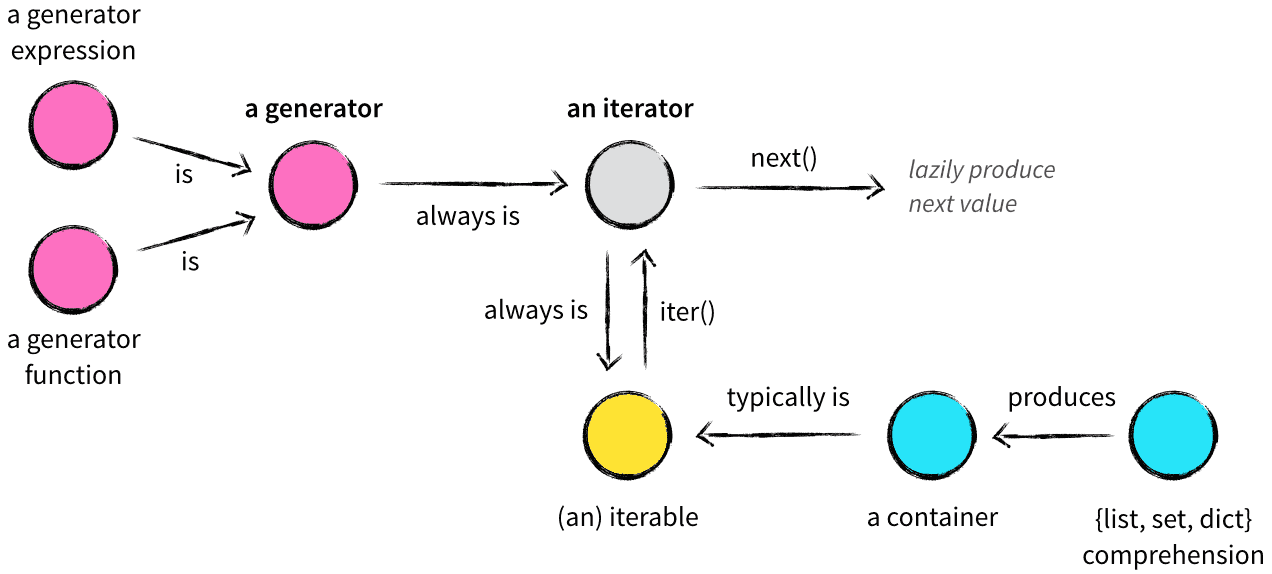
Python 迭代对象、迭代器、生成器

http://nvie.com/posts/iterators-vs-generators/

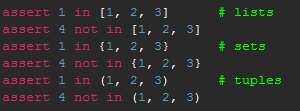
在Python中，迭代对象、迭代器、生成器是常用但又容易混淆的几个概念。

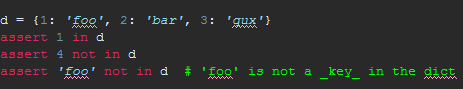


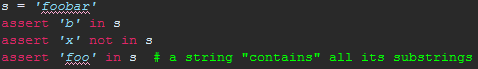
1. **容器(container)**

容器是一种把多个元素组织在一起的数据结构，容器中的元素可以逐个地迭代获取，可以用in, not in关键字判断元素是否包含在容器中。通常这类数据结构把所有的元素存储在内存中（也有一些特例，并不是所有的元素都放在内存，比如**迭代器和生成器**对象）在**Python**中，常见的容器对象有：

* list, deque, ….
* set, frozensets, ….
* dict, defaultdict, OrderedDict, Counter, ….
* tuple, namedtuple, …
* str

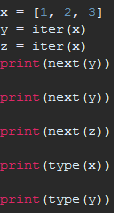
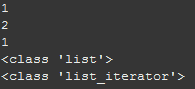
容器比较容易理解，因为你就可以把它看作是一个盒子、一栋房子、一个柜子，里面可以塞任何东西。从技术角度来说，当它可以用来询问某个元素是否包含在其中时，那么这个对象就可以认为是一个容器，比如 list, set, tuples都是容器对象：

询问某元素是否在dict中用dict的中key：

询问某substring是否在string中：

尽管绝大多数容器都提供了某种方式来获取其中的每一个元素，但这并不是容器本身提供的能力，而是**可迭代对象**赋予了容器这种能力，当然并不是所有的容器都是可迭代的，比如：[Bloom filter](https://zh.wikipedia.org/wiki/%E5%B8%83%E9%9A%86%E8%BF%87%E6%BB%A4%E5%99%A8)，虽然Bloom filter可以用来检测某个元素是否包含在容器中，但是并不能从容器中获取其中的每一个值，因为Bloom filter压根就没把元素存储在容器中，而是通过一个散列函数映射成一个值保存在数组中。

