python 元组 tuple

元组在所有方面都与列表相同，除了以下属性:

> 元组的定义是将元素括在圆括号`()`中，而不是方括号`[]`。
> 元组是不可变的。

元组的创建

```
t = ('foo', 'bar', 'baz', 'qux', 'quux', 'corge')

print(type(t))
print(t)
print(t[0])
print(t[-1])
print(t[1::2])
```

```
<class 'tuple'>
('foo', 'bar', 'baz', 'qux', 'quux', 'corge')
foo
corge
('bar', 'qux', 'corge')
```

```
print(t[::-1])
```

```
('corge', 'quux', 'qux', 'baz', 'bar', 'foo')
```

元组不可变(元素不可修改)

```
t = ('foo', 'bar', 'baz', 'qux', 'quux', 'corge')
t[2] = 'Bark!'
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-4-793de0908259> in <module>()
      1 t = ('foo', 'bar', 'baz', 'qux', 'quux', 'corge')
----> 2 t[2] = 'Bark!'
```

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

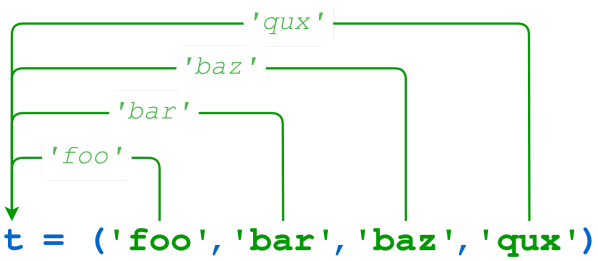
为什么使用元组而不是列表?

```
当操作一个元组时，程序执行速度要比处理等价列表时快。(当列表或元组很小时，这一点可能不会很明显。)
有时不希望修改数据。如果集合中的值在程序生命周期内保持不变，那么使用元组而不是列表可以防止意外修改。
不久将遇到另一种Python数据类型dictionary，它要求其组件之一具有不可变类型的值。
```

元组分配、打包和拆包

```
t = ('foo', 'bar', 'baz', 'qux')
print(t)
```

```
('foo', 'bar', 'baz', 'qux')
```

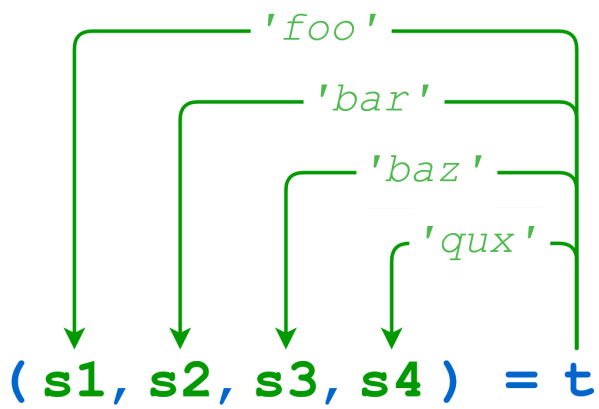


如果那个“打包”的对象随后被分配给一个新的元组，各个项目被“解压缩”到元组中的对象中:

```
(s1, s2, s3, s4) = t

print(s1)
print(s2)
print(s3)
print(s4)
```

```
foo
bar
baz
qux
```



当解包时，左边的变量数量必须与元组中的值数量匹配:

```
(s1, s2, s3) = t
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-10-42d3e84cbe00> in <module>()
----> 1 (s1, s2, s3) = t
```

```
ValueError: too many values to unpack (expected 3)
```

```
(s1, s2, s3, s4, s5) = t
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-11-426d2f4abb58> in <module>()
----> 1 (s1, s2, s3, s4, s5) = t
```

```
ValueError: not enough values to unpack (expected 5, got 4)
```

包装和拆包可以合并成一个语句，构成一个复合赋值:

```
(s1, s2, s3, s4) = ('foo', 'bar', 'baz', 'qux')

print(s1)
print(s2)
print(s3)
print(s4)
```

```
foo
bar
baz
qux
```

