Think Python 2e, Chapter 12 Notes

Tuples

October 7, 2022

Tuples are like immutable lists

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

```
>>> t = ’a’, ’b’, ’c’, ’d’, ’e’
```

It is not necessary, but common to enclose tuples in parentheses:

```
>>> t = (’a’, ’b’, ’c’, ’d’, ’e’)
```

To create a tuple with a single element, you need a final comma:

```
>>> t1 = ’a’,
>>> type(t1)
<class ’tuple’>
```

A value in parentheses is not a tuple:

```
>>> t2 = (’a’)
>>> type(t2)
<class ’str’>
```

Creating tuples with `tuple`

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

```
>>> t = tuple()
>>> t
()
```

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

```
>>> t = tuple(’lupins’)
>>> t
(’l’, ’u’, ’p’, ’i’, ’n’, ’s’)
```

List and string operators also work on tuples

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
>>> ('a', 'b', 'c') + ('d', 'e')
('a', 'b', 'c', 'd', 'e')
```

And the slice operator selects a range of elements.

```
>>> t[1:3]
('b', 'c')
```

Tuples can contain any type:

```
('a', 99, [1, 2, 3], ('a', 'b', 'c'), 'hello')
```

Like strings, tuples are immutable

```
>>> t[0] = ’A’
TypeError: object doesn’t support item assignment
```

But you can replace one tuple with another:

```
>>> t = (’A’,) + t[1:]
>>> t
(’A’, ’b’, ’c’, ’d’, ’e’)
```

This statement makes a new tuple and then makes t refer to it.

Relational operators work on tuples

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> (1, 2, 3) < (1, 2, 3, 4)
True
```

Tuple assignment

Sometimes it is necessary to swap the values of variables. Normally this is done like this:

```
>>> temp = a
>>> a = b
>>> b = temp
```

Tuple assignment makes this more elegant:

```
>>> a, b = b, a
```

Tuple assignment

Sometimes it is necessary to swap the values of variables. Normally this is done like this:

```
>>> temp = a
>>> a = b
>>> b = temp
```

Tuple assignment makes this more elegant:

```
>>> a, b = b, a
```

The right hand side can be any sequence (string, list, or tuple):

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
>>> uname
'monty'
>>> domain
'python.org'
```

Fast and easy Fibonacci

```
def fib(n):
 if n < 2:
   return n
 else:
   return fib(n-1)+fib(n-2)
```

```
def fibfast(n):
  a, b = 0, 1
  while n > 0:
    a,b,n = b, a+b, n-1
  return a
```

Timings in seconds:

n	fib(n)	fibfast(n)
30	0.35	0.0
31	0.51	0.0
32	0.82	0.0
33	1.25	0.0
34	2.15	0.0
35	3.54	0.0
36	6.06	0.0
37	10.37	0.0
38	16.70	0.0
39	26.14	0.0
40	41.76	0.0

Recursive fibfast

```python
def fibfast(n):
    a, b = 0, 1
    while n > 0:
        a,b,n = b, a+b, n-1
    return a
```

```python
def recfibfast(n):
    return fibhelper(0, 1, n)
def fibhelper(a, b, n):
    if n < 1:
        return a
    else:
        return fibhelper(b, a+b, n-1)
```

Timings are the same into the hundreds.

Tuples as return values

Instead of computing both `7 // 3` and `7 % 3`, we can compute both at the same time:

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

Or use tuple assignment to store the elements separately:

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

Here is an example of a user-defined function that returns a tuple:

```
def min_max(t):
    return min(t), max(t)
```

gather parameter

```
def printall(*args):
    print(args)
```

```
>>> printall(1, 2.0, ’3’)
(1, 2.0, ’3’)
```

The gather parameter can have any name you like, but args is conventional.

scatter argument

```
>>> divmod(7, 3)
(2, 1)
>>> t = (7, 3)
>>> divmod(t[0], t[1])
(2, 1)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
>>> divmod(*t)
(2, 1)
```

Builtins differ

```
>>> max(1, 2, 3)
3
>>> max([1, 2, 3])
3
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
>>> sum([1, 2, 3])
6
```

zip

```
>>> s = ’abc’
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

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:

```
>>> for pair in zip(s, t):
...     print(pair)
...
(’a’, 0)
(’b’, 1)
(’c’, 2)
```

Iterators

A zip object is a kind of **iterator**.

