第5章 列表、元组和字符串

5.1 列表：一个“打了激素”的数组


视频讲解

有时候可能需要将一些相互之间有关联的数据保存到一起，很多接触过编程的读者脑海里浮现出来的第一个概念应该就是数组。数组允许把一些相同类型的数据挨个儿摆在一起，然后通过下标进行索引。

Python 也有类似数组的东西，不过更为强大。由于 Python 的变量没有数据类型，所以 Python 的“数组”可以同时存放不同类型的变量。这么厉害的东西，Python 将其称为列表，姑且可以认为列表就是一个“打了激素”的数组。

5.1.1 创建列表

创建一个列表非常简单，只需要使用中括号将数据包裹起来（数据之间用逗号分隔）就可以了。

```
>>> [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

上面创建了一个匿名的列表，因为没有名字，所以创建完也没办法再次使用它。为了可以随时对它进行引用和修改，可以给它贴上一个变量名：

```
>>> number = [1, 2, 3, 4, 5]
>>> type(number)
<class 'list'>
>>> for each in number:
    print(each)

1
2
```

```
3
4
5
```


注意：

type()函数用于返回指定参数的类型，list 即列表的意思。

没有哪一项规定要求 Python 的列表保存同一类型的数据，因此，它支持将各种不同的数据存放到一起：

```
>>> mix = [520, "小甲鱼", 3.14, [1, 2, 3]]
```

可以看到这个列表里有整型、字符串、浮点型数据，甚至还可以包含另一个列表。

当实在想不到要往列表里面塞什么数据的时候，可以先创建一个空列表：

```
>>> empty = []
```

5.1.2 向列表添加元素

列表并不是一成不变的，可以随意地往里面添加新的元素。添加元素到列表中，可以使用 append()方法：

```
>>> number = [1, 2, 3, 4, 5]
>>> number.append(6)
>>> number
[1, 2, 3, 4, 5, 6]
```

可以看到，数字 6 已经被添加到列表 number 的末尾了。有读者可能会问，这个 append()的调用怎么跟平时的 BIF 内置函数调用不一样呢？

因为 append()并不是一个 BIF，它是属于列表对象的一个方法。中间这个“.”，暂时可以理解为范围的意思：append()这个方法是属于一个叫 number 的列表对象的。关于对象的知识，暂时只需要理解这么多，后面小甲鱼会再详细地来介绍对象。

下面代码试图将数字 8 和 9 同时添加进 number 列表中：

```
>>> number.append(8, 9)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    number.append(8, 9)
TypeError: append() takes exactly one argument (2 given)
```

出错了，这是因为 append()方法只支持一个参数。

如果希望同时添加多个数据，可以使用 extend()方法向列表末尾添加多个元素：

```
>>> number.extend([8, 9])
>>> number
[1, 2, 3, 4, 5, 6, 8, 9]
```

注意：

extend()事实上是使用一个列表来扩充另一个列表，所以它的参数是另一个列表。

无论是 append()还是 extend()方法，都是往列表的末尾添加数据，那么是否可以将数据插入到指定的位置呢？

当然没问题，想要往列表的任意位置插入元素，可以使用到 insert()方法。

insert()方法有两个参数：第一个参数指定待插入的位置（索引值），第二个参数是待插入的元素值。

下面代码将数字 0 插入到 number 列表的最前面：

```
>>> number.insert(0, 0)
>>> number
[0, 1, 2, 3, 4, 5, 6, 8, 9]
```

在计算机编程中常常会出现一些“反常识”的知识点，如在 Python 列表中，第一个位置的索引值是 0，第二个是 1，第三个是 2，以此类推……

下面代码将数字 7 插入到 6 和 8 之间：

```
>>> number.insert(7, 7)
>>> number
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

insert()方法中代表位置的第一个参数还支持负数，表示与列表末尾的相对距离：

```
>>> number.insert(-1, 8.5)
>>> number
[0, 1, 2, 3, 4, 5, 6, 7, 8, 8.5, 9]
```

5.1.3　从列表中获取元素

视频讲解

通过索引值可以直接获取列表中的某个元素：

```
>>> eggs = ["鸡蛋", "鸭蛋", "鹅蛋", "铁蛋"]
>>> eggs[0]
'鸡蛋'
>>> eggs[3]
'铁蛋'
```

如果想要访问列表中最后一个元素，怎么办？可以使用 len()函数获取该列表的长度（元素个数），再减 1 就是这个列表最后一个元素的索引值：

```
>>> eggs = ["鸡蛋", "鸭蛋", "鹅蛋", "铁蛋"]
>>> eggs[len(eggs)-1]
'铁蛋'
>>> eggs[len(eggs)-2]
'鹅蛋'
```

len()函数的调用直接省去也可以实现同样的效果，即当索引值为负数时，表示从列表的末尾反向索引：

```
>>> eggs = ["鸡蛋", "鸭蛋", "鹅蛋", "铁蛋"]
>>> eggs[-1]
'铁蛋'
>>> eggs[-2]
'鹅蛋'
```

如果要将“鸭蛋”和“铁蛋”的位置进行调换，通常可以这么写：

```
>>> eggs = ["鸡蛋", "鸭蛋", "鹅蛋", "铁蛋"]
>>> temp = eggs[1]
>>> eggs[1] = eggs[3]
>>> eggs[3] = temp
>>> eggs
['鸡蛋', '铁蛋', '鹅蛋', '鸭蛋']
```

这里的 temp 是一个临时变量，避免相互覆盖。不过 Python 允许适当地“偷懒”，下面代码可以实现相同的功能：

```
>>> eggs[1], eggs[3] = eggs[3], eggs[1]
>>> eggs
['鸡蛋', '铁蛋', '鹅蛋', '鸭蛋']
```

