

迭代器、生成器、装饰器

一、实验介绍

1.1 实验内容

在这个实验里我们学习迭代器、生成器、装饰器有关知识。

1.2 实验知识点

- 迭代器
- 生成器
- 生成器表达式
- 闭包
- 装饰器

1.3 实验环境

- python3.5
- Xfce终端
- Vim

1.4 适合人群

本课程属于初级级别课程，不仅适用于那些有其它语言基础的同学，对没有编程经验的同学也非常友好

二、实验步骤

2.1 迭代器

Python 迭代器 (*Iterators*) 对象在遵守迭代器协议时需要支持如下两种方法。

`__iter__()` , 返回迭代器对象自身。这用在 `for` 和 `in` 语句中。

`__next__()` , 返回迭代器的下一个值。如果没有下一个值可以返回, 那么应该抛出 `StopIteration` 异常。

```
class Counter(object):
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        #返回下一个值直到当前值大于 high
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1
```

现在我们能把这个迭代器用在我们的代码里。

```
>>> c = Counter(5,10)
>>> for i in c:
...     print(i, end=' ')
...
5 6 7 8 9 10
```

请记住迭代器只能被使用一次。这意味着迭代器一旦抛出 `StopIteration` , 它会持续抛出相同的异常。

```
>>> c = Counter(5,6)
>>> next(c)
5
>>> next(c)
6
>>> next(c)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 11, in next
StopIteration
>>> next(c)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 11, in next
StopIteration
```

我们已经看过在 `for` 循环中使用迭代器的例子了，下面的例子试图展示迭代器被隐藏的细节：

```
>>> iterator = iter(c)
>>> while True:
...     try:
...         x = iterator.__next__()
...         print(x, end=' ')
...     except StopIteration as e:
...         break
...
5 6 7 8 9 10
```

2.2 生成器

在这一节我们学习有关 Python 生成器 (*Generators*) 的知识。生成器是更简单的创建迭代器的方法，这通过在函数中使用 `yield` 关键字完成：

```
>>> def my_generator():
...     print("Inside my generator")
...     yield 'a'
...     yield 'b'
...     yield 'c'
...
>>> my_generator()
<generator object my_generator at 0x7fbcfa0a6aa0>
```

在上面的例子中我们使用 `yield` 语句创建了一个简单的生成器。我们能在 `for` 循环中使用它，就像我们使用任何其它迭代器一样。

```
>>> for char in my_generator():
...     print(char)
...
Inside my generator
a
b
c
```

在下一个例子里，我们会使用一个生成器函数完成与 `Counter` 类相同的功能，并且把它用在 `for` 循环中。

```
>>> def counter_generator(low, high):
...     while low <= high:
...         yield low
...         low += 1
...
>>> for i in counter_generator(5,10):
...     print(i, end=' ')
...
5 6 7 8 9 10
```

在 `While` 循环中，每当执行到 `yield` 语句时，返回变量 `low` 的值并且生成器状态转为挂起。在下一次调用生成器时，生成器从之前冻结的地方恢复执行然后变量 `low` 的值增一。生成器继续 `while` 循环并且再次来到 `yield` 语句...

当你调用生成器函数时它返回一个生成器对象。如果你把这个对象传入 `dir()` 函数，你会在返回的结果中找到 `__iter__` 和 `__next__` 两个方法名。

```
>>> c = counter_generator(5,10)
>>> dir(c)
['__class__', '__del__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__lt__', '__name__', '__ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'close', 'gi_code', 'gi_frame', 'gi_running', 'send', 'throw']
>>>
```

我们通常使用生成器进行惰性求值。这样使用生成器是处理大数据的好方法。如果你不想在内存中加载所有数据，你可以使用生成器，一次只传递给你一部分数据。

`os.path.walk()` 函数是最典型的这样的例子，它使用一个回调函数和当前的 `os.walk` 生成器。使用生成器实现节约内存。

我们可以使用生成器产生无限多的值。以下是一个这样的例子。

```
>>> def infinite_generator(start=0):
...     while True:
...         yield start
...         start += 1
...
>>> for num in infinite_generator(4):
...     print(num, end=' ')
...     if num > 20:
...         break
...
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

如果我们回到 `my_generator()` 这个例子，我们会发现生成器的一个特点：它们是不可重复使用的。

