

集合与循环

BBCLOUD 林凡

https://github.com/lvancer/course_python

lin029011@163.com



大纲

集合类型

列表

元组

字典

集合

循环

练习



集合类型

基础语法中介绍的几种python的基础类型都属于数值类型。

另外一种基础类型是集合类型，主要包括列表、元组、字典、集合。

其中列表和元组是一种序列。序列是一种将一串元素按顺序组合在一起的数据结构。

序列的特点是拥有索引操作与切片操作。

索引操作是从序列中通过编号获得其中的一个元素，语法是中括号 [编号]。

切片操作是获取一部分序列，语法为 [编号1:编号2]。

回忆字符串，是不是也有相同的操作，获取其中某个或某段字符。

所以字符串其实也是一种序列，其元素就是单个字符。

```
print(s[1])  
print(s[3:5])  
print(s[2:])  
print(s[:4])
```

lin029011@163.com

列表

列表是一种有序的序列，其元素可以是任意类型，编号从0开始到长度-1结束。

创建一个列表：Python中使用中括号[]来表示列表。注意与索引切片区分。

```
name_list = ['Green', 'Lucy', 'Lily']    # 字符串列表
score_list = [98, 67, 80]                # 整数列表
x_list = ['hello', 98, 2.15, True]       # 任意类型列表
y_list = [score_list, name_list]         # 列表的列表
```

获取列表长度：len。

```
print(len(name_list))    # 列表长度
```

索引与切片：

```
print(score_list[1])    # 索引，返回元素
print(x_list[0:1])      # 切片，返回列表
```

输出：

```
3
67
['hello']
```

lin029011@163.com

列表

如何理解列表的列表？

```
y_list = [score_list, name_list]
```

列表中可以放任意类型的数据，放一个序列类型肯定是没有问题的。

专业的说法是**二维列表**，当然还可以继续嵌套形成**多维列表**。

访问方式：

```
print(y_list[1][2])
```

 # 访问多维列表

通过第一个[]获得第一维的列表，再用[]就可以得到第二维的数据了。

等价于

```
_name_list = y_list[1]  
print(_name_list[2])
```

修改元素：直接通过赋值就可以修改其中的元素。

```
name_list[2] = 'Kate'
```

列表

逻辑判断：`in`，`not in` 判断列表中是否存在某个元素。

```
if 'Green' in name_list:  
    print('Green is here')
```

```
if 'Lucy' not in name_list:  
    print('Lucy is not here')
```

加法与乘法：

加号（`+`）会将两个列表组合起来产生一个新列表。

乘号（`*`）则会将原列表进行复制，产生新列表。

```
# 加法  
x = [1, 3, 2]  
y = [4, 6, 5]  
z = x + y  
print(x + y)
```

输出：

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

```
# 乘法  
x = [1, 2, 3]  
z = x * 3  
print(z)
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

列表

最大值与最小值：`max`、`min`。

```
x = [1, 2, 3]
print(max(x))
print(min(x))
```

输出：

```
3
1
```

如果列表内的元素必须是可比较的。试试字符串列表能不能用？

排序：`sort`。将列表内的元素从小到大进行排序。

```
# 排序
x = [1, 2, 4, 5, 3, 2]
x.sort()
print(x)
```

输出：

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

翻转列表：`reverse`。把列表进行翻转，后面到前面。

```
x.reverse()
```

将上面排序好的进行翻转就得到

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

列表

索引：**index**，获得某个元素在列表中**第一次**出现的**位置**。

```
x = [1, 2, 3, 2]
i = x.index(2)
print(i)
```

2第一次出现在位置1，所以输出 1。
当元素**不存在**时，该方法会**报错**。

```
if 9 in x:
    i = x.index(9)
    print(i)
```

可以先判断是否存在，再做索引。
或者使用后面介绍的**try**、**except**来处理。

统计元素个数：**count**。

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

x列表中元素2出现了2次，所有该方法返回2。

列表

尾部添加：`append`、`extend`。

两个都是在尾部添加，区别在于`append`添加的是元素，`extend`添加的是列表。