有时候可能需要开发一个具有“抽奖”功能的程序，只需要先将“奖项/参与者”放到列表里面，然后配合 random 模块即可实现：

```
>>> import random
>>> prizes = ['鸡蛋', '铁蛋', '鹅蛋', '鸭蛋']
>>> random.choice(prizes)
'鹅蛋'
>>> random.choice(prizes)
'鸭蛋'
```

random 的 choice()方法可以从一个非空的序列（如列表）中随机获取一个元素。

列表中还可以包含另一个列表，如果要获取内部子列表的某个元素，应该使用两次索引：

```
>>> eggs = ['鸡蛋', '铁蛋', ['天鹅蛋', '企鹅蛋', '加拿大鹅蛋'], '鸭蛋']
>>> eggs[2][2]
'加拿大鹅蛋'
```

5.1.4 从列表删除元素

从列表中删除元素，可以有三种方法实现：remove()、pop()和 del。

remove()方法需要指定一个待删除的元素：

```
>>> eggs
['鸡蛋', '铁蛋', '鹅蛋', '鸭蛋']
>>> eggs.remove("铁蛋")
>>> eggs
['鸡蛋', '鹅蛋', '鸭蛋']
```

使用 remove()删除元素，并不需要知道这个元素在列表中的具体位置。但是如果指定的元素不存在于列表中，程序就会报错：

```
>>> eggs.remove("卤蛋")
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    eggs.remove("卤蛋")
ValueError: list.remove(x): x not in list
```

pop()方法是将列表中的指定元素“弹”出来，也就是取出并删除该元素的意思，它的参数是一个索引值：

```
>>> eggs.pop(1)
'鹅蛋'
>>> eggs
['鸡蛋', '鸭蛋']
```

如果不带参数，pop()方法默认是弹出列表中的最后一个元素：

```
>>> eggs.pop()
'鸭蛋'
>>> eggs
['鸡蛋']
```

最后一个是 del 语句，注意，它是一个 Python 语句，而不是 del 列表的方法，或者 BIF：

```
>>> del eggs[0]
>>> eggs
[]
```

del 语句在 Python 中的用法非常丰富，不仅可以用来删除列表中的某个（些）元素，还可以直接删除整个变量：

```
>>> del eggs
>>> eggs
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    eggs
NameError: name 'eggs' is not defined
```

分析：上面代码由于 eggs 整个变量被 del 语句删除了，所以再次引用时，Python 由于找不到该变量，便会报错。

5.1.5 列表切片

切片（slice）语法的引入，使得 Python 的列表真正地走向了高端。这个连 Python 之父都爱不释手的语法真有那么神奇吗？不妨来试一试。

现在要求将列表 list1 中的三个元素取出来，放到列表 list2 里面。学了前面的知识，可以使用“笨”方法来实现：

```
>>> list1 = ["钢铁侠", "蜘蛛侠", "蝙蝠侠", "绿灯侠", "神奇女侠"]
>>> list2 = [list1[2], list1[3], list1[4]]
>>> list2
['蝙蝠侠', '绿灯侠', '神奇女侠']
```

像这样，从一个列表中取出部分元素是非常常见的操作，但这里是取出三个元素，如果要求取出列表中最后 200 个元素，那不是很心酸？

其实动动脑筋还是可以实现的：

```
>>> list2 = []
>>> for i in range(-200, 0):
        list2.append(list1[i])
```

虽然可以实现，但是每次都要套个循环跑一圈，未免也太烦琐了！切片的引入，大大地简化了这种操作：

```
>>> list1 = ["钢铁侠", "蜘蛛侠", "蝙蝠侠", "绿灯侠", "神奇女侠"]
>>> list2 = list1[2:5]
>>> list2
['蝙蝠侠', '绿灯侠', '神奇女侠']
```

很简单对吧？只不过是用一个冒号隔开两个索引值，左边是开始位置，右边是结束位置。这里要注意的一点是：结束位置上的元素是不包含的（如上面例子中，“神奇女侠”的索引值是 4，如果写成 list1[2:4]，便不能将其包含进来）。

使用列表切片也可以“偷懒”，之前提到过 Python 是以简洁而闻名于世，所以你能想到的“便捷方案”，Python 的作者以及 Python 社区的小伙伴们都已经想到了，并付诸实践，你要做的就是验证一下是否可行：

```
>>> list1 = ["钢铁侠", "蜘蛛侠", "蝙蝠侠", "绿灯侠", "神奇女侠"]
>>> list1[:2]
['钢铁侠"', '蜘蛛侠']
>>> list1[2:]
['蝙蝠侠', '绿灯侠', '神奇女侠']
>>> list1[:]
['钢铁侠', '蜘蛛侠', '蝙蝠侠', '绿灯侠', '神奇女侠']
```

如果省略了开始位置，Python 会从 0 这个位置开始。同样道理，如果要得到从指定索引值到列表末尾的所有元素，把结束位置也省去即可。如果啥都没有，只有一个冒号，

Python 将返回整个列表的拷贝。

这种方法有时候非常方便，如想获取列表最后的几个元素，可以这么写：

```
>>> list1 = list(range(100))
>>> list1[-10:]
[90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

注意：

列表切片并不会修改列表自身的组成结构和数据，它其实是为列表创建一个新的拷贝（副本）并返回。

5.1.6　进阶玩法

列表切片操作实际上还可以接受第三个参数，其代表的是步长，默认值为 1。下面将步长修改为 2，看看有什么神奇的效果？

```
>>> list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list1[0:9:2]
[1, 3, 5, 7, 9]
```

其实该代码还可以直接写成 list1[::2]，实现效果是一样的。

如果将步长设置为负数，如−1，结果会是怎样呢？

```
>>> list1[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

这就很有意思了，将步长设置为−1，相当于将整个列表翻转过来。

上面这些列表切片操作都是获取列表加工后（切片）的拷贝，并不会影响到原有列表的结构：

```
>>> list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list1[::-2]
[9, 7, 5, 3, 1]
>>> list1
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

但如果将 del 语句作用于列表切片，其结果又让人大跌眼镜：

