

Python Bootcamp & Masterclass

Tuples



tuples

A tuple is an ordered sequence of objects

```
t1 = (1, 2, 3, 3, 'Hi', tuple())           # a tuple is typically enclosed in a pair of parentheses
print("object t1:", t1, "is of type:", type(t1))
```

```
object t1: (1, 2, 3, 3, 'Hi', ()) is of type: <class 'tuple'>
```

```
t2 = 4, 5, 6, 4, 4, 'Hello!'           # parentheses are optional for tuple creation
print("object t2:", t2, "is of type:", type(t2))   # on output tuples are always enclosed in parentheses
```

```
object t2: (4, 5, 6, 4, 4, 'Hello!') is of type: <class 'tuple'>
```

Python's container objects contain references (memory addresses) of their objects, **not** the objects themselves. So, when a tuple is created with x objects, it only has references of those x objects. It does not know what those objects are.

Generally, tuples are enclosed in a pair of parentheses (also called round brackets), but parentheses are optional.

Python's creator intended lists for homogenous data and tuples for heterogenous data. Tuples are Python's way of packaging heterogeneous pieces of information in a composite object.

For example, `socket = ('www.python.org', 80)` brings together a string and a number so that the host/port pair can be passed around as a socket, a composite object



empty tuple

An empty tuple can be created by using tuple constructor, `tuple ()` or by using a pair of parentheses with no values in them.

```
t3 = tuple()           # an empty tuple
print("object t3:", t3, "is of type:", type(t3), "and is of length:", len(t3))
```

```
object t3: () is of type: <class 'tuple'> and is of length: 0
```

```
t4 = ()              # an empty tuple
print("object t4:", t4, "is of type:", type(t4), "and is of length:", len(t4))
```

```
object t4: () is of type: <class 'tuple'> and is of length: 0
```

```
t5 = (tuple())       # an empty tuple
print("object t5:", t5, "is of type:", type(t5), "and is of length:", len(t5))
```

```
object t5: () is of type: <class 'tuple'> and is of length: 0
```

singleton

A tuple with just one element (singleton tuple) can be created by using a pair of parentheses with the value in them.

```
t9 = (4,)          # singleton tuple (a tuple with one element)
print("object t9:", t9, "is of type:", type(t9), "and is of length:", len(t9))
print("object t9[0]:", t9[0], "is of type:", type(t9[0]))
```

```
object t9: (4,) is of type: <class 'tuple'> and is of length: 1
object t9[0]: 4 is of type: <class 'int'>
```

```
t10 = 4,          # singleton tuple (a tuple with one element)
print("object t10:", t10, "is of type:", type(t10), "and is of length:", len(t10))
print("object t10[0]:", t10[0], "is of type:", type(t10[0]))
```

```
object t10: (4,) is of type: <class 'tuple'> and is of length: 1
object t10[0]: 4 is of type: <class 'int'>
```

Tuples are immutable, but can contain mutable elements

Though a tuple is immutable, it can contain mutable objects. For example, it can contain one or more lists. If one of those lists is modified, its reference that is kept in the tuple does not change. So, the tuple would not even get notified of the change (the list doesn't know whether it is referred to by a variable, a tuple, or another list).

```
t2 = ('TX', ['75060', '75061'])
print("t2 is: ", t2, " at: \t", hex(id(t2)))
print("List is: ", t2[1], " at: \t\t", hex(id(t2[1])))
t2[1].append('75062') # Adding a new zipcode to the list
print("t2 is: ", t2, "at: ", hex(id(t2)))
print("List is: ", t2[1], " at: \t", hex(id(t2[1])))
```

```
t2 is: ('TX', ['75060', '75061']) at: 0x192127782c0
List is: ['75060', '75061'] at: 0x19212785600
t2 is: ('TX', ['75060', '75061', '75062']) at: 0x19212783f40
List is: ['75060', '75061', '75062'] at: 0x19212785600
```

```
t3 = ('TX', ['75060', '75061'])
print("t3 is: ", t3, " at: ", hex(id(t3)))
print("List is: ", t3[1], " at: \t", hex(id(t3[1])))
del t3[1][0] # delete the first element of the list
print("t3 is: ", t3, "at: \t\t", hex(id(t3)))
print("List is: ", t3[1], " at: \t\t", hex(id(t3[1])))
#del t3[1] # cannot delete the list as 'tuple' doesn't support item deletion
t3[1][0] = '77777' # list elements can be modified, appended, inserted, deleted
print("t3 is: ", t3, "at: \t\t", hex(id(t3)))
del t3[1][0] # all elements of the list can be deleted
print("t3 is: ", t3, "at: \t\t\t", hex(id(t3)))
```

```
t3 is: ('TX', ['75060', '75061']) at: 0x19212799400
List is: ['75060', '75061'] at: 0x1921278a9c0
t3 is: ('TX', ['75061']) at: 0x19212799400
List is: ['75061'] at: 0x1921278a9c0
t3 is: ('TX', ['77777']) at: 0x19212799400
t3 is: ('TX', []) at: 0x19212799400
```

