4.3 使用生成器创建新的迭代模式¶

问题¶

你想实现一个自定义迭代模式，跟普通的内置函数比如 `range()` , `reversed()` 不一样。

解决方案¶

如果你想实现一种新的迭代模式，使用一个生成器函数来定义它。 下面是一个生产某个范围内浮点数的生成器：

```
def frange(start, stop, increment):
    x = start
    while x < stop:
        yield x
        x += increment
```

为了使用这个函数， 你可以用for循环迭代它或者使用其他接受一个可迭代对象的函数(比如 `sum()` , `list()` 等)。示例如下：

```
>>> for n in frange(0, 4, 0.5):
...     print(n)
...
0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
>>> list(frange(0, 1, 0.125))
[0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875]
>>>
```

讨论¶

一个函数中需要有一个 `yield` 语句即可将其转换为一个生成器。 跟普通函数不同的是，生成器只能用于迭代操作。 下面是一个实验，向你展示这样的函数底层工作机制：

```
>>> def countdown(n):
...     print('Starting to count from', n)
...     while n > 0:
...         yield n
...         n -= 1
...     print('Done!')
...

>>> # Create the generator, notice no output appears
>>> c = countdown(3)
>>> c
<generator object countdown at 0x1006a0af0>

>>> # Run to first yield and emit a value
>>> next(c)
Starting to count from 3
3

>>> # Run to the next yield
>>> next(c)
2

>>> # Run to next yield
>>> next(c)
1
```

```
>>> # Run to next yield (iteration stops)
>>> next(c)
Done!
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
StopIteration
>>>
```

一个生成器函数主要特征是它只会回应在迭代中使用到的 *next* 操作。 一旦生成器函数返回退出，迭代终止。我们在迭代中通常使用的for语句会自动处理这些细节，所以你无需担心。