03-列表和元组，到底用哪一个？

你好，我是景霄。

前面的课程，我们讲解了Python语言的学习方法，并且带你了解了Python必知的常用工具——Jupyter。那么从这节课开始，我们将正式学习Python的具体知识。

对于每一门编程语言来说，数据结构都是其根基。了解掌握Python的基本数据结构，对于学好这门语言至关重要。今天我们就一起来学习，Python中最常见的两种数据结构：列表（list）和元组（tuple）。

列表和元组基础

首先，我们需要弄清楚最基本的概念，什么是列表和元组呢？

实际上，列表和元组，都是**一个可以放置任意数据类型的有序集合**。

在绝大多数编程语言中，集合的数据类型必须一致。不过，对于Python的列表和元组来说，并无此要求：

```
l = [1, 2, 'hello', 'world'] # 列表中同时含有int和string类型的元素
l
[1, 2, 'hello', 'world']

tup = ('jason', 22) # 元组中同时含有int和string类型的元素
tup
('jason', 22)
```

其次，我们必须掌握它们的区别。

- **列表是动态的**，长度大小不固定，可以随意地增加、删减或者改变元素（mutable）。
- **而元组是静态的**，长度大小固定，无法增加删减或者改变（immutable）。

下面的例子中，我们分别创建了一个列表与元组。你可以看到，对于列表，我们可以很轻松地让其最后一个元素，由4变为40；但是，如果你对元组采取相同的操作，Python 就会报错，原因就是元组是不可变的。

```
l = [1, 2, 3, 4]
l[3] = 40 # 和很多语言类似，python中索引同样从0开始，l[3]表示访问列表的第四个元素
l
[1, 2, 3, 40]

tup = (1, 2, 3, 4)
tup[3] = 40
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

可是，如果你想对已有的元组做任何"改变"，该怎么办呢？那就只能重新开辟一块内存，创建新的元组了。

比如下面的例子，我们想增加一个元素5给元组，实际上就是创建了一个新的元组，然后把原来两个元组的值依次填充进去。

而对于列表来说，由于其是动态的，我们只需简单地在列表末尾，加入对应元素就可以了。如下操作后，会修改原来列表中的元素，而不会创建新的列表。

```
tup = (1, 2, 3, 4)
new_tup = tup + (5, ) # 创建新的元组new_tup，并依次填充原元组的值
new _tup
(1, 2, 3, 4, 5)

l = [1, 2, 3, 4]
l.append(5) # 添加元素5到原列表的末尾
l
[1, 2, 3, 4, 5]
```

通过上面的例子，相信你肯定掌握了列表和元组的基本概念。接下来我们来看一些列表和元组的基本操作和注意事项。

首先，和其他语言不同，**Python中的列表和元组都支持负数索引**，-1表示最后一个元素，-2表示倒数第二个元素，以此类推。

```
l = [1, 2, 3, 4]
l[-1]
4

tup = (1, 2, 3, 4)
tup[-1]
4
```

除了基本的初始化，索引外，**列表和元组都支持切片操作**：

```
list = [1, 2, 3, 4]
l[1:3] # 返回列表中索引从1到2的子列表
[2, 3]

tup = (1, 2, 3, 4)
tup[1:3] # 返回元组中索引从1到2的子元组
(2, 3)
```

另外，列表和元组都**可以随意嵌套**：

```
l = [[1, 2, 3], [4, 5]] # 列表的每一个元素也是一个列表
```

```
tup = ((1, 2, 3), (4, 5, 6)) # 元组的每一个元素也是一元组
```

当然，两者也可以通过list()和tuple()函数相互转换：

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

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

最后，我们来看一些列表和元组常用的内置函数：

```
l = [3, 2, 3, 7, 8, 1]
l.count(3)
2
l.index(7)
3
l.reverse()
l
[1, 8, 7, 3, 2, 3]
l.sort()
l
[1, 2, 3, 3, 7, 8]

tup = (3, 2, 3, 7, 8, 1)
tup.count(3)
2
tup.index(7)
3
list(reversed(tup))
[1, 8, 7, 3, 2, 3]
sorted(tup)
[1, 2, 3, 3, 7, 8]
```

这里我简单解释一下这几个函数的含义。

- count(item)表示统计列表/元组中item出现的次数。
- index(item)表示返回列表/元组中item第一次出现的索引。
- list.reverse()和list.sort()分别表示原地倒转列表和排序（注意，元组没有内置的这两个函数)。
- reversed()和sorted()同样表示对列表/元组进行倒转和排序，但是会返回一个倒转后或者排好序的新的列表/元组。

列表和元组存储方式的差异

前面说了，列表和元组最重要的区别就是，列表是动态的、可变的，而元组是静态的、不可变的。这样的差异，势必会影响两者存储方式。我们可以来看下面的例子：

