迭代器

> 迭代是访问集合元素的一种方式。迭代器是一个可以记住遍历的位置的对象。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

1. 可迭代对象

我们已经知道可以对list、tuple、str等类型的数据使用for...in...的循环语法从其中依次拿到数据进行使用，我们把这样的过程称为遍历，也叫**迭代**。

但是，是否所有的数据类型都可以放到for...in...的语句中，然后让for...in...每次从中取出一条数据供我们使用，即供我们迭代吗?

```
>>> for i in 100:
...     print(i)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>>
# int整型不是iterable，即int整型不是可以迭代的
```

我们把可以通过for...in...这类语句迭代读取一条数据供我们使用的对象称之为可迭代对象(Iterable)。

2. 如何判断一个对象是否可以迭代

可以使用 isinstance() 判断一个对象是否是 Iterable 对象：

```
In [50]: from collections import Iterable

In [51]: isinstance([], Iterable)
Out[51]: True

In [52]: isinstance({}, Iterable)
Out[52]: True

In [53]: isinstance('abc', Iterable)
Out[53]: True

In [54]: isinstance(mylist, Iterable)
Out[54]: False

In [55]: isinstance(100, Iterable)
Out[55]: False
```

3. 可迭代对象的本质

我们分析对可迭代对象进行迭代使用的过程，发现每迭代一次（即在for...in...中每循环一次）都会返回对象中的下一条数据，一直向后读取数据直到迭代了所有数据后结束。那么，在这个过程中就应该有一个“人”去记录每次访问到了第几条数据，以便每次迭代都可以返回下一条数据。我们把这个能帮助我们进行数据迭代的“人”称为**迭代器(Iterator)**。

可迭代对象的本质就是可以向我们提供一个这样的中间“人”即迭代器帮助我们对其进行迭代遍历使用。

可迭代对象通过 `__iter__` 方法向我们提供一个迭代器，我们在迭代一个可迭代对象的时候，实际上就是先获取该对象提供的一个迭代器，然后通过这个迭代器来依次获取对象中的每一个数据.

那么也就是说，一个具备了 `__iter__` 方法的对象，就是一个可迭代对象。

```
from collections.abc import Iterable
class Demo(object):
    def __init__(self, n):
        self.n = n
        self.current = 0
    def __iter__(self):
        pass

demo = Demo(10)
print(isinstance(demo, Iterable))  # True

for d in demo:   # 重写了 __iter__ 方法以后，demo就是一个一个可迭代对象了，可以放在for...in的后面
    print(d)

# 此时再使用for...in循环遍历，会提示 TypeError: iter() returned non-iterator of type 'NoneType'
# 这是因为，一个可迭代对象如果想要被for...in循环，它必须要有一个迭代器
```

4. 迭代器Iterator

通过上面的分析，我们已经知道，迭代器是用来帮助我们记录每次迭代访问到的位置，当我们对迭代器使用next()函数的时候，迭代器会向我们返回它所记录位置的下一个位置的数据。实际上，在使用next()函数的时候，调用的就是迭代器对象的 `__next__` 方法（Python3中是对象的 `__next__` 方法，Python2中是对象的next()方法）。**所以，我们要想构造一个迭代器，就要实现它的next方法**。但这还不够，python要求迭代器本身也是可迭代的，所以我们还要为迭代器实现 `__iter__` 方法，而 `__iter__` 方法要返回一个迭代器，迭代器自身正是一个迭代器，所以迭代器的 `__iter__` 方法返回自身即可。

一个实现了iter方法和next方法的对象，就是迭代器。

```
class MyIterator(object):
    def __init__(self, n):
        self.n = n
        self.current = 0

    # 自定义迭代器需要重写__iter__和__next__方法
    def __iter__(self):
        return self

    def __next__(self):
        if self.current < self.n:
            value = self.current
            self.current += 1
            return value
        else:
            raise StopIteration

my_it = MyIterator(10)

for i in my_it:    # 迭代器重写了__iter__方法，它本身也是一个可迭代对象
    print(i)
```

5. 如何判断一个对象是否是迭代器

调用一个对象的 `__iter__` 方法，或者调用iter()内置函数，可以获取到一个可迭代对象的迭代器。

```
names = ['hello', 'good', 'yes']
print(names.__iter__())  # 调用对象的__iter__()方法
print(iter(names))  # 调用iter()内置函数
```

可以使用 isinstance() 判断一个对象是否是 Iterator 对象：

```
from collections.abc import Iterator
names = ['hello', 'good', 'yes']
print(isinstance(iter(names), Iterator))
```

6. for...in...循环的本质

for **item** in **Iterable** 循环的本质就是先通过iter()函数获取可迭代对象Iterable的迭代器，然后对获取到的迭代器不断调用next()方法来获取下一个值并将其赋值给item，当遇到StopIteration的异常后循环结束。

7. 迭代器的应用场景

我们发现迭代器最核心的功能就是可以通过next()函数的调用来返回下一个数据值。如果每次返回的数据值不是在一个已有的数据集合中读取的，而是通过程序按照一定的规律计算生成的，那么也就意味着可以不用再依赖一个已有的数据集合，也就是说不用再将所有要迭代的数据都一次性缓存下来供后续依次读取，这样可以节省大量的存储（内存）空间。

举个例子，比如，数学中有个著名的斐波拉契数列(Fibonacci)，数列中第一个数为0，第二个数为1，其后的每一个数都可由前两个数相加得到：

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

现在我们想要通过for...in...循环来遍历迭代斐波那契数列中的前n个数。那么这个斐波那契数列我们就可以用迭代器来实现，每次迭代都通过数学计算来生成下一个数。

```
class FibIterator(object):
    """斐波那契数列迭代器"""
    def __init__(self, n):
        """
        :param n: int, 指明生成数列的前n个数
        """
        self.n = n
        # current用来保存当前生成到数列中的第几个数了
        self.current = 0
        # num1用来保存前前一个数，初始值为数列中的第一个数0
        self.num1 = 0
        # num2用来保存前一个数，初始值为数列中的第二个数1
        self.num2 = 1

    def __next__(self):
        """被next()函数调用来获取下一个数"""
        if self.current < self.n:
            num = self.num1
            self.num1, self.num2 = self.num2, self.num1+self.num2
            self.current += 1
            return num
        else:
            raise StopIteration

    def __iter__(self):
        """迭代器的__iter__返回自身即可"""
        return self


if __name__ == '__main__':
    fib = FibIterator(10)
    for num in fib:
        print(num, end=" ")
```