```
>>> g = my_generator()
>>> for c in g:
...     print(c)
...
Inside my generator
a
b
c
>>> for c in g:
...     print(c)
...
...
```

一个创建可重复使用生成器的方式是不保存任何状态的基于对象的生成器。任何一个生成数据的含有 `__iter__` 方法的类都可以用作对象生成器。在下面的例子中我们重新创建了 `counter` 生成器。

```
>>> class Counter(object):
...     def __init__(self, low, high):
...         self.low = low
...         self.high = high
...     def __iter__(self):
...         counter = self.low
...         while self.high >= counter:
...             yield counter
...             counter += 1
...
>>> gobj = Counter(5, 10)
>>> for num in gobj:
...     print(num, end=' ')
...
5 6 7 8 9 10
>>> for num in gobj:
...     print(num, end=' ')
...
5 6 7 8 9 10
```

2.3 生成器表达式

在这一节我们学习生成器表达式 (*Generator expressions*)，生成器表达式是列表推导式和生成器的一个高性能，内存使用效率高的推广。

举个例子，我们尝试对 1 到 9 的所有数字进行平方求和。

```
>>> sum([x*x for x in range(1,10)])
```

这个例子实际上首先在内存中创建了一个平方数值的列表，然后遍历这个列表，最终求和后释放内存。你能理解一个大列表的内存占用情况是怎样的。

我们可以通过使用生成器表达式来节省内存使用。

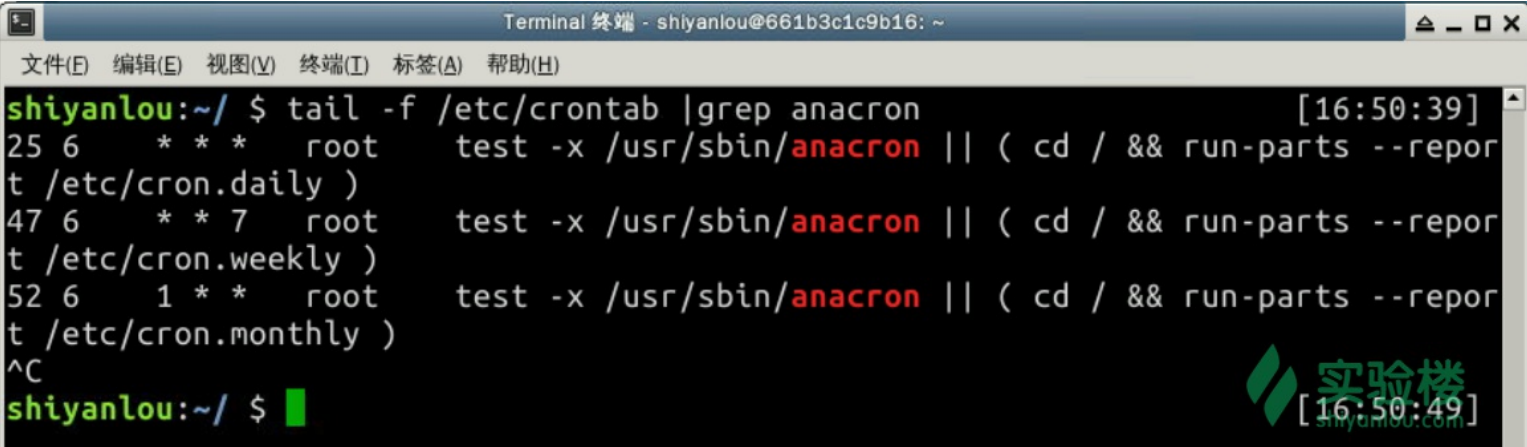
```
>>> sum(x*x for x in range(1,10))
```

生成器表达式的语法要求其总是直接在一对括号内，并且不能在两边有逗号。这基本上意味着下面这些例子都是有效的生成器表达式用法示例：

