迭代器

iter()函数

内置函数iter()：生成迭代器

参数为容器(比如：列表)，则返回一个迭代器对象

参数为迭代器，则返回自己

```
a = [1, 2, 3, 4]
it = iter(a) # 返回一个迭代器<listiterator at 0x102496e10>
it2 = iter(it) # 返回自己
id(it) == id(it2) # True
```

迭代器

迭代器都包含一个next()：惰性计算(迭代之前不存在，迭代之后可被销毁)，返回制定序列的下一个元素

如果next第一次被调用，返回第一个元素

如果next最后一次调用，则会报错：StopIteration

```
# 循环的本质
it = iter([1, 2, 3, 4])
print(it)
try:
    while True:
        print(it.__next__())
except StopIteration:
    pass

# for循环是python的语法糖
# 注意：如果继续执行遍历，下面这段for循环不会执行，因为迭代器用一次之后就会被销毁
# 如果用列表，则不会销毁，可以循环使用
for i in it:
    print(i)
    print("hello")
```

解释：for 循环的时候，首先对循环对象实现迭代器包装，返回一个迭代器对象，然后每循环一步，就调用那个迭代器对象的next方法，循环结束的时候，自动处理了 StopIteration这个异常。for循环是对迭代器进行迭代的**语法糖**。

迭代器内部

迭代器对象(class类)，包含两个方法：`__iter__` 和 `next`，next是迭代器的定义，必须存在；`__iter__`是python的迭代协议的要求

1. `__iter__`方法必须返回一个迭代器对象，否则会报错

```
# 直接返回1
class only_iter(object):
    def __iter__(self):
        return 1

# 会报错：iter() returned non-iterator of type 'int'
# 返回的对象不是iterator，而是int
print(iter(only_iter()))

# 返回自己
class iterator_cls(object):
    def __init__(self, num):
        self.num = num

    def __iter__(self):
        return self

# 会报错：iter() returned non-iterator of type 'iterator_cls'
# 返回的对象不是iterator类型而是：cls
print(iter(iterator_cls(10))
```

iter()函数，会去检查传入对象的`__iter__`方法的返回内容，检查该内容是否是迭代器，如果是迭代器，则直接返回该迭代器，如果不是迭代器则报错。其实for x in y也是一样，针对y，也会进行同样的判断。

2. 如果只包含`__next__`而不包含 `__iter__`

```
# 定义一个迭代器，但是这个迭代器是不可以对其进行for循环的
class iterator_cls(object):
    def __init__(self, num):
        self.num = num

    def __next__(self):
        self.num += 1
        if self.num > 15:
            raise StopIteration
        return self.num

# 报错：'iterator_cls' object is not iterable
# 发现cls对象，是不可以for循环的！！！
print(iter(iterator_cls(10)))
```

3. 关于`__iter__`和`__next__`的完整解释：

```
class iterator_cls(object):
    def __init__(self, num):
        self.num = num

    # def __iter__(self):
    #     return self

    def __next__(self):
        self.num += 1
        if self.num > 15:
            raise StopIteration
        return self.num

# 如果缺少__next__方法，则会提示：non-iterator，因为iterator必须包含next方法
# 如果缺少__iter__方法，则会提示：不是iterable，因为for循环这个语法糖，会先把 in ..
# 后面的东西，先执行一下：iter()，该方法期望获取到一个iterator，会进行如下操作：
# 1. 去检查对象是否包含__iter__方法
# 2. 检查__iter__方法是否返回了一个iterator，但我们这里没有__iter__方法，所以得到
None
# 3. for循环/iter() 会抛出异常：not iterable，不可迭代
# 方式1
print(iter(iterator_cls(10)))
# 方式2
# for i in iterator_cls(10):
#     print(i)
```

4. 完整版的迭代器：

```
from collections import Iterable
from collections import Iterator


class iterator_cls(object):
    def __init__(self, num):
        self.num = num

    def __iter__(self):
        return self

    def __next__(self):
        self.num += 1
        if self.num > 15:
            raise StopIteration
        return self.num
# 当一个类包含__next__方法的时候，则这个类就是一个迭代器
# 受限于python的协议规定，期待这个类的__iter__方法返回这个迭代器
# 上面两种结果结合，导致这个迭代器是可以进行for循环的，所以它就成为了Iterable对象
print(isinstance(iterator_cls(10), Iterable))
```

```
print(isinstance(iterator_cls(10), Iterator))
```

总结

Iterator只要要求实现`__next__`即可，但在python的Iterator里如果只实现`__next__`会很不方便，因为python的for循环机制，要求被迭代的对象必须是Iterable，上面执行发现：如果一个类A只实现了`__next__`方法，那么它是一个Iterator，但不是一个可迭代对象(Iterable)。如果要它是一个可迭代对象，则必须把整个类A(Iterator)当作一个对象来返回，比如下面这样：

```
from collections import Iterable
from collections import Iterator


class only_next(object):
    def __init__(self, num):
        self.num = num

    def __next__(self):
        self.num += 1
        if self.num > 15:
            raise StopIteration
        return self.num


# 当我们让这个类通过__iter__方法返回一个Iterator对象时，这个类就是一个Iterable对象了
class only_iter():
    def __iter__(self):
        # 把上面定义的Iterator返回
        return only_next(10)


# 发现是Iterable对象
print(isinstance(only_iter(), Iterable))
# 但不是Iterator对象，因为only_iter中没有next方法
print(isinstance(only_iter(), Iterator))
# 发现可以正常进行for循环
for i in only_iter():
    print(i)
```

所以，为了方便使用，我们通常直接在Iterator中加上`__iter__`让它返回自己整个类，如下：

```
class iterator_cls(object):
    def __init__(self, num):
        self.num = num

    def __iter__(self):
        return self

    def __next__(self):
        self.num += 1
        if self.num > 15:
            raise StopIteration
        return self.num
```

这样，我们的这个类，即是Iterator，又是Iterable，并且可以方便的进行for循环。总结来说：Iterator中添加的`__iter__`是为了兼容Iterable的接口。

补充：

1. 某个类下的`__iter__`方法期望返回一个Iterator对象，如果你强制不返回Iterator，则这个类：不能进行for循环遍历；也不能用iter()方法获取到一个迭代对象