```
x = [1, 2, 3]
x.append(100)
print(x)
x.extend([200, 300])
print(x)
```

输出：

```
[1, 2, 3, 100]
[1, 2, 3, 100, 100, 200, 300]
```

`extend`的效果相当于 `x = x + [200, 300]`

插入元素：`insert`。在列表的某个位置插入某个元素，`insert(位置，元素)`

```
x = [1, 2, 3]
x.insert(0, 0)
x.insert(2, 100)
print(x)
```

输出：

```
[0, 1, 100, 2, 3]
```

x列表先在位置0（即列表开头）插入了0，
然后又在位置2插入了100。

列表

删除：`remove`，`del`，`pop`，`clear`。

`remove` 删除第一个出现的该元素，如果不存在时与`index`一样会报错。

`del` 则是根据索引位置进行删除，当索引超出范围时会报错。

`pop` 会删除最后一个元素，并将其返回。

`clear` 删除所有列表中元素。

```
x = [1, 2, 3, 2]
x.remove(2)      # 删除第一个2元素
del x[1]         # 删除位置1的元素
p = x.pop()      # 删除最后一个2，同时p被赋值2
x.clear()        # 删除所有元素
```

元组

元组与列表是一个的区别就是元组是一种不可变序列，其他都与列表一样。

元组拥有序列的通用操作，但会改变序列的方法都不存在。

创建一个元组：Python中使用括号()来表示元组。单只有一个元素时，要加一个逗号。

```
a = (1, )      # 只有一个元素的元组
b = (1, 3, 4)  # 元组
```

序列通用操作：

```
print(len(a))    # 长度
print(b[1])      # 索引
print(b[0:2])    # 切片
print(max(b))    # 最大值
print(min(b))    # 最小值
print(2 in b)    # in
```

输出：

```
1
3
(1, 3)
4
1
False
```

元组主要用来实现方法的多返回值。
在实际编程中使用的并不多。

元组

元组与列表的转换：`list`、`tuple`。

```
tuple([1, 2, 3])    # 转换为元组  
list((1, 2, 3))     # 转换为列表
```

拆包：拆包是将元组或列表中的**所有元素**逐个**赋值**给**多个变量**。

```
x = ('a', 'b', 'c')  输出：  
a, b, c = x  
print(a, b, c)
```

a, b, c三个变量用**逗号**隔开，并按顺序被赋值上x元组中的三个值。

变量的数量必须与元组的**长度一样**。

字典

字典是一种**无序**的结构，同样可以放入**任意类型**元素。

与**列表**使用0，1，2...进行**有序**索引不同，**字典**使用**自定义**的**关键字**进行**索引**。

key-value结构：**key**代表用来索引的**关键字**，value代表存储的**元素**。【**键值对**】

创建一个字典：使用**大括号**{ }包裹字典，每个**键值对**用**逗号**隔开，**键值对**中用**冒号**分隔

```
user = {  
    'name': 'Green',  
    'age': 12,  
    'gender': 'male'  
}
```

key与value，前面的是key，后面的是value。写成一行

```
user = {'Green': 80, 'Lily': 90, 'Lucy': 76}
```

也是可以的，但大部分情况分行书写会比较清晰。

如何取值：同样是通过 **[]** 来进行，只不过将列表里的编号换成**key**值就可以取到value。

```
print(user['name'])
```

当key**不存在**时，会**报错**。

字典

安全取值：`get`，如果key不存在时，使用`get`方法可以避免报错，并返回一个空值`None`。

```
address = user.get('address') # 返回None，因为没有address这个key  
name = user.get('name') # 返回Green
```

添加与修改：直接赋值。

```
user['address'] = 'china' # 添加一个address键值对  
user['age'] = 18 # age被修改为18
```

删除：`del`。

```
del user['gender'] # 删除了gender这个键值对
```

清空：`clear`。

```
user.clear() # 清空了所有键值对
```

字典

判断存在key : **in** 。

```
if 'age' in user:           # 判断age是否在user里面
    print(user['age'])
```

获得字典的所有内容：**keys**、**values**、**items** 。

```
print(user.keys())         # 获得所有key值
print(user.values())       # 获得所有value值
print(user.items())        # 获得所有键值对
```

输出：