deleting a tuple

If a tuple needs to be deleted, `del` keyword can be used. A tuple can be emptied (delete all elements) by assigning an empty tuple to it. Another way to empty a tuple is by assigning the product of the tuple and `0` to the tuple itself.

```
t1 = ('TX', ['75060', '75061'])
print("t1 is: ", t1, " at: ", hex(id(t1)))
del t1
```

```
t1 is: ('TX', ['75060', '75061']) at: 0x23ec81c5c40
```

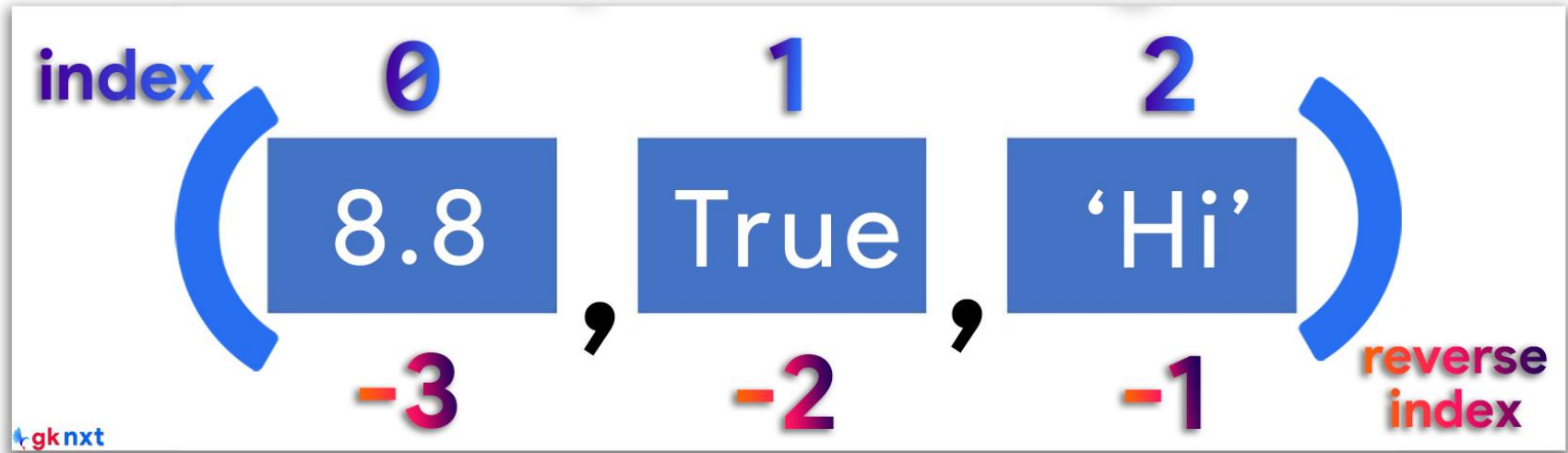
```
t2 = ('TX', ['75060', '75061'])
t2 = ()
t2
```

```
()
```

```
t3 = ('TX', ['75060', '75061'])
t3 = t3 * 0
t3
```

```
()
```


tuple indexing



A tuple is a sequence, so each item in a tuple has a numbered position called **index** that starts at `0`. Elements of a tuple can be accessed in the reverse order by using **negative index** starting with `-1` (last element has index `-1`, second-to-last element has index `-2`, and so on)

```
t1 = (8.8, True, 'Hi')
print("Data at t1[0]:", t1[0])
print("Data at t1[1]:", t1[1])
print("Data at t1[2]:", t1[2])
print('\n')
print("Data at t1[-1]:", t1[-1])
print("Data at t1[-2]:", t1[-2])
print("Data at t1[-3]:", t1[-3])
```

```
Data at t1[0]: 8.8
Data at t1[1]: True
Data at t1[2]: Hi
```

```
Data at t1[-1]: Hi
Data at t1[-2]: True
Data at t1[-3]: 8.8
```

```
t2 = ('a', 'b', 'c', 'd')
t2[ : -1]
t2[ : :-1]
t2[-1]
```

```
('a', 'b', 'c')
```

```
('d', 'c', 'b', 'a')
```

```
'd'
```

