高级特性

掌握了Python的数据类型、语句和函数，基本上就可以编写出很多有用的程序了。

比如构造一个1, 3, 5, 7, ..., 99的列表，可以通过循环实现：

In [1]:
```
L = []
n = 1
while n <= 99:
    L.append(n)
    n = n + 2
```

取list的前一半的元素，也可以通过循环实现。

但是在Python中，代码不是越多越好，而是越少越好。代码不是越复杂越好，而是越简单越好。

基于这一思想，我们来介绍Python中非常有用的高级特性，1行代码能实现的功能，决不写5行代码。请始终牢记，代码越少，开发效率越高。

切片 (slice)

取一个list或tuple的部分元素是非常常见的操作。

比如，对以下list，欲取前3个元素，应该怎么做？

In [2]:
```
L = ['Michael', 'Sarah', 'Tracy', 'Bob', 'Jack']
```

笨办法：

In [3]:
```
[L[0], L[1], L[2]]
```

Out[3]:
```
['Michael', 'Sarah', 'Tracy']
```

之所以是笨办法是因为扩展一下，取前N个元素就没辙了。

取前N个元素，也就是索引为0~(N-1)的元素，可以用循环：

In [4]:
```
r = []
n = 3
for i in range(n):
    r.append(L[i])

r
```

Out[4]:
```
['Michael', 'Sarah', 'Tracy']
```

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。

```
In [5]: L[0:3]
```

```
Out[5]: ['Michael', 'Sarah', 'Tracy']
```

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：

```
In [6]: L[:3]
```

```
Out[6]: ['Michael', 'Sarah', 'Tracy']
```

也可以从索引1开始，取出2个元素出来：

```
In [10]: L[1:3]
```

```
Out[10]: ['Sarah', 'Tracy']
```

取倒数第一个元素，可以这样做：

```
In [7]: L[-1]
```

```
Out[7]: 'Jack'
```

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：

```
In [11]: L[-2:]
```

```
Out[11]: ['Bob', 'Jack']
```

```
In [12]: L[-2:-1]
```

```
Out[12]: ['Bob']
```

记住倒数第一个元素的索引是-1。

切片的用途

切片操作十分有用。我们先创建一个0-99的数列：

```
In [16]: L = list(range(100))
```

可以通过切片轻松取出某一段数列。

In [ ]:
```
print( L[:10] )      # 取前10个数
print( L[-10:] )     # 取后10个数
print( L[10:20] )    # 取第11-20个数
print( L[:10:2] )    # 前10个数，每两个取一个
print( L[::5] )      # 所有数，每5个取一个
print( L[:] )        # 复制一个list
```

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：

In [29]:
```
(0, 1, 2, 3, 4, 5)[:3]
```

Out[29]: (0, 1, 2)

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：

In [31]:
```
print( 'ABCDEFG'[:3] )
print( 'ABCDEFG'[::2] )
```

```
ABC
ACEG
```

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。

小结

有了切片操作，很多地方循环就不再需要了。Python的切片非常灵活，一行代码就可以实现很多行循环才能完成的操作。

迭代

如果给定一个list或tuple，我们可以通过for循环来遍历这个list或tuple，这种遍历我们称为迭代（Iteration）。

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

In [ ]:
```
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。

对于list, 可以这么迭代：

In [8]:
```
list = [1, 4, 9, 16]
for item in list:
    print(item)
```

```
1
4
9
16
```

list这种数据类型有下标，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。

对于dict，可以这样迭代：

In [34]:
```
d = {'a': 1, 'b': 2, 'c': 3}
for key in d:
    print(key)
```

```
c
a
b
```

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():

In [35]:
```
for value in d.values():
    print(value)
```

```
3
1
2
```

如果要同时迭代key和value，可以用for k, v in d.items()

In [36]:
```
for key, value in d.items():
    print(key, value)
```

```
c 3
a 1
b 2
```

对于字符串，它也是可迭代对象，也可作用于 for 循环：

In [38]:
```
for ch in 'ABC':
    print(ch)
```