```
l = [1, 2, 3]
l.__sizeof__()
64
tup = (1, 2, 3)
tup.__sizeof__()
48
```

你可以看到，对列表和元组，我们放置了相同的元素，但是元组的存储空间，却比列表要少16字节。这是为什么呢？

事实上，由于列表是动态的，所以它需要存储指针，来指向对应的元素（上述例子中，对于int型，8字节）。另外，由于列表可变，所以需要额外存储已经分配的长度大小（8字节），这样才可以实时追踪列表空间的使用情况，当空间不足时，及时分配额外空间。

```
l = []
l.__sizeof__() // 空列表的存储空间为40字节
40
l.append(1)
l.__sizeof__()
72 // 加入了元素1之后，列表为其分配了可以存储4个元素的空间 (72 - 40)/8 = 4
l.append(2)
l.__sizeof__()
72 // 由于之前分配了空间，所以加入元素2，列表空间不变
l.append(3)
l.__sizeof__()
72 // 同上
l.append(4)
l.__sizeof__()
72 // 同上
l.append(5)
l.__sizeof__()
104 // 加入元素5之后，列表的空间不足，所以又额外分配了可以存储4个元素的空间
```

上面的例子，大概描述了列表空间分配的过程。我们可以看到，为了减小每次增加/删减操作时空间分配的开销，Python每次分配空间时都会额外多分配一些，这样的机制（over-allocating）保证了其操作的高效性：增加/删除的时间复杂度均为O(1)。

但是对于元组，情况就不同了。元组长度大小固定，元素不可变，所以存储空间固定。

看了前面的分析，你也许会觉得，这样的差异可以忽略不计。但是想象一下，如果列表和元组存储元素的个数是一亿，十亿甚至更大数量级时，你还能忽略这样的差异吗？

列表和元组的性能

通过学习列表和元组存储方式的差异，我们可以得出结论：元组要比列表更加轻量级一些，所以总体上来说，元组的性能速度要略优于列表。

另外，Python会在后台，对静态数据做一些**资源缓存**（resource caching）。通常来说，因为垃圾回收机制的存在，如果一些变量不被使用了，Python就会回收它们所占用的内存，返还给操作系统，以便其他变量或其他应用使用。

但是对于一些静态变量，比如元组，如果它不被使用并且占用空间不大时，Python会暂时缓存这部分内存。这样，下次我们再创建同样大小的元组时，Python就可以不用再向操作系统发出请求，去寻找内存，而是可以直接分配之前缓存的内存空间，这样就能大大加快程序的运行速度。

下面的例子，是计算**初始化**一个相同元素的列表和元组分别所需的时间。我们可以看到，元组的初始化速度，要比列表快5倍。

```
python3 -m timeit 'x=(1,2,3,4,5,6)'
20000000 loops, best of 5: 9.97 nsec per loop
python3 -m timeit 'x=[1,2,3,4,5,6]'
5000000 loops, best of 5: 50.1 nsec per loop
```

但如果是**索引操作**的话，两者的速度差别非常小，几乎可以忽略不计。

```
python3 -m timeit -s 'x=[1,2,3,4,5,6]' 'y=x[3]'
10000000 loops, best of 5: 22.2 nsec per loop
python3 -m timeit -s 'x=(1,2,3,4,5,6)' 'y=x[3]'
10000000 loops, best of 5: 21.9 nsec per loop
```

当然，如果你想要增加、删减或者改变元素，那么列表显然更优。原因你现在肯定知道了，那就是对于元组，你必须得通过新建一个元组来完成。

列表和元组的使用场景

那么列表和元组到底用哪一个呢？根据上面所说的特性，我们具体情况具体分析。

1. 如果存储的数据和数量不变，比如你有一个函数，需要返回的是一个地点的经纬度，然后直接传给前端渲染，那么肯定选用元组更合适。

```
def get_location():
    .....
    return (longitude, latitude)
```

2. 如果存储的数据或数量是可变的，比如社交平台上的一个日志功能，是统计一个用户在一周之内看了哪些用户的帖子，那么则用列表更合适。

```
viewer_owner_id_list = [] # 里面的每个元素记录了这个viewer一周内看过的所有owner的id
```

```
records = queryDB(viewer_id) # 索引数据库，拿到某个viewer一周内的日志
for record in records:
    viewer_owner_id_list.append(record.id)
```

总结

关于列表和元组，我们今天聊了很多，最后一起总结一下你必须掌握的内容。

总的来说，列表和元组都是有序的，可以存储任意数据类型的集合，区别主要在于下面这两点。

- 列表是动态的，长度可变，可以随意的增加、删减或改变元素。列表的存储空间略大于元组，性能略逊于元组。
- 元组是静态的，长度大小固定，不可以对元素进行增加、删减或者改变操作。元组相对于列表更加轻量级，性能稍优。

思考题

1. 想创建一个空的列表，我们可以用下面的A、B两种方式，请问它们在效率上有什么区别吗？我们应该优先考虑使用哪种呢？可以说说你的理由。