slicing

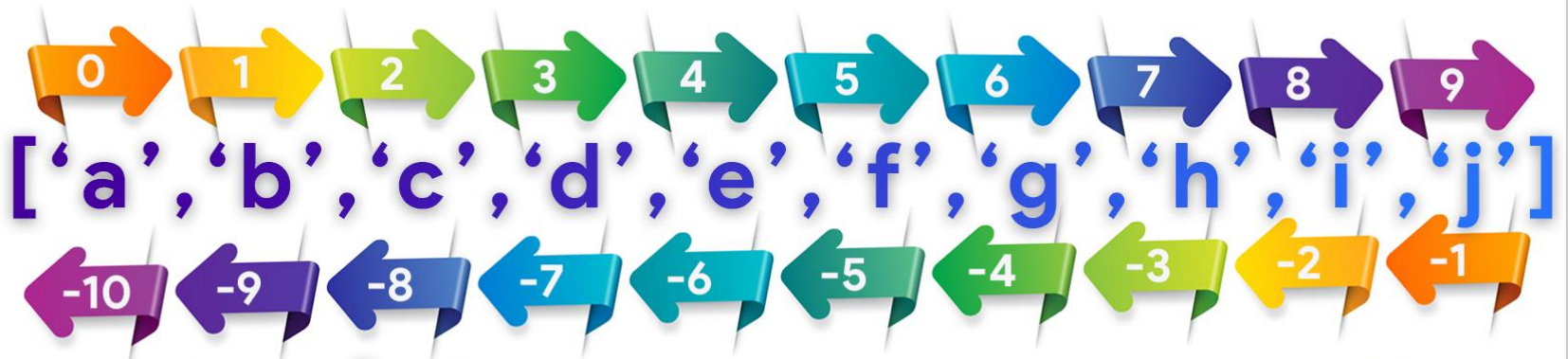
tuple[start: stop: step]

default:

0

len(tuple)

1



tuple[

:

:

-1

]

default:

-1

$-(\text{len}(\text{tuple}) + 1)$

A slice can be extracted from a tuple using the slice operator (`[]`) and colon(`:`) to separate start, stop and step options as integers (step cannot be zero)

```
t1 = ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j')
t1[2:]
t1[:8]
t1[: : 2]
```

```
('a', 'c', 'e', 'g', 'i')
```

```
t2 = ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j')
t2[:]
t2[::]
t2[0:10:]
t2[0:10:1]
t2[:10:1]
t2[0::1]
```

```
('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j')
```

```
t3 = ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j')
t3[100:]           # if the start is out of range, empty list will be the slice
t3[-100: : -1]    # if the start is out of range for reverse indexing, empty list will be the slice
t3[:100]          # if the stop is out of range, stop defaults to length of the list
t3[: -100: -1]    # if the stop is out of range for reverse indexing, stop defaults to -(length of the list + 1)
t3[: : 100]       # step can also be out of range
```

```
('a',)
```



Multiplication operator (*) can be used to replicate tuples by a factor.

```
t1 = (1, 2, 3, 'Hi')
print("t1 is:", t1, "at:\t\t", hex(id(t1)))
t2 = t1 * 2
print("t2 is:", t2, "at:\t", hex(id(t2)))
```

```
t1 is: (1, 2, 3, 'Hi') at:          0x23ec82ff9a0
t2 is: (1, 2, 3, 'Hi', 1, 2, 3, 'Hi') at:    0x23ec66d0430
```

```
t3 = ( )
print("t3 is:", t3, "at:", hex(id(t3)))
t4 = t3 * 100
print("t4 is:", t4, "at:", hex(id(t4)))
```

```
t3 is: () at: 0x23ec3544040
t4 is: () at: 0x23ec3544040
```

concatinating tuples

Tuples can be concatenated by using concatenation operator (+) or `Itertools.chain()` method. To concatenate multiple tuples, the preferred way is to use unpacking operator (*)

```
a = (1, 2, 3)
b = (4, 5, 6)
c = a + b
c
(1, 2, 3, 4, 5, 6)

import itertools

d = (1, 2, 3)
e = (4, 5, 6)
f = tuple(itertools.chain(d, e))
f
(1, 2, 3, 4, 5, 6)

p = (1, 2, 3)
q = ('a', 'b', 'c')
r = (4, 5, 6)
s = (*p, *q, *r)
s
t = (*p, *r, *q)
t
(1, 2, 3, 'a', 'b', 'c', 4, 5, 6)
(1, 2, 3, 4, 5, 6, 'a', 'b', 'c')
```

sorting a tuple

Since tuples do not have sort method, the built-in sorted function can be used for sorting tuples

```
t1 = ((7,8,9), (4,5,6), (1,2,3))  
# t1.sort() # tuples do not have sort method  
sorted(t1)  
sorted(t1, reverse=True)
```

```
[(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