```
A
B
C
```

总之，当我们使用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，而我们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？

可通过collections模块的Iterable类型判断：

In [43]:
```
from collections import Iterable
print( isinstance('abc', Iterable) )
print( isinstance([1, 2, 3], Iterable) )
print( isinstance(123, Iterable) )
```

```
True
True
False
```

如果你想同时获取列表的下标及对应的元素，该怎么做？

Python内置的enumerate()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。

In [48]:
```
L = ['A', 'B', 'C']
for i, value in enumerate(L):
    print(i, value)
```

```
0 A
1 B
2 C
```

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：

In [49]:
```
for x, y in [(1, 1), (2, 4), (3, 9)]:
    print(x, y)
```

```
1 1
2 4
3 9
```

小结

任何可迭代对象都可以作用于for循环，包括我们自定义的数据类型，只要符合迭代条件，就可以使用for循环。

列表生成式 (List Comprehension)

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。

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

In [ ]:
```
list(range(1, 11))
```

但如果要生成[1x1, 2x2, 3x3, ..., 10x10]，该怎么做？

方法一：用循环

In [73]:
```
L = []
for x in range(1, 11):
    L.append(x * x)

L
```

Out[73]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

方法二：列表生成式

用循环太繁琐，而列表生成式则可以用一行语句代替循环，生成上面的list：

In [74]:
```
[x * x for x in range(1, 11)]
```

Out[74]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

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。如：

In [75]:
```
[x * x for x in range(1, 11) if x % 2 == 0]
```

Out[75]:
```
[4, 16, 36, 64, 100]
```

还可以使用两层循环。

In [12]:
```
[m + n for m in 'ABC' for n in 'XYZ']
```

Out[12]:
```
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

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：

In [78]:
```
import os
[d for d in os.listdir('.')]
```

Out[78]:
```
['function.ipynb',
 'basic.ipynb',
 'oop.ipynb',
 'advance.ipynb',
 'lec.pdf',
 'slide.pdf',
 'slide',
 'code',
 'lec1.pdf',
 'lec.sh',
 '.ipynb_checkpoints',
 'images',
 'slide.sh',
 '.log',
 'README',
 'lec.tex',
 'lec',
 'slide.tex']
```

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：

In [82]:
```
d = {'x': 'A', 'y': 'B', 'z': 'C' }
for key, value in d.items():
    print(key, '=', value)
```

```
y = B
z = C
x = A
```

于是，列表生成式也可以使用两个变量来生成list：

In [83]:
```
[key + '=' + value for key, value in d.items()]
```

Out[83]:
```
['y=B', 'z=C', 'x=A']
```

再看一个例子，把一个list中所有的字符串变成小写：

In [84]:
```
L = ['Hello', 'World', 'IBM', 'Apple']
[s.lower() for s in L]
```

Out[84]:
```
['hello', 'world', 'ibm', 'apple']
```

练习

如果list中既包含字符串，又包含整数，由于非字符串类型没有lower()方法，所以列表生成式会报错：

In [86]:
```
L = ['Hello', 'World', 18, 'IBM', 'Apple']
[s.lower() for s in L]
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-86-8b69214ef009> in <module>()
      1 L = ['Hello', 'World', 18, 'IBM', 'Apple']
----> 2 [s.lower() for s in L]

<ipython-input-86-8b69214ef009> in <listcomp>(.0)
      1 L = ['Hello', 'World', 18, 'IBM', 'Apple']
----> 2 [s.lower() for s in L]

AttributeError: 'int' object has no attribute 'lower'
```

使用内建的isinstance函数可以判断一个变量是不是字符串：

In [88]:
```
x = 'abc'
y = 123
print( isinstance(x, str))
print( isinstance(y, str))
```

```
True
False
```

请修改列表生成式，通过添加if语句保证列表生成式能正确地执行：

In [91]:
```
L = ['Hello', 'World', 18, 'IBM', 'Apple']
L2 = [s.lower() for s in L if isinstance(s, str)]
L2
```

Out[91]:
```
['hello', 'world', 'ibm', 'apple']
```

生成器

通过列表生成式，我们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，如果我们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，那我们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：

