**第三部分 高级特性**

##### 切片

取一个list或tuple的部分元素是非常常见的操作。比如，一个list如下：

L = ['Michael', 'Sarah', 'Tracy', 'Bob', 'Jack']

取前3个元素，应该怎么做？

[L[0], L[1], L[2]]

取前N个元素，也就是索引为0-(N-1)的元素，可以用循环：

r = []

n = 3

for i in range(n):

... r.append(L[i])

...

r

对这种经常取指定索引范围的操作，用循环十分繁琐，因此，Python提供了切片（Slice）操作符，能大大简化这种操作。

对应上面的问题，取前3个元素，用一行代码就可以完成切片：

L[0:3]

L[0:3]表示，从索引0开始取，直到索引3为止，但不包括索引3。即索引0，1，2，正好是3个元素。

如果第一个索引是0，还可以省略：

L[:3]

也可以从索引1开始，取出2个元素出来：

L[1:3]

类似的，既然Python支持L[-1]取倒数第一个元素，那么它同样支持倒数切片，试试：

L[-2:]

L[-2:-1]

记住倒数第一个元素的索引是-1。

切片操作十分有用。先创建一个0-99的数列：

L = list(range(100))

L

可以通过切片轻松取出某一段数列。比如前10个数：

L[:10]

后10个数：

L[-10:]

前11-20个数：

L[10:20]

前10个数，每两个取一个：

L[:10:2]

所有数，每5个取一个：

L[::5]

甚至什么都不写，只写[:]就可以原样复制一个list：

L[:]

tuple也是一种list，唯一区别是tuple不可变。因此，tuple也可以用切片操作，只是操作的结果仍是tuple：

(0, 1, 2, 3, 4, 5)[:3]

字符串'xxx'也可以看成是一种list，每个元素就是一个字符。因此，字符串也可以用切片操作，只是操作结果仍是字符串：

'ABCDEFG'[:3]

'ABCDEFG'[::2]

在很多编程语言中，针对字符串提供了很多各种截取函数（例如，substring），其实目的就是对字符串切片。Python没有针对字符串的截取函数，只需要切片一个操作就可以完成，非常简单。

##### 迭代

给定一个list或tuple，可以通过for循环来遍历这个list或tuple，这种遍历称为迭代（Iteration）。

在Python中，迭代是通过for ... in来完成的，而很多语言比如C语言，迭代list是通过下标完成的，比如Java代码：

for (i=0; i<list.length; i++) {

n = list[i];

}

可以看出，Python的for循环抽象程度要高于C的for循环，因为Python的for循环不仅可以用在list或tuple上，还可以作用在其他可迭代对象上。

list这种数据类型虽然有下标，但很多其他数据类型是没有下标的，但是，只要是可迭代对象，无论有无下标，都可以迭代，比如dict就可以迭代：

d = {'a': 1, 'b': 2, 'c': 3}

for key in d:

... print(key)

因为dict的存储不是按照list的方式顺序排列，所以，迭代出的结果顺序很可能不一样。

默认情况下，dict迭代的是key。如果要迭代value，可以用for value in d.values()，如果要同时迭代key和value，可以用for k, v in d.items()。

由于字符串也是可迭代对象，因此，也可以作用于for循环：

for ch in 'ABC':

... print(ch)

所以，当使用for循环时，只要作用于一个可迭代对象，for循环就可以正常运行，而不太关心该对象究竟是list还是其他数据类型。

那么，如何判断一个对象是可迭代对象呢？方法是通过collections模块的Iterable类型判断：

from collections import Iterable

isinstance('abc', Iterable) # str是否可迭代

isinstance([1,2,3], Iterable) # list是否可迭代

isinstance(123, Iterable) # 整数是否可迭代

如果要对list实现类似Java那样的下标循环怎么办？Python内置的enumerate函数可以把一个list变成索引-元素对，这样就可以在for循环中同时迭代索引和元素本身：

for i, value in enumerate(['A', 'B', 'C']):

... print(i, value)

上面的for循环里，同时引用了两个变量，在Python里是很常见的，比如下面的代码：

for x, y in [(1, 1), (2, 4), (3, 9)]:

... print(x, y)

##### 列表生成式

列表生成式即List Comprehensions，是Python内置的非常简单却强大的可以用来创建list的生成式。

举个例子，要生成list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]可以用list(range(1, 11))：

list(range(1, 11))

但如果要生成[1x1, 2x2, 3x3, ..., 10x10]怎么做？方法一是循环：

L = []

for x in range(1, 11):

... L.append(x \* x)

L

但是循环太繁琐，而列表生成式则可以用一行语句代替循环生成上面的list：

[x \* x for x in range(1, 11)]

写列表生成式时，把要生成的元素x \* x放到前面，后面跟for循环，就可以把list创建出来，十分有用，多写几次，很快就可以熟悉这种语法。

for循环后面还可以加上if判断，这样就可以筛选出仅偶数的平方：

[x \* x for x in range(1, 11) if x % 2 == 0]

还可以使用两层循环，可以生成全排列：

[m + n for m in 'ABC' for n in 'XYZ']

三层和三层以上的循环就很少用到了。

运用列表生成式，可以写出非常简洁的代码。例如，列出当前目录下的所有文件和目录名，可以通过一行代码实现：

import os # 导入os模块，模块的概念后面讲到

[d for d in os.listdir('.')] # os.listdir可以列出文件和目录

for循环其实可以同时使用两个甚至多个变量，比如dict的items()可以同时迭代key和value：

d = {'x': 'A', 'y': 'B', 'z': 'C' }

for k, v in d.items():

... print(k, '=', v)