同样，赋值左侧元组中的元素数量必须等于右侧的元素数量:

```
(s1, s2, s3, s4, s5) = ('foo', 'bar', 'baz', 'qux')
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-13-18c26382b845> in <module>()
----> 1 (s1, s2, s3, s4, s5) = ('foo', 'bar', 'baz', 'qux')
```

```
ValueError: not enough values to unpack (expected 5, got 4)
```

在这样的赋值和少数其他情况下，Python允许省略圆括号，圆括号通常用于表示一个元组:

```
t = 1, 2, 3
print(t)
```

```
(1, 2, 3)
```

```
x1, x2, x3 = t
print(x1, x2, x3)
```

```
1 2 3
```

```
x1, x2, x3 = 4, 5, 6
print(x1, x2, x3)
```

```
4 5 6
```

单一元素的元组在创建过程中注意添加区分逗号 `,`

```
t = 2
print(type(t))
print(t)
```

```
<class 'int'>
2
```

```
t = (2)
print(type(t))
print(t)
```

```
<class 'int'>
2
```

```
t = (2,)
print(type(t))
print(t)
```

```
<class 'tuple'>
(2,)
```

```
t = 2,
print(type(t))
print(t)
```

```
<class 'tuple'>
(2,)
```

交换赋值

```
a = 'foo'
b = 'bar'
print(a, b)
```

```
foo bar
```

```
a, b = b, a

print(a, b)
```

```
bar foo
```

```

```

Python Tuple

https://www.techbeamers.com/python-tuple/

```
# create an empty tuple
py_tuple = ()
print("A blank tuple:", py_tuple)

# create a tuple without using round brackets
py_tuple = 33, 55, 77
print("A tuple set without parenthesis:", py_tuple, "type:", type(py_tuple))

# create a tuple of numbers
py_tuple = (33, 55, 77)
print("A tuple of numbers:", py_tuple)

# create a tuple of mixed numbers
# such as integer, float, imaginary
py_tuple = (33, 3.3, 3+3j)
print("A tuple of mixed numbers:", py_tuple)

# create a tuple of mixed data types
# such as numbers, strings, lists
py_tuple = (33, "33", [3, 3])
print("A tuple of mixed data types:", py_tuple)

# create a tuple of tuples
# i.e. a nested tuple
py_tuple = (('x', 'y', 'z'), ('X', 'Y', 'Z'))
print("A tuple of tuples:", py_tuple)
```

```
A blank tuple: ()
A tuple set without parenthesis: (33, 55, 77) type: <class 'tuple'>
A tuple of numbers: (33, 55, 77)
A tuple of mixed numbers: (33, 3.3, (3+3j))
A tuple of mixed data types: (33, '33', [3, 3])
A tuple of tuples: (('x', 'y', 'z'), ('X', 'Y', 'Z'))
```

```