```
# 创建空列表
# option A
empty_list = list()

# option B
empty_list = []
```

2. 你在平时的学习工作中，是在什么场景下使用列表或者元组呢？欢迎留言和我分享。

精选留言：

- 胡峣 2019-05-15 00:20:42
  老师能不能讲一下list和tuple的内部实现，里边是linked list 还是array，还是把array linked一下这种。
  最后那个问题，类比java，new 是在heap，直接声明就可能在常量区了。老师能讲下Python的vm么，比如内存分配，gc算法之类的。 [32赞]

  作者回复2019-05-15 15:45:25
  1. list和tuple的内部实现都是array的形式，list因为可变，所以是一个over-allocate的array，tuple因为不可变，所以长度大小固定。具体可以参照源码list: https://github.com/python/cpython/blob/master/Objects/listobject.c. tuple: https://github.com/python/cpython/blob/master/Objects/tupleobject.c

  2. 最后的思考题：
  区别主要在于list()是一个function call，Python的function call会创建stack，并且进行一系列参数检查的操作，比较expensive，反观[]是一个内置的C函数，可以直接被调用，因此效率高。
  内存分配，GC等等知识会在第二章进阶里面专门讲到。

- Python高效编程 2019-05-15 12:11:34
  元素不需要改变时:
  两三个元素，使用 tuple，元素多一点使用namedtuple。
  元素需要改变时:
  需要高效随机读取，使用list。需要关键字高效查找，采用 dict。去重，使用 set。大型数据节省空间，使用标准库 array。大型数据高效操作，使用 numpy.array。 [15赞]

- 看，有只猪 2019-05-15 08:17:51
  []比list()更快，因为调用list函数有一定的开销，而[]却没有。
  这个有点像C语言中的内联函数与函数的差异 [11赞]

- adapt 2019-05-15 10:30:31
  如果一个列表在元组中的话，其实这个元组是”可变”的，只是这个可变只是能改变该列表里的内容。
  这一点作者没有讲到哦 。 [8赞]

- 对方正在输入中… 2019-05-15 09:22:25
  python -m timeit 'empty_list = list()'
  10000000 loops, best of 3: 0.0829 usec per loop

  python -m timeit 'empty_list = []'
  10000000 loops, best of 3: 0.0218 usec per loop

  python -m timeit 'empty_list = ()'
  100000000 loops, best of 3: 0.0126 usec per loop
  测试结果，虽然直接创建元组初始化速度最快，但是由于要用list函数转一道反而不如直接创建列表的速度快。 [7赞]

- Geek_59f23e 2019-05-15 14:10:39
  实测被打脸了😁函数构建和直接构建一个空列表或数组速度上并没有什么差别，有时前者快些，有时后者快些。。。

  In [1]: timeit 'lst1 = []'
  9.86 ns ± 0.721 ns per loop (mean ± std. dev. of 7 runs, 100000000 loops each)

In [2]: timeit 'lst2 = list()'
9.82 ns ± 0.43 ns per loop (mean ± std. dev. of 7 runs, 100000000 loops each)

In [3]: timeit 'tup1 = (,)'
9.59 ns ± 0.294 ns per loop (mean ± std. dev. of 7 runs, 100000000 loops each)

In [4]: timeit 'tup2 = tuple()'
9.75 ns ± 0.464 ns per loop (mean ± std. dev. of 7 runs, 100000000 loops each) [3赞]

作者回复2019-05-15 15:57:14
你的命令有些奇怪。在程序里也应该是timeit(...)，试过用文中的命令测试的结果吗？

另外你python的版本和运行环境的截图能贴一下吗？

- 2019-05-15 11:05:26
  1. 测试了一下， []快于list()
  2. 一般在key中使用元祖，其他情况多数都使用列表 [3赞]

- kevinsu 2019-05-15 17:20:46
  import timeit
  print(timeit.timeit('list(x for x in range(1,1000))',number=10000))
  print(timeit.timeit('[x for x in range(1,1000)]',number=10000))
  0.6829426919994148
  0.36637431800045306
  看着是[]更快些，😄 [2赞]

- DUDUANWANGGU198.com 2019-05-15 12:34:04
  老师请问一下，为什么l = [1, 2, 3]消耗的空间为64字节，而l.append(1), l.append(2), l.append(3)消耗的空间为72字节，这不是相同的列表吗？ [2赞]

  作者回复2019-05-15 15:50:48