```
>>> del list1[::2]
>>> list1
[2, 4, 6, 8]
```

是的，del 直接作用于原始列表了，因为不这样做的话，代码就失去意义了，不是吗？同样会作用于原始列表的操作还有为切片后的列表赋值：

```
>>> list1 = ["钢铁侠", "蜘蛛侠", "蝙蝠侠", "绿灯侠", "神奇女侠"]
>>> list1[0:2] = ["超人", "闪电侠"]
```

```
>>> list1
['超人', '闪电侠', '蝙蝠侠', '绿灯侠', '神奇女侠']
```


视频讲解

5.1.7 一些常用操作符

此前学过的大多数操作符都可以运用到列表上：

```
>>> list1 = [123]
>>> list2 = [234]
>>> list1 > list2
False
>>> list1 <= list2
True
>>> list3 = ['apple']
>>> list4 = ['pineapple']
>>> list3 < list4
True
```

列表好像挺聪明的，不仅懂得比大小，还知道菠萝（pineapple）比苹果（apple）大？那如果列表中不止一个元素呢？结果又会如何？

```
>>> list1 = [123, 456]
>>> list2 = [234, 123]
>>> list1 > list2
False
```

怎么会这样？Python做出这样的判断是基于什么根据呢？总不会是随机瞎猜的吧？

list1 列表两个元素的和是 579，按理应该比 list2 列表的和 357 要大，那为什么 list1>list2 还会返回 False 呢？

其实，Python 的列表原来并没有我们想象中那么“智能”，当列表包含多个元素的时候，默认是从第一个元素开始比较，只要有一个 PK 赢了，就算整个列表赢了。字符串比较也是同样的道理（字符串比较的是每一个字符对应的 ASCII 码值的大小）。

前面演示过字符串可以使用加号（+）进行拼接，使用乘号（*）来实现自我复制。这两个操作符也可以作用于列表：

```
>>> list1 = [123, 456]
>>> list2 = [234, 123]
>>> list3 = list1 + list2
>>> list3
[123, 456, 234, 123]
```

加号(+)也叫连接操作符，它允许把多个列表对象合并在一起，其实就相当于 extend()方法实现的效果。一般情况下建议使用 extend()方法来扩展列表，因为这样显得更为规范和专业。另外，连接操作符并不能实现列表添加新元素的操作：

```
>>> list1 = [123, 456]
```

```
>>> list2 = list1 + 789
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    list2 = list1 + 789
TypeError: can only concatenate list (not "int") to list
```

乘号（*）也叫重复操作符，重复操作符同样可以用于列表中：

```
>>> list1 = ["FishC"]
>>> list1 * 3
['FishC', 'FishC', 'FishC']
```

另外有个成员关系操作符大家也不陌生了，我们是在谈 for 循环的时候认识它的，成员关系操作符就是 in 和 not in：

```
>>> list1 = ["小猪", "小猫", "小狗", "小甲鱼"]
>>> "小甲鱼" in list1
True
>>> "小乌龟" not in list1
True
```

之前说过列表里边可以包含另一个列表，那么对于列表中的列表的元素，能不能使用 in 和 not in 测试呢？试试便知：

```
>>> list1 = ["小猪", "小猫", ["小甲鱼", "小乌龟"], "小狗"]
>>> "小甲鱼" in list1
False
>>> "小乌龟" not in list1
True
```

可见 in 和 not in 只能判断一个层次的成员关系，这跟 break 和 continue 语句只能跳出一个层次的循环是一个道理。

在开发中，有时候需要去除列表中重复的数据，只要利用好 in 和 not in，就可以巧妙地实现：

```
>>> old_list = ['西班牙', '葡萄牙', '葡萄牙', '牙买加', '匈牙利']
>>> new_list = []
>>> for each in old_list:
    if each not in new_list:
        new_list.append(each)

>>> print(new_list)
['西班牙', '葡萄牙', '牙买加', '匈牙利']
```

分析：代码先迭代遍历 old_list 的每一个元素，如果该元素不存在于 new_list 中，便调用列表的 append()方法添加进去。

5.1.8 列表的小伙伴们

接下来认识一下列表的小伙伴们，列表有多少小伙伴呢？不妨让 Python 自己告诉我们：

```
>>> dir(list)
['__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']
```

产生了一个熟悉又陌生的列表，很多熟悉的方法似曾相识，如 append()、extend()、insert()、pop()、remove()都是学过的。下面再给大家介绍几个常用的方法。

count()方法的作用是统计某个元素在列表中出现的次数：

```
>>> list1 = [1, 1, 2, 3, 5, 8, 13, 21]
>>> list1.count(1)
2
```

index()方法的作用是返回某个元素在列表中第一次出现的索引值：

```
>>> list1.index(1)
0
```

index()方法可以限定查找的范围：

```
>>> start = list1.index(1) + 1
>>> stop = len(list1)
>>> list1.index(1, start, stop)
1
```

reverse()方法的作用是将整个列表原地翻转：

```
>>> list1 = [1, 1, 2, 3, 5, 8, 13, 21]
>>> list1.reverse()
>>> list1
[21, 13, 8, 5, 3, 2, 1, 1]
```

sort()方法的作用是对列表元素进行排序：

```
>>> list1 = [8, 9, 3, 5, 2, 6, 10, 1, 0]
>>> list1.sort()
>>> list1
```

```
[0, 1, 2, 3, 5, 6, 8, 9, 10]
```

那如果需要从大到小排队呢？很简单，先调用 sort()方法，列表会先从小到大排好队，然后调用 reverse()方法原地翻转就可以啦。

什么？太麻烦？好吧，大家真是越来越懒了……很好，“懒”有时候确实是发明创新的原动力。其实，sort()这个方法有三个参数，语法形式为：

