# 迭代器与生成器

## next—手动遍历

为了手动的遍历可迭代对象，使用next() 函数并在代码中捕获StopIteration 异常

# 手动读取一个文件中的所有行：  
*def* manual\_iter():  
 *with* open('/etc/passwd') *as* f:  
 *try*:  
 *while True*:  
 line = next(f)  
 print(line, end='')  
 *except* StopIteration:  
 *pass*

StopIteration 用来指示迭代的结尾。然而，如果你手动使用上面演示的next() 函数的话，你还可以通过返回一个指定值来标记结尾，比如None

line = next(f, None)

## \_\_iter\_\_—代理迭代

构建一个自定义容器对象，里面包含有列表、元组或其他可迭代对象。在这个新容器对象上执行迭代操作。定义一个\_\_iter\_\_ () 方法，将迭代操作代理到容器内部的对象上去。

*class* Node:  
 *def \_\_init\_\_*(self, *value*):  
 self.\_value = *value* self.\_children = []  
 *def \_\_repr\_\_*(self):  
 *return* 'Node({!r})'.format(self.\_value)  
 *def* add\_child(self, *node*):  
 self.\_children.append(*node*)  
 *def \_\_iter\_\_*(self):  
 *return* iter(self.\_children)  
 # Example  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 root = Node(0)  
 child1 = Node(1)  
 child2 = Node(2)  
 root.add\_child(child1)  
 root.add\_child(child2)  
 *for* ch *in* root:  
 print(ch)

## yield—生成器

一个函数中需要有一个yield 语句即可将其转换为一个生成器。跟普通函数不同的是，生成器只能用于迭代操作。

一个生成器函数主要特征是它只会回应在迭代中使用到的next 操作。一旦生成器函数返回退出，迭代终止

*def* frange(*start*, *stop*, *increment*):  
 x = *start  
 while* x < *stop*:  
 *yield* x  
 x += *increment  
for* n *in* frange(0, 4, 0.5):  
 print(n)

## 迭代器协议

目前为止，在一个对象上实现迭代最简单的方式是使用一个生成器函数。

*class* Node:  
 *def \_\_init\_\_*(self, *value*):  
 self.\_value = *value* self.\_children = []  
 *def \_\_repr\_\_*(self):  
 *return* 'Node({!r})'.format(self.\_value)  
 *def* add\_child(self, *node*):  
 self.\_children.append(*node*)  
 *def \_\_iter\_\_*(self):  
 *return* iter(self.\_children)  
 *def* depth\_first(self):  
 *yield* self  
 *for* c *in* self:  
 *yield from* c.depth\_first()  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 root = Node(0)  
 child1 = Node(1)  
 child2 = Node(2)  
 root.add\_child(child1)  
 root.add\_child(child2)  
 child1.add\_child(Node(3))  
 child1.add\_child(Node(4))  
 child2.add\_child(Node(5))  
 *for* ch *in* root.depth\_first():  
 print(ch)

# Outputs Node(0), Node(1), Node(3), Node(4), Node(2), Node(5)

depth first() 方法简单直观。它首先返回自己本身并迭代每一个子节点并通过调用子节点的depth first() 方法(使用yield from 语句) 返回对应元素。

## reversed—反向迭代

反向迭代仅仅当对象的大小可预先确定或者对象实现了reversed () 的特殊方法时才能生效。如果两者都不符合，那你必须先将对象转换为一个列表才行, 如果可迭代对象元素很多的话，将其预先转换为一个列表要消耗大量的内存。

*class* Countdown:  
 *def \_\_init\_\_*(self, *start*):  
 self.start = *start* # Forward iterator  
 *def \_\_iter\_\_*(self):  
 n = self.start  
 *while* n > 0:  
 *yield* n  
 n -= 1  
 *def \_\_reversed\_\_*(self):  
 n = 1  
 *while* n <= self.start:  
 *yield* n  
 n += 1  
*for* rr *in* reversed(Countdown(30)):  
 print(rr)  
*for* rr *in* Countdown(30):  
 print(rr)

定义一个反向迭代器可以使得代码非常的高效，因为它不再需要将数据填充到一个

列表中然后再去反向迭代这个列表。

## 带有外部状态的生成器

如果想让生成器暴露外部状态，可以简单的将它实现为一个类，然后把生成器函数放到\_\_iter\_\_() 方法中