  列表的over-allocate是在你加入了新元素之后解释器判断得出当前存储空间不够，给你分配额外的空间，因此
  l=[], l.append(1), l.append(2), l.append(3)实际分配了4个元素的空间。但是l=[1, 2, 3]直接初始化列表，并没有增加元素的操作，因此只会分配3个元素的空间

- 許敲敲 2019-05-15 08:13:16
  思考题 试了运行timeit，发现在我电脑上两个创建列表的时间一样，所以答案是什么呢？ [2赞]

- leechaochao 2019-05-16 16:27:55
  list存的是元素的指针，那么
  1. 存指针的地方是array吗
  2. 存元素的地方也是array吗 [1赞]

- 呼啦啦 2019-05-16 06:40:45
  老师，我测试了一下，好像只有0.1nsec秒之差。没有五倍之多，我是用python3.7测试的。
  C:\Users\wuzhaoming>python -m timeit 'x=(1,2,3,4,5,6)'
  50000000 loops, best of 5: 8.1 nsec per loop

  C:\Users\wuzhaoming>python -m timeit 'x=[1,2,3,4,5,6]'

50000000 loops, best of 5: 8.2 nsec per loop [1赞]

作者回复2019-05-16 15:45:31
怀疑windows平台是不是加了什么限制，你放到linux或者mac上看看（或者用前面一讲的云端jupyter或者colab上试试）

- 不瘦到140不改名 2019-05-15 23:09:56
print([].__sizeof__()) # 40
print(().__sizeof__()) # 24
老师 我想问一下，列表比元组多了16个字节，由于列表是可变的，所以需要分配8字节来存储已经分配的长度大小，那剩余的8字节干什么了呢？ [1赞]

作者回复2019-05-16 16:12:27
文中有提到。元祖是直接存储的元素，但是列表存储的是指向元素的指针，这就是你说的剩余的8字节。
可以参考源码：
列表：https://github.com/python/cpython/blob/3.7/Include/listobject.h
元祖：https://github.com/python/cpython/blob/3.7/Include/tupleobject.h

- 安亚明 2019-05-15 21:02:53
老师，PYTHON学校为何要从元组和列表开始。 [1赞]

作者回复2019-05-16 03:30:56
因为这是最基本的数据结构啊，学语言肯定得先了解数据结构啊

- 汤尼房 2019-05-15 20:25:34
景老师，一直在想一个tuple元组如何拥有大数据量的元素，比如千万个元素、上亿个元素。因为tuple是静态的，不能添加元素，于是今天实践将[i for i in xrange(1000000000)]给初始化成tuple，发现初始化的过程相当耗时间，之前也希望利用tuple的性能比list好的优点，想把含有大数据量的list给转换成tuple来处理，今天实践发现初始化过程非常耗时间，请问景老师，平时在工作过程中遇到的含有大数据量个元素的tuple是如何形成的呢？ [1赞]

- Danpier 2019-05-15 18:32:38
>>> l=list()
>>> l.__sizeof__()
20
>>> l.append(1)
>>> l.__sizeof__()
36
老师，我电脑实测结果都比你的空间分配少了一半，是因为32-bit系统的缘故吗？

还有，对于思考题，按老师的示例测试固定的loop次数（分别是100000、200000、500000、1000000、5000000）,两者的性能差别竟然不相上下（测试环境python2.7.1和3.7.1都一样），而进入交互模式才能明显测出[ ]比list( )高效，这个好奇怪？ [1赞]

- Brigand 2019-05-15 18:23:53
元组更像底层数组，列表更像数组的封装！ [1赞]

- Geek_59f23e 2019-05-15 18:14:46
我用的是ipython测试timeit，元组生成速度是列表的5倍，这个测试结果是一致的。空列表和list方法差异不大，我感觉这两种生成方法应该被优化过了等价，运行环境如下:

Python 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.2.0 -- An enhanced Interactive Python. Type '?' for help.
[1赞]

- 刘朋 2019-05-15 16:26:27
  import timeit
  timeit.timeit('a=list()',number=10000) 返回 0.0006914390251040459
  timeit.timeit('a=[]',number=10000) 返回 0.00018375739455223083
  timeit.timeit('a=()',number=10000) 返回 0.00010870955884456635 [1赞]

- leechaochao 2019-05-15 16:22:29
  list的内部实现是over-allocate array的形式
  1. 那在需要扩容的时候，是不是也是需要重新开辟一块连续的内存空间呢？
  2. 每次扩容都会预留一些空间，这里面有没有公式，公式是什么呢 [1赞]

  作者回复2019-05-16 16:03:36
  1. 是的
  2. 你可以自己overallocate的pattern一般是0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...