**Tuples in Python**

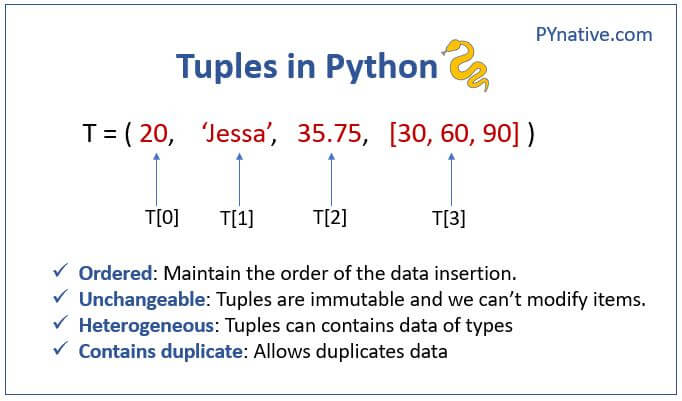
We will learn how to use a tuple data structure in Python. Also, learn how to create, access, and modify a tuple in Python and all other operations we can perform on a tuple.

## **What is a Tuple**

**Tuples are ordered collections of heterogeneous data that are unchangeable**. Heterogeneous means tuple can store variables of all types.

Tuple has the following characteristics

* **Ordered**: Tuples are part of sequence data types, which means they hold the order of the data insertion. It maintains the index value for each item.
* **Unchangeable**: Tuples are unchangeable, which means that we cannot add or delete items to the tuple after creation.
* **Heterogeneous**: Tuples are a sequence of data of different data types (like integer, float, list, string, etc;) and can be accessed through indexing and slicing.
* **Contains Duplicates**: Tuples can contain duplicates, which means they can have items with the same value.



## **Table of contents**

* [What is a Tuple](https://pynative.com/python-tuples/#h-what-is-a-tuple)
* [Creating a Tuple](https://pynative.com/python-tuples/#h-creating-a-tuple)
  + [Create a tuple with a single item](https://pynative.com/python-tuples/#h-create-a-tuple-with-a-single-item)
  + [Packing and Unpacking](https://pynative.com/python-tuples/#h-packing-and-unpacking)
* [Length of a Tuple](https://pynative.com/python-tuples/#h-length-of-a-tuple)
* [Iterating a Tuple](https://pynative.com/python-tuples/#h-iterating-a-tuple)
* [Accessing items of a Tuple](https://pynative.com/python-tuples/#h-accessing-items-of-a-tuple)
  + [Indexing](https://pynative.com/python-tuples/#h-indexing)
  + [Negative Indexing](https://pynative.com/python-tuples/#h-negative-indexing)
  + [Slicing a tuple](https://pynative.com/python-tuples/#h-slicing-a-tuple)
* [Finding an item in a Tuple](https://pynative.com/python-tuples/#h-finding-an-item-in-a-tuple)
  + [Find within a range](https://pynative.com/python-tuples/#h-find-within-a-range)
  + [Checking if an item exists](https://pynative.com/python-tuples/#h-checking-if-an-item-exists)
* [Adding and changing items in a Tuple](https://pynative.com/python-tuples/#h-adding-and-changing-items-in-a-tuple)
  + [Modify nested items of a tuple](https://pynative.com/python-tuples/#h-modify-nested-items-of-a-tuple)
* [Removing items from a tuple](https://pynative.com/python-tuples/#h-removing-items-from-a-tuple)
  + [Using del keyword](https://pynative.com/python-tuples/#h-using-del-keyword)
  + [By converting it into a List](https://pynative.com/python-tuples/#h-by-converting-it-into-a-list)
* [Count the occurrence of an item in a tuple](https://pynative.com/python-tuples/#h-count-the-occurrence-of-an-item-in-a-tuple)
* [Copying a tuple](https://pynative.com/python-tuples/#h-copying-a-tuple)
* [Concatenating two Tuples](https://pynative.com/python-tuples/#h-concatenating-two-tuples)
  + [Using the + operator](https://pynative.com/python-tuples/#h-using-the-operator)
  + [Using the sum() function](https://pynative.com/python-tuples/#h-using-the-sum-function)
  + [Using the chain() function](https://pynative.com/python-tuples/#h-using-the-chain-function)
* [Nested tuples](https://pynative.com/python-tuples/#h-nested-tuples)
* [Use built-in functions with tuple](https://pynative.com/python-tuples/#h-use-built-in-functions-with-tuple)
  + [min() and max()](https://pynative.com/python-tuples/#h-min-and-max)
  + [all()](https://pynative.com/python-tuples/#h-all)
  + [any()](https://pynative.com/python-tuples/#h-any)
* [When to use Tuple?](https://pynative.com/python-tuples/#h-when-to-use-tuple)
* [Summary of tuples operations](https://pynative.com/python-tuples/#h-summary-of-tuples-operations)

