

# Chapter 28: Tuple

A tuple is an immutable list of values. Tuples are one of Python's simplest and most common collection types, and can be created with the comma operator (`value = 1, 2, 3`).

## Section 28.1: Tuple

Syntactically, a tuple is a comma-separated list of values:

```
t = 'a', 'b', 'c', 'd', 'e'
```

Although not necessary, it is common to enclose tuples in parentheses:

```
t = ('a', 'b', 'c', 'd', 'e')
```

Create an empty tuple with parentheses:

```
t0 = ()  
type(t0)           # <type 'tuple'>
```

To create a tuple with a single element, you have to include a final comma:

```
t1 = 'a',  
type(t1)           # <type 'tuple'>
```

Note that a single value in parentheses is not a tuple:

```
t2 = ('a')  
type(t2)           # <type 'str'>
```

To create a singleton tuple it is necessary to have a trailing comma.

```
t2 = ('a',)  
type(t2)           # <type 'tuple'>
```

Note that for singleton tuples it's recommended (see [PEP8 on trailing commas](#)) to use parentheses. Also, no white space after the trailing comma (see [PEP8 on whitespaces](#))

```
t2 = ('a',)         # PEP8-compliant  
t2 = 'a',           # this notation is not recommended by PEP8  
t2 = ('a', )        # this notation is not recommended by PEP8
```

Another way to create a tuple is the built-in function `tuple`.

```
t = tuple('lupins')  
print(t)           # ('l', 'u', 'p', 'i', 'n', 's')  
t = tuple(range(3))  
print(t)           # (0, 1, 2)
```

These examples are based on material from the book [Think Python by Allen B. Downey](#).

## Section 28.2: Tuples are immutable

One of the main differences between `lists` and `tuples` in Python is that tuples are immutable, that is, one cannot add or modify items once the tuple is initialized. For example:

```
>>> t = (1, 4, 9)
>>> t[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Similarly, tuples don't have `.append` and `.extend` methods as `list` does. Using `+=` is possible, but it changes the binding of the variable, and not the tuple itself:

```
>>> t = (1, 2)
>>> q = t
>>> t += (3, 4)
>>> t
(1, 2, 3, 4)
>>> q
(1, 2)
```

Be careful when placing mutable objects, such as `lists`, inside tuples. This may lead to very confusing outcomes when changing them. For example:

```
>>> t = (1, 2, 3, [1, 2, 3])
(1, 2, 3, [1, 2, 3])
>>> t[3] += [4, 5]
```

Will **both** raise an error and change the contents of the list within the tuple:

```
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, 3, [1, 2, 3, 4, 5])
```

You can use the `+=` operator to "append" to a tuple - this works by creating a new tuple with the new element you "appended" and assign it to its current variable; the old tuple is not changed, but replaced!

This avoids converting to and from a list, but this is slow and is a bad practice, especially if you're going to append multiple times.

## Section 28.3: Packing and Unpacking Tuples

Tuples in Python are values separated by commas. Enclosing parentheses for inputting tuples are optional, so the two assignments

```
a = 1, 2, 3 # a is the tuple (1, 2, 3)
```

and

```
a = (1, 2, 3) # a is the tuple (1, 2, 3)
```

are equivalent. The assignment `a = 1, 2, 3` is also called *packing* because it packs values together in a tuple.

Note that a one-value tuple is also a tuple. To tell Python that a variable is a tuple and not a single value you can use

a trailing comma

```
a = 1 # a is the value 1
a = 1, # a is the tuple (1,)
```

A comma is needed also if you use parentheses

```
a = (1,) # a is the tuple (1,)
a = (1) # a is the value 1 and not a tuple
```

To unpack values from a tuple and do multiple assignments use

```
# unpacking AKA multiple assignment
x, y, z = (1, 2, 3)
# x == 1
# y == 2
# z == 3
```

The symbol `_` can be used as a disposable variable name if one only needs some elements of a tuple, acting as a placeholder:

```
a = 1, 2, 3, 4
_, x, y, _ = a
# x == 2
# y == 3
```

Single element tuples:

```
x, = 1, # x is the value 1
x = 1, # x is the tuple (1,)
```

In Python 3 a target variable with a `*` prefix can be used as a [catch-all](#) variable (see Unpacking Iterables):

Python 3.x Version  $\geq 3.0$

```
first, *more, last = (1, 2, 3, 4, 5)
# first == 1
# more == [2, 3, 4]
# last == 5
```

## Section 28.4: Built-in Tuple Functions

Tuples support the following build-in functions

### Comparison

If elements are of the same type, python performs the comparison and returns the result. If elements are different types, it checks whether they are numbers.

- If numbers, perform comparison.
- If either element is a number, then the other element is returned.
- Otherwise, types are sorted alphabetically .

If we reached the end of one of the lists, the longer list is "larger." If both list are same it returns 0.

```
tuple1 = ('a', 'b', 'c', 'd', 'e')
tuple2 = ('1', '2', '3')
```

```
tuple3 = ('a', 'b', 'c', 'd', 'e')
```

```
cmp(tuple1, tuple2)
```

```
Out: 1
```

```
cmp(tuple2, tuple1)
```

```
Out: -1
```

```
cmp(tuple1, tuple3)
```

```
Out: 0
```

## Tuple Length

The function `len` returns the total length of the tuple

```
len(tuple1)
```

```
Out: 5
```

## Max of a tuple

The function `max` returns item from the tuple with the max value

```
max(tuple1)
```

```
Out: 'e'
```

```
max(tuple2)
```

```
Out: '3'
```

## Min of a tuple

The function `min` returns the item from the tuple with the min value

```
min(tuple1)
```

```
Out: 'a'
```

```
min(tuple2)
```

```
Out: '1'
```

## Convert a list into tuple

The built-in function `tuple` converts a list into a tuple.

```
list = [1,2,3,4,5]
```

```
tuple(list)
```

```
Out: (1, 2, 3, 4, 5)
```

## Tuple concatenation

Use `+` to concatenate two tuples

```
tuple1 + tuple2
```

```
Out: ('a', 'b', 'c', 'd', 'e', '1', '2', '3')
```

# Section 28.5: Tuple Are Element-wise Hashable and Equatable

```
hash( (1, 2) ) # ok
```

```
hash( ([], {"hello"}) ) # not ok, since lists and sets are not hashable
```

Thus a tuple can be put inside a `set` or as a key in a `dict` only if each of its elements can.

```
{ (1, 2) } # ok
```

```
{ ([], {"hello"}) ) # not ok
```

## Section 28.6: Indexing Tuples

```
x = (1, 2, 3)
x[0] # 1
x[1] # 2
x[2] # 3
x[3] # IndexError: tuple index out of range
```

Indexing with negative numbers will start from the last element as -1:

```
x[-1] # 3
x[-2] # 2
x[-3] # 1
x[-4] # IndexError: tuple index out of range
```

Indexing a range of elements

```
print(x[:-1]) # (1, 2)
print(x[-1:]) # (3,)
print(x[1:3]) # (2, 3)
```

## Section 28.7: Reversing Elements

Reverse elements within a tuple

```
colors = "red", "green", "blue"
rev = colors[::-1]
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Or using reversed (reversed gives an iterable which is converted to a tuple):

```
rev = tuple(reversed(colors))
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```