IIT Madras
BSc Degree

Home Lesson-6.1

Lesson-5.6

Tuples

Introduction

A tuple is an immutable sequence of values:

```
family = ('father', 'mother', 'child')
print(type(family))
print(isinstance(family, tuple))
```

Tuples share a close resemblance to lists. They can be indexed and sliced just like lists:

```
print(family[0])
print(family[:2])
```

The main point of difference between lists and tuples is that tuples cannot be updated in-place since they are immutable. So, the following operation will throw an error:

```
##### Alarm! Wrong code snippet! #####
numbers = ('one', 'two', 'four')
numbers[2] = 'three'
##### Alarm! Wrong code snippet! #####
```

The interpreter throws a `TypeError` with the following message: `TypeError: 'tuple' object does not support item assignment`. As a consequence, we cannot append or insert elements into a tuple. Likewise, elements in a tuple cannot be deleted. `count` and `index` are the only two methods which are defined for `tuple` and they carry the usual meaning:

```
numbers = (1, 2, 3, 1, 1)
print(numbers.count(1))
print(numbers.index(2))
```

We can iterate through a tuple using `for`:

```
for num in (1, 2, 3):
    print(num)
```

Since tuples are immutable, they are passed by value in functions similar to other immutable types such as strings and numbers. As for functions that operate on tuples, `sum`, `max`, `min` are useful ones.

More on Tuples

A few more points on tuples.

- A singleton tuple should be defined as follows:

```
i_am_single = (1, )
print(len(i_am_single))
print(isinstance(i_am_single, tuple))
```

Note the presence of a comma after the element. Let us see what happens if it is removed:

```
i_am_single = (1)
print(isinstance(i_am_single, int))
```

It is an integer!

- A list can be converted into a tuple and vice versa:

```
a_list = [1, 2, 3]
a_tuple = tuple(a_list)
b_tuple = (1, 2, 3)
b_list = list(b_tuple)
```

- A tuple can hold a non-homogeneous sequence of items:

```
a_tuple = (1, 'cool', True)
```

- Membership can be determined using the `in` keyword:

```
1 in (1, 2, 3)
'hello' not in ('some', 'random', 'sequence')
```

- Tuples can be nested:

```
a = ((1, 2, 3), (4, 5, 6))
print(a[0][2])
```

- A tuple can hold mutable objects.

```
a_tuple = ([0, 1, 2], [4, 5, 6])
a_tuple[0][0] = 100
```

The code given above runs without any errors. But we are trying to update the tuple in line-2. Aren't tuples immutable? Though `a_tuple` is immutable, the element inside it is mutable. In any case, we aren't trying to change the sequence of objects inside the tuple, i.e., `a_tuple[0]` continues to point to the same object. Let us verify this:

```
a_tuple = ([0, 1, 2], [4, 5, 6])
print(id(a_tuple[0]))
a_tuple[0][0] = 100
print(id(a_tuple[0]))
```

We see that the `id` of the element inside the tuple remains unchanged. Thus the identities of the sequence of objects that make up a tuple can never change, and the interpreter will never allow that to change. If the objects inside the sequence are mutable — such as lists — then the values that they hold might change, but they continue to retain their identities.

Lists and Tuples

We have seen the close kinship between lists and tuples. Here is a brief summary that highlights the points of agreement and disagreement:

List	Tuple
Mutable	Immutable
`L = [1, 2, 3]`	`T = (1, 2, 3)`
Supports indexing and slicing	Supports indexing and slicing
Supports item assignment	Doesn't support item assignment
Supported methods: `count, index, append, insert, remove, pop` and others	Supported methods: `count, index`
To get a list: `list(obj)`	To get a tuple: `tuple(obj)`

The partnership between lists and tuples is quite interesting and can be explored further with another example.

> Populate a list that contains all ordered pairs of positive integers whose product is 100. Note that order matters: (2, 50) and (50, 2) are two different pairs.

Solution

```
pairs = [ ]
for a in range(1, 101):
    for b in range(1, 101):
        if a * b == 100:
            pairs.append((a, b))
print(pairs)
```

`pairs` is a list of tuples. We could have stored each pair as a list. But a tuple is the better choice here since the two elements in the pair have a well defined relationship and we don't want to accidentally modify them.

Packing and Unpacking

At first sight, tuples might seem redundant members in the Python family, but they do occupy a significant place. For that, we have to look at tuples in more detail. Consider the following code:

```
T = 1, 2, 3
print(T)
print(isinstance(T, tuple))
```

At first sight, line-1 seems to be an error. We have seen multiple assignment on the same line, perhaps we are two variables short on the LHS? But on execution, we see that there is no error. `T` is in fact the tuple `(1, 2, 3)`. This is called **tuple packing**. The values `1`, `2` and `3` are packed into a tuple. The reverse operation is called **sequence unpacking**:

```
x, y, z = T
print(x, y, z)
```

Here, the tuple `T` is unpacked into the corresponding variables `x`, `y` and `z`. This is the principle behind multiple assignment. From the Python documentation, we have [refer]:

> Multiple assignment is a combination of tuple packing and sequence unpacking.

```
x, y, z = 1, 2, 3
```

In the line given above, the RHS is first packed into a tuple and the sequence is then unpacked into the variables `x`, `y` and `z`. But why does the unpacking operation have the qualifier sequence before it? This is because any sequence can be unpacked:

```
l1, l2, l3, l4 = 'good'          # string
num1, num2, num3 = [1, 2, 3]     # list
b1, b2 = (True, False)           # tuple
x, y, z = range(3)               # range
```

That's fun! The same operations are invoked when multiple values are returned from functions:

```
1 def max_min(a, b):
2     if a > b:
3         return a, b
4     return b, a
5
6 x = max_min(1, 2)
7 print(x)
8 print(isinstance(x, tuple))
```

We see that `x` is a tuple. In the return statements at lines 3 and 4, the multiple values are packed into tuples. So, the function is essentially returning a tuple.