```
sort(func, key, reverse)
```

func 和 key 参数用于设置排序的算法和关键字，默认是使用归并排序，算法问题不在这里讨论，感兴趣的朋友可以参考小甲鱼的另一部视频教程——《数据结构和算法》。这里讨论 sort()方法的第三个参数：reverse，没错，就是刚刚学的那个 reverse()方法的 reverse。不过这里作为 sort()的一个默认参数，它的默认值是 sort(reverse=False)，表示不颠倒顺序。因此，只需要把 False 改为 True，列表就相当于从大到小排序：

```
>>> list1 = [8, 9, 3, 5, 2, 6, 10, 1, 0]
>>> list1.sort(reverse=True)
>>> list1
[10, 9, 8, 6, 5, 3, 2, 1, 0]
```

5.2　元组：戴上了“枷锁”的列表


视频讲解

接下来介绍的是列表的“表亲”——元组。

元组和列表的最大区别是：元组只可读，不可写。也就是说，可以任意修改（插入/删除）列表中的元素，而对于元组来说这些操作是不行的，元组只可以被访问，不能被修改。

5.2.1　创建和访问一个元组

元组和列表，除了不可改变这个显著特征之外，还有一个明显的区别：创建列表用的是中括号，而创建元组大部分时候使用的是小括号：

```
>>> tuple1 = (1, 2, 3, 4, 5, 6, 7, 8)
>>> tuple1
(1, 2, 3, 4, 5, 6, 7, 8)
>>> type(tuple1)
<class 'tuple'>
```


注意：

tuple 即元组的意思。

访问元组的方式与列表无异，也是通过索引值访问一个或多个（切片）元素：

```
>>> tuple1[1]
2
>>> tuple1[5:]
(6, 7, 8)
>>> tuple1[:5]
(1, 2, 3, 4, 5)
```

复制一个元组，通常可以使用切片来实现：

```
>>> tuple2 = tuple1[:]
>>> tuple2
(1, 2, 3, 4, 5, 6, 7, 8)
```

如果试图修改元组，那么抱歉，Python 会很快通过报错来回应：

```
>>> tuple1[1] = 1
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    tuple1[1] = 1
TypeError: 'tuple' object does not support item assignment
```

列表的标识符是中括号（[]），那么元组的标识符号是什么呢？

小甲鱼相信 90%的朋友都会不假思索地回答：小括号！是这样吗？不妨来做个实验：

```
>>> tuple1 = (520)
>>> type(tuple1)
<class 'int'>
```

这里，type()函数告诉我们 temp 变量是 int（整型）。

是的，小括号还有其他的功能，在这里它就被当作操作符使用了……所以，如果想要元组中只包含一个元素，可以在该元素后面添加一个逗号（,）来实现：

```
>>> tuple1 = (520,)
>>> type(tuple1)
<class 'tuple'>
```

其实小括号也是可以不要的：

```
>>> tuple2 = 520,
>>> tuple1 == tuple2
True
>>> tuple3 = 1, 2, 3, 4, 5
>>> type(tuple3)
<class 'tuple'>
```

发现了吧？逗号（,）才是关键，小括号只是起到补充的作用。再举个例子来对比：

```
>>> 8 * (8)
64
>>> 8 * (8,)
```

```
(8, 8, 8, 8, 8, 8, 8, 8)
```

5.2.2　更新和删除元组

有的读者可能会说，刚才不是说“元组是板上钉钉不能修改的吗？”现在又来谈更新一个元组，小甲鱼你这不是自己自相矛盾吗？

大家不要激动……我们只是讨论一个相对灵活的做法，与元组的定义并不冲突。由于元组中的元素是不允许被修改的，但这并不妨碍我们创建一个新的同名元组：

```
>>> x_men = ("金刚狼", "X 教授", "暴风女", "火凤凰", "镭射眼")
>>> x_men[1] = "小甲鱼"
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    x_men[1] = "小甲鱼"
TypeError: 'tuple' object does not support item assignment
>>> # 上面这样做是不行的
>>> # 下面的做法是可行的
>>> x_men = (x_men[0], "小甲鱼") + x_men[2:]
>>> x_men
('金刚狼', '小甲鱼', '暴风女', '火凤凰', '镭射眼')
```

这段代码其实是利用切片和拼接实现更新元组的目的，它并不是修改元组自身，而是耍了“狸猫换太子”的小手段。

下面代码可以证明小甲鱼所言非虚：

```
>>> x_men = ("金刚狼", "X 教授", "暴风女", "火凤凰", "镭射眼")
>>> id(x_men)
2325773492160
>>> x_men = (x_men[0], "小甲鱼") + x_men[2:]
>>> id(x_men)
2325773560296
```

id()函数用于返回指定对象的唯一 id 值，这个 id 值可以理解为现实生活中的身份证，在同一生命周期中，Python 确保每个对象的 id 值是唯一的。上面两个元组虽然都叫 x_men，但是 id 值出卖了它们——两者并不是同一个对象。

5.1.4 节介绍了三种方法删除列表里边的元素，但是由于元组具有不可以被修改的原则，所以删除元素的操作理论来说是不存在的。如果非要这么做，建议使用上面的技巧实现：

```
>>> temp = temp[:2] + temp[3:]
>>> x_men = x_men[:1] + x_men[2:]
>>> x_men
('金刚狼', '暴风女', '火凤凰', '镭射眼')
```

删除整个元组，只需要使用 del 语句：

```
>>> del x_men
>>> x_men
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    x_men
NameError: name 'x_men' is not defined
```

其实在日常开发中，很少使用 del 去删除整个元组，因为 Python 的垃圾回收机制会在某个对象不再被使用的时候自动进行清理。

最后小结一下哪些操作符可以使用在元组上，拼接操作符和重复操作符刚刚演示过了，关系操作符、逻辑操作符和成员关系操作符（in 和 not in）也可以直接应用在元组上，这与列表是一样的，大家自己实践一下就知道了。关于列表和元组，今后会谈得更多，目前，就先聊到这里。


视频讲解

5.3 字符串

或许现在又回过头来谈字符串，有些读者可能会觉得没必要。其实关于字符串，还有很多你可能不知道的秘密，由于字符串在日常使用中是如此常见，因此小甲鱼抱着负责任的态度在本节把所知道的都与大家分享一下。

在一些编程语言中，字符和字符串是两个不同的概念，如 C 语言使用单引号将字符括起来，使用双引号包含字符串。但在 Python 中，只有字符串这一个概念：