## **Creating a Tuple**

We can create a tuple using the two ways

1. **Using parenthesis ():**A tuple is created by enclosing comma-separated items inside rounded brackets.
2. **Using a tuple() constructor:** Create a tuple by passing the comma-separated items inside the tuple().

**Example**

A tuple can have items of different data type integer, float, list, string, etc;

*# create a tuple using ()  
# number tuple*

number\_tuple = (10, 20, 25.75)  
print(number\_tuple)

*# Output (10, 20, 25.75)*

*# string tuple*

string\_tuple = ('Jessa', 'Emma', 'Kelly')  
print(string\_tuple)

*# Output ('Jessa', 'Emma', 'Kelly')  
  
# mixed type tuple*

sample\_tuple = ('Jessa', 30, 45.75, [25, 78])  
print(sample\_tuple)  
*# Output ('Jessa', 30, 45.75, [25, 78])  
  
# create a tuple using tuple() constructor*

sample\_tuple2 = tuple(('Jessa', 30, 45.75, [23, 78]))  
print(sample\_tuple2)

*# Output ('Jessa', 30, 45.75, [23, 78])*

As we can see in the above output, the different items are added in the tuple like integer, string, and list.

### **Create a tuple with a single item**

A single item tuple is created by enclosing one item inside parentheses followed by a comma. If the tuple time is a string enclosed within parentheses and not followed by a comma, Python treats it as a str type. Let us see this with an example.

*# without comma*

single\_tuple = ('Hello')  
print(type(single\_tuple))  
*# Output class 'str'*

print(single\_tuple)  
*# Output Hello  
  
# with comma*

single\_tuple1 = ('Hello',)

*# output class 'tuple'*

print(type(single\_tuple1))

*# Output ('Hello',)*

print(single\_tuple1)

As we can see in the above output the first time, we did not add a comma after the “Hello”. So the variable type was class str, and the second time it was a class tuple.

### **Packing and Unpacking**

A tuple can also be created without using a tuple() constructor or enclosing the items inside the parentheses. It is called the variable “Packing.”

In Python, we can create a tuple by packing a group of variables. Packing can be used when we want to collect multiple values in a single variable. Generally, this operation is referred to as tuple packing.

Similarly, we can unpack the items by just assigning the tuple items to the same number of variables. This process is called “Unpacking.”

Let us see this with an example.

*# packing variables into tuple*tuple1 = 1, 2, "Hello"

*# display tuple*print(tuple1)  
*# Output (1, 2, 'Hello')*print(type(tuple1))  
*# Output class 'tuple'  
  
# unpacking tuple into variable*i, j, k = tuple1

*# printing the variables*print(i, j, k)  
*# Output 1 2 Hello*

As we can see in the above output, three tuple items are assigned to individual variables i, j, k, respectively.

In case we assign fewer variables than the number of items in the tuple, we will get the value error with the message too many values to unpack.

**Length of a Tuple**

We can find the length of the tuple using the len() function. This will return the number of items in the tuple.

tuple1 = ('P', 'Y', 'T', 'H', 'O', 'N')

# length of a tuple

**print**(**len**(tuple1))

# Output 6

## **Iterating a Tuple**

We can iterate a tuple using a for loop Let us see this with an example.

# create a tuple

sample\_tuple = **tuple**((1, 2, 3, "Hello", [4, 8, 16]))

# iterate a tuple

**for** item **in** sample\_tuple:

**print**(item)

**Output**

1

2

3

Hello

[4, 8, 16]

As we can see in the above output we are printing each and every item in the tuple using a loop.

