Topic 5.3: Tuples

A tuple in Python is an ordered collection of objects, much like a list. The primary difference is that tuples are immutable – once created, their elements cannot be changed, added, or removed. It is important to understand that although the elements of a tuple are immutable, if the element is a mutable object, then the tuple can seem to change. Examples in this section will hopefully make this very clear.

Creating an Empty Tuple

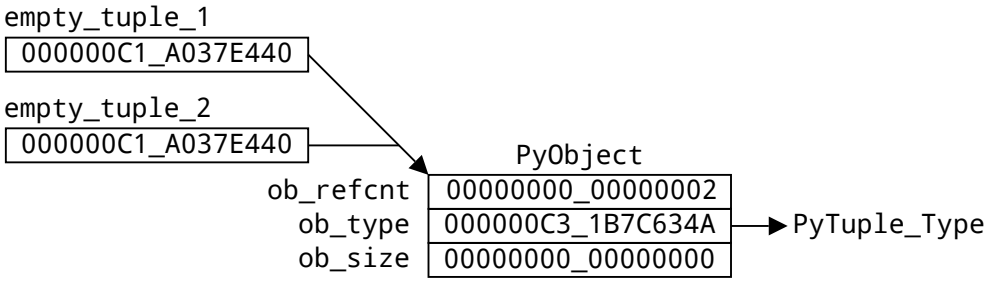
An empty tuple can be created by calling the `tuple` constructor or, with an empty pair of parentheses.

```
empty_tuple_1 = tuple()
empty_tuple_2 = ()
```

Unlike with an empty list, no elements can be added to an empty tuple after it has been created. This allows Python to use a singleton for the empty list, meaning there is a single empty tuple object, and any variable that is an empty list can refer to the same empty tuple object.

```
>>> empty_list_1 = []
>>> id(empty_list_1)
4337812032
>>> empty_list_2 = []
>>> id(empty_list_2)
4337774336
>>> empty_list_1 == empty_list_2
True
>>> empty_list_1 is empty_list_2
False
>>> empty_tuple_1 = ()
>>> id(empty_tuple_1)
4356951008
>>> empty_tuple_2 = ()
>>> id(empty_tuple_2)
4356951008
>>> empty_tuple_1 == empty_tuple_2
True
>>> empty_tuple_1 is empty_tuple_2
True
```

The memory structure of two variables, each referring to the empty tuple object, is shown diagrammatically below.



Tuple Literals

A tuple literal is usually written enclosed in parentheses, though the parentheses are optional when the context makes it clear (e.g., in a comma-separated list). Elements are separated by commas.

```
>>> car_brands = ("Volkswagen", "BYD", "Xiaomi")
>>> car_brands
('Volkswagen', 'BYD', 'Xiaomi')
>>> primes = (2, 3, 5, 7, 11)
>>> primes
(2, 3, 5, 7, 11)
>>> count = 1, 2, 3, 4, 5
>>> count
(1, 2, 3, 4, 5)
```