```
>>> str1 = "I love FishC.com!"
>>> str1
'I love FishC.com!'
```


注意：

可以使用单引号将字符串包裹起来，也可以使用双引号，但务必要成对编写，不能一边是单引号而另一边是双引号。

在学习了列表和元组之后，我们掌握了一个新的操作——切片，事实上也可以应用到字符串上：

```
>>> str1[7:]
'FishC.com!'
```

字符串与元组一样，都是属于“一言既出，驷马难追”的家伙。所以，一旦确定下来就不能再对它进行修改。如果非要这么做，仍然可以利用切片和拼接来实现：

```
>>> str2 = "一只穿云箭，千军万马来相见！"
>>> str2 = str2[:1] + '支' + str2[2:]
>>> str2
'一支穿云箭，千军万马来相见！'
```

这种通过拼接旧字符串的各个部分组合得到新字符串的方式，并不是真正意义上的修改字符串。原来的那个旧的字符串其实还在，只不过我们将变量名指向了拼接后的新字符串。旧的字符串一旦失去了变量的引用，就会被 Python 的垃圾回收机制释放掉。

比较操作符、逻辑操作符、成员关系操作符的操作和列表、元组是一样的，这里就不再赘述。

5.3.1　各种内置方法

列表和元组都有一些内置方法，大家可能觉得它们的方法已经非常多了，其实字符串的方法更多。表 5-1 总结了字符串的所有方法及对应的含义。

表 5-1　Python 字符串的方法及含义

方　法	含　义
capitalize()	将字符串的第一个字符修改为大写，其他字符全部改为小写
casefold()	将字符串的所有字符修改为小写
center(width[,fillchar])	当字符个数大于 width 时，字符串不变； 当字符个数小于 width 时，字符串居中，并在左右填充空格以达到 width 指定宽度； fillchar 参数可选，指定填充的字符（默认是空格）
count(sub[,start[,end]])	返回 sub 参数在字符串里边出现的次数； start 和 end 参数可选，指定统计范围
encode(encoding='utf-8',errors='strict')	以 encoding 参数指定的编码格式对字符串进行编码，并返回 errors 参数指定出错时的处理方式，默认是抛出 UnicodeError 异常，还可以使用'ignore'、'replace'、'xmlcharrefreplace'、'backslashreplace'等处理方式
endswith(sub[,start[,end]])	检查字符串是否以 sub 参数结束，如果是返回 True，否则返回 False；start 和 end 参数可选，指定范围
expandtabs(tabsize=8)	把字符串中的制表符（\t）转换为空格代替
find(sub[,start[,end]])	检查 sub 参数是否包含在字符串中，如果有则返回第一个出现位置的索引值，否则返回–1；start 和 end 参数可选，表示范围
index(sub[,start[,end]])	跟 find()方法一样，不过该方法如果找不到将抛出一个 ValueError 异常
isalnum()	如果字符串仅由字母或数字构成则返回 True，否则返回 False
isalpha()	如果字符串仅由字母构成则返回 True，否则返回 False
isdecimal()	如果字符串仅由十进制数字构成则返回 True，否则返回 False
isdigit()	如果字符串仅由数字构成则返回 True，否则返回 False
islower()	如果字符串仅由小写字母构成则返回 True，否则返回 False
isnumeric()	如果字符串仅由数值构成则返回 True，否则返回 False
isspace()	如果字符串仅由空白字符构成则返回 True，否则返回 False
istitle()	如果是标题化（所有的单词均以大写字母开始，其余字母皆小写）字符串则返回 True，否则返回 False

续表

方　法	含　义
isupper()	如果字符串仅由大写字母构成则返回 True，否则返回 False
join(iterable)	以字符串作为分隔符，插入到 iterable 参数迭代出来的所有字符串之间； 如果 iterable 中包含任何非字符串值，将抛出 TypeError 异常
ljust(width[,fillchar])	当字符个数大于 width 时，字符串不变； 当字符个数小于 width 时，左对齐字符串，并在右边填充空格以达到 width 指定宽度； fillchar 参数可选，指定填充的字符（默认是空格）
lower()	将字符串的所有大写字母修改为小写字母
lstrip([chars])	删除字符串左边的所有空白字符； chars 参数可选，指定待删除的字符集
partition(sep)	找到 sep 参数第一次出现的位置，并将字符串切分成一个三元组（sep 前面的子字符串,sep,sep 后面的子字符串）； 如果字符串中不包含 sep，则返回三元组('原字符串','','')
replace(old,new[,count])	将字符串中的 old 参数指定的字符串替换成 new 参数指定的字符串； count 参数可选，表示最多替换次数不超过 count
rfind(sub[,start[,end]])	类似于 find()方法，不过是从右边开始查找
rindex(sub[,start[,end]])	类似于 index()方法，不过是从右边开始查找
rjust(width[,fillchar])	当字符个数大于 width 时，字符串不变； 当字符个数小于 width 时，右对齐字符串，并在右边填充空格以达到 width 指定宽度； fillchar 参数可选，指定填充的字符（默认是空格）
rpartition(sep)	类似于 partition()方法，不过是从右边开始查找
rstrip([chars])	删除字符串右边的所有空白字符； chars 参数可选，指定待删除的字符集
split(sep=None,maxsplit=-1)	以空白字符作为分隔符对字符串进行分割； sep 参数指定分隔符，默认是空白字符； maxsplit 参数设置最大分割次数，默认是不限制
splitlines([keepends])	以换行符作为分隔符对字符串进行分割； keepends 参数设置最大分割次数
startswith(prefix[,start[,end]])	检查字符串是否以 prefix 参数开头，如果是则返回 True，否则返回 False； start 和 end 参数可选，表示范围
strip([chars])	删除字符串前边和后边所有空白字符； chars 参数可选，指定待删除的字符集
swapcase()	将字符串中所有的大写字母修改为小写，将小写字母修改为大写
title()	以标题化（所有的单词均以大写字母开始，其余字母皆小写）的形式格式化字符串
translate(table)	根据 table 的规则（可以由 str.maketrans('a','b')定制）转换字符串中的字符
upper()	将字符串的所有小写字母修改为大写字母
zfill(width)	当字符个数大于 width 时，字符串不变； 当字符个数小于 width 时，返回长度为 width 的字符串，原字符串右对齐，前边用 0 进行填充

这里选几个常用的字符串方法给大家演示一下用法，其他的可以根据上述文档的注释依葫芦画瓢。

casefold()方法用于将字符串中所有的英文字母修改为小写：