Range is another iterator:

```
>>> type(range(10))
<class ’range’>
```

You can get the list of values with list:

```
>>> list(zip(’abc’, [1, 2, 3]))
[(’a’, 1), (’b’, 2), (’c’, 3)]
>>> list(range(4))
[0, 1, 2, 3]
```

Zipping

```
>>> s = ’hello’
>>> for a,b in zip(s, range(len(s))):
        print(b,a)
0 h
1 e
2 l
3 l
4 o
```

enumerate

```
for index, element in enumerate(’abc’):
    print(index, element)
0 a
1 b
2 c
```

Traversing two (or more) sequences at the same time

```
>>> a = 'hello'
>>> b = [11,22,33,44,55]
>>> for index in range(len(a)):
          print(a[index], b[index])
h 11
e 22
l 33
l 44
o 55
```

```
>>> for x,y in zip(a,b):
        print(x,y)
h 11
e 22
l 33
l 44
o 55
```

```
for i,x in enumerate(a):
    print(x, b[i])
h 11
e 22
l 33
l 44
o 55
```

items

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

The result is a `dict_items` object, which is an iterator that iterates the key-value pairs.

```
>>> for key, value in d.items():
...     print(key, value)
...
c 2
a 0
b 1
```

Initializing a dict with tuples

```
>>> t = [(’a’, 0), (’c’, 2), (’b’, 1)]
>>> d = dict(t)
>>> d
{’a’: 0, ’c’: 2, ’b’: 1}
```

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

```
>>> d = dict(zip(’abc’, range(3)))
>>> d
{’a’: 0, ’c’: 2, ’b’: 1}
```

update

```
>>> d = dict(zip(’abc’,(1,2,3)))
>>> d
{’a’: 1, ’b’: 2, ’c’: 3}
>>> d.update(zip(’xyz’, (24,25,26)))
>>> d
{’a’: 1, ’b’: 2, ’c’: 3, ’x’: 24, ’y’: 25, ’z’: 26}
```

Tuples as keys

```
>>> last = 'Matthews'
>>> first = 'Geoffrey'
>>> number = '555-1234'
```

We can use a tuple as a key:

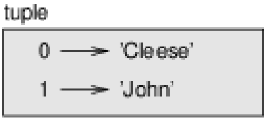
```
>>> directory[last, first] = number
```

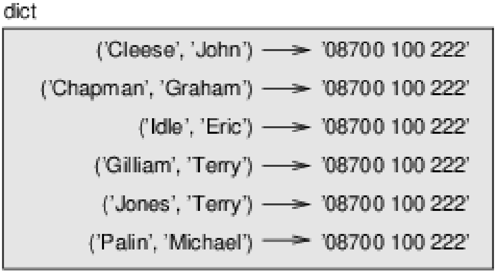
We can use tuple assignment from the keys:

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

State diagrams for tuples

```python
('Cleese', 'John')
```




dict
('Cleese', 'John') → '08700 100 222'
('Chapman', 'Graham') → '08700 100 222'
('Idle', 'Eric') → '08700 100 222'
('Gilliam', 'Terry') → '08700 100 222'
('Jones', 'Terry') → '08700 100 222'
('Palin', 'Michael') → '08700 100 222'

Sequences: random advice

- Lists, tuples, and strings are often interchangeable.
- Strings are the most constrained, lists the least.
- Lists are the only mutable one.
- Sometimes it's syntactically simpler to use a tuple:

  ```
  return a, b, c
  ```
- Dictionary keys can't be lists.
- Passing a tuple to a function instead of a list reduces aliasing.
- Tuples can't use `sort` and `reverse`, but they can use `sorted` and `reversed`

Textbook provides a `structshape` module

```
>>> from structshape import structshape
>>> t = [1, 2, 3]
>>> structshape(t)
’list of 3 int’
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
’list of 3 list of 2 int’
>>> t3 = [1, 2, 3, 4.0, ’5’, ’6’, [7], [8], 9]
>>> structshape(t3)
’list of (3 int, float, 2 str, 2 list of int, int)’
>>> s = ’abc’
>>> lt = list(zip(t, s))
>>> structshape(lt)
’list of 3 tuple of (int, str)’
>>> d = dict(lt)
>>> structshape(d)
’dict of 3 int->str’
```

Vocabulary

tuple: An immutable sequence of elements.

tuple assignment: An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.

gather: An operation that collects multiple arguments into a tuple.

scatter: An operation that makes a sequence behave like multiple arguments.

Vocabulary

zip object: The result of calling a built-in function zip; an object that iterates through a sequence of tuples.

iterator: An object that can iterate through a sequence, but which does not provide list operators and methods.

data structure: A collection of related values, often organized in lists, dictionaries, tuples, etc.

shape error: An error caused because a value has the wrong shape; that is, the wrong type or size.