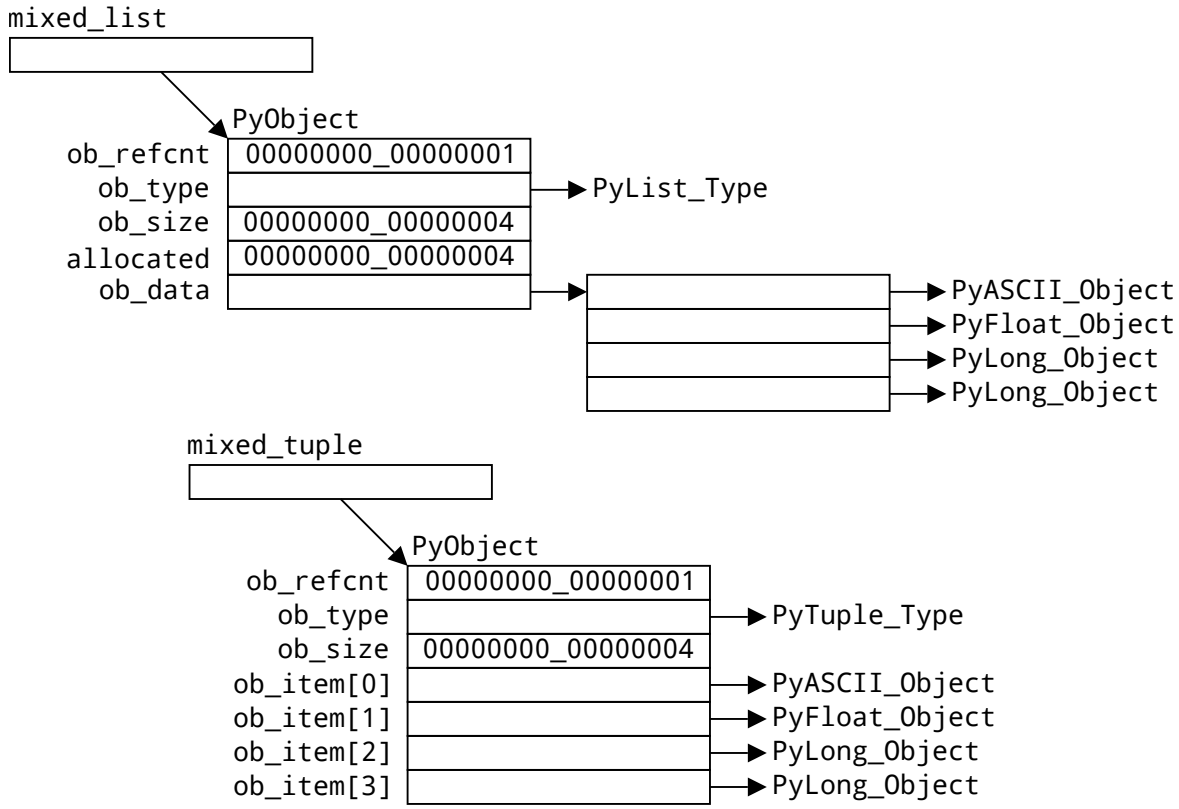
Single-element tuples require a comma to distinguish them from parentheses used for precedence.

```
>>> str = ("Hello")
>>> str
'Hello'
>>> type(str)
<class 'str'>
>>> tuple_str = ("Hello",)
>>> tuple_str
('Hello',)
>>> type(tuple_str)
<class 'tuple'>
>>> number_5 = ( 2 + 3 )
>>> number_5
5
>>> number_6 = ( 6 )
>>> number_6
6
>>> tuple_6 = ( 6, )
>>> tuple_6
(6,)
```

The `list` constructor can take any iterable object (a tuple, for example) and creates a new `list` object containing the elements of the parameter object. The `tuple` constructor does the same, except it creates a new `tuple` object containing the elements of the parameter object.

```
>>> numbers_tuple = ( 1, 2, 3, 4, 5 )
>>> numbers_list = list(numbers_tuple)
>>> numbers_tuple
(1, 2, 3, 4, 5)
>>> numbers_list
[1, 2, 3, 4, 5]
>>> mixed_list = [ "banana", 1.25, 100, True ]
>>> mixed_tuple = tuple(mixed_list)
>>> mixed_list
['banana', 1.25, 100, True]
>>> mixed_tuple
('banana', 1.25, 100, True)
```

Compare the memory structure diagrams for `mixed_list` and `mixed_tuple`:



Because the contents of a list may change, the array storing the contents of the list is allocated separately from the memory allocated for the `list` object. This allows the array storing list elements to be changed (re-allocating a different amount of memory). For tuples, the `tuple` object is allocated together with the tuple element storage, so changing the contents of the tuple requires creation of a new `tuple` object.