因此，列表生成式也可以使用两个变量来生成list：

d = {'x': 'A', 'y': 'B', 'z': 'C' }

[k + '=' + v for k, v in d.items()]

最后把一个list中所有的字符串变成小写：

L = ['Hello', 'World', 'IBM', 'Apple']

[s.lower() for s in L]

##### 生成器

通过列表生成式，可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器：generator。

要创建一个generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的[]改成()，就创建了一个generator：

L = [x \* x for x in range(10)]

L

g = (x \* x for x in range(10))

g

创建L和g的区别仅在于最外层的[]和()，L是一个list，而g是一个generator。

可以直接打印出list的每一个元素，但怎么打印出generator的每一个元素呢？

如果要一个一个打印出来，可以通过next()函数获得generator的下一个返回值：

next(g)

next(g)

next(g)

讲过，generator保存的是算法，每次调用next(g)，就计算出g的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration的错误。

当然，上面这种不断调用next(g)实在是太麻烦了，正确的方法是使用for循环，因为generator也是可迭代对象：

g = (x \* x for x in range(10))

for n in g:

... print(n)

所以，创建了一个generator后，基本上永远不会调用next()，而是通过for循环来迭代它，并且不需要关心StopIteration的错误。

generator非常强大。如果推算的算法比较复杂，用类似列表生成式的for循环无法实现的时候，还可以用函数来实现。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

def fib(max):

n, a, b = 0, 0, 1

while n < max:

print(b)

a, b = b, a + b

n = n + 1

return 'done'

注意，赋值语句：

a, b = b, a + b

相当于：

t = (b, a + b) # t是一个tuple

a = t[0]

b = t[1]

但不必显式写出临时变量t就可以赋值。

上面的函数可以输出斐波那契数列的前N个数：

fib(6)

仔细观察，可以看出，fib函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

也就是说，上面的函数和generator仅一步之遥。要把fib函数变成generator，只需要把print(b)改为yield b就可以了：

def fib(max):

n, a, b = 0, 0, 1

while n < max:

yield b

a, b = b, a + b

n = n + 1

return 'done'

这就是定义generator的另一种方法。如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator：

f = fib(6)

f

这里，最难理解的就是generator和函数的执行流程不一样。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。

举个简单的例子，定义一个generator，依次返回数字1，3，5：

def odd():

print('step 1')

yield 1

print('step 2')

yield(3)

print('step 3')

yield(5)

调用该generator时，首先要生成一个generator对象，然后用next()函数不断获得下一个返回值：

o = odd()

next(o)

next(o)

step 2

next(o)

step 3

next(o)

可以看到，odd不是普通函数，而是generator，在执行过程中，遇到yield就中断，下次又继续执行。执行3次yield后，已经没有yield可以执行了，所以，第4次调用next(o)就报错。

回到fib的例子，在循环过程中不断调用yield，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator后，基本上从来不会用next()来获取下一个返回值，而是直接使用for循环来迭代：

for n in fib(6):

... print(n)

但是用for循环调用generator时，发现拿不到generator的return语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration的value中：

g = fib(6)

while True:

... try:

... x = next(g)

... print('g:', x)

... except StopIteration as e:

... print('Generator return value:', e.value)

... break

##### 迭代器

已经知道，可以直接作用于for循环的数据类型有以下几种：

一类是集合数据类型，如list、tuple、dict、set、str等；

一类是generator，包括生成器和带yield的generator function。

这些可以直接作用于for循环的对象统称为可迭代对象：Iterable。

可以使用isinstance()判断一个对象是否是Iterable对象：

from collections import Iterable

isinstance([], Iterable)

isinstance({}, Iterable)

isinstance('abc', Iterable)

isinstance((x for x in range(10)), Iterable)

isinstance(100, Iterable)

而生成器不但可以作用于for循环，还可以被next()函数不断调用并返回下一个值，直到最后抛出StopIteration错误表示无法继续返回下一个值了。

可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator。

可以使用isinstance()判断一个对象是否是Iterator对象：

from collections import Iterator

isinstance((x for x in range(10)), Iterator)

isinstance('abc', Iterator)

生成器都是Iterator对象，但list、dict、str虽然是Iterable，却不是Iterator。

把list、dict、str等Iterable变成Iterator可以使用iter()函数：

isinstance(iter([]), Iterator)

isinstance(iter('abc'), Iterator)

为什么list、dict、str等数据类型不是Iterator？

**这是因为Python的Iterator对象表示的是一个数据流，Iterator对象可以被next()函数调用并不断返回下一个数据，直到没有数据时抛出StopIteration错误。可以把这个数据流看做是一个有序序列，但却不能提前知道序列的长度，只能不断通过next()函数实现按需计算下一个数据，所以Iterator的计算是惰性的，只有在需要返回下一个数据时它才会计算。**

Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的。

**小结**

凡是可作用于for循环的对象都是Iterable类型；

凡是可作用于next()函数的对象都是Iterator类型，它们表示一个惰性计算的序列；

集合数据类型如list、dict、str等是Iterable但不是Iterator，不过可以通过iter()函数获得一个Iterator对象。

Python的for循环本质上就是通过不断调用next()函数实现的

##### 错误

###### 异常处理

try:

print('try...')

r = 10 / 0

print('result:', r)

except ZeroDivisionError as e:

print('except:', e)

finally:

print('finally...')

print('END')

###### 日志

import logging

def foo(s):

return 10 / int(s)

def bar(s):

return foo(s) \* 2

def main():

try:

bar('0')

except Exception as e:

logging.exception(e)

main()

print('END')

###### 抛出异常

class FooError(ValueError):

pass

def foo(s):

n = int(s)

if n==0:

raise FooError('invalid value: %s' % s)

return 10 / n

foo('0')