Chapter_12_Tuples

March 6, 2024

One note: there is no consensus on how to pronounce "tuple". Some people say "tuhple", which rhymes with "supple". But in the context of programming, most people say "too-ple", which rhymes with "quadruple".

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable.

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

[20]:
```
t = 'a', 'b', 'c', 'd', 'e'
```

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

[21]:
```
t = ('a', 'b', 'c', 'd', 'e')
```

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

[22]:
```
t1 = 'a',
>>> type(t1)
```

[22]:
```
tuple
```

[23]:
```
t2 = ('a')
```

[24]:
```
type(t2)
```

[24]:
```
str
```

Another way to create a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

[25]:
```
t = tuple()
t
```

[25]:
```
()
```

[26]:
```
t = tuple('lupins')
t
```

[26]:
```
('l', 'u', 'p', 'i', 'n', 's')
```

Most list operators also work on tuples. The bracket operator indexes an element:

[27]:
```
t = ('a', 'b', 'c', 'd', 'e')
t[0]
```

[27]:
```
'a'
```

[28]:
```
t[1:3]
```

[28]:
```
('b', 'c')
```

[29]:
```
t[0] = 'A'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-29-7e674cdf20e6> in <module>
----> 1 t[0] = 'A'

TypeError: 'tuple' object does not support item assignment
```

Because tuples are immutable, you can't modify the elements. But you can replace one tuple with another:

[30]:
```
>>> t = ('A',) + t[1:]
>>> t
```

[30]:
```
('A', 'b', 'c', 'd', 'e')
```

The relational operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

[9]:
```
(0, 1, 2) < (0, 3, 4)
```

[9]:
```
True
```

[7]:
```
(0, 1, 2000000) < (0, 3, 4)
```

[7]:
```
True
```

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap a and b:

[34]:
```
# >>> temp = a
# >>> a = b
# >>> b = temp
```

This solution is cumbersome; tuple assignment is more elegant

[36]:
```
# a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right have to be the same:

[37]:
```
>>> a, b = 1, 2, 3
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-37-8904fd2ea925> in <module>
----> 1 a, b = 1, 2, 3

ValueError: too many values to unpack (expected 2)
```

[39]:
```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
print(uname,domain)
```

```
monty python.org
```

Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute x//y and then x%y. It is better to compute them both at the same time.

The built-in function divmod takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as a tuple:

[40]:
```
t = divmod(7, 3)
t
```

[40]:
(2, 1)

Or use tuple assignment to store the elements separately:

[41]:
```
quot, rem = divmod(7, 3)
```

[42]:
```
quot
```

[42]:
2

[43]:
```
rem
```

[43]:
1

[44]:
```
#Here is an example of a function that returns a tuple:
def min_max(t):
    return min(t), max(t)
```

Functions can take a variable number of arguments. A parameter name that begins with "*" gathers arguments into a tuple. For example, printall takes any number of arguments and prints them:

[45]:
```
def printall(*args):
    print(args)
printall(1, 2.0, '3')
```

```
(1, 2.0, '3')
```

The complement of gather is scatter. If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the * operator. For example, divmod takes exactly two arguments; it doesn't work with a tuple:

[46]:
```
t = (7, 3)
>>> divmod(t)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-46-b9b469f7c0ca> in <module>
      1 t = (7, 3)
----> 2 divmod(t)

TypeError: divmod expected 2 arguments, got 1
```

But if you scatter the tuple, it works:

[47]:
```
divmod(*t)
```

[47]:
(2, 1)

Many of the built-in functions use variable-length argument tuples. For example, max and min can take any number of arguments:

[48]:
```
max(1, 2, 3)
```

[48]:
3

But sum does not.

[49]:
```
sum(1, 2, 3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-49-c44a16fb57bf> in <module>
----> 1 sum(1, 2, 3)

TypeError: sum expected at most 2 arguments, got 3
```

zip is a built-in function that takes two or more sequences and interleaves them. The name of the function refers to a zipper, which interleaves two rows of teeth.

This example zips a string and a list:

[50]:
```
s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
```

[50]:
```
<zip at 0x26d7b4dd208>
```

The result is a zip object that knows how to iterate through the pairs. The most common use of zip is in a for loop:

[52]:
```
for pair in zip(s, t):
    print(pair)
```