Length of a Tuple (`len`)

The same len function that gives the length of a list, also provides the length of a tuple.

```
>>> mixed_tuple = ( "banana", 1.25, 100, True )
>>> len(mixed_tuple)
4
```

Accessing Tuple Elements

Tuples use zero-based indexing and are accessed using the index enclosed in square brackets. Negative indexing starts with the last element of the tuple at index `-1`. This is exactly the same syntax as used when accessing lists.

```
>>> mixed_list = [ "banana", 1.25, 100, True ]
>>> mixed_list[0]
'banana'
>>> mixed_tuple = ( "banana", 1.25, 100, True )
>>> mixed_tuple[0]
'banana'
>>> mixed_tuple[-2]
100
>>> mixed_tuple[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

Mutating Elements

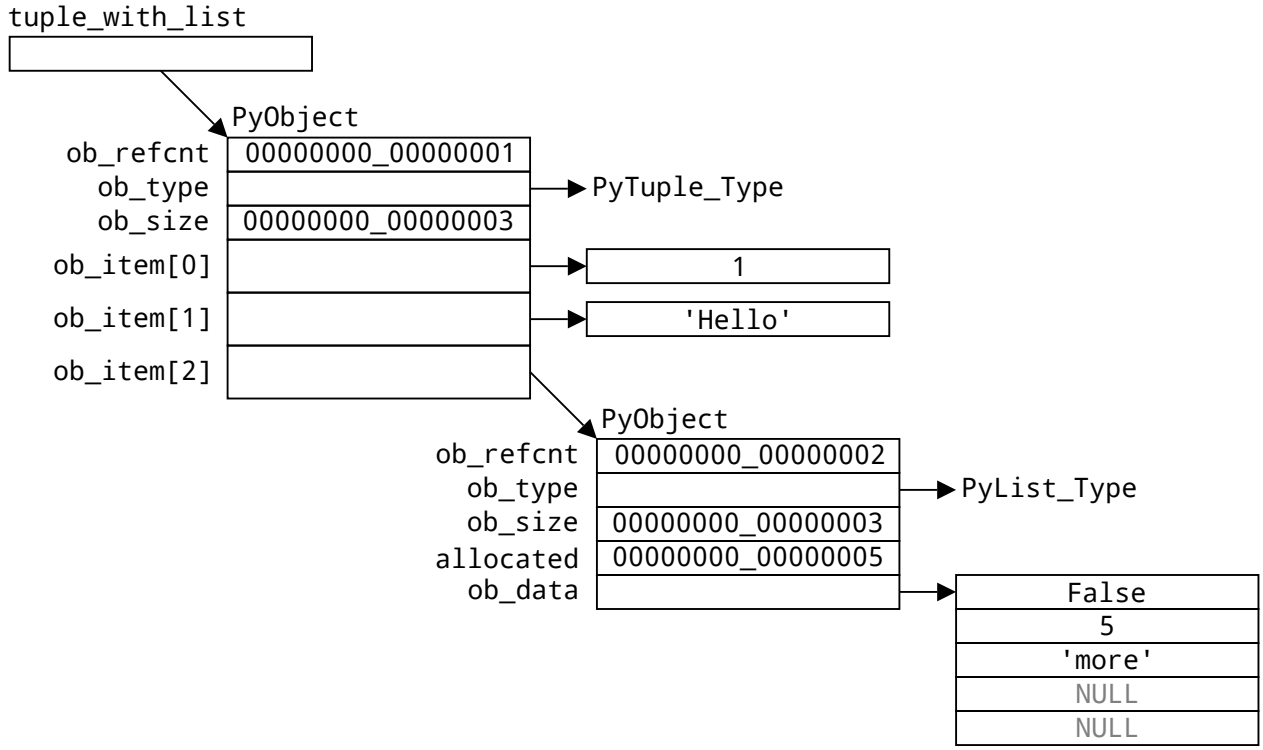
Tuples are immutable, so is not possible to change an element reference in a tuple. The methods and language constructs used to mutate a list are not available for a tuple (`append`, `insert`, `remove`, `pop`, `extend`, `reverse`, `clear`, `sort`, `del`). Also, there is no tuple `copy` method because tuples are immutable, so sharing references is always safe, unlike lists, which require `copy` to create a new, independent, modifiable `list` object.

```
 (>>> fruits = ( "apple", "banana", "orange" )
>>> fruits[2] = "kiwi"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> fruits.remove("banana")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'remove'
```

However – and this is an extremely important concept to understand – if an element of a tuple refers to a mutable object, such as a list, the values stored in the element object can still change.

