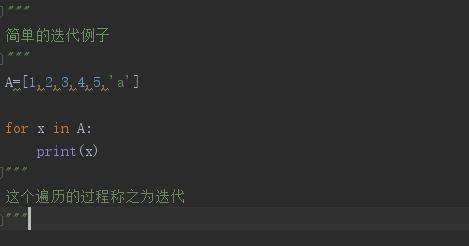
# Python-yield学习与用法

1. 带有yield的函数在Python中被称之为generator（生成器）。

同时在我们学习Python-yield前，我们首先需要了解迭代“Iterables”

如果给定一个list或tuple，我们可以通过for循环来遍历这个list或tuple，这种遍历我们称之为迭代。

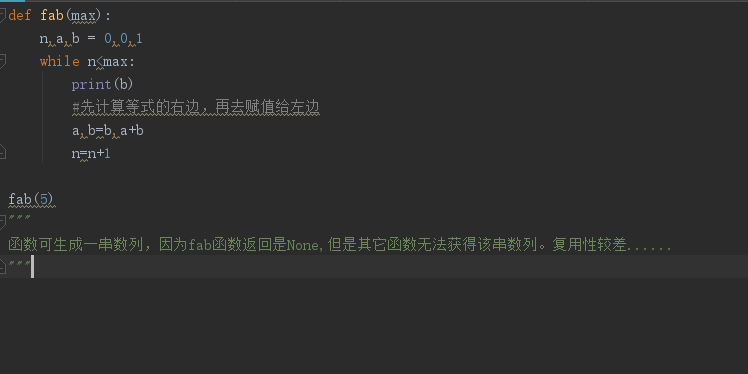


输出结果：

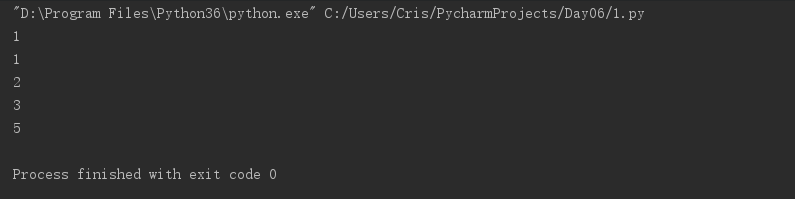


假设我们需要写出一个函数，得到一个数列。

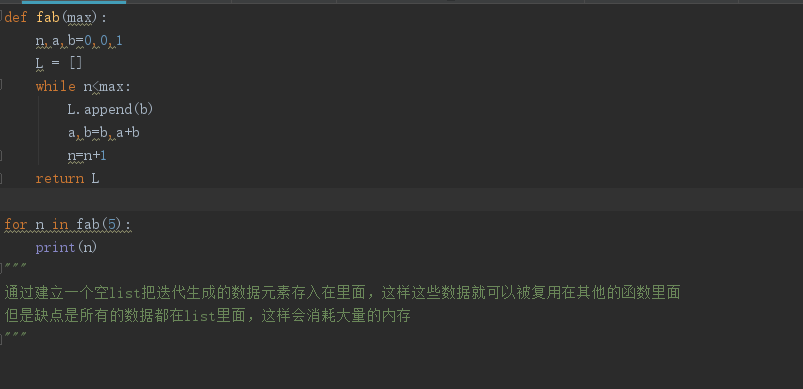
Demo：



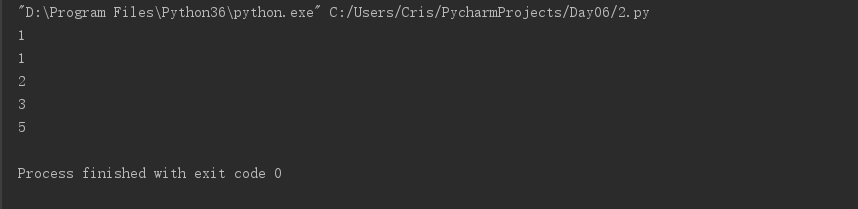
运行结果：



这个时候我们为了将得到的数列这个数据存储起来，以致于可以被其他的函数调用，所以我们拓展出下面的demo;