```
('a', 0)
('b', 1)
('c', 2)
```

A zip object is a kind of iterator, which is any object that iterates through a sequence. Iterators are similar to lists in some ways, but unlike lists, you can't use an index to select an element from an iterator.

If you want to use list operators and methods, you can use a zip object to make a list:

[53]:
```
>>> list(zip(s, t))
```

[53]:
```
[('a', 0), ('b', 1), ('c', 2)]
```

If the sequences are not the same length, the result has the length of the shorter one.

[54]:
```
>>> list(zip('Anne', 'Elk'))
```

[54]:
```
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

You can use tuple assignment in a for loop to traverse a list of tuples:

[55]:
```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

```
0 a
1 b
2 c
```

If you combine zip, for and tuple assignment, you get a useful idiom for traversing two (or more) sequences at the same time. For example, has_match takes two sequences, t1 and t2, and returns True if there is an index i such that t1[i] == t2[i]:

[57]:
```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
        return False
```

If you need to traverse the elements of a sequence and their indices, you can use the built-in function enumerate:

[58]:
```
for index, element in enumerate('abc'):
    print(index, element)
```

```
0 a
1 b
2 c
```

The result from enumerate is an enumerate object, which iterates a sequence of pairs; each pair contains an index (starting from 0) and an element from the given sequence.

Dictionaries have a method called items that returns a sequence of tuples, where each tuple is a key-value pair.

[59]:
```
d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
```

[59]:
```
dict_items([('a', 0), ('b', 1), ('c', 2)])
```

The result is a dict_items object, which is an iterator that iterates the key-value pairs. You can use it in a for loop like this:

[60]:
```
for key, value in d.items():
    print(key, value)
```

```
a 0
b 1
c 2
```

Going in the other direction, you can use a list of tuples to initialize a new dictionary:

[61]:
```
t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
```

[61]:
```
{'a': 0, 'c': 2, 'b': 1}
```

Combining dict with zip yields a concise way to create a dictionary:

[62]:
```
d = dict(zip('abc', range(3)))
>>> d
```

[62]:
```
{'a': 0, 'b': 1, 'c': 2}
```

The dictionary method update also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary.

It is common to use tuples as keys in dictionaries (primarily because you can't use lists). For example, a telephone directory might map from last-name, first-name pairs to telephone numbers. Assuming that we have defined last, first and number, we could write:

directory[last, first] = number

The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

[65]:
```
# for last, first in directory:
#     print(first, last, directory[last,first])
```

This loop traverses the keys in directory, which are tuples. It assigns the elements of each tuple to last and first, then prints the name and corresponding telephone number.

I have focused on lists of tuples, but almost all of the examples in this chapter also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.

[1]:
```
# ! pip install structshape
```

```
ERROR: Could not find a version that satisfies the requirement structshape (from
versions: none)
ERROR: No matching distribution found for structshape
```

[2]:
```
# from structshape import structshape
# >>> t = [1, 2, 3]
# >>> structshape(t)
```

[3]:
```
import string
>>> string.punctuation
```

[3]:
```
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be deterministic. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be difficult, but there are ways to make it at least seem nondeterministic. One of them is to use algorithms that generate pseudorandom numbers.

Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The random module provides functions that generate pseudorandom numbers (which I will simply call "random" from here on). The function random returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call random, you get the next number in a long series. To see a sample, run this loop:

[4]:
```
import random
for i in range(10):
    x = random.random()
    print(x)
```

```
0.2624002780273291
0.0705053258756193
0.4506181371055107
0.9767748838582239
0.7223005719019827
0.2680359920740464
0.3720829711670174
0.13081381043274765
0.9930932076698827
0.2255998182086113
```

The function randint takes parameters low and high and returns an integer between low and high (including both).

[5]:
```
random.randint(5, 10)
```

[5]: 6

[6]:
```
random.randint(5, 10)
```

[6]: 10

To choose an element from a sequence at random, you can use choice:

[7]:
```
>>> t = [1, 2, 3]
>>> random.choice(t)
```

[7]: 2

[8]:
```
>>> random.choice(t)
```

[8]: 1

The random module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

[ ]:
```

```