通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含 100 万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的 list，从而节省大量的空间。在 Python 中，这种一边循环一边计算的机制，称为生成器：generator。

要创建一个 generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的`[]`改成`()`，就创建了一个 generator：

```
g = (x * x for x in range(10))
```

我们可以直接打印出 list 的每一个元素，但我们怎么打印出 generator 的每一个元素呢？

如果要一个一个打印出来，可以通过`next()`函数获得 generator 的下一个返回值。当然，上面这种不断调用`next(g)`实在是太变态了，正确的方法是使用`for`循环，因为 generator 也是可迭代对象。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        print(b)
        a, b = b, a + b
        n = n + 1
    return 'done'
```

仔细观察，可以看出，`fib`函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似 generator。

也就是说，上面的函数和 generator 仅一步之遥。要把`fib`函数变成 generator，只需要把`print(b)`改为`yield b`就可以了：

```
def fib(max):
```

```
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
    return 'done'
```

这就是定义 generator 的另一种方法。如果一个函数定义中包含 `yield` 关键字，那么这个函数就不再是一个普通函数，而是一个 generator。

这就是定义 generator 的另一种方法。