```
>>> str1 = "FishC"
>>> str1.casefold()
'fishc'
```

提示：

只要涉及字符串修改的方法，并不是修改原字符串，而是返回字符串修改后的一个拷贝。

count(sub[,start[,end]])方法用于查找 sub 参数在字符串中出现的次数，可选参数 start 和 end 表示查找的范围：

```
>>> str2 = "上海自来水来自海上"
>>> str2.count('上')
2
>>> str2.count('上', 0, 5)
1
```

find(sub[,start[,end]])或 index(sub[,start[,end]])方法用于查找 sub 参数在字符串中第一次出现的位置，如果找到了，返回位置索引值；如果找不到，find()方法会返回–1，而 index()方法会抛出异常（注：异常是可以被捕获并处理的错误）：

```
>>> str3 = "床上女子叫子女上床"
>>> str3.find("女子")
2
>>> str3.index("男子")
Traceback (most recent call last):
  File "<pyshell#53>", line 1, in <module>
    str3.index("男子")
ValueError: substring not found
```

replace(old,new[,count])方法用于将字符串中的 old 参数指定的字符串替换成 new 参数指定的字符串：

```
>>> str4 = "I love you."
>>> str4.replace("you", "fishc.com")
'I love fishc.com.'
```

split(sep=None, maxsplit=-1)方法用于拆分字符串：

```
>>> str5 = "肖申克的救赎/1994 年/9.6 分/美国"
>>> str5.split(sep='/')
['肖申克的救赎', '1994 年', '9.6 分', '美国']
```

和 split()方法相反，join(iterable)方法用于拼接字符串：

```
>>> countries = ['中国', '俄罗斯', '美国', '日本', '韩国']
>>> '-'.join(countries)
'中国-俄罗斯-美国-日本-韩国'
>>> ','.join(countries)
'中国,俄罗斯,美国,日本,韩国'
>>> ''.join(countries)
'中国俄罗斯美国日本韩国'
```

这种语法看上去可能会比较奇怪，很多读者可能会觉得被拼接的对象应该放在 join()方法的左侧更合适（如写成这样 countries.join('-')）？

但是因为 join()被指定为字符串的其中一个方法，所以只能这么写。另外还有一个重要的原因是，join()的参数支持一切可迭代对象（如列表、元组、字典、文件、集合或生成器等），如果将它们写在左侧，那就必须为这些对象都创建一个 join()方法，显然这样做是没有必要的。

其实，Python 程序员更喜欢使用 join()方法代替加号（+）来拼接字符串，这是因为使用加号（+）去拼接大量的字符串，效率相对会比较低，这种操作会频繁进行内存复制和触发垃圾回收机制。


视频讲解

5.3.2 格式化

什么是字符串的格式化，又为什么需要对字符串进行格式化？讲个小故事给大家听：某天小甲鱼心血来潮，试图召开一个“鱼 C 跨物种互联交流大会”，到会的朋友有来自各个物种的精英人士，有小乌龟、喵星人、汪星人，当然还有米奇和唐老鸭，那气势简直跟小甲鱼开了个动物园一样……但是问题来了，大家交流起来简直是鸡同鸭讲，不知所云！不过最后聪明的小甲鱼还是把问题给解决了，其实也很简单，各界都找一个翻译就行了，统一将发言都翻译成普通话，那么问题就解决了……最后我们这个大会当然取得了成功并被载入了“吉尼斯世界动物大全”。

好吧，举这个例子其实就是想跟大家说，格式化字符串，就是按照统一的规格去输出一个字符串。如果规格不统一，就很可能造成误会，例如，十六进制的 10 跟十进制的 10 或二进制的 10 完全是不同的概念（十六进制的 10 等于十进制的 16，二进制的 10 却等于十进制的 2）。字符串格式化，正是帮助我们纠正并规范这类问题而存在的。

1. format()

format()方法接收位置参数和关键字参数（位置参数和关键字参数在第 6 章中有详细讲解），二者均传递到一个名为 replacement 的字段。而这个 replacement 字段在字符串内用大括号（{}）表示。先看一个例子：

```
>>> "{0} love {1}.{2}".format("I", "FishC", "com")
'I love FishC.com'
```

怎么回事呢？仔细看一下，字符串中的{0}、{1}和{2}应该与位置有关，依次被 format()的三个参数替换，那么 format()的三个参数就称为位置参数。那什么是关键字参

数呢？再来看一个例子：

```
>>> "{a} love {b}.{c}".format(a="I", b="FishC", c="com")
'I love FishC.com'
```

{a}、{b}和{c}就相当于三个目标标签，format()将参数中等值的字符串替换进去，这就是关键字参数。另外，也可以综合位置参数和关键字参数在一起使用：

```
>>> "{0} love {b}.{c}".format("I", b="FishC", c="com")
'I love FishC.com'
```

但要注意的是，如果将位置参数和关键字参数综合在一起使用，那么位置参数必须在关键字参数之前，否则就会出错：