```
>>> list_1 = [ False ]
>>> tuple_with_list = ( 1, "Hello", list_1 )
>>> tuple_with_list
(1, 'Hello', [False])
>>> list_1.append(5)
>>> tuple_with_list
(1, 'Hello', [False, 5])
>>> tuple_with_list[2].append("more")
>>> tuple_with_list
(1, 'Hello', [False, 5, 'more'])
```

Examine the memory structure diagram that is given below to understand how the list within the tuple can change and expand or contract without the contents of the tuple changing. Objects other than the relevant list and tuple have been abstracted to make the diagram more concise and compact.



Tuple Concatenation: (+)

The basic tuple concatenation (+) works the same as lists, because concatenation creates a new tuple.

```
>>> workdays = ("Mon", "Tue", "Wed", "Thu", "Fri")
>>> weekend = ("Sat", "Sun")
>>> days_of_week = workdays + weekend
>>> days_of_week
('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun')
>>> workdays
('Mon', 'Tue', 'Wed', 'Thu', 'Fri')
>>> weekend
('Sat', 'Sun')
```

The concatenation assignment operator (+=) works differently for tuples and lists. For a list, the operator modifies the existing list, the same way the append method does. For a tuple, it works the same way as the concatenation (+) operator, creating a new tuple and changing the variable to refer to that new tuple.

```
>>> number_list_1 = [ 1, 2, 3 ]
>>> number_list_2 = number_list_1
>>> number_list_1 += [4, 5]
>>> number_list_1
[1, 2, 3, 4, 5]
>>> number_list_2
[1, 2, 3, 4, 5]
>>> number_tuple_1 = ( 1, 2, 3 )
>>> number_tuple_2 = number_tuple_1
>>> number_tuple_1 += ( 4, 5 )
>>> number_tuple_1
(1, 2, 3, 4, 5)
>>> number_tuple_2
(1, 2, 3)
```

List assignment concatenation works with any iterable on the right-hand side, to a tuple can be concatenated to a list. For a tuple, assignment concatenation only works with two tuples. A list can be appended to a tuple by using the `tuple` constructor to create a tuple from a list.

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

>>> number_tuple = 1, 2, 3
>>> number_tuple
(1, 2, 3)

>>> number_tuple += (4, )
>>> number_tuple
(1, 2, 3, 4)

>>> number_tuple += [ 5 ]
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "list") to tuple

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

Repetition: The * Operator

The * operator creates a new tuple that contains repeats of the elements of the original tuple (i.e.: in the same way it works for lists).

```
>>> tuple_1 = ( "a", 1 )
>>> tuple_2 = tuple_1 * 4

>>> tuple_1
('a', 1)

>>> tuple_2
('a', 1, 'a', 1, 'a', 1, 'a', 1)
```

Membership: the `in` Operator

The in operator returns `True` if the element is in the tuple, otherwise it returns `False`.

```
>>> fruits = ("apple", "banana", "orange")
>>> print("banana" in fruits)
True
>>> print("grape" in fruits)
False
>>> print("grape" not in fruits)
True
```

Tuple Slicing

Slicing works exactly as it does for lists, producing a new tuple containing the selected elements. The syntax is `tuple[start:end:step]`, and all the rules (half-open intervals, negative indices, omitted boundaries) apply.

```
>>> numbers = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 )
>>> numbers[2:6]
[(2, 3, 4, 5)
>>> numbers[:4]
(0, 1, 2, 3)
>>> numbers[1:-1]
(1, 2, 3, 4, 5, 6, 7, 8)
>>> numbers[6:2:-1]
(6, 5, 4, 3)
>>> numbers[::-1]
(9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

Tuple Unpacking

A very useful feature of tuples is unpacking: you can assign the elements of a tuple to a corresponding number of variables in one statement. Actually, unpacking works with any iterable, including lists, but is especially common with tuples.

```
>>> coordinates = (3, 4)
>>> x, y = coordinates
>>> x
3
>>> y
4
```

Tuple unpacking can be used to swap two variables elegantly.

```
>>> a, b = 10, 20
>>> a
10
>>> b
20
>>> a, b = b, a
>>> a
20
>>> b
10
```

Recall how this would be accomplished in most other languages. Below shows how it takes three lines of code and declaring a temporary variable to accomplish the same task in Java.

```
int a = 10;
int b = 20;

int temp = a;
a = b;
b = temp;
```

When unpacking an iterable, every element of the iterable must be unpacked into a variable.

```
>>> coordinate_3d = ( 3, 5, 2 )
>>> x, y = coordinate_3d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

When unpacking an iterable, the remaining elements can be collected using `*`. Notice that no matter what type the iterable on the right-hand side is, the remaining elements are collected in a `list` object.