```
dict_keys(['name', 'age', 'address'])
dict_values(['Green', 18, 'china'])
dict_items([('name', 'Green'), ('age', 18), ('address', 'china')])
```

输出的值并不是列表，可以用
list方法转换为列表，方便使用。

字典

复杂结构：通过字典和列表的互相嵌套，就能实现实际开发中常用的结构。

```
users = [  
    {'name': 'Green', 'age': 12, 'like': ['apple']},  
    {'name': 'Lucy', 'age': 13, 'like': ['apple', 'pear']},  
    {'name': 'Lily', 'age': 11, 'like': ['orange', 'melon']}  
]
```

获得某个用户的第一个喜好。

```
print(users[1]['like'][0])
```

通过多个 `[]` 的提取，最后就提取到了用户1【Lucy】的第一个喜好。

集合

集合是一组key组成的数据，与字典的区别在于没有value，

与列表的区别在于每个元素key不可重复。

在数学上可以进行交集、并集、差集等运算，是一种在特定场合很方便的数据结构。

创建集合：与字典一样，使用大括号{}表示集合，但不需要有value。

```
set1 = {'apple', 'orange', 'melon'}  
set2 = {'apple', 'pear'}
```

列表去重：使用set方法将列表转换为集合，可以很方便的列表去重。

```
print(set([1, 3, 3, 4]))    # 列表去重  
print(set('class'))        # 字符串也是一种序列
```

```
{1, 3, 4}  
{ 'c', 'l', 's', 'a' }
```

输出的结果自动删除了重复的值，然后还可以再用list转换为列表，就是去重后的列表了。

lin029011@163.com

集合

交集：set1 & set2，计算两个集合中相同的元素。

并集：set1 | set2，计算两个集合所有的不重复元素。

差集：set1 - set2，计算set1中有，但set2中没有的元素。

对称差集：set1 ^ set2，计算只存在于set1或set2中的元素。

```
set1 = {'apple', 'orange', 'melon'}
set2 = {'apple', 'pear'}
print(set1 & set2)    # 交集
print(set1 | set2)    # 并集
print(set1 - set2)    # 差集
print(set1 ^ set2)    # 对称差集
```

输出：

```
{'apple'}
{'pear', 'melon', 'apple', 'orange'}
{'orange', 'melon'}
{'pear', 'orange', 'melon'}
```

集合

添加元素：`add`。

```
s = {'apple', 'orange'}  
s.add('pear')
```

删除元素：`remove`、`discard`、`pop`、`clear`。

```
s.remove('apple')      # 如果不存在该元素，会报错。  
s.discard('apple1')    # 更安全的删除，即使不存在也不报错  
x = s.pop()            # 随机删除集合中的一个元素，并返回它  
s.clear()              # 删除集合中的所有元素
```

子集与超集的判断：`issubset`、`issuperset`。

```
set1 = {'a', 'b', 'c'}  
set2 = {'a'}          # set2是set1的子集，set1是set2的超集  
set2.issubset(set1)    # 返回True  
set1.issuperset(set2)  # 返回True
```

循环

循环语句用于处理具有重复性的工作，经常用于处理列表、字典等结构。

如一个列表中有几十万个元素，这时候就需要用循环来遍历所有元素。

遍历是一种从集合中逐个获取元素的操作。

for 循环：for、in。

```
names = ['Green', 'Lucy', 'Lily']  
for name in names:  
    print('Happy Birthday, {}'.format(name))
```

```
Happy Birthday, Green  
Happy Birthday, Lucy  
Happy Birthday, Lily
```

for打头开始一个循环，in后面的names表示要遍历的列表。

for后面的name是一个临时变量，用来存储遍历过程中的每个元素。

从0到N-1的元素按顺序依次赋值给临时变量，并重复执行下面的代码块N次。

(N等于列表的长度)

循环

编写一个程序，计算列表中所有元素的总和。

```
numbers = [10, 53, 45, 44, 907] # 要计算的列表
answer = 0 # 总和初始化为0
for i in numbers: # 开始循环
    answer = answer + i # 逐个相加
print(answer)
```

每次循环 **i** 都会被**赋值**为numbers里的一个值。

将当前的answer加上 i，并将结果赋值给answer。

每次循环都更新了answer，当**循环结束**时就是列表的总和了。

sum方法，同样也可以完成这个功能，可以将结果进行对比。