```
>>> "{a} love {b}.{0}".format(a="I", b="FishC", "com")
SyntaxError: non-keyword arg after keyword arg
```

如果要把大括号打印出来，有办法吗？没错，这与字符串转义字符有点像，只需要用多一层大括号包起来即可：

```
>>> "{{0}}".format("不打印")
'{0}'
```

位置参数“不打印”没有被输出，这是因为{0}的特殊功能被外层的大括号（{}）所剥夺，因此没有字段可以输出。注意，这并不会产生错误哦。最后来看另一个例子：

```
>>> "{0}：{1:.2f}".format("圆周率", 3.14159)
'圆周率：3.14'
```

可以看到，位置参数{1}跟平常有些不同，后边多了个冒号。在替换域中，冒号表示格式化符号的开始，“.2”的意思是四舍五入到保留两位小数点，而 f 的意思是浮点数，所以按照格式化符号的要求打印出了 3.14。

2．格式化操作符：%

刚才讲的是字符串的格式化方法，现在来谈谈字符串所独享的一个操作符：%。有人说，这不是求余数的操作符吗？是的，没错。当%的左右均为数字的时候，它表示求余数的操作；但当它出现在字符中的时候，它表示的是格式化操作符。表 5-2 列举了 Python 的格式化符号及含义。

表 5-2　Python 格式化符号及含义

符　　号	含　　义
%c	格式化字符及其 ASCII 码
%s	格式化字符串
%d	格式化整数
%o	格式化无符号八进制数
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写）

续表

符　号	含　义
%f	格式化浮点数字，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e
%g	根据值的大小决定使用%f 或%e
%G	作用同%g

下面给大家举几个例子供参考：

```
>>> '%c' % 97
'a'
>>> '%c%c%c%c%c' % (70, 105, 115, 104, 67)
'FishC'
>>> '%d 转换为八进制是：%o' % (123, 123)
'123 转换为八进制是：173'
>>> '%f 用科学计数法表示为：%e' % (149500000, 149500000)
'149500000.000000 用科学计数法表示为：1.495000e+08'
```

所以，使用格式化的方法也可以对字符串进行拼接：

```
>>> str1 = "一支穿云箭，千军万马来相见："
>>> str2 = "两副忠义胆，刀山火海提命现。"
>>> "%s%s" % (str1, str2)
'一支穿云箭，千军万马来相见：两副忠义胆，刀山火海提命现。'
```

那么结合前面提到的两种方法，现在共有三种方法可以对字符串进行拼接了。什么时候用哪种方法，根据不同情况，可以参考下面三条准则进行选择：

- 简单字符串连接时，直接使用加号（+），例如：full_name = prefix + name。
- 复杂的，尤其有格式化需求时，使用格式化操作符（%）进行格式化连接，例如：result = "result is %s:%d" % (name, score)。
- 当有大量字符串拼接，尤其发生在循环体内部时，使用字符串的 join()方法无疑是最棒的，例如：result = "".join(iterator)。

另外，Python 还提供了格式化操作符的辅助指令，如表 5-3 所示。

表 5-3　格式化操作符的辅助命令

符　号	含　义
m.n	m 显示的是最小总宽度，n 是小数点后的位数
-	结果左对齐
+	在正数前面显示加号（+）
#	在八进制数前面显示'0o'，在十六进制数前面显示'0x'或'0X'
0	显示的数字前面填充'0'代替空格

同样给大家举几个例子供参考：

```
>>> '%5.1f' % 27.658
' 27.7'
```

```
>>> '%.2e' % 27.658
'2.77e+01'
>>> '%10d' % 5
'         5'
>>> '%-10d' % 5
'5         '
>>> '%010d' % 5
'0000000005'
>>> '%#X' % 100
'0X64'
```

3．Python 的转义字符及含义

Python 的部分转义字符已经使用了一段时间，是时候来总结一下了，如表 5-4 所示。

表 5-4　转义字符及含义

符　号	说　明	符　号	说　明
\'	单引号	\r	回车符
\"	双引号	\f	换页符
\a	发出系统响铃声	\o	八进制数代表的字符
\b	退格符	\x	十六进制数代表的字符
\n	换行符	\0	表示一个空字符
\t	横向制表符（TAB）	\\	反斜杠
\v	纵向制表符		

5.4　序列

视频讲解

聪明的你可能已经发现，小甲鱼把列表、元组和字符串放在一块儿来讲解是有道理的，因为它们之间有很多共同点：

- 都可以通过索引得到每一个元素。
- 默认索引值总是从 0 开始（当然灵活的 Python 还支持负数索引）。
- 可以通过切片的方法得到一个范围内的元素的集合。
- 有很多共同的操作符（重复操作符、拼接操作符、成员关系操作符）。

我们把它们统称为：序列！下面介绍一些关于序列的常用 BIF（内建方法）。

1．list([iterable])

list()方法用于把一个可迭代对象转换为列表，很多读者可能经常听到“迭代”这个词，但要是让你解释的时候，却又可能会含糊其词了：迭代……迭代不就是 for 循环嘛……

这里小甲鱼帮大家科普一下：所谓迭代，是重复反馈过程的活动，其目的通常是为了接近并达到所需的目标或结果。每一次对过程的重复被称为一次“迭代”，而每一次迭代得到的结果会被用来作为下一次迭代的初始值……就目前来说，迭代还真的就是一个 for 循环，但今后会介绍到迭代器，那个功能，才叫惊艳！

好了，这里说 list()方法要么不带参数，要么带一个可迭代对象作为参数，而这个序列天生就是可迭代对象（迭代这个概念实际上就是从序列中泛化而来的）。

下面仍然通过几个例子给大家讲解一下：

