目录

17 生成器表达式和列表生成式

更新时间：2019-09-25 10:03:10

> 一个不注意小事情的人，永远不会成功大事业。
>
> ——戴尔·卡耐基

列表生成式

如果我们想要构造一个包含指定元素或者具有某种规则的列表，比如 2 的指数幂序列 [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]， 该怎么做？

- 最简单的办法，直接将这些数原样写入代码来创建列表：

```
nums = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

这种办法当然是可行的，不过也有很大的局限。像这样直接将数据写入代码的做法，叫做硬编码。很多情况下我们不会使用硬编码的方式来创建列表（或者其它容器），因为列表中有什么数据往往在写代码时是不能确定的，通常在程序运行过程中通过计算得到，或从程序外部读入（比如从数据库 / 文件 / 网络中读入）。另外当数据量很大时，使用硬编码也是一件繁琐低效的事。

- 还有一种办法，创建一个空的列表，之后通过计算（或其它操作）获得各个元素，并添加到列表中：

```
nums = []
exponent = 1
while exponent <= 10:
    nums.append(2 ** exponent)
    exponent += 1
```

> [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]

这样方式完全可行，也没有什么缺陷。

- 或者，对一个现有的可迭代对象中的各个元素做处理，构造出一个新的列表：

```
nums = []
for i in range(1, 11):          # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    nums.append(2 ** i)
```

> >>> nums
> [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]

这段代码可以用一种简洁的方式写出，只需要一行代码：

```
nums = [2 ** i for i in range(1, 11)]
```

> >>> nums
> [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]

这行代码就是我们这个章节要所讲的**列表生成式**。顾名思义，列表生成式最终生成的是一个列表，它是用已有的可迭代对象来构造新列表的便捷方法。

> 提示：如果不清楚什么是可迭代对象，可以看一下上一篇文章《深入理解迭代器和生成器》。

列表生成式的写法

列表生成式的语法如下：

```
[对项的操作 for 项 in 可迭代对象]
```

这个写法怎么理解呢？

首先，这句代码的阅读顺序是： for 项 in 可迭代对象 -> 对项的操作 。其次，外围的方括号（ [] ）表明这是列表生成式，最终的结果是一个列表。

for 项 in 可迭代对象 这部分和 for 循环很相似，通过迭代可迭代对象，每次取出一个项。对于取出的项，我们可以对它做一些处理，也就是表达式中的 对项的操作 部分。最终，可迭代对象中的所有项都会被迭代和处理，并被收集起来形成一个新的列表。

这个过程用伪代码来描述的话是这样的：

```
for 项 in 可迭代对象:
    新项 = 对项的操作(项)
  列表.append(新项)
```

来看一个例子：

这里有个列表：['a', 'b', 'c', 'd', 'e']，怎样把其中的每个小写字母转换为大写？可以这样：

```
[char.upper() for char in ['a', 'b', 'c', 'd', 'e']]
```

```
>>> [char.upper() for char in [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]]
[‘A’, ‘B’, ‘C’, ‘D’, ‘E’]
```

如果你不能一下子理解，不妨比较一下用 for 循环来实现的版本。它们之间是等价的：

```
chars = []
for char in ['a', 'b', 'c', 'd', 'e']:
    chars.append(char.upper())
```

```
>>> chars
[‘A’, ‘B’, ‘C’, ‘D’, ‘E’]
```

再来谈谈 [对项的操作 for 项 in 可迭代对象] 中的 对项的操作，这个操作它可以简单，也可以很复杂。

简单来看，我们可以直接使用 项 本身而不做任何处理。如：

```
>>> [char for char in ‘ABCDEF’]
[‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’]
```

当然如果是要得到这个结果，我们应该直接使用 list('ABCDEF')：

```
>>> list(‘ABCDEF’)
[‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’]
```

复杂来看，我们可以对 项 进行一系列的处理。如分别将 'abcde' 中每个字母的大写形式和小写形式放到元组中：

```
[(char.upper(), char) for char in 'abcde']
```

> [(‘A’ , ‘a’), (‘B’ , ‘b’), (‘C’ , ‘c’), (‘D’ , ‘d’), (‘E’ , ‘e’)]

这里我们将每个 char 转换为了 (char.upper(), char) ，并且其中 char 被多次用到。

上个例子等价于：

```
result = []
for char in 'abcde':
    result.append((char.upper(), char))
```

```
>>> result
[( ‘A’ , ‘a’ ), ( ‘B’ , ‘b’ ), ( ‘C’ , ‘c’ ), ( ‘D’ , ‘d’ ), ( ‘E’ , ‘e’ )]
```

列表生成式中使用 if

在列表生成式的中，每次迭代的 项 是可以被筛选过滤的，使用 if 关键字。如：

```
[对项的操作 for 项 in 可迭代对象 if 对项的判断]
```

它的阅读顺序是： for 项 in 可迭代对象 -> if 对项的判断 -> 对项的操作 。

每次迭代时所取出的 项 ，要先经过 对项的判断 ，如果结果为 True ，才会由 对项的操作 处理。如果 对项的判断 的结果为 False ，后续 对项的操作 会被跳过，此时最终列表的长度也会减少。

举个例子， [2 ** i for i in range(1, 11)] 可以生成出 20～210 间的整数，如果我们只想要其中的奇数次方的值，该怎么做？

这时就可以在列表中使用 if 关键字：