1. **可迭代对象(iterable)**

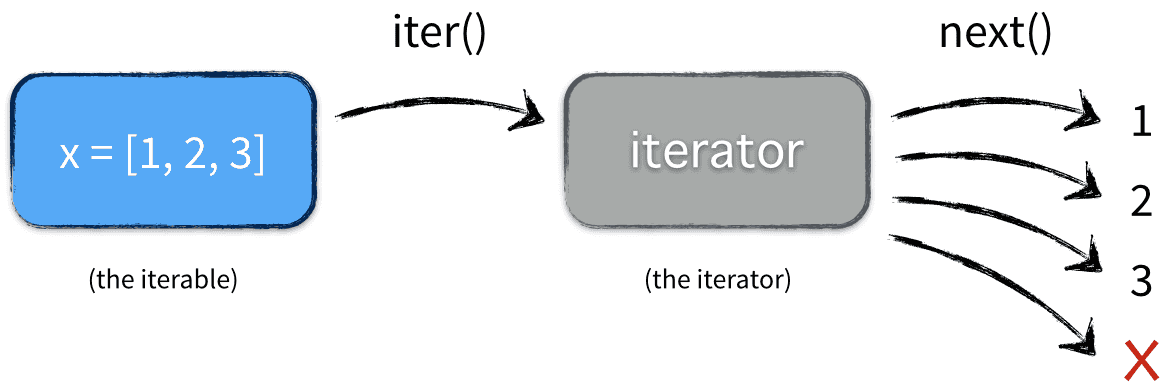
刚才说过，很多容器都是可迭代对象，此外还有更多的对象同样也是可迭代对象，比如处于打开状态的files，sockets等等。但凡是可以返回一个迭代器的对象都可称之为可迭代对象，听起来可能有点困惑，没关系，先看一个例子：

这里x是一个可迭代对象，可迭代对象和容器一样是一种通俗的叫法，并不是指某种具体的数据类型，list是可迭代对象，dict是可迭代对象，set也是可迭代对象。y和z是两个独立的迭代器，迭代器内部持有一个状态，该状态用于记录当前迭代所在的位置，以方便下次迭代的时候获取正确的元素。迭代器有一种具体的迭代器类型，比如list\_iterator，set\_iterator。可迭代对象实现了\_\_iter\_\_方法，该方法返回一个迭代器对象。

当运行代码：

x = [1, 2, 3]

for elem in x:...

实际执行情况是：

反编译该段代码，你可以看到解释器显示地调用GET\_ITER指令，相当于调用iter(x)，FOR\_ITER指令就是调用next()方法，不断地获取迭代器中的下一个元素，但是你没法直接从指令中看出来，因为他被解释器优化过了。

>>> **import** dis

>>> x **=** [1, 2, 3]

>>> dis**.**dis('for \_ in x: pass')

1 0 SETUP\_LOOP 14 (to 17)

3 LOAD\_NAME 0 (x)

6 GET\_ITER

>> 7 FOR\_ITER 6 (to 16)

10 STORE\_NAME 1 (\_)

13 JUMP\_ABSOLUTE 7

>> 16 POP\_BLOCK

>> 17 LOAD\_CONST 0 (None)

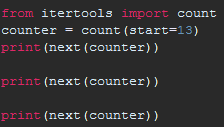
20 RETURN\_VALUE

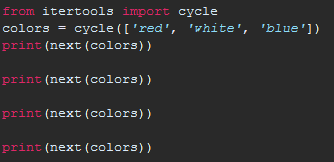
1. **迭代器(iterator)**

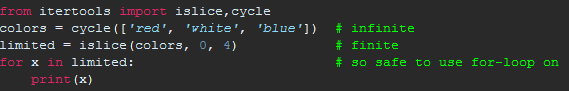
那么什么迭代器呢？它是一个带状态的对象，他能在你调用next()方法的时候返回容器中的下一个值，任何实现了\_\_iter\_\_和\_\_next\_\_()（python2中实现next()）方法的对象都是迭代器，\_\_iter\_\_返回迭代器自身，\_\_next\_\_返回容器中的下一个值，如果容器中没有更多元素了，则抛出StopIteration异常，至于它们到底是如何实现的这并不重要。

所以，迭代器就是实现了工厂模式的对象，它在你每次你询问要下一个值的时候给你返回。有很多关于迭代器的例子，比如itertools函数返回的都是迭代器对象。

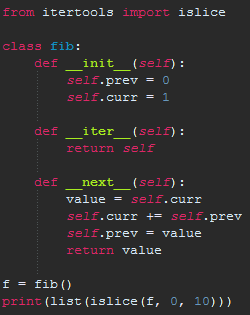
C:\Users\xiaoguan\AppData\Local\Temp\1522737024(1).png生成无限序列：



C:\Users\xiaoguan\AppData\Local\Temp\1522737246(1).png从一个有限序列中生成无限序列：

C:\Users\xiaoguan\AppData\Local\Temp\1522737507(1).png从无限的序列中生成有限序列：

C:\Users\xiaoguan\AppData\Local\Temp\1522738056(1).png为了更直观地感受迭代器内部的执行过程，我们自定义一个迭代器，以斐波那契数列为例：