## **Accessing items of a Tuple**

Tuple can be accessed through indexing and slicing. This section will guide you by accessing tuple using the following two ways

* **Using indexing**, we can access any item from a tuple using its index number
* **Using slicing**, we can access a range of items from a tuple

### **Indexing**

A tuple is an ordered sequence of items, which means they hold the order of the data insertion. It maintains the index value for each item.

We can access an item of a tuple by using its index number inside the index operator [] and this process is called **“Indexing”**.

**Note**:

* As tuples are ordered sequences of items, the index values start from 0 to the tuple’s length.
* Whenever we try to access an item with an index more than the tuple’s length, it will throw the 'Index Error'.
* Similarly, the index values are always integer. If we give any other type, then it will throw Type Error.



In the above image, we can see that the index values start from zero and it goes till the last item whose index value will be len(tuple) - 1 .

tuple1 = ('P', 'Y', 'T', 'H', 'O', 'N')

**for** i **in** **range**(4):

**print**(tuple1[i])

**Output**

P

Y

T

H

As seen in the above example, we print the tuple’s first four items with the indexing.

**Note**: If we mention the index value greater than the length of a tuple then it will throw an index error.

tuple1 = ('P', 'Y', 'T', 'H', 'O', 'N')  
  
*# IndexError: tuple index out of range*print(tuple1[7])

Also, if you mention any index value other than integer then it will throw Type Error.

tuple1 = ('P', 'Y', 'T', 'H', 'O', 'N')

# TypeError: tuple indices must be integers or slices, not float

**print**(tuple1[2.0])

### **Negative Indexing**

The index values can also be negative, with the last but the first items having the index value as -1 and second last -2 and so on.

For example, We can access the last item of a tuple using tuple\_name[-1].

Let’s do two things here

* Access tuple items using the negative index value
* Iterate tuple using negative indexing

**Example**

tuple1 = ('P', 'Y', 'T', 'H', 'O', 'N')

*# Negative indexing  
# print last item of a tuple*print(tuple1[-1]) *# N*

*# print second last*print(tuple1[-2]) *# O  
  
# iterate a tuple using negative indexing*for i in range(-6, 0):  
 print(tuple1[i], end=", ")  
*# Output P, Y, T, H, O, N,*

### **Slicing a tuple**

We can even specify a range of items to be accessed from a tuple using the technique called ‘Slicing.’ The operator used is ':'.

We can specify the start and end values for the **range of items to be accessed from the tuple**. The output will be a tuple, and it includes the range of items with the index values from the start till the end of the range. The end value item will be excluded.

We should keep in mind that the index value always starts with a 0.

For easy understanding, we will be using an integer tuple with values from 0 to 9 similar to how an index value is assigned.

**tuple1 = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
  
*# slice a tuple with start and end index number*print(tuple1[1:5])  
*# Output (1, 2, 3, 4)***

As seen in the above output the values starting from 1 to 4 are printed. Here the last value in the range 5 is excluded.

**Note**:

* If the start value is not mentioned while slicing a tuple, then the values in the tuples start from the first item until the end item in the range. Again the end item in the range will be excluded.
* Similarly, we can mention a slicing range without the end value. In that case, the item with the index mentioned in the start value of the range till the end of the tuple will be returned.

**Example:**

tuple1 = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
  
*# slice a tuple without start index*print(tuple1[:5])  
*# Output (0, 1, 2, 3, 4)  
  
# slice a tuple without end index*print(tuple1[6:])  
*# Output (6, 7, 8, 9, 10)*

Similarly, we can slice tuple using negative indexing as well. The last but first item will have the index -1.

tuple1 = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

# slice a tuple using negative indexing

**print**(tuple1[-5:-1])

# Output (6, 7, 8, 9)

Here we can see that the items with the negative indexes starting from -1 till -4 are printed excluding -5.

## **Finding an item in a Tuple**

We can search for a certain item in a tuple using the index() method and it will return the position of that particular item in the tuple.

The index() method accepts the following three arguments

1. **item** – The item which needs to be searched
2. **start** – (Optional) The starting value of the index from which the search will start
3. **end** – (Optional) The end value of the index search

**Example**

tuple1 = (10, 20, 30, 40, 50)

# get index of item 30

position = tuple1.index(30)

**print**(position)

# Output 2

As seen in the above output the index value of item 30 is printed.

