第八章迭代生成

> 本文是Python通用编程系列教程，已全部更新完成，实现的目标是从零基础开始到精通Python编程语言。本教程不是对Python的内容进行泛泛而谈，而是精细化，深入化的讲解，共5个阶段，25章内容。所以，需要有耐心的学习，才能真正有所收获。虽不涉及任何框架的使用，但是会对操作系统和网络通信进行全局的讲解，甚至会对一些开源模块和服务器进行重写。学完之后，你所收获的不仅仅是精通一门Python编程语言，而且具备快速学习其他编程语言的能力，无障碍阅读所有Python源码的能力和对

> 计算机与网络的全面认识。对于零基础的小白来说，是入门计算机领域并精通一门编程语言的绝佳教材。对于有一定Python基础的童鞋，相信这套教程会让你的水平更上一层楼。

一 迭代器

1. 迭代器说明

迭代器就是迭代的工具，迭代是一个重复的过程，并且每次重复都是基于上一次的结果而来。

```
# 这是一个迭代过程，虽然在重复，但是每次结果不一样
dict1 = {'x': 1, 'y': 2}
n = 0
for i in dict1:
    if n < len(dict1):
        print(dict1[i])
    n += 1

# 这不是迭代过程，一直在重复，却没有变化
while True:
    print('======>')
```

2. 可迭代对象

要想了解迭代器到底是什么？必须先要清楚一个概念，即什么是可迭代的对象？在python中，只要内置有iter方法的对象，都是可迭代的对象。

```
# 这不是可迭代对象
num = 1

# 以下都是可迭代的对象
str1 = 'hello'
list1 = [1, 2, 3]
tup1 = (1, 2, 3)
dict1 = {'x': 1}
set1 = {'a', 'b', 'c'}
file1 = open('a.txt', 'w', encoding='utf-8')
```

3. 迭代器用法

可迭代的对象执行iter方法得到的返回值就是迭代器对象。

```
dict1 = {'x': 1, 'y': 2, 'z': 3}
iter_dict1 = dict1.__iter__()  # 双下划线开头和结尾在python中称为魔法函数，之后在面向对象的章节中会详述，好奇的同学请自行百度
print(iter_dict1.__next__())
print(iter_dict1.__next__())
print(iter_dict1.__next__())
# print(iter_dict1.__next__())  # 停止迭代

set1 = {'a', 'b', 'c'}
iter_set1 = set1.__iter__()
print(iter_set1.__next__())
print(iter_set1.__next__())
print(iter_set1.__next__())
# print(iter_set1.__next__())  # 停止迭代

list1 = [1, 2, 3]
iter_list1 = list1.__iter__()
print(iter_list1.__next__())
print(iter_list1.__next__())
print(iter_list1.__next__())
```

4. 可迭代对象VS迭代器对象

(1) 可迭代对象

可迭代对象无须获取，Python内置str，list，tuple，dict，set，file都是可迭代对象，它的特点是内置有iter方法，执行该方法会拿到一个返回值就是迭代器对象。

(2) 迭代器对象

文件对象本身既是可迭代对象又是迭代器对象，可迭代对象执行iter方法，拿到的返回值就是迭代器对象。迭代器对象的特点是内置有next方法，执行该方法会拿到迭代器对象中的一个值，迭代器对象内置有iter方法，执行该方法会拿到迭代器本身。

```
str1 = 'hello'  # 可迭代对象

iter_str1 = str1.__iter__()  # 迭代器对象
print(iter_str1.__next__())  # 取出迭代器对象中的一个值
print(iter_str1.__iter__() is iter_str1)
print(iter_str1.__iter__().__iter__() is iter_str1)
print(iter_str1.__iter__().__iter__().__iter__() is iter_str1)

# 文件本身既是迭代器对象又是可迭代对象
f = open('a.txt', 'r', encoding='utf-8')
print(f.__iter__() is f)
print(f.__next__())
print(f.__next__())
print(f.__next__())
print(f.__next__())
print(f.__next__())
```

5. 迭代器优缺点分析

(1) 优点

<1> 提供了一种可以不依赖索引取值的方式

假如你现在没有学过for循环，对于没有索引的可迭代对象如set，dict或者file这些应该怎么单独取出里面的每一个值?

```
# 集合
set1 = {1, 2, 3, 4, 5, }
iter_set1 = set1.__iter__()
while True:

    # try和except是第三阶段面向对象最后一个章节的内容，这里先简单使用一下

    try:  # 监测try下面的代码块是否出现异常
        print(iter_set1.__next__())
    except StopIteration:  # 相当于if判断，如果出现的异常是StopIteration
        break

# 字典
dict1 = {'x': 1, 'y': 2, 'z': 3}
```

```
iter_dict1 = dict1.__iter__()
while True:
    try:
        print(iter_dict1.__next__())
    except StopIteration:
        break

# 文件

# 文件内容有五行，每行分别是一个数字1，2，3，4，5
file1 = open('a.txt', 'r', encoding='utf-8')
iter_file1 = file1.__iter__()
while True:
    try:
        print(iter_file1.__next__())
    except StopIteration:
        break
file1.close()

# 列表
# 有索引的可迭代对象自然也可以使用
list1 = [1, 2, 3, 4, 4, 5, 5, 6, 6, 6]
iter_list1 = list1.__iter__()
while True:
    try:
        print(iter_list1.__next__())
    except StopIteration:
        break
```