```
>>> number_list = [ 5, 4, 3, 2, 1 ]
>>> first, *rest = number_list
>>> first
5
>>> rest
[4, 3, 2, 1]
>>> fruit_tuple = [ "apple", "banana", "cherry", "date" ]
>>> first, *rest = fruit_tuple
>>> first
'apple'
>>> rest
['banana', 'cherry', 'date']
```

Other Tuple Methods (`count`, `index`)

Because tuples are immutable, they have only two methods, `count` and `index`, both of which are non-mutating and operate the same as they do for lists. For the `index` method, a `ValueError` is raised if the element is not found in the range of indexes of the tuple.

```
>>> letters_tuple = ("a", "b", "c") * 3
>>> letters_tuple
('a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c')
>>> letters_tuple.count("a")
3
>>> letters_tuple.count("d")
0
>>> letters_tuple.index('c');
2
>>> letters_tuple.index('c', 3);
5
>>> letters_tuple.index('c', 3, 8)
5
>>> letters_tuple.index('a',1,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: tuple.index(x): x not in tuple
```

Summary of Tuples in Python

A list literal is delimited by parentheses, `(` and `)`, which are optional in some contexts, and elements are separated by commas:

```
( 1, "two", True, 4.0 )
```

Operators:

`+`	The concatenation operator is used to combine two tuples into a new tuple.
`+=`	Creates a new tuple by concatenating the right-hand side tuple to the left-hand side tuple, and changing the left-hand variable to refer to this new tuple.
`a * n`	Creates a new `tuple` object that contains the elements of tuple `a` repeated `n` times.
`x in a`	Returns `True` if element `x` is found in tuple `a`.
`a[i]`	The subscript operator, `[` and `]`, is used to access: `print(a[2])` elements of the tuple at index `i`. Indexing starts at zero, and negative indexing starts at `-1` for the last element, `-2` for the second to last, etc.

`=`	Unpacking can assign the elements of an iterable such as a tuple to multiple variables:
	`coordinates = ( 3, 6, 1 )`
	`x, y, z = coordinates`
	`first, *rest = coordinates`

Constructor, Functions:

`tuple(i)`	The constructor to create a new `tuple` object out of any iterable object `i`. (`i` may be a list, set, string, dictionary, etc.)
`len(a)`	Returns the number of elements of the tuple `a`.
`min(a)`	Returns the smallest item in the tuple `a`.
`max(a)`	Returns the largest item in the tuple `a`.
`sum(a)`	Returns the sum of all elements in the tuple `a`.

Language Constructs:

`a[s:e]`	Creates a new `tuple` object that contains the elements from tuple `a` starting with the element at index `s` and ending with the element one prior to index `e`. (a half-open interval). Negative indexing is valid.
`a[s:e:c]`	The same as the previous operator, but indexing increments by the step count given in `c`. A negative step size, `c`, is valid, but then ensure the start index, `s`, is later in the tuple than the end index, `e`, as the Python interpreter will be decrementing the index starting with index `s`.

Methods:

`count(x)`	Counts the number of occurrences of item `x` in the list.
`index(x)`	Returns the index of the first occurrence of item `x` in the list.
`index(x, s)`	Returns the index of the first occurrence of item `x` in the list, starting the search from index `s` and ending at the end of the list.
`index(x, s, e)`	Returns the index of the first occurrence of item `x` in the list, starting the search from index `s`, and ending at the element one prior to index `e` (a half-open interval).

The immutability of tuples allows some implementation efficiencies when compared to lists, so use tuples when dealing with a sequence that will not grow, shrink, or change.