### **Find within a range**

We can mention the start and end values for the index() method so that our search will be limited to those values.

Example

tuple1 = (10, 20, 30, 40, 50, 60, 70, 80)

# Limit the search locations using start and end

# search only from location 4 to 6

# start = 4 and end = 6

# get index of item 60

position = tuple1.index(60, 4, 6)

**print**(position)

# Output 5

As seen in the above output we have limited the search from the index position 4 to 6 as the number 60 is present in this range only. In case we mention any item that is not present then it will throw a value error.

tuple1 = (10, 20, 30, 40, 50, 60, 70, 80)

#index out of range

position= tuple1 .index(10)

**print**(postion)

# Output ValueError: tuple.index(x): x not in tuple

### **Checking if an item exists**

We can check whether an item exists in a tuple by using the in operator. This will return a boolean True if the item exists and False if it doesn’t.

tuple1 = (10, 20, 30, 40, 50, 60, 70, 80)

# checking whether item 50 exists in tuple

**print**(50 **in** tuple1)

# Output True

**print**(500 **in** tuple1)

# Output False

As seen in the above output we can see that the item ’50’ exists in the tuple so we got True and ‘500’ doesn’t and so we got False.

## **Adding and changing items in a Tuple**

A list is a mutable type, which means we can add or modify values in it, but tuples are immutable, so they cannot be changed.

Also, because a tuple is immutable there are no built-in methods to add items to the tuple.

If you try to modify the value you will get an error.

**Example**

tuple1 = (0, 1, 2, 3, 4, 5)

tuple1[1] = 10

# Output TypeError: 'tuple' object does not support item assignment

As a workaround solution, we can convert the tuple to a list, add items, and then convert it back to a tuple. As tuples are ordered collection like lists the items always get added in the end.

tuple1 = (0, 1, 2, 3, 4, 5)

# converting tuple into a list

sample\_list = **list**(tuple1)

# add item to list

sample\_list.append(6)

# converting list back into a tuple

tuple1 = **tuple**(sample\_list)

**print**(tuple1)

# Output (0, 1, 2, 3, 4, 5, 6)

As we can see in the above output the item is added to the tuple in the end.

### **Modify nested items of a tuple**

One thing to remember here, If one of the items is itself a mutable data type as a list, then we can change its values in the case of a nested tuple.

For example, let’s assume you have the following tuple which has a list as its last item and you wanted to modify the list items.

tuple1 = (10, 20, **[25, 75, 85]**)

Let’s see how to modify the set item if it contains mutable types.

**Example**

tuple1 = (10, 20, [25, 75, 85])

# before update

**print**(tuple1)

# Output (10, 20, [25, 75, 85])

# modify last item's first value

tuple1[2][0] = 250

# after update

**print**(tuple1)

# Output (10, 20, [250, 75, 85])

As tuples are immutable we cannot change the values of items in the tuple. Again with the same workaround we can convert it into a list, make changes and convert it back into a tuple.

tuple1 = (0, 1, 2, 3, 4, 5)

# converting tuple into a list

sample\_list = **list**(tuple1)

# modify 2nd item

sample\_list[1] = 10

# converting list back into a tuple

tuple1 = **tuple**(sample\_list)

**print**(tuple1)

# Output (0, 10, 2, 3, 4, 5)

As we can see in the above output the last item has been updated from 1 to 10.

## **Removing items from a tuple**

Tuples are immutable so there are no pop() or remove() methods for the tuple. We can remove the items from a tuple using the following two ways.

1. Using del keyword
2. By converting it into a list

### **Using del keyword**

The del keyword will delete the entire tuple.

sampletup1 =(0,1,2,3,4,5,6,7,8,9,10)

**del** sampletup1

**print**(sampletup1)

**OUTPUT:**

NameError: name 'sampletup1' is not defined

As seen in the above output we are getting error when we try to access a deleted tuple.

### **By converting it into a List**

We can convert a tuple into a list and then remove any one item using the remove() method. Then again we will convert it back into a tuple using the tuple() constructor.

tuple1 = (0, 1, 2, 3, 4, 5)

# converting tuple into a list

sample\_list = **list**(tuple1)

# reomve 2nd item

sample\_list.remove(2)

# converting list back into a tuple

tuple1 = **tuple**(sample\_list)

**print**(tuple1)

# Output (0, 1, 3, 4, 5)

As seen in the above output item 3 has been removed from the tuple.