```
[2 ** i for i in range(1, 11) if i % 2 == 1 ]
```

```
>>> [2 ** i for i in range(1, 11) if i % 2 == 1 ]
[2, 8, 32, 128, 512]
```

这里的阅读顺序是：

1. for e in range(1, 11)
2. if e % 2 == 1
3. 2 ** e

上述代码等价于：

```
nums = []
for i in range(1, 11):
```

```
>>> nums
[2, 8, 32, 128, 512]
```

列表生成式中嵌套 for

列表生成式中的 `for` 中还可以再嵌套 `for` 。如：

```
[对项1和(或)项2的操作 for 项1 in 可迭代对象1 for 项2 in 可迭代对象2]
```

它等价于：

```
列表 = []
for 项1 in 可迭代对象1:
    for 项2 in 可迭代对象2:
        新项 = 对项1和(或)项2的操作()
        列表.append(新项)
```

看起来有点复杂，我们看个例子：

```
nums = [1, 2, 3]
chars = ['a', 'b', 'c']
[c * n for n in nums for c in chars]
```

```
>>> nums = [1, 2, 3]
>>> chars = [ ‘a’ , ‘b’ , ‘c’ ]
>>> [c * n for n in nums for c in chars]
[ ‘a’ , ‘b’ , ‘c’ , ‘aa’ , ‘bb’ , ‘cc’ , ‘aaa’ , ‘bbb’ , ‘ccc’ ]
```

它等价于：

```
nums = [1, 2, 3]
chars = ['a', 'b', 'c']
result = []
for n in nums:
    for c in chars:
        result.append(c * n)
```

```
>>> result
[ ‘a’ , ‘b’ , ‘c’ , ‘aa’ , ‘bb’ , ‘cc’ , ‘aaa’ , ‘bbb’ , ‘ccc’ ]
```

`[对项1和(或)项2的操作 for 项1 in 可迭代对象1 for 项2 in 可迭代对象2]` 中的 `可迭代对象2` 可以是 `项1` 本身。也就是可以写成：

```
[对项和(或)子项的操作 for 项 in 可迭代对象 for 子项 in 项]
```

```
strings = ['aa', 'bb', 'cc']
[char for string in strings for char in string]
```

> >>> strings = [‘aa’ , ‘bb’ , ‘cc’]
> >>> [char for string in strings for char in string]
> [‘a’ , ‘a’ , ‘b’ , ‘b’ , ‘c’ , ‘c’]

它等价于：

```
strings = ['aa', 'bb', 'cc']
result = []
for string in strings:
    for char in string:
        result.append(char)
```

> >>> result
> [‘a’ , ‘a’ , ‘b’ , ‘b’ , ‘c’ , ‘c’]

字典生成式

便捷地构造列表可以使用列表生成式，同样的，想要通过已有的可迭代对象来便捷地构造字典，可以使用**字典生成式**。

字典生成式的写法是：

```
{键: 值 for 项 in 可迭代对象}
```

和列表生成式非常相似，不同之处在于它使用的是花括号（ {} ），另外还使用 键: 值 形式。

举个例子，有字符串 'abcde' ，以每个小字母作为键，对应大写字母作为值的来构造个字典：

```
{char: char.upper() for char in 'abcde'}
```

> >>> {char: char.upper() for char in ‘abcde’ }
> { ‘a’ : ‘A’ , ‘b’ : ‘B’ , ‘c’ : ‘C’ , ‘d’ : ‘D’ , ‘e’ : ‘E’ }

同样的，字典生成式中也可以使用 if 和嵌套 for ，使用方法参照列表生成式。

集合生成式

想要通过已有的可迭代对象来构造集合，可以使用**集合生成式**。

```
{对项的操作 for 项 in 可迭代对象}
```

例如：

```
{char.lower() for char in 'ABCDABCD'}
```

```
>>> {char.lower() for char in ‘ABCDABCD’}
{‘c’, ‘a’, ‘d’, ‘b’}
```

> 提示：通过这个例子也能看到集合的重要特性——无序且无重复。

同样的，集合生成式中也可以使用 if 和嵌套 for 。

生成器表达式

上面有列表生成式、字典生成式、集合生成式，那么是不是也有「元组生成式」？是不是用圆括号来表示就可以了？

不是的，Python 中并没有「元组生成式」！虽然 Python 中确实有类似的圆括号的写法：

```
(对项的操作 for 项 in 可迭代对象)
```

但这可不是什么「元组生成式」，而是我们上一章节学习过的**生成器表达式**。

生成器表达式是一种创建生成器的便捷方法。虽然写法上和列表生成式、字典生成式、集合生成式相似，却有着本质的不同，因为它创建出来的是生成器，而不是列表、字典、集合这类容器。

```
(char.lower() for char in 'ABCDEF')
```

```
>>> g = (char.lower() for char in ‘ABCDEF’)
>>> g
<generator object at 0x103da6c78>
>>> next(g)
‘a’
>>> next(g)
‘b’
```

> 提示：如果你对生成器有些遗忘，不妨看下前一篇文章《深入理解迭代器和生成器》。

生成器表达式中同样可以使用 if 和嵌套 for ，使用方法和列表生成式相同。

目录

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论

千学不如一看，千看不如一练