<2> 迭代器更加节省内存

```
# range用法
for i in range(0, 10):
    print(i)

# item是一个可迭代对象，指的是从0到10000000000000000000000的所有数字
item = range(0, 10000000000000000000000)
iter_item = item.__iter__()  # 迭代器对象
while True:
    try:
        print(iter_item.__next__())
    except StopIteration:
        break
```

(2) 缺点

<1> 取值有缺陷

取值麻烦，只能一个一个取，只能往后取，并且是一次性的。

```
x = [1, 2, 3]
iter_x = x.__iter__()
while True:
    try:
        print(iter_x.__next__())
    except StopIteration:
        break

print('第二次==================================>')

# iter_x = x.__iter__()  # 注释这行第二次取不到，像小孩玩滑梯一样，要重新爬上去
while True:
    try:
        print(iter_x.__next__())
    except StopIteration:
        break
```

<2> 无法用len获取长度

迭代器对象不取到最后一个值，你永远不能知道它的长度。

> 人生就像是一个各式各样的巧克力，你永远不知道你下一块是什么口味。

```
x = [1, 2, 3]
iter_x = x.__iter__()
# print(len(iter_x))  # 没有获取长度方法
```

6. for循环的原理

for循环称之为迭代器循环，in后跟的必须是可迭代的对象，for循环会执行in后对象的**iter**方法，拿到迭代器对象，然后调用迭代器对象的**next**方法，拿到一个返回值赋值给一个变量，周而复始，直到取值完毕，for循环会检测到异常自动结束循环。

```
file1 = open('a.txt', 'r', encoding='utf-8')
for line in file1:  # iter_file1=file1.__iter__()
    print(line)

for item in {'x': 1, 'y': 2}:
    print(item)
```

二 生成器

1. 生成器说明

> 我们可以把上面讲过的迭代器理解为一只老母鸡，理论上讲，老母鸡的肚子里可以有无穷个蛋，但是它需要一个一个的下蛋，Python给我们内置了几种老母鸡数据类型。

生成器其实本质就是迭代器，或者说生成器是特殊的迭代器，因为生成器是我们自己制造的迭代器。

2. yield两个用法

1. yield为我们提供了一种自定义迭代器的方式，可以在函数内用yield关键字，调用函数拿到的结果就是一个生成器。
2. yield可以像return一样用于返回值，区别是return只能返回一次值，而yield可返回多次，因为yield可以保存函数执行的状态。

yield与return用法比较

```
# yield
def test_yield():
    print('=======>first')
    yield 1
    print('=======>second')
    yield 2
```

```
    print('=======>third')
    yield 3


# 使用yield返回，调用函数时，不会执行函数体代码，拿到的返回值就是一个生成器对象
res = test_yield()
print(res)  # <generator object test_yield at 0x1078f7660>
print(res.__iter__() is res)
print(res.__next__())
print(res.__next__())
print(res.__next__())

# return
def test_return():
    print('=======>first')
    return 1  # 使用return返回，函数执行结束
    print('=======>second')
    return 2
    print('=======>third')
    return 3


res = test_return()
```

3. 生成器的构造

函数内包含有yield关键字，再调用函数，就不会执行函数体代码，拿到的返回值就是一个生成器对象。

```
def chicken():
    print('=====>first')
    yield 1
    print('=====>second')
    yield 2
    print('=====>third')
    yield 3


obj = chicken()
print(obj)
print(obj.__iter__() is obj)
print(obj.__next__())
```

```
print(obj.__next__())
print(obj.__next__())
```

4. 自定义range

(1) range的用法

```
for i in range(1, 10, 1):
    """
    range最多可以接收三个参数，第一个是起始位置，默认值为0，
    第二个是结束位置，无默认值，必须指定，
    第三个是步长，默认值为1，
    如果只传一个位置参数，那就是指的结束位置，
    如果传两个位置参数，第一个为起始位置，
    第二个为结束位置，
    range第一个能取到，最后一个取不到，顾头不顾尾
    """
    print(i)
```

(2) 自定义range

```
# 简易版本range，只能接收两个位置参数或者三个位置参数，起始位置没有默认值
def show_my_range(start, stop, step=1):
    n = start
    while n < stop:
        yield n
        n += step


for item in show_my_range(1, 10, 3):
    print(item)
```

