where we don’t need to change, add or remove any element. Using tuples also indicated developers that the value is not meant to change.

**2.** If you have to update, add or remove an element in a collection then lists should be used.

**3.** Tuples are faster than lists when iterating over the elements. So if you are defining a constant set of values that you need to just iterate to then tuples should be the better choice for you.

**4.** You can’t create a dictionary with lists as keys.

**Creating a Tuple**

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.).

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) # 3

print(b) # 4.6

print(c) # 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'>

## 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 index

print(n\_tuple[0][3]) # 's'

print(n\_tuple[1][1]) # 4

**Output**

p

t

s

4

### Accessing elements from nested tuples

Lets understand how the double indexes are used to access the elements of nested tuple. The first index represents the element of main tuple and the second index represent the element of the nested tuple.

In the following example, when I used my\_data[2][1], it accessed the second element of the nested tuple. Because 2 represented the third element of main tuple which is a tuple and the 1 represented the second element of that tuple.

my\_data = (1, "Steve", (11, 22, 33))

# prints 'v'

print(my\_data[1][3])

# prints 22

print(my\_data[2][1])

Output:

v

22

### 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])

**Output**

t

p

### 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

### Slicing operation in tuples

my\_data = (11, 22, 33, 44, 55, 66, 77, 88, 99)

print(my\_data)

# elements from 3rd to 5th

# prints (33, 44, 55)

print(my\_data[2:5])

# elements from start to 4th

# prints (11, 22, 33, 44)

print(my\_data[:4])

# elements from 5th to end

# prints (55, 66, 77, 88, 99)

print(my\_data[4:])

# elements from 5th to second last

# prints (55, 66, 77, 88)

print(my\_data[4:-1])

# displaying entire tuple

print(my\_data[:])

## 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 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#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 tuple

del my\_tuple

# NameError: name 'my\_tuple' is not defined

print(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: 2

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

**Output**

2

3

### Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword in.

# Membership test in tuple

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

# In operation

print('a' in my\_tuple)

print('b' in my\_tuple)

# Not in operation

print('g' not in my\_tuple)

**Output**

True

False

True

### 2. Iterating Through a Tuple

We can use a for loop to iterate through each item in a tuple.

# Using a for loop to iterate through a tuple

for name in ('John', 'Kate'):

print("Hello", name)

**Output**

Hello John

Hello Kate

### Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword in.

# Membership test in tuple

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

# In operation

print('a' in my\_tuple)

print('b' in my\_tuple)

# Not in operation

print('g' not in my\_tuple)

**Output**

True

False

True

Creating empty tuple

* There are two ways to initialize an **empty tuple**.
* You can initialize an **empty tuple** by having () with no values in them.
* You can also initialize an **empty tuple** by using the **tuple** function.
* A **tuple** with values can be initialized by **making** a sequence of values separated by commas.

But one more important thing is, in most of the cases, empty tuple does nothing, because it cannot be manipulated. The only allowed methods on a tuple are:

1. ***count()***
2. ***index()***

**Concatenation of Tuples**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| # Code for concatenating 2 tuples    tuple1 = (0, 1, 2, 3)  tuple2 = ('python', 'geek')    # Concatenating above two  print(tuple1 + tuple2) |

# Code for creating nested tuples

tuple1 = (0, 1, 2, 3)

tuple2 = ('python', 'geek')

tuple3 = (tuple1, tuple2)

print(tuple3)

# Code to create a tuple with repetition

tuple3 = ('python',)\*3

print(tuple3)

# code to test slicing

tuple1 = (0 ,1, 2, 3)

print(tuple1[1:])

print(tuple1[::-1])

print(tuple1[2:4])

# Code for deleting a tuple

tuple3 = ( 0, 1)

del tuple3

print(tuple3)

# Code for printing the length of a tuple

tuple2 = ('python', 'geek')

print(len(tuple2))

# Code for converting a list and a string into a tuple

list1 = [0, 1, 2]

print(tuple(list1))

print(tuple('python')) # string 'python'

# A python program to demonstrate the use of

# cmp(), max(), min()

tuple1 = ('python', 'geek')

tuple2 = ('coder', 1)

if (cmp(tuple1, tuple2) != 0):

# cmp() returns 0 if matched, 1 when not tuple1

# is longer and -1 when tuple1 is shoter

print('Not the same')

else:

print('Same')

print ('Maximum element in tuples 1,2: ' +

str(max(tuple1)) + ',' +

str(max(tuple2)))

print ('Minimum element in tuples 1,2: ' +

str(min(tuple1)) + ',' + str(min(tuple2)))

the sys.getsizeof() function includes the garbage collector overhead if any:

getsizeof() calls the object’s \_\_sizeof\_\_ method and adds an additional garbage collector overhead if the object is managed by the garbage collector.

The overhead of an empty tuple is 56 bytes vs. the 72 of a listThis is only the space required to store the list structure itself (which is an array of pointers to the Python objects for each element). A 32-bit system will require 4 bytes per element, a 64-bit system will use 8 bytes per element.

An empty string takes 37 bytes, and each additional character adds another byte. That says a lot about the tradeoffs of keeping multiple short strings where you’ll pay the 37 bytes overhead for each one vs. a single long string where you pay the overhead only once.

Unicode strings behave similarly, except the overhead is 50 bytes and each additional character adds 2 bytes. That’s something to consider if you use libraries that return Unicode strings, but your text can be represented as simple strings.

By the way, in Python 3, strings are always Unicode and the overhead is 49 bytes (they saved a byte somewhere).

The list doesn’t contain the int objects themselves. It just contains an 8-byte (on 64-bit versions of CPython) pointer to the actual int object. What that means is that the getsizeof() function doesn’t return the actual memory of the list and all the objects it contains, but only the memory of the list and the pointers to its objects