```

```
# creating a tuple from a set
py_tuple = tuple({33, 55 , 77})
type(py_tuple)
```

```
tuple
```

```
py_tuple
```

```
(33, 77, 55)
```

```
# creating a tuple from a list
py_tuple = tuple([33, 55 , 77])
type(py_tuple)
```

```
tuple
```

```
py_tuple
```

```
(33, 55, 77)
```

```
# A single element surrounded by parenthesis will create a string instead of a tuple
py_tuple = ('single')
type(py_tuple)
```

```
str
```

```
# You need to place a comma after the first element to create a tuple of size "one"
py_tuple = ('single',)
type(py_tuple)
```

```
tuple
```

```
# You can use a list of one element and convert it to a tuple
py_tuple = tuple(['single'])
type(py_tuple)
```

```
tuple
```


```
# You can use a list of one element and convert it to a tuple
py_tuple = tuple(['single'])
type(py_tuple)
```


```
tuple
```

```
vowel_tuple = ('a','e','i','o','u')
print("The tuple:", vowel_tuple, "Length:", len(vowel_tuple))

# Indexing the first element
print("OP(vowel_tuple[0]):", vowel_tuple[0])

# Indexing the last element
print("OP(vowel_tuple[length-1]):", vowel_tuple[len(vowel_tuple) - 1])

# Indexing a non-existent member
# will raise the IndexError
try:
    print(vowel_tuple[len(vowel_tuple)+1])
except Exception as ex:
    print("OP(vowel_tuple[length+1]) Error:", ex)

# Indexing with a non-integer index
# will raise the TypeError
try:
    print(vowel_tuple[0.0])
except Exception as ex:
    print("OP(vowel_tuple[0.0]) Error:", ex)

# Indexing in a tuple of tuples
t_o_t = (('jan', 'feb', 'mar'), ('sun', 'mon', 'wed'))

# Accessing elements from the first sub tuple
print("OP(t_o_t[0][2]):", t_o_t[0][2])

# Accessing elements from the second sub tuple
print("OP(t_o_t[1][2]):", t_o_t[1][2])
```

```
The tuple: ('a', 'e', 'i', 'o', 'u') Length: 5
OP(vowel_tuple[0]): a
OP(vowel_tuple[length-1]): u
OP(vowel_tuple[length+1]) Error: tuple index out of range
OP(vowel_tuple[0.0]) Error: tuple indices must be integers or slices, not float
OP(t_o_t[0][2]): mar
OP(t_o_t[1][2]): wed
```

```
vowels = ('a','e','i','o','u')
vowels
```

```
('a', 'e', 'i', 'o', 'u')
```

```
first_tuple = ('p', 'y', 't')
second_tuple = ('h', 'o', 'n')
full_tuple = first_tuple + second_tuple
full_tuple
```

```
('p', 'y', 't', 'h', 'o', 'n')
```

```
init_tuple = ("fork", )
fork_tuple = init_tuple * 5
fork_tuple
```

```
('fork', 'fork', 'fork', 'fork', 'fork')
```

元组与列表的区别

Python: What is the Difference between a List and a Tuple?

Python List vs. Tuples

```
list_num = [1,2,3,4]
tup_num = (1,2,3,4)

print(list_num)
print(tup_num)
```

```
[1, 2, 3, 4]
(1, 2, 3, 4)
```

列表和元组之间的主要区别在于列表是可变的，而元组是不可变的。

```
a = ["apples", "bananas", "oranges"]
print(type(a))
print(a[0])
print(a)
```

```
<class 'list'>
apples
['apples', 'bananas', 'oranges']
```

```
a[0] = "berries"
print(a)
```

```
['berries', 'bananas', 'oranges']
```

元组不可修改

```
a = ("apples", "bananas", "oranges")
a[0] = "berries"
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-25-4a9d04861544> in <module>()
      1 a = ("apples", "bananas", "oranges")
----> 2 a[0] = "berries"
```

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

```
a = ("apples", "bananas", "oranges")
a = ("berries", "bananas", "oranges")
```