5. yield表达式

yield可以把函数暂停住，那么自然就能保存函数的运行状态，我们可以使用yield表达式形式来做一些有意思的操作。

```
def eat(name):
    print('【1】%s is ready for eating' % name)
    while True:
        food = yield  # 这是yield表达式形式，yield可以赋值给一个变量
        print('【2】%s starts to eat %s' % (name, food))


person1 = eat('Albert')

# 函数暂停在food = yield这行代码
person1.__next__()

# 继续执行代码，由于yield没有值，即yield = None，则food = None
person1.__next__()
```

yield肯定不能一直为空，肯定有一种方法给yield传值，这种方法就是send。

```
def eat(name):
    print('【1】%s is ready for eating' % name)
    while True:
        food = yield
        print('【2】%s starts to eat %s' % (name, food))


person1 = eat('Albert')
"""
对于表达式形式的yield，在使用前必先初始化
即第一次必须传None，或者用__next__方法
"""
# person1.send(None)  # 初始化，和下面一行代码同等效果
person1.__next__()

person1.send('蒸羊羔')  # send有两个功能：1 传值，2 初始化
person1.send('蒸鹿茸')
person1.send('蒸熊掌')
person1.send('烧素鸭')
person1.close()  # 关闭之后，后面的就吃不了了，也不能兜着走
# person1.send('烧素鹅')
# person1.send('烧鹿尾')
```

我们原本就知道yield可以有返回值，那么能否与yield表达式形式连用呢？如果我们需要记录吃过的东西，就要用到这种用法。

```
def eat(name):
    print('%s is ready for eating' % name)
    food_list = []
    while True:
        food = yield food_list
        print('%s starts to eat %s' % (name, food))
        food_list.append(food)


name = 'Albert'

person1 = eat(name)

person1.send(None)
# person1.__next__()

res1 = person1.send('蒸羊羔')
print('%s has eaten %s' % (name, res1))

res2 = person1.send('蒸鹿茸')
print('%s has eaten %s' % (name, res2))

res3 = person1.send('蒸熊掌')
print('%s has eaten %s' % (name, res3))

res4 = person1.send('烧素鸭')
print('%s has eaten %s' % (name, res4))

person1.close()  # 关闭之后，后面的就吃不了了，也不能兜着走
# person1.send('烧素鹅')
# person1.send('烧鹿尾')
```

以上这种写法能够帮助你更好的理解yield的执行过程，但是明显有点啰嗦，为了实现同样的功能，我们还有更加简介的写法。

```
def eat(name):
    print('%s is ready for eating' % name)
    food_list = []
    while True:
```

```
        food = yield food_list
        print('%s starts to eat %s' % (name, food))
        food_list.append(food)
        print('%s has eaten %s' % (name, food_list))


person1 = eat('Albert')

person1.send(None)

person1.send('蒸羊羔')

person1.send('蒸鹿茸')

person1.send('蒸熊掌')

person1.send('烧素鸭')

person1.close()
```

三 面向过程编程

(1) 编程范式

面向过程编程是一种编程的思想，或者叫编程范式，本篇末尾所讲解的就是编程范式的一种，即面向过程编程。

> 编程范式就像是武林中的各个流派，没有高下之分，比如我是明教的，你们都是峨眉的，不一定我明教的武功就比峨眉的高，能够区分的使用的人或者不同的场景。

面向过程的编程思想核心是`过程`二字，过程即解决问题的步骤，即先干什么，再干什么，最后干什么。基于面向过程编写程序就好比在设计一条流水线，是一种机械式的思维方式。在本教程中，我们过去所讲解的示例都是面向过程编程。

(2) 面向过程编程优缺点

- 优点：复杂的问题流程化，进而简单化。
- 缺点：修改一个阶段，其他阶段都有可能需要做出修改，牵一发而动全身，即扩展性差。

对于大部分人来说，写的程序都是都需不断迭代的(只要需求发生变化，就要修改)，如果全部使用面向过程编程，那么程序的扩展讲会变得不现实，但是一些很小的功能肯定使用面向过程更简单。除此之外，与面向过程编程相对应的是面向对象编程，这也是我们后面要讲解的内容，使用面向对象编程会让程序设计变得更为复杂。在有些场景下，当一个项目的设计非常庞大，而且并不需要经常改动，可能两三年才会换代一次，为了使程序设计设计变得简洁，可能会采用面向过程编程，但是也需要使用面向对象编程思想。Windows系统到目前为止，代码大约为七千万行代码(可能你的整个职业生涯也写不了这么多代码)，这里面使用大量的C++和少量的汇编语言。往往我们在写程序的时候都需要两者结合，但是归根结底从整体来看程序设计还是面向过程，一定要有明确的先后顺序，当你们学习到了面向对象就会有更深的体会。