```
>>> # 创建一个空列表
>>> a = list()
>>> a
[]
>>> # 将字符串的每个字符迭代存放到列表中
>>> b = list("FishC")
>>> b
['F', 'i', 's', 'h', 'C']
>>> # 将元组中的每个元素迭代存放到列表中
>>> c = list((1, 1, 2, 3, 5, 8, 13))
>>> c
[1, 1, 2, 3, 5, 8, 13]
```

事实上这个 list()方法大家自己也可以动手实现，对不对？很简单嘛，实现过程大概就是新建一个列表，然后循环通过索引迭代参数的每一个元素并加入列表，迭代完毕后返回列表即可。大家不妨自己动手来尝试一下。

2．tuple([iterable])

tuple()方法用于把一个可迭代对象转换为元组，具体的用法和 list()一样，这里就不再赘述了。

3．str(obj)

str()方法用于把 obj 对象转换为字符串，这个方法 3.9.4 节中讲过，还记得吧？

4．len(sub)

len()方法前面已经使用过几次了，该方法用于返回 sub 参数的长度：

```
>>> str1 = "I love fishc.com"
>>> len(str1)
16
>>> list1 = [1, 1, 2, 3, 5, 8, 13]
>>> len(list1)
7
>>> tuple1 = "这", "是", "一", "个", "元祖"
>>> len(tuple1)
5
```

5．max()

max()方法用于返回序列或者参数集合中的最大值，也就是说，max()的参数可以是一个序列，返回值是该序列中的最大值；也可以是多个参数，那么，max()将返回这些参

数中最大的一个：

```
>>> list1 = [1, 18, 13, 0, -98, 34, 54, 76, 32]
>>> max(list1)
76
>>> str1 = "I love fishc.com"
>>> max(str1)
'v'
>>> max(5, 8, 1, 13, 5, 29, 10, 7)
29
```

6．min()

min()方法跟 max()用法一样，但效果相反：返回序列或者参数集合中的最小值。这里需要注意的是，使用 max()方法和 min()方法都要保证序列或者参数的数据类型统一，否则会出错：

```
>>> list1 = [1, 18, 13, 0, -98, 34, 54, 76, 32]
>>> list1.append("x")
>>> max(list1)
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    max(list1)
TypeError: '>' not supported between instances of 'str' and 'int'
>>> min(123, 'oo', 456, 'xx')
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    min(123, 'oo', 456, 'xx')
TypeError: '<' not supported between instances of 'str' and 'int'
```

俗话说：外行看热闹，内行看门道。

不妨分析一下这个报错信息“TypeError: '<' not supported between instances of 'str' and 'int'”，意思是说不能拿字符串和整型进行比较。这说明了什么呢？说明 max()方法和 min()方法的内部实现事实上类似于之前提到的，通过索引得到每一个元素，然后将各个元素进行对比。

所以，根据上述猜想，可以写出类似的实现代码：

```
# 猜想下 max(tuple1)的实现方式
temp = tuple1[0]

for each in tuple1:
    if each > temp:
        temp = each

return temp
```

由此可见，Python 的内置方法其实也没什么了不起的，仔细思考一下也是可以独立

实现的嘛。所以，只要认真地跟着本书的内容学习下去，很多看似“如狼似虎”的问题，将来都能迎刃而解！

7．sum(iterable[, start])

sum()方法用于返回序列 iterable 的所有元素值的总和，用法跟 max()和 min()一样。但 sum()方法有一个可选参数(start)，如果设置该参数，表示从该值开始加起，默认值是 0：

```
>>> tuple1 = 1, 2, 3, 4, 5
>>> sum(tuple1)
15
>>> sum(tuple1, 10)
25
```

8．sorted(iterable, key=None, reverse=False)

sorted()方法用于返回一个排序的列表，大家还记得列表的内建方法 sort()吗？它们的实现效果一致，但列表的内建方法 sort()是实现列表原地排序；而 sorted()是返回一个排序后的新列表。

```
>>> list1 = [1, 18, 13, 0, -98, 34, 54, 76, 32]
>>> list2 = list1[:]
>>> list1.sort()
>>> list1
[-98, 0, 1, 13, 18, 32, 34, 54, 76]
>>> sorted(list2)
[-98, 0, 1, 13, 18, 32, 34, 54, 76]
>>> list2
[1, 18, 13, 0, -98, 34, 54, 76, 32]
```

9．reversed(sequence)

reversed()方法用于返回逆向迭代序列的值。同样的道理，实现效果跟列表的内建方法 reverse()一致。区别是：列表的内建方法是原地翻转，而 reversed()是返回一个翻转后的迭代器对象。你没看错，它不是返回一个列表，而是返回一个迭代器对象。

```
>>> list1 = [1, 18, 13, 0, -98, 34, 54, 76, 32]
>>> reversed(list1)
<list_reverseiterator object at 0x000000000324F518>
>>> for each in reversed(list1):
    print(each, end=',')

32,76,54,34,-98,0,13,18,1,
```

10．enumerate(iterable)

enumerate()方法生成由二元组（二元组就是元素数量为 2 的元组）构成的一个迭代

对象，每个二元组由可迭代参数的索引号及其对应的元素组成，举个例子：

```
>>> str1 = "FishC"
>>> for each in enumerate(str1):
    print(each)

(0, 'F')
(1, 'i')
(2, 's')
(3, 'h')
(4, 'C')
```

11．zip(iter1 [,iter2 [...]])

zip()方法用于返回由各个可迭代参数共同组成的元组，举个例子：

```
>>> list1 = [1, 3, 5, 7, 9]
>>> str1 = "FishC"
>>> for each in zip(list1, str1):
    print(each)

 (1, 'F')
(3, 'i')
(5, 's')
(7, 'h')
(9, 'C')
>>> tuple1 = (2, 4, 6, 8, 10)
>>> for each in zip(list1, str1, tuple1):
    print(each)

 (1, 'F', 2)
(3, 'i', 4)
(5, 's', 6)
(7, 'h', 8)
(9, 'C', 10)
```