运行结果：



Python创建列表时，解释器在内存中生成一个类似数组的数据结构储存数据，数据项自下而上存储。

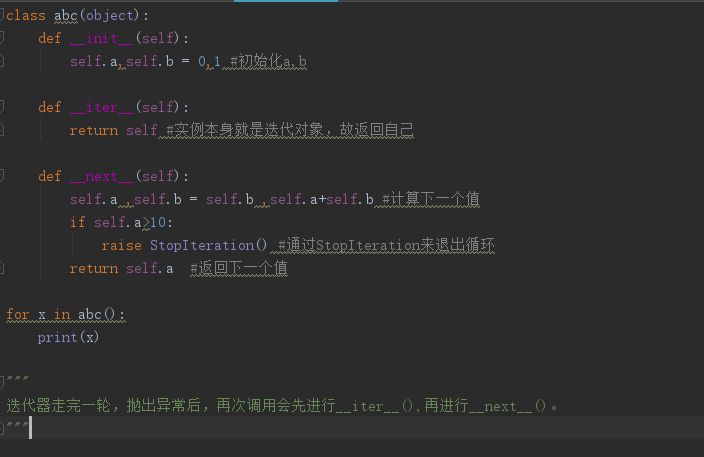
它的缺陷就是所有数据都存在内存中，如果有海量数据的话会非常消耗内存。

这个时候我们需要创建一个迭代器来解决这样的问题。

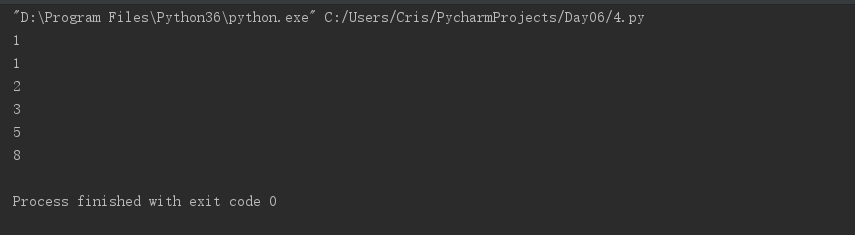
Python中关于迭代有两个概念，第一个是Iterable（迭代），第二个是Iterator（迭代器），协议规定Iterable的\_\_iter\_\_方法会返回一个Iterator, Iterator的\_\_next\_\_方法会返回下一个迭代对象，如果迭代结束则抛出StopIteration异常

迭代器就是重复地做一些事情，可以简单的理解为循环，在python中实现了\_\_iter\_\_方法的对象是可迭代的，实现了\_\_next\_\_()方法的对象是迭代器，这样说起来有点拗口，实际上要想让一个迭代器工作，至少要实现\_\_iter\_\_方法和next方法。很多时候使用迭代器完成的工作使用列表也可以完成，但是如果有很多值列表就会占用太多的内存，而且使用迭代器也让我们的程序更加通用、优雅、pythonic（注释：具有python的个性特色）。

Demo：



运行结果：



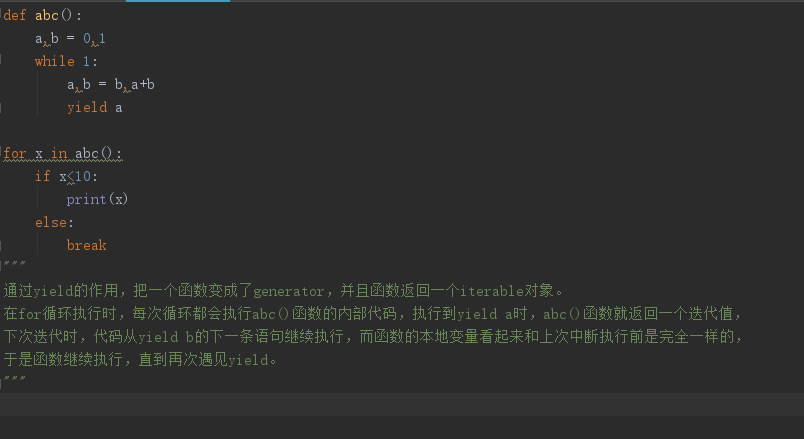
在Python里iterable被认为是一类对象，这类对象能够一次返回它的一个成员（元素）。抽象一点就是适合迭代的对象实际上，任何具有\_\_iter\_\_()或\_\_getitem\_\_()方法的对象，Python就认为它是一个iterator。

Python里有大量内置的iterable类型，如： list，str，tuple，dict，file，xrange等。使用内置的iter()函数来生成iterator。

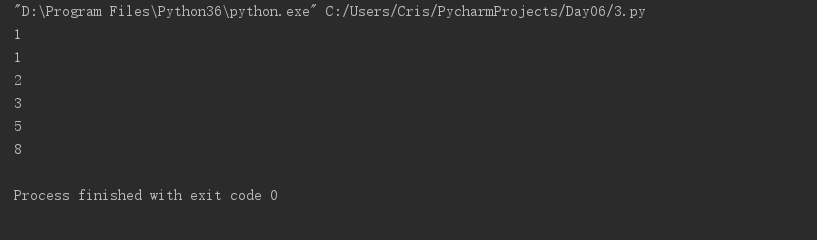
如果一个类想被用于for ... in循环，类似list或tuple那样，就必须实现一个\_\_iter\_\_()方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的\_\_next\_\_()方法拿到循环的下一个值，直到遇到StopIteration错误时退出循环。

迭代器是一个对象，而生成器是一个函数，迭代器和生成器是python中两个非常强大的特性，编写程序时你可以不使用生成器达到同样的效果，但是生成器让你的程序更加pythonic。创建生成器非常简单，只要在函数中加入yield语句即可。函数中每次使用yield产生一个值，函数就返回该值，然后停止执行，等待被激活，被激活后继续在原来的位置执行。

Demo:



运行结果;



一个带有 yield 的函数就是一个 generator(生成器)，它和普通函数不同，生成一个 generator 看起来像函数调用，但不会执行任何函数代码，直到对其调用 next()（在 for 循环中会自动调用 next()）才开始执行。虽然执行流程仍按函数的流程执行，但每执行到一个 yield 语句就会中断，并返回一个迭代值，下次执行时从 yield 的下一个语句继续执行。看起来就好像一个函数在正常执行的过程中被 yield 中断了数次，每次中断都会通过 yield 返回当前的迭代值。

yield 的好处是显而易见的，把一个函数改写为一个 generator 就获得了迭代能力，比起用类的实例保存状态来计算下一个 next() 的值，不仅代码简洁，而且执行流程异常清晰。