In [17]:
```
L = [x * x for x in range(10)]
G = (x * x for x in range(10))
print(L)
print(G)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
<generator object <genexpr> at 0x7f734c0aec50>
```

我们可以直接打印出list的每一个元素，但我们怎么打印出generator的每一个元素呢？

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

In [18]:
```
next(G)
```

Out[18]:
```
0
```

In [19]:
```
next(G)
```

Out[19]:
```
1
```

In [27]:
```
next(G)
```

Out[27]:
```
81
```

In [28]:
```
next(G)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-28-380e167d6934> in <module>()
----> 1 next(G)

StopIteration:
```

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)实在是太变态了，正确的方法是使用for循环，因为generator也是可迭代对象：

In [106]:
```
G = (x * x for x in range(10))

for n in G:
    print(n)
```

```
0
1
4
9
16
25
36
49
64
81
```

所以，我们创建了一个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, ...

Fibonacci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：

In [107]:
```
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：

In [ ]:
```
a, b = b, a + b
```

相当于

In [ ]:
```
t = (a, a + b)
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：

In [109]:
```
fib(6)
```

```
1
1
2
3
5
8
```

Out[109]: 'done'

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()变成generator，只需要把print(b)改为yield b就可以了：

In [110]:
```
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：

In [111]:
```
f = fib(6)
f
```

Out[111]: <generator object fib at 0x7fbe703de200>

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：

In [29]:
```
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：

In [116]:
```
o = odd()
print( next(o) )
print( next(o) )
print( next(o) )
print( next(o) )
```

```
step 1
1
step 2
3
step 3
5

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-116-bd09e6d3ef35> in <module>()
      3 print( next(o) )
      4 print( next(o) )
----> 5 print( next(o) )

StopIteration:
```

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，我们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，我们基本上从来不会用next()来获取下一个返回值，而是直接使用for循环来迭代：

In [117]:
```
for n in fib(6):
    print(n)
```

```
1
1
2
3
5
8
```

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：

In [119]:
```
g = fib(6)
while True:
    try:
        x = next(g)
        print('g:', x)
    except StopIteration as e:
        print('Generator return value:', e.value)
        break
```

```
g: 1
g: 1
g: 2
g: 3
g: 5
g: 8
Generator return value: done
```

迭代器

可迭代对象

我们已经知道，可以直接作用于for循环的数据类型有以下几种：

- 集合数据类型，如
  - list
  - tuple
  - dict
  - set
  - str
- generator，包括
  - 生成器
  - 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：

In [125]:
```
from collections import Iterable
print( isinstance([], Iterable) )
print( isinstance((), Iterable) )
print( isinstance({}, Iterable) )
print( isinstance('', Iterable) )
print( isinstance((x for x in range(10)), Iterable) )
print( isinstance(123, Iterable))
```

```
True
True
True
True
True
False
```

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。

迭代器

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：

In [127]:
```
from collections import Iterator
print( isinstance([], Iterator) )
print( isinstance((), Iterator) )
print( isinstance({}, Iterator) )
print( isinstance('', Iterator) )
print( isinstance((x for x in range(10)), Iterator) )
print( isinstance(123, Iterator))
```

```
False
False
False
False
True
False
```

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()：

In [128]:
```
print( isinstance(iter([]), Iterator) )
print( isinstance(iter(()), Iterator) )
print( isinstance(iter({}), Iterator) )
print( isinstance(iter(''), Iterator) )
```

```
True
True
True
True
```

为什么**list**、**dict**、**str**等数据类型不是**Iterator**？

- 因为Iterator对象表示一个数据流，它可以被next()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，但却不能预先确定序列的长度，只能不断通过next()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。例如：

In [129]:
```
for x in [1, 2, 3, 4, 5]:
    pass
```

实际上完全等价于：

In [130]:
```
# 首先获得Iterator对象：
it = iter([1, 2, 3, 4, 5])
# 循环：
while True:
    try:
        # 获得下一个值：
        x = next(it)
    except StopIteration:
        # 遇到StopIteration就退出循环
        break
```