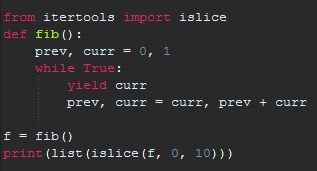
Fib既是一个可迭代对象（因为它实现了\_\_iter\_\_方法），又是一个迭代器（因为实现了\_\_next\_\_方法）。实例变量prev和curr用户维护迭代器内部的状态。每次调用next()方法的时候做两件事：

1. 为下一次调用next()方法修改状态
2. 为当前这次调用生成返回结果

迭代器就像一个懒加载的工厂，等到有人需要的时候才给它生成值返回，没调用的时候就处于休眠状态等待下一次调用。

**4.** **生成器(generator)**

生成器算得上是Python语言中最吸引人的特性之一，生成器其实是一种特殊的迭代器，不过这种迭代器更加优雅。它不需要再像上面的类一样写\_\_iter\_\_()和\_\_next\_\_()方法了，只需要一个yiled关键字。 生成器一定是迭代器（反之不成立），因此任何生成器也是以一种懒加载的模式生成值。用生成器来实现斐波那契数列的例子是：

C:\Users\xiaoguan\AppData\Local\Temp\1522738731(1).pngfib就是一个普通的**python**函数，它特殊的地方在于函数体中没有return关键字，函数的返回值是一个生成器对象。当执行f=fib()返回的是一个生成器对象，此时函数体中的代码并不会执行，只有显示或隐示地调用next的时候才会真正执行里面的代码。

生成器在**Python**中是一个非常强大的编程结构，可以用更少地中间变量写流式代码，此外，相比其它容器对象它更能节省内存和CPU，当然它可以用更少的代码来实现相似的功能。关键字yield是核心，我们来了解下代码运行的过程：

当执行 f = fib() 的时候，一个生成器创建了，但是只是进行了空白的初始化过程，并未实际执行任何代码。

紧接着生成器实例被分割包装了通过islice方法，但结果仍是一个迭代器。之后，list方法生成一个list通过对islice使用next() 方法，而islice则会对f调用n ext()方法。

在对f调用next()的时候，第一次先执行第一行命令，然后进入while循环，当第一次执行到yield语句时，就暂停执行了，并返回当前的curr值。再次对f调用next()时，重启循环语句，继续从上次暂停的位置向下执行其它语句，直到再次遇到yield语句。这就是生成器的工作过程。

**5.** **总结**

* 容器是一系列元素的集合，str、list、set、dict、file、sockets对象都可以看作是容器，容器都可以被迭代（用在for，while等语句中），因此他们被称为可迭代对象。
* 可迭代对象实现了\_\_iter\_\_方法，该方法返回一个迭代器对象。
* 迭代器持有一个内部状态的字段，用于记录下次迭代返回值，它实现了\_\_next\_\_和\_\_iter\_\_方法，迭代器不会一次性把所有元素加载到内存，而是需要的时候才生成返回结果。
* 生成器是一种特殊的迭代器，它的返回值不是通过return而是用yield。

附录：

python的迭代器为什么一定要实现\_\_iter\_\_方法？

一般迭代器里的\_\_iter\_\_下只是return self，为什么还要保留这个方法呢？

这是个和多态有关的问题，Python中关于迭代有两个概念，第一个是Iterable，第二个是Iterator，协议规定Iterable的\_\_iter\_\_方法会返回一个Iterator, Iterator的\_\_next\_\_方法（Python 2里是next）会返回下一个迭代对象，如果迭代结束则抛出StopIteration异常。  
同时，Iterator自己也是一种Iterable，所以也需要实现Iterable的接口，也就是\_\_iter\_\_，这样在for当中两者都可以使用。Iterator的\_\_iter\_\_只需要返回自己就行了。

Python中许多方法直接返回iterator，比如itertools里面的izip等方法，如果Iterator自己不是Iterable的话，就很不方便，需要先返回一个Iterable对象，再让Iterable返回Iterator。生成器表达式也是一个iterator，显然对于生成器表达式直接使用for是非常重要的。那么为什么不只保留Iterator的接口而还需要设计Iterable呢？许多对象比如list、dict，是可以重复遍历的，甚至可以同时并发地进行遍历，通过\_\_iter\_\_每次返回一个独立的迭代器，就可以保证不同的迭代过程不会互相影响。而生成器表达式之类的结果往往是一次性的，不可以重复遍历，所以直接返回一个Iterator就好。让Iterator也实现Iterable的兼容就可以很灵活地选择返回哪一种。总结来说Iterator实现的\_\_iter\_\_是为了兼容Iterable的接口，从而让Iterator成为Iterable的一种实现。