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。如：list、set、dict、tuple、str。

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

迭代（iteration）：

用简单的话讲，它就是从某个地方（比如一个列表）取出一个元素的过程。当我们使用一个循环来遍历某个东西时，这个过程本身就叫迭代。

迭代器（iterator）：

迭代器是可以遍历一个容器的对象，只要定义了next(Python2) 或者\_\_next\_\_方法，它就是一个迭代器。

可迭代对象（iterable）：

Python中任意的对象，只要它定义了可以返回一个迭代器的\_\_iter\_\_方法，或者定义了可以支持下标索引的\_\_getitem\_\_方法，那么它就是一个可迭代对象。简单说，可迭代对象就是能提供迭代器的任意对象。

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。

生成无限序列：

>>> from itertools import count

>>> counter = count(start=13)

>>> next(counter)

13

>>> next(counter)

14

从一个有限序列中生成无限序列：

>>> from itertools import cycle

>>> colors = cycle(['red', 'white', 'blue'])

>>> next(colors)

'red'

>>> next(colors)

'white'

>>> next(colors)

'blue'

>>> next(colors)

'red'

从无限的序列中生成有限序列：

>>> from itertools import islice

>>> colors = cycle(['red', 'white', 'blue']) # infinite

>>> limited = islice(colors, 0, 4) # finite

>>> for x in limited:

... print(x)

red

white

blue

red

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：

class Fib:

def \_\_init\_\_(self):

self.prev = 0

self.curr = 1

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

value = self.curr

self.curr += self.prev

self.prev = value

return value

>>> f = Fib()

>>> list(islice(f, 0, 10))

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

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：

为下一次调用 next() 方法修改状态

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。

<http://www.open-open.com/lib/view/open1463668934647.html>

<https://eastlakeside.gitbooks.io/interpy-zh/content/Generators/>

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，但是你只能对其迭代一次。生成器一定是迭代器，反之不一定。

这是因为它们并没有把所有的值存在内存中，而是在运行时生成值。你通过遍历来使用它们，要么用一个“for”循环，要么将它们传递给任意可以进行迭代的函数和结构。

生成器最佳应用场景是：你不想同一时间将所有计算出来的大量结果集分配到内存当中，特别是结果集里还包含循环。

注：许多Python 2里的标准库函数都会返回列表，而Python 3都修改成了返回生成器，因为生成器占用更少的资源。

Python有两种不同的方式提供生成器：

生成器函数：常规函数定义，但是，使用yield语句而不是return语句返回结果（它也不需要再像迭代器类一样写 \_\_iter\_\_() 和 \_\_next\_\_() 方法了）。yield语句一次返回一个结果，在每个结果中间，挂起函数的状态，以便下次重它离开的地方继续执行

生成器表达式：类似于列表推导，但是，生成器返回按需产生结果的一个对象，而不是一次构建一个结果列表

生成器函数：

下面是一个计算斐波那契数列的生成器：

# generator version

def fibon(n):

a = b = 1

for i in range(n):

yield a

a, b = b, a + b

函数使用方法如下：

for x in fibon(1000000):

print(x)

用这种方式，我们可以不用担心它会使用大量资源。然而，之前如果我们这样来实现的话：

def fibon(n):

a = b = 1

result = []

for i in range(n):

result.append(a)

a, b = b, a + b

return result

这也许会在计算很大的输入参数时，用尽所有的资源。

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，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。现在就可以动手重构你的代码了，但凡看到类似：

def something():

result = []

for ... in ...:

result.append(x)

return result

都可以用生成器函数来替换：

def iter\_something():

for ... in ...:

yield x

生成器表达式：

使用列表推导，将会一次产生所有结果：

>>> squares = [x\*\*2 for x in range(5)]

>>> squares

[0, 1, 4, 9, 16]

将列表推导的中括号，替换成圆括号，就是一个生成器表达式：

>>> squares = (x\*\*2 for x in range(5))

>>> squares

<generator object at 0x00B2EC88>

>>> next(squares)

0

>>> next(squares)

1

>>> next(squares)

4

>>> list(squares)

[9, 16]

Python不但使用迭代器协议，让for循环变得更加通用。大部分内置函数，也是使用迭代器协议访问对象的。例如， sum函数是Python的内置函数，该函数使用迭代器协议访问对象，而生成器实现了迭代器协议，所以，我们可以直接这样计算一系列值的和：

>>> sum(x \*\* 2 for x in xrange(4))

而不用多此一举的先构造一个列表：

>>> sum([x \*\* 2 for x in xrange(4)])

注意：

1. 我们这就去看看：

my\_string = "Yasoob"

next(my\_string)

# Output: Traceback (most recent call last):

# File "<stdin>", line 1, in <module>

# TypeError: str object is not an iterator

好吧，这不是我们预期的。这个异常说那个str对象不是一个迭代器。对，就是这样！它是一个可迭代对象，而不是一个迭代器。这意味着它支持迭代，但我们不能直接对其进行迭代操作。那我们怎样才能对它实施迭代呢？是时候学习下另一个内置函数，iter。它将根据一个可迭代对象返回一个迭代器对象。这里是我们如何使用它：

my\_string = "Yasoob"

my\_iter = iter(my\_string)

next(my\_iter)

# Output: 'Y'

2. 我们直接来看例子，假设文件中保存了每个省份的人口总数，现在，需要求每个省份的人口占全国总人口的比例。显然，我们需要先求出全国的总人口，然后在遍历每个省份的人口，用每个省的人口数除以总人口数，就得到了每个省份的人口占全国人口的比例。

如下所示：

def get\_province\_population(filename):

with open(filename) as f:

for line in f:

yield int(line)

gen = get\_province\_population('data.txt')

all\_population = sum(gen)

#print all\_population

for population in gen:

print population / all\_population

执行上面这段代码，将不会有任何输出，这是因为，生成器只能遍历一次。在我们执行sum语句的时候，就遍历了我们的生成器，当我们再次遍历我们的生成器的时候，将不会有任何记录。所以，上面的代码不会有任何输出。

因此，生成器的唯一注意事项就是：生成器只能遍历一次。

<https://www.zhihu.com/question/24807364/answer/133606304>

<https://eastlakeside.gitbooks.io/interpy-zh/content/Generators/Generators.html>