变量和对象之间的区别

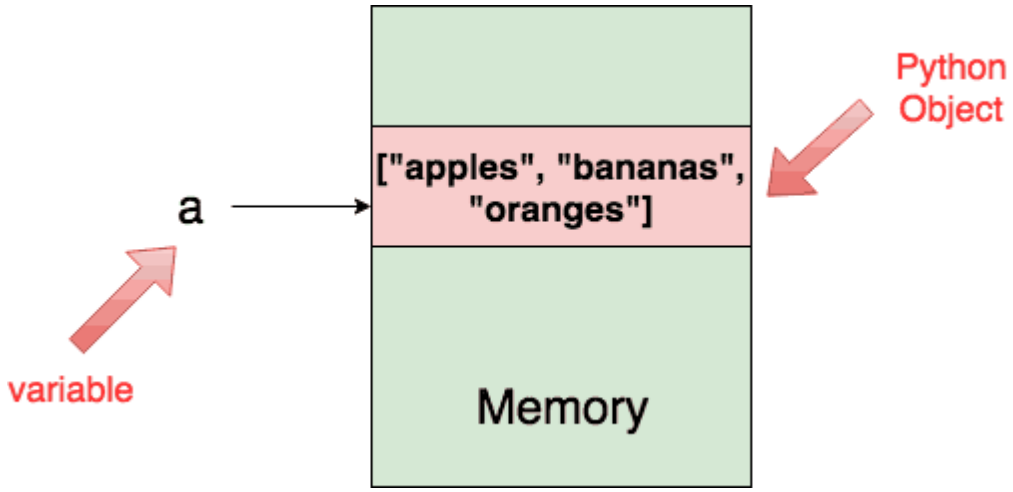
请记住，变量只是对内存中实际python对象的引用。

变量本身不是对象。

例如，让我们试着可视化当您将一个列表分配给一个变量a时会发生什么。

```
a = ["apples", "bananas", "oranges"]
```

当这样做时，将在内存中创建一个类型为list的python对象，变量a通过在内存中保存它的位置来引用这个对象。



实际上，可以通过使用id()函数检查a来检索list对象在内存中的位置。

```
a = ["apples", "bananas", "oranges"]
print(id(a))
```

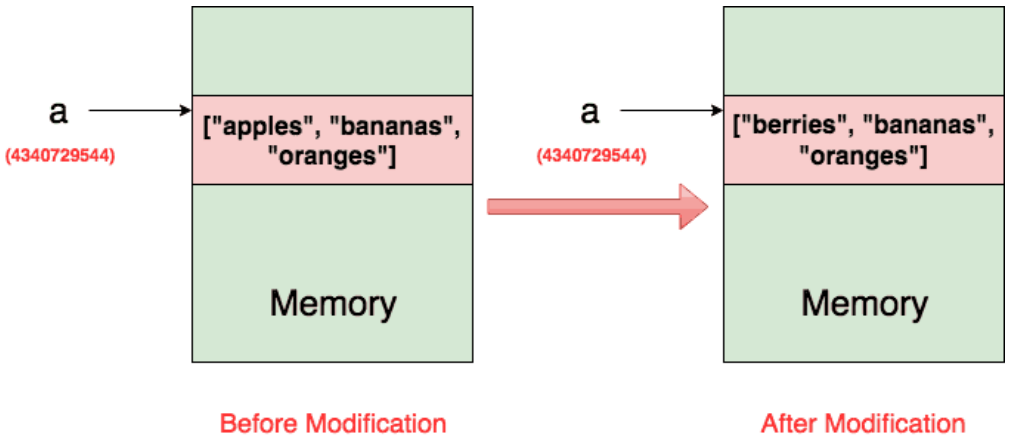
```
99700552
```

现在，如果修改列表的第一个索引，并再次检查id()，你将得到相同的值，因为a仍然引用相同的对象。

```
a[0] = "berries"
print(id(a))
```

```
99700552
```

下图准确地显示了修改后的情况。



现在，让我们看看如果对元组执行相同的操作会发生什么。

```
a = ("apples", "bananas", "oranges")
print(id(a))
```

```
130380640
```

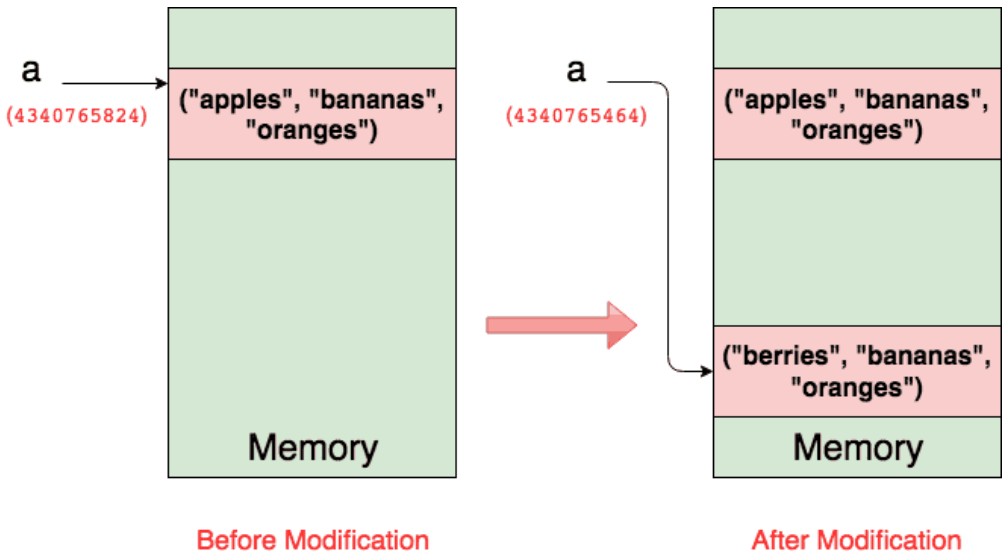
```
a = ("berries", "bananas", "oranges")
print(id(a))
```

```
130102184
```