## **Count the occurrence of an item in a tuple**

As we learned, a tuple can contain duplicate items. To determine how many times a specific item occurred in a tuple, we can use the count() method of a tuple object.

The count() method accepts any value as a parameter and returns the number of times a particular value appears in a tuple.

**Example**

tuple1 = (10, 20, 60, 30, 60, 40, 60)

# Count all occurrences of item 60

count = tuple1.count(60)

**print**(count)

# Output 3

count = tuple1.count(600)

**print**(count)

# Output 0

## **Copying a tuple**

We can create a copy of a tuple using the assignment operator '=' . This operation will create only a reference copy and not a deep copy because tuples are immutable.

tuple1 = (0, 1, 2, 3, 4, 5)

# copy tuple

tuple2 = tuple1

**print**(tuple2)

# Output (0, 1, 2, 3, 4, 5)

# changing tuple2

# converting it into a list

sample\_list = **list**(tuple2)

sample\_list.append(6)

# converting list back into a tuple2

tuple2 = **tuple**(sample\_list)

# printing the two tuples

**print**(tuple1)

# Output (0, 1, 2, 3, 4, 5)

**print**(tuple2)

# Output (0, 1, 2, 3, 4, 5, 6)

As we can see in the above output the tuple1 is not affected by the changes made in tuple2.

## **Concatenating two Tuples**

We can concatenate two or more tuples in different ways. One thing to note here is that tuples allow duplicates, so if two tuples have the same item, it will be repeated twice in the resultant tuple. Let us see each one of them with a small example.

### **Using the + operator**

We can add two tuples using the + operator. This is a very and straightforward method and the resultant tuple will have items from both the tuples.

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

tuple2 = (3, 4, 5, 6, 7)

# concatenate tuples using + operator

tuple3 = tuple1 + tuple2

**print**(tuple3)

# Output (1, 2, 3, 4, 5, 3, 4, 5, 6, 7)

As seen in the above output the resultant tuple has items from both the tuples and the item 3, 4, 5 are repeated twice.

### **Using the sum() function**

We can also use the Python built-in function sum to concatenate two tuples. But the sum function of two iterables like tuples always needs to start with Empty Tuple. Let us see this with an example.

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

tuple2 = (3, 4, 5, 6, 7)

# using sum function

tuple3 = **sum**((tuple1, tuple2), ())

**print**(tuple3)

# Output (1, 2, 3, 4, 5, 3, 4, 5, 6, 7)

As we can see in the above output the sum function takes an Empty tuple as an argument and it returns the items from both the tuples.

### **Using the chain() function**

The chain() function is part of the itertools module in python. It makes an iterator, which will return all the first iterable items (a tuple in our case), which will be followed by the second iterable. We can pass any number of tuples to the chain() function.

**import** itertools

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

tuple2 = (3, 4, 5, 6, 7)

# using itertools

tuple3 = **tuple**(item **for** item **in** itertools.chain(tuple1, tuple2))

**print**(tuple3)

# Output (1, 2, 3, 4, 5, 3, 4, 5, 6, 7)

As seen in the above output we can concatenate any number of tuples using the above method and it is more time-efficient than other methods.

## **Nested tuples**

Nested tuples are tuples within a tuple i.e., when a tuple contains another tuple as its member then it is called a nested tuple.

In order to retrieve the items of the inner tuple we need a nested for loop

nested\_tuple = ((20, 40, 60), (10, 30, 50), "Python")

# access the first item of the third tuple

**print**(nested\_tuple[2][0]) # P

# iterate a nested tuple

**for** i **in** nested\_tuple:

**print**("tuple", i, "elements")

**for** j **in** i:

**print**(j, end=", ")

**print**("\n")

**Output**

P

tuple (20, 40, 60) items

20, 40, 60,

tuple (10, 30, 50) items

10, 30, 50,

tuple Python items

P, y, t, h, o, n,

## **Use built-in functions with tuple**

### **min() and max()**

As the name suggests the max() function returns the maximum item in a tuple and min() returns the minimum value in a tuple.

tuple1 = ('xyz', 'zara', 'abc')