```
[(7, 8, 9), (4, 5, 6), (1, 2, 3)]
```

reversing a tuple

Since tuples do not have reverse method, the built-in reversed function can be used for reversing tuples

```
t1 = (1, 2, 3)
#t1.reverse()    # tuples do not have reverse method
reversed(t1)
tuple(reversed(t1))
```

```
<reversed at 0x23ec826edc0>
```

```
(3, 2, 1)
```


list into a tuple

Tuple constructor function `tuple()` can be used to convert a list into a tuple

```
t1 = tuple([1, 2, 3, 4])  
t1
```

```
(1, 2, 3, 4)
```

```
t2 = tuple([1, 2, 3, 4, ['a', 'b']])  
t2
```

```
(1, 2, 3, 4, ['a', 'b'])
```

string into a tuple

Tuple constructor function `tuple()` can be used to convert a string into a tuple

```
t1 = tuple('Hello World!')
t1
t2 = tuple("He said, 'Hi'")
t2
('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!')
('H', 'e', ' ', 's', 'a', 'i', 'd', ',', ' ', "'", 'H', 'i', "'")
```

comparing tuples

Sequences of the same type also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. If the first elements are comparable and different, comparison is over. If the first elements are comparable and same, then only comparison continues to the next element(s)

```
(0) < (1)    # < means 'is before'  
(0) > (1)    # > means 'is after'
```

True

False

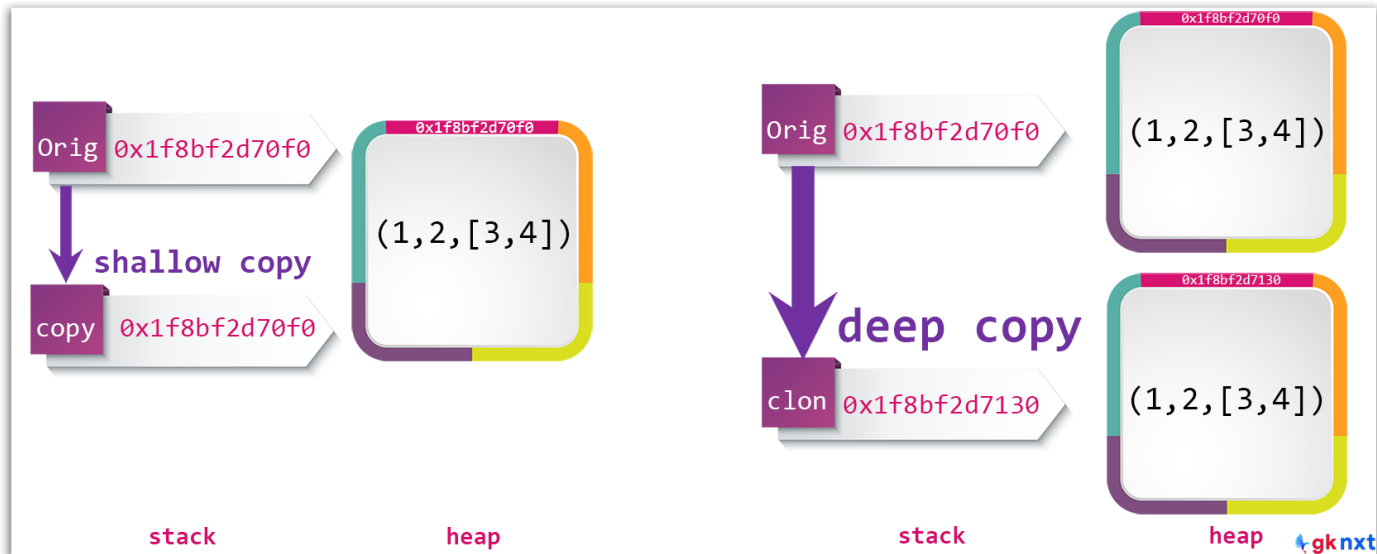
```
(0, 1) > (1, 0) # if the first elements are comparable and different, comparison is over  
(1, 1) > (1, 0) # if the first elements are comparable and same, comparison continues to the next element(s)
```

False

True

Shallow copy & deep copy


Assignment statements in Python do not create copies - they only bind names to objects. Sometimes copies of mutable objects or collections of mutable objects would be needed for data processing.



A **shallow copy** constructs a new compound object (objects that contain other objects) and then (to the extent possible) inserts references into it to the objects found in the original.

A **shallow copy** can be created in many ways.

<pre>orig = [1, 2, 3.4, True] # original list</pre>	1	<pre>import copy copy1 = copy.copy(orig)</pre>	4	<pre>copy4 = orig + []</pre>
<pre>import copy copy1 = copy.copy(orig)</pre>	2	<pre>copy2 = tuple(orig)</pre>	5	<pre>copy5 = orig * 1</pre>
<pre>copy2 = tuple(orig)</pre>	3	<pre>copy3 = orig[:]</pre>	6	<pre>copy6 = [i for i in orig]</pre>
<pre>copy3 = orig[:]</pre>				
<pre>copy4 = orig + []</pre>				
<pre>copy5 = orig * 1</pre>				
<pre>copy6 = [i for i in orig]</pre>				