如你所见，这两个地址是不同的。

这意味着在第二次赋值之后，a引用了一个全新的对象。

这个图准确地显示了发生了什么。



此外，如果程序中没有其他变量引用旧元组，那么python的垃圾收集器将从内存中完全删除旧元组。这就是可变性的概念了这就是列表和元组的关键区别。

在CPython (Python最流行的实现)中，如果创建具有相同值的不可变对象，Python(在某些条件下)可能会将这些不同的对象捆绑到一起。

```
a = "Karim"
b = "Karim"

print(id(a))
print(id(b))
```

```
100011568
100011568
```

请记住，字符串(以及整数、浮点数和bool)也是不可变对象的所有示例。
正如您所看到的，即使在我们的python程序中，我们显式地创建了两个不同的字符串对象，python内部还是将它们绑定到一个字符串对象中。
我们怎么知道的?
因为a的恒等式和b的恒等式是一样的。

Python之所以能够做到这一点，是因为字符串的不可变性使得执行这种绑定是安全的。
不仅这将节省一些内存(不是多次存储的字符串在内存中),而且每次你想创建一个新的对象相同的值,python会创建一个引用的对象在内存中已经存在,绝对是更有效率。

这不仅适用于字符串，也适用于整数(在某些条件下)。

```
a = 1
b = 1

print(id(a))
print(id(b))
```

```
502066288
502066288
```

不会自动将两个等价的元组捆绑在一起

```
a = (1, 2)
b = (1, 2)

print(id(a))
print(id(b))
```

```
138761736
130360328
```

内建函数

列表具有比元组更多的内建函数。

```
list_num = [1,2,3,4]
tup_num = (1,2,3,4)

print(list_num)
print(tup_num)
```

```
[1, 2, 3, 4]
(1, 2, 3, 4)
```

```
print(dir(list_num))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
'__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

```
print(dir(tup_num))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count', 'index']
```

内存占用比较

```
a= (1,2,3,4,5,6,7,8,9,0)
b= [1,2,3,4,5,6,7,8,9,0]

print('a=',a.__sizeof__())
print('b=',b.__sizeof__())
```

```
a= 104
b= 120
```

元组也可以在字典中用作键，因为它们具有可hashable和不可变的特性，而列表在字典中不用作键，因为列表不能处理_hash__()，并且具有可变的特性。

```
key_val= {('alpha','bravo'):123} # Valid

key_val = {['alpha','bravo']:123} # Invalid
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-40-e0da2559cce6> in <module>()
      1 key_val= {('alpha','bravo'):123} #Valid
      2
----> 3 key_val = {['alpha','bravo']:123} #Invalid
```

```
TypeError: unhashable type: 'list'
```

当列表作为元组的元素时，该元素的值可以改变，修改内容为列表对象的元素而不是元组的元素对象

```
my_tuple = (4, 2, 3, [6, 5])