# The Maximum value in a string tuple

**print**(**max**(tuple1))

# Output zara

# The minimum value in a string tuple

**print**(**min**(tuple1))

# Output abc

tuple2 = (11, 22, 10, 4)

# The Maximum value in a integer tuple

**print**(**max**(tuple2))

# Output 22

# The minimum value in a integer tuple

**print**(**min**(tuple2))

# Output 4

**Note**: We can’t find the max() and min() for a heterogeneous tuple (mixed types of items). It will throw Type Error

tuple3 = ('a', 'e', 11, 22, 15)

# max item

**print**(**max**(tuple3))

### **all()**

In the case of all() function, the return value will be true only when all the values inside are true. Let us see the different item values and the return values.

| **Item values in a tuple** | **Return value** |
| --- | --- |
| All values are True | True |
| One or more False values | False |
| All False values | False |
| Empty tuple | True |

*# all() with All True values*tuple1 = (1, 1, True)  
print(all(tuple1)) *# True  
  
# all() All True values*tuple1 = (1, 1, True)  
print(all(tuple1)) *# True  
  
# all() with One false value*tuple2 = (0, 1, True, 1)  
print(all(tuple2)) *# False  
  
# all() with all false values*tuple3 = (0, 0, False)  
print(all(tuple3)) *# False  
  
# all() Empty tuple*tuple4 = ()  
print(all(tuple4)) *# True*

### **any()**

The any() method will return true if there is at least one true value. In the case of the empty tuple, it will return false. Let us see the same possible combination of values for any() function in a tuple and its return values.

| **Item values in a tuple** | **Return value** |
| --- | --- |
| All values are True | True |
| One or more False values | True |
| All False values | False |
| Empty tuple | False |

Similarly, let’s see each one of the above scenarios with a small example.

*# any() with All True values*tuple1 = (1, 1, True)  
print(any(tuple1)) *# True  
  
# any() with One false value*tuple2 = (0, 1, True, 1)  
print(any(tuple2)) *# True  
  
# any() with all false values*tuple3 = (0, 0, False)  
print(any(tuple3)) *# False  
  
# any() with Empty tuple*tuple4 = ()  
print(any(tuple4)) *# False*

## **When to use Tuple?**

As tuples and lists are similar data structures, and they both allow sequential data storage, tuples are often referred to as immutable lists. So the tuples are used for the following requirements instead of lists.

* There are no append() or extend() to add items and similarly no remove() or pop() methods to remove items. This ensures that the data is write-protected. As the tuples are Unchangeable, they can be used to represent read-only or fixed data that does not change.
* As they are immutable, they can be used as a key for the dictionaries, while lists cannot be used for this purpose.
* As they are immutable, the search operation is much faster than the lists. This is because the id of the items remains constant.
* Tuples contain heterogeneous (all types) data that offers huge flexibility in data that contains combinations of data types like alphanumeric characters.

## **Summary of tuples operations**

For the following examples, we assume that t1 and t2 are tuples, x, i, j, k, n are integers.

t1 = (10, 20, 30, 40, 50) and t2 = (60, 70, 80, 60)

| **Operation** | **Description** |
| --- | --- |
| x in t1 | Check if the tuple t1 contains the item x. |
| x not in t2 | Check if the tuple t1 does not contain the item x. |
| t1 + t2 | Concatenate the tuples t1 and t2. Creates a new tuple containing the items from t1 and t2. |
| t1 \* 5 | Repeat the tuple t1 5 times. |
| t1[i] | Get the item at the index i. Example, t1[2] is 30 |
| t1[i:j] | Tuple slicing. Get the items from index i up to index j (excluding j) as a tuple. An example t1[0:2] is (10, 20) |
| t1[i:j:k] | Tuple slicing with step. Return a tuple with the items from index i up to index j taking every k-th item. An example t1[0:4:2] is (10, 30) |
| len(t1) | Returns a count of total items in a tuple |
| t2.count(60) | Returns the number of times a particular item (60) appears in a tuple. Answer is 2 |
| t1.index(30) | Returns the index number of a particular item(30) in a tuple. Answer is 2 |
| t1.index(40, 2, 5) | Returns the index number of a particular item(30) in a tuple. But search only from index number 2 to 5. |
| min(t1) | Returns the item with a minimum value from a tuple |
| max(t1) | Returns the item with maximum value from a tuple |