```
import copy
orig = (1, 2, 3.4, True) # original tuple
copy1 = copy.copy(orig) # shallow copy
print("orig is: ", orig, "at: \t\t\t", hex(id(orig)))
print("copy1 is: ", copy1, "at: \t\t\t", hex(id(copy1)))
```

```
orig is: ('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!') at: 0x19212756130
copy1 is: ('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!') at: 0x19212756130
```

```
orig = (1, 2, 3.4, True) # original tuple
copy2 = tuple(orig) # shallow copy
print("orig is: ", orig, "at: \t\t\t", hex(id(orig)))
print("copy2 is: ", copy2, "at: \t\t\t", hex(id(copy2)))
```

```
orig is: ('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!') at: 0x19212756040
copy2 is: ('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!') at: 0x19212756040
```

```
orig = (1, 2, 3.4, True) # original tuple
copy3 = orig[:] # shallow copy
print("orig is: ", orig, "at: \t\t\t", hex(id(orig)))
print("copy3 is: ", copy3, "at: \t\t\t", hex(id(copy3)))
```

```
orig is: ('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!') at: 0x1921273f9f0
copy3 is: ('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!') at: 0x1921273f9f0
```

A **deep copy** constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original. So, **deep copy** creates a new copy at a different memory location with no connection to the original object whatsoever. It can be created with the **deepcopy()** method from the **copy** module.

```
import copy
orig = (1, ['a', 2, 5.5], True)
d_copy = copy.deepcopy(orig)
print("orig is: ", orig, "at: ", hex(id(orig)))
print("d_copy is: ", d_copy, "at:", hex(id(d_copy)))
d_copy[1][0] = 'z'
d_copy
orig
```

```
orig is: (1, ['a', 2, 5.5], True) at: 0x19212193200
d_copy is: (1, ['a', 2, 5.5], True) at: 0x19212204cc0

(1, ['a', 2, 5.5], True)
```

index()

The `tuple.index(x [, i [, j]])` method returns the first occurrence of `x` in the tuple (at or after index `i` and before index `j`) It is one of the only two methods of tuple (count is the other)

It raises **ValueError** if `x` is not in the `tuple`

```
t1 = (3, 'six', 9, -1, 9)
t1.index(9)
```

```
2
```

```
t2 = (3, 'six', 9, -1, '0')
t2.index(0)
```

```
-----
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16040\1299861094.py in <module>
      1 t2 = (3, 'six', 9, -1, '0')
----> 2 t2.index(0)
```

```
ValueError: tuple.index(x): x not in tuple
```


count()

The `tuple.count(x)` method returns the total number of occurrences of `x` in the tuple. It is one of the only two methods of tuple (index is the other)

```
t1 = (5, 2, 2, 3, 5, 2, 5)
t1.count(5)
```

```
3
```

```
t2 = (5, 2, 2, 3, 5, 2, 5)
t2.count(0)
```

```
0
```

sum()

The `sum(iterable, start=0)` built-in function returns the total of items in the `iterable` starting with `start`, if given. This method should not be used for concatenation, as there are efficient alternatives available for that.