```
>>> sum(x*x for x in range(1,10))
285
>>> g = (x*x for x in range(1,10))
>>> g
<generator object <genexpr> at 0x7fc559516b90>
```

我们可以把生成器和生成器表达式联系起来，在下面的例子中我们会读取文件 `/var/log/cron` 并且查看任意指定任务（例中我们搜索 `'anacron'`）是否成功运行。

我们可以用 shell 命令 `tail -f /etc/crontab |grep anacron` 完成同样的事（按 `Ctrl + C` 终止命令执行）。

A terminal window titled "Terminal 终端 - shiyanlou@661b3c1c9b16: ~" showing the command `tail -f /etc/crontab |grep anacron` being executed. The output shows three lines from the crontab file, each containing the word `anacron` in red. The lines are: `25 6 * * * root test -x /usr/sbin/anacron || (cd / && run-parts --report /etc/cron.daily)`, `47 6 * * 7 root test -x /usr/sbin/anacron || (cd / && run-parts --report /etc/cron.weekly)`, and `52 6 1 * * root test -x /usr/sbin/anacron || (cd / && run-parts --report /etc/cron.monthly)`. The terminal also shows the prompt `shiyanlou:~/ $` and a green cursor. A watermark "实验楼" is visible in the bottom right corner.

```
Terminal 终端 - shiyanlou@661b3c1c9b16: ~
文件(F) 编辑(E) 视图(V) 终端(T) 标签(A) 帮助(H)
shiyanlou:~/ $ tail -f /etc/crontab |grep anacron [16:50:39]
25 6 * * * root test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.daily )
47 6 * * 7 root test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.weekly )
52 6 1 * * root test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.monthly )
^C
shiyanlou:~/ $
```

```
>>> jobtext = 'anacron'
>>> all = (line for line in open('/etc/crontab', 'r'))
>>> job = ( line for line in all if line.find(jobtext) != -1)
>>> text = next(job)
>>> text
'25 6\t* * *\troot\ttest -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.daily )\n'
>>> text = next(job)
>>> text
'47 6\t* * 7\troot\ttest -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.weekly )\n'
>>> text = next(job)
>>> text
'52 6\t1 * *\troot\ttest -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.monthly )\n'
```

你可以写一个 `for` 循环遍历所有行。

2.4 闭包

闭包 (*Closures*) 是由另外一个函数返回的函数。我们使用闭包去除重复代码。在下面的例子中我们创建了一个简单的闭包来对数字求和。

```
>>> def add_number(num):  
...     def adder(number):  
...         #adder 是一个闭包  
...         return num + number  
...     return adder  
...  
>>> a_10 = add_number(10)  
>>> a_10(21)  
31  
>>> a_10(34)  
44  
>>> a_5 = add_number(5)  
>>> a_5(3)  
8
```

`adder` 是一个闭包，把一个给定的数字与预定义的一个数字相加。

2.5 装饰器

装饰器 (*Decorators*) 用来给一些对象动态的添加一些新的行为，我们使用过的闭包也是这样的。

我们会创建一个简单的示例，将在函数执行前后打印一些语句。


```
>>> def my_decorator(func):
...     def wrapper(*args, **kwargs):
...         print("Before call")
...         result = func(*args, **kwargs)
...         print("After call")
...         return result
...     return wrapper
...
>>> @my_decorator
... def add(a, b):
...     #我们的求和函数
...     return a + b
...
>>> add(1, 3)
Before call
After call
4
```

三、总结

本实验我们学习了迭代器和生成器以及装饰器这几个高级特性的定义方法和用法，也了解了怎样使用生成器表达式和怎样定义闭包。

**本课程内容，由作者授权实验楼发布，未经允许，禁止转载、下载及非法传播。*

上一节：[PEP8 代码风格指南 \(/courses/596/labs/2049/document\)](/courses/596/labs/2049/document)

下一节：[Virtualenv \(/courses/596/labs/2051/document\)](/courses/596/labs/2051/document)