```
sum(numbers)
```

 这里我们初步了解到了方法到底在做什么。

循环

停止循环：**break**。

在for循环代码块中出现**break**，就会停止**这个for循环**。

一般我们会与**条件判断**一起使用，实现**满足条件**后停止循环。

下面我们一起来实现一个**index**功能。【返回第一次出现的位置，不存在时返回-1】

```
numbers = [10, 53, 45, 44, 907]
i = 0           # 记录当前遍历到的位置
index = -1      # 结果默认为-1不存在
for number in numbers:
    if number == 53:  # 满足条件
        index = i    # 赋值位置
        break        # 停止循环
    i = i + 1        # 位置+1
print(index)
```

这里的i记录了当前遍历到的位置，所以在每次循环执行的最后要加1。
如果你打印出i的值，会发现i停止在了满足条件的位置，即循环停止了。

lin029011@163.com

循环

中断循环：`continue`。

在循环中出现`continue`，就会停止该次循环，马上进行下一轮循环。

下面我们还是实现一个求和的功能，但如果数字大于100时不计算在内。

```
numbers = [10, 53, 45, 44, 907]
answer = 0
for i in numbers:
    if i > 100:
        continue # 马上开始下一个循环
    answer = answer + i
print(answer)
```

当大于100时，就跳过，
不进行相加

```
if i <= 100:
    answer = answer + i
```

也许你已经发现其实不用`continue`也可以实现这个功能。

所以`continue`的出镜率并不高，一般在逻辑异常复杂时非常好用。

循环

while循环：**while**。

while循环是根据**条件判断**来是否**继续循环**的。

```
names = ['Green', 'Lucy', 'Lily']
i = 0    # 索引位置从0开始
while i < len(names):    # 条件为True时，继续循环
    name = names[i]      # 通过索引获得元素
    print('Happy Birthday, {}'.format(name))
    i = i + 1            # 索引+1
```

while后面的**条件语句**就是要判断的内容，为**True**时循环**继续**，为**False**时循环**停止**。

这是我们写的第一个for循环的while版本。条件为索引值小于列表长度。

while同样支持**break**和**continue**。

while和for循环是基本等价的不同写法，根据需要选择更方便的写法。

```
# 无限循环
while True:
    print(1)
```

lin029011@163.com

循环

range：生成一个整数序列。在for循环中常用。

```
for i in range(10):    # 0到9的序列
    print(i)
range(3, 10)          # 3到9的序列
range(3, 10, 2)       # 3,5,7,9 步长为2的序列
```

range可以生成从**a**到**b**，且步长为**n**的整数序列。

for循环的range版本：

```
names = ['Green', 'Lucy', 'Lily']
for i in range(len(names)):
    name = names[i]
    print('Happy Birthday, {}'.format(name))
```

这里使用**len(names)**来生成**全部的索引**，经常处理列表时有需要索引值时可以这样做。

lin029011@163.com

循环

循环的嵌套：

```
for i in range(1, 10):  
    for j in range(1, 10):  
        print('{} * {} = {}'.format(i, j, i*j))
```

九九乘法表的生成。

上面的乘法表有个问题，就是生成了1*2和2*1两个，这时我们通过加入`continue`解决。

```
for i in range(1, 10):  
    for j in range(1, 10):  
        if j < i:  
            continue  
        print('{} * {} = {}'.format(i, j, i*j))
```

`break`和`continue`在循环中只作用于它所处的循环，这个要特别注意。

循环

字典的循环：字典类型同样可以用循环来处理。

1、通过`keys`循环遍历所有`key`。

```
user = {'name': 'Green', 'age': 12, 'address': 'china'}  
for key in user.keys():  
    print('{} : {}'.format(key, user[key]))
```

2、通过`items`循环遍历所有`键值对`。

```
for key, value in user.items():  
    print('{} : {}'.format(key, value))
```

与之前的for循环不一样的地方就在于`items`方法返回的列表中元素是一个`元组【键值对】`。

所以在接收时就要用`key`和`value`两个变量去将其`拆包`，这样就可以同时遍历到`key`和`value`。

练习

编写一个获得列表中最大值的程序。

编写一个获得列表中最小值的程序。

编写一个计算列表所有值的平均值的程序。

