Python程序设计

第五讲 批量数据类型与操作
生成器与迭代器

张 华
WHU

生成器和迭代器

- 迭代
- 生成器
- 迭代器

迭代

迭代

- 如果给定一个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。
- 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这种数据有下标，而有些类型的数据是没有下标的，但是，只要是可迭代对象，无论有无下标，都可以迭代，比如zip：

```
>>> z = zip('abc',[12,34,56])
>>> for c,v in z:
...     print(c,':',v)
...
a : 12
b : 34
c : 56
```

生成器

列表推导式的问题

- 通过列表推导式，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。

生成器

- 在Python中，这种一边循环一边计算的机制，称为生成器（generator）。

生成器

创建生成器

- 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
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> g = (x*x for x in range(10))
>>> g
<generator object <genexpr> at 0x000002498B8DA7D8>
```

生成器

显示生成器的元素

- 可以直接打印出list的每一个元素，但是怎么打印出generator的每一个元素呢？
- 如果要一个一个打印出来，可以通过内置函数next()获得generator的下一个返回值：

```
>>> next(g)
0
>>> next(g)
1
>>> next(g)
4
…
>>> next(g)
81
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

生成器

显示生成器的元素

- 生成器保存的是算法，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。
- 因为生成器是可迭代对象，也可以使用for循环：

```
>>> for v in g:
...     print(v)
...
0
1
4
9
16
25
36
49
64
81
```

生成器

生成器的特点

- 使用生成器对象的元素时，可以根据需要将其转化为列表或元组，也可以使用生成器对象的__next__()方法或者内置函数next()进行遍历，或者直接使用for循环来遍历其中的元素。
- 但是不管用哪种方法访问其元素，只能从前往后正向访问每个元素，没有任何方法可以再次访问已访问过的元素，也不支持使用下标访问其中的元素。
- 当所有元素访问结束以后，如果需要重新访问其中的元素，必须重新创建该生成器对象，enumerate、filter、map、zip等其他迭代器对象也具有同样的特点。

生成器

生成器的特点

```
>>> g = (x*x for x in range(10))
>>> l1 = list(g)
>>> l1
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> list(g)                                  #生成器对象已遍历结束，没有元素了
[]

>>> x = map(str, range(20))                  #map对象也具有类似的特点
>>> '0' in x
True
>>> '0' in x                                 #不可再次访问已访问过的元素
False
```

可迭代对象

- 可以直接用于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；
  - 一类是生成器，包括生成器和带yield的生成器函数。
  - 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.abc import Iterable
>>> isinstance(g, Iterable)
True
>>> isinstance('abc', Iterable)
True
```

迭代器

迭代器

- 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。
- 可以被内置函数next()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.abc import Iterator
>>> isinstance(g, Iterator)
True
>>> isinstance([1,2,3], Iterator)
False
```

迭代器

创建迭代器

- 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可以使用内置函数iter()：

```
>>> isinstance(iter([1,2,3]), Iterator)
True
>>> isinstance(iter('abc'), Iterator)
True
```

迭代器

迭代器的特点

- 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；
- 批量数据类型（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。

小结

- Python的for循环本质上就是通过不断调用next()函数实现的。
  - 例如：

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

  - 等价于：

```
# 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
```