```
t1 = (1, 2, 3, 4, 5)
sum(t1)
```

```
15
```

```
t2 = (1, 2, 3, 4, 5)
sum(t2, 10)      # start = 10
```

```
25
```

```
t3 = ((1, 2), (3, 4), ('s', 'u'))
sum(t3, ())
```

```
(1, 2, 3, 4, 's', 'u')
```

```
t4 = (1, 2, 3)
t5 = (44, 55)
sum(sum((t4, t5), ()))
```

```
105
```

len ()

The built-in function `len(iterable)` returns the length (the number of items) of the `iterable`. The length of an empty tuple is `0`

```
t1 = (  
len(t1)
```

```
0
```

```
t2 = (0,  
len(t2)
```

```
1
```

```
t3 = (23, True, ('a', 'b'), ['Hello', 44])  
len(t3)
```

```
4
```

```
t4 = (int, 'Sydney', 4+3, ['a', 'e', 'i'], print)  
len(t4)
```

```
5
```

```
t5 = ((1, 3, (4, 5), 6), 7)  
len(t5)
```

```
2
```

min()

The built-in function `min(iterable)` returns the smallest object of the `iterable`. If the objects are not comparable, `TypeError` will be raised.

```
t1 = ('xyz', 'zara', 'abc')
t2 = (23, 44, 11, 456, -111)
t3 = (23, 44, 11, True, 255)
t4 = (0.3, False, 0.44, 0.11, True, 0.255)
t5 = ('2022', '9', '890', '70', '891', '898')
print(f"Min of {t1} is: {min(t1)}")           # strings are compared lexicographically
print(f"Min of {t2} is: {min(t2)}")
print(f"Min of {t3} is: {min(t3)}")         # True evaluates to 1
print(f"Min of {t4} is: {min(t4)}")         # False evaluates to 0
print(f"Min of {t5} is: {min(t5)}")
```

```
Min of ('xyz', 'zara', 'abc') is: abc
Min of (23, 44, 11, 456, -111) is: -111
Min of (23, 44, 11, True, 255) is: True
Min of (0.3, False, 0.44, 0.11, True, 0.255) is: False
Min of ('2022', '9', '890', '70', '891', '898') is: 2022
```

```
t6 = [23, 44, 11, 'xyz', 'zara', 'abc']
min(t6)
```

```
-----
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16040\49786927.py in <module>
      1 t6 = [23, 44, 11, 'xyz', 'zara', 'abc']
----> 2 min(t6)

TypeError: '<' not supported between instances of 'str' and 'int'
```

max()

The built-in function `max(iterable)` returns the largest object of the `iterable`. If the objects are not comparable, `TypeError` will be raised.

```
t1 = ('xyz', 'zara', 'abc')
t2 = (23, 44, 11, 456, -111)
t3 = (-23, False, -44, -1, -0.255)
t4 = (0.3, False, 0.44, 0.11, True, 0.255)
t5 = ('2022', '9', '890', '70', '891', '898')
print(f"Max of {t1} is: {max(t1)}")           # strings are compared Lexicographically
print(f"Max of {t2} is: {max(t2)}")
print(f"Max of {t3} is: {max(t3)}")         # True evaluates to 1
print(f"Max of {t4} is: {max(t4)}")         # False evaluates to 0
print(f"Max of {t5} is: {max(t5)}")
```

```
Max of ('xyz', 'zara', 'abc') is: zara
Max of (23, 44, 11, 456, -111) is: 456
Max of (-23, False, -44, -1, -0.255) is: False
Max of (0.3, False, 0.44, 0.11, True, 0.255) is: True
Max of ('2022', '9', '890', '70', '891', '898') is: 9
```

```
t6 = [23, 44, 11, 'xyz', 'zara', 'abc']
max(t6)
```

```
-----
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16040\4288880778.py in <module>
      1 t6 = [23, 44, 11, 'xyz', 'zara', 'abc']
----> 2 max(t6)
```

```
TypeError: '>' not supported between instances of 'str' and 'int'
```

in & not in

The `in` and `not in` operators can be used to test whether a value is in a tuple or not.

```
t1 = ('a', 'e', 'i', 'o', 'u')  
'i' in t1
```

True

```
t2 = ('a', 'c', 'e', 'g', 'i')  
'x' not in t2
```

True

**Tuples are compact and
don't over-allocate.**

**Hence they are efficient
both in speed and space**

If the data in the collection is meant to remain constant for the life of the program, using a tuple instead of a list prevents accidental modification of the data (e.g. blocked IP Addresses)



Online Resources

For best python resources, please visit:



gknxt.com/python/

Python Bootcamp & Masterclass

Thank You
for your Rating & Review