# TypeError: 'tuple' object does not support item assignment
# my_tuple[1] = 9

# However, item of mutable element can be changed
my_tuple[3][0] = 9    # Output: (4, 2, 3, [9, 5])
print(my_tuple)

# Tuples can be reassigned
my_tuple = ('p','r','o','g','r','a','m','i','z')

# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple)
```

```
(4, 2, 3, [9, 5])
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

使用 `+` 实现元组的拼接

```
# Concatenation
# Output: (1, 2, 3, 4, 5, 6)
print((1, 2, 3) + (4, 5, 6))

# Repeat
# Output: ('Repeat', 'Repeat', 'Repeat')
print(("Repeat",) * 3)
```

```
(1, 2, 3, 4, 5, 6)
('Repeat', 'Repeat', 'Repeat')
```

统计和索引

```
my_tuple = ('a','p','p','l','e',)

print(my_tuple.count('p'))  # Output: 2
print(my_tuple.index('l'))  # Output: 3
```

```
2
3
```

判断元素是否在元组中

```
my_tuple = ('a','p','p','l','e',)

# In operation
# Output: True
print('a' in my_tuple)

# Output: False
print('b' in my_tuple)

# Not in operation
# Output: True
print('g' not in my_tuple)
```

```
True
False
True
```

```
```

```
import timeit
print(timeit.timeit('x=(1,2,3,4,5,6,7,8,9)', number=100000))
print(timeit.timeit('x=[1,2,3,4,5,6,7,8,9]', number=100000))
```

```
0.0013953274557718153
0.006988655603198196
```

```
# Tuple 'n_tuple' with a list as one of its item.
n_tuple = (1, 1, [3,4])

#Items with same value have the same id.
print(id(n_tuple[0]) == id(n_tuple[1]))
```

```
True
```

元组没有append

```
n_tuple.append(5)
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-47-3cd258e024ff> in <module>()
----> 1 n_tuple.append(5)
```

```
AttributeError: 'tuple' object has no attribute 'append'
```

```
n_tuple[2].append(5)
print(n_tuple)
```

```
(1, 1, [3, 4, 5])
```

不能向元组添加元素，因为它们具有不可变的属性。元组没有append()或extend()方法，
不能从元组中删除元素，这也是由于它们的不可变性。元组没有remove()或pop()方法，
可以在元组中找到元素，因为这不会更改元组。
还可以使用in操作符检查元组中是否存在元素。
所以，如果你要定义一个常量值集你要做的就是迭代它，使用元组而不是列表。它将比使用列表更快，也更安全

添加元素

```
x = (1,2,3,4)
y = (5,6,7,8)

# Combining two tuples to form a new tuple
z = x + y

print(z)
```

```
(1, 2, 3, 4, 5, 6, 7, 8)
```

乘法复制

```
x = (1,2,3,4)
z = x*2
print(z)
```

```
(1, 2, 3, 4, 1, 2, 3, 4)
```

与Python列表不同，元组没有append()、remove()、extend()、insert()和pop()等方法，这是由于其不可变的特性。然而，还有许多其他内置的方法可以使用

count() and len()

```
a = [1,2,3,4,5,5]
print(a.count(5))
```

```
2
```

```
a = (1,2,3,4,5)
print(len(a))
```

```
5
```

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

```
4
```

any()

可以使用any()来发现元组中的任何元素是否可迭代。如果是这种情况，返回True，否则返回False。

```
a = (1,)
print(any(a))
```

```
True
```

注意，在上面元组a的声明中。如果在初始化元组中的单个项时没有指定逗号，Python假定您错误地添加了一对额外的括号(这是无害的)，但是数据类型不是元组。因此，在元组中声明单个项时，请记住添加逗号。

现在，回到any()函数:在布尔上下文中，项的值是不相关的。空元组为假，任何至少有一个项的元组为真。

```
b = ()
print(any(b))
```

```
False
```

min() and max()

```
print(max(a))
print(min(a))
```

```
1
1
```

```
# The string 'Apple' is automatically converted into a sequence of characters.
a = ('Apple')
print(max(a))
```

```
p
```

sum()

```
a = (1,2,3,4,5)

print(sum(a))
```

```
15
```

sorted()

```
a = (6,7,4,2,1,5,3)
print(sorted(a))
```

```
[1, 2, 3, 4, 5, 6, 7]
```