# 生成器、迭代器

先看一个列表：

```
举例： 求列表中每个元素的平方
# 普通列表
l = [i * i for i in range(10000000)]
print(l)
```

> 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，而且如果要创建一个包含100万个元素的列表，会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。
> 想一下：
> 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。

```
# 生成器
g = (i * i for i in range(10000000))
print(g)  # <generator object <genexpr> at 0x7f90291c6430>
```

列表和生成器取数据的方式相似，但内部原理不同：

```
# 列表取数据
l = [i * i for i in range(10000000)]
# 我们在对l遍历的时候，l就已经占用了大量的空间。如果我们只需要前面几个数据，
# 那么就造成了空间浪费。
for i in l:
  print(i)
next(l)  # TypeError: 'list' object is not an iterator （迭代器）
```

```
# 生成器取数据
g = (i * i for i in range(10000000))
print(g)

# 获取方式1
print(next(g))  # 0*0
print(next(g))  # 1*1
print(next(g))  # 2*2

# 获取方式2
# 而且是接着上面的继续运行的。
for i in g:
    print(i)  # 3*3


# 打印结果：
开始使用next()获取生成器g中的数据
0
1
4
开始遍历生成器g
9
16
...
```

> 虽然列表和生成器都可以使用for循环遍历，但是区别在于，在遍历列表之前，列表中保存的就已经是它的一个一个的元素。而遍历生成器之前，生成器保存的是一个算法，只有开始遍历的时候（或者调用next()函数），它才会一次计算出一个元素，直到计算出最后一个结果。
> 不同于列表，生成器保存的是算法，而且一次只能产生一个值，所以消耗的内存大大减小。
>
> 比喻：
> 有个人他每天制作1000个面包，假如他制做一个面包的速度很快，只需要1秒。
> 第一种方式：他先制作出来1000个面包才开始卖，这样会占用店铺里的空间。
> 第二种方式：每来一个客人，他就立马制作一个面包卖出去

> （1s）。这样就不会占用店铺空间。
> 甚至有可能，每天都卖不了1000个包子。。。所以第二种方式显然更合适。

## yield ，就是一个生成器

函数举例：

```
# 普通函数
def get_num():
    nums = []
    for i in range(10000):
        nums.append(i*i)
    return nums


# 使用了生成器的函数
def get_num():
    for i in range(0, 10000):
        print("i:",i)
        yield i*i

nums = get_num()
print(nums)
print(next(nums))
print(next(nums))
print(next(nums))
print(next(nums))
print(next(nums))
print(next(nums))
```

> 迭代器（迭代就是循环）
> 我们已经知道，可以直接作用于for循环的数据类型有以下几种：
> 一类是集合数据类型，如`list`，`tuple`，`dict`，`set`，`str`等
> 一类是`generator`，包括生成器、和带有生成器的函数。
> 这些可以直接作用于for 循环的对象统称为可迭代对象：`Iterable`
> 可以被`next()`函数调用并不断返回下一个值的对象称为迭代器：

> `Iterator`。
> 可以使用`isinstance()`判断一个对象是否为可`Iterable`对象

```
# 拓展：set
nums = [4, 2, 3, 5, 0, 1, 3, 9, 5, 5, 1, 0]
print(set(nums))  # 去重、排序

# 拓展：isinstance():判断类型
print(isinstance(nums, list))  # true

# 判断是不是可迭代对象
from collections.abc import Iterable
print(isinstance([], Iterable))  # true
print(isinstance({}, Iterable))  # true
print(isinstance('abc', Iterable))  # true
print(isinstance((x for x in range(10)), Iterable))  # true
print(isinstance(100, Iterable))  # false

# 判断是不是迭代器
from collections.abc import Iterator

print(isinstance([], Iterator))  # False
print(isinstance({}, Iterator))  # False
print(isinstance('abc', Iterator))  # False
print(isinstance((x for x in range(10)), Iterator))  # True
print(isinstance(100, Iterator))  # false
```

为什么`list`、`dict`、`str`等数据类型不是`Iterator`？

这是因为Python的`Iterator`对象表示的是一个**数据流**，`Iterator`对象可以被`next()`函数调用并不断返回下一个数据，直到没有数据时抛出`StopIteration`错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过`next()`函数实现按需计算下一个数据，所以`Iterator`的计算是惰性的，只有在需要返回下一个数据时它才会计算。

`Iterator`甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的。

集合数据类型如`list`、`dict`、`str`等是可迭代对象（`Iterable`）但不是迭代器（`Iterator`），不过可以通过`iter()`函数获得一个`Iterator`对象。

其实Python3的for循环本质上就是通过不断调用next()函数实现的，例如：

```
for x in [1, 2, 3, 4, 5]:
    pass
```

实际上完全等价于：

```
# 首先获得Iterator对象:
it = iter([1, 2, 3, 4, 5])

# 循环:
while True:
    try:
        # 获得下一个值:
        x = next(it)
    except StopIteration:
        # 遇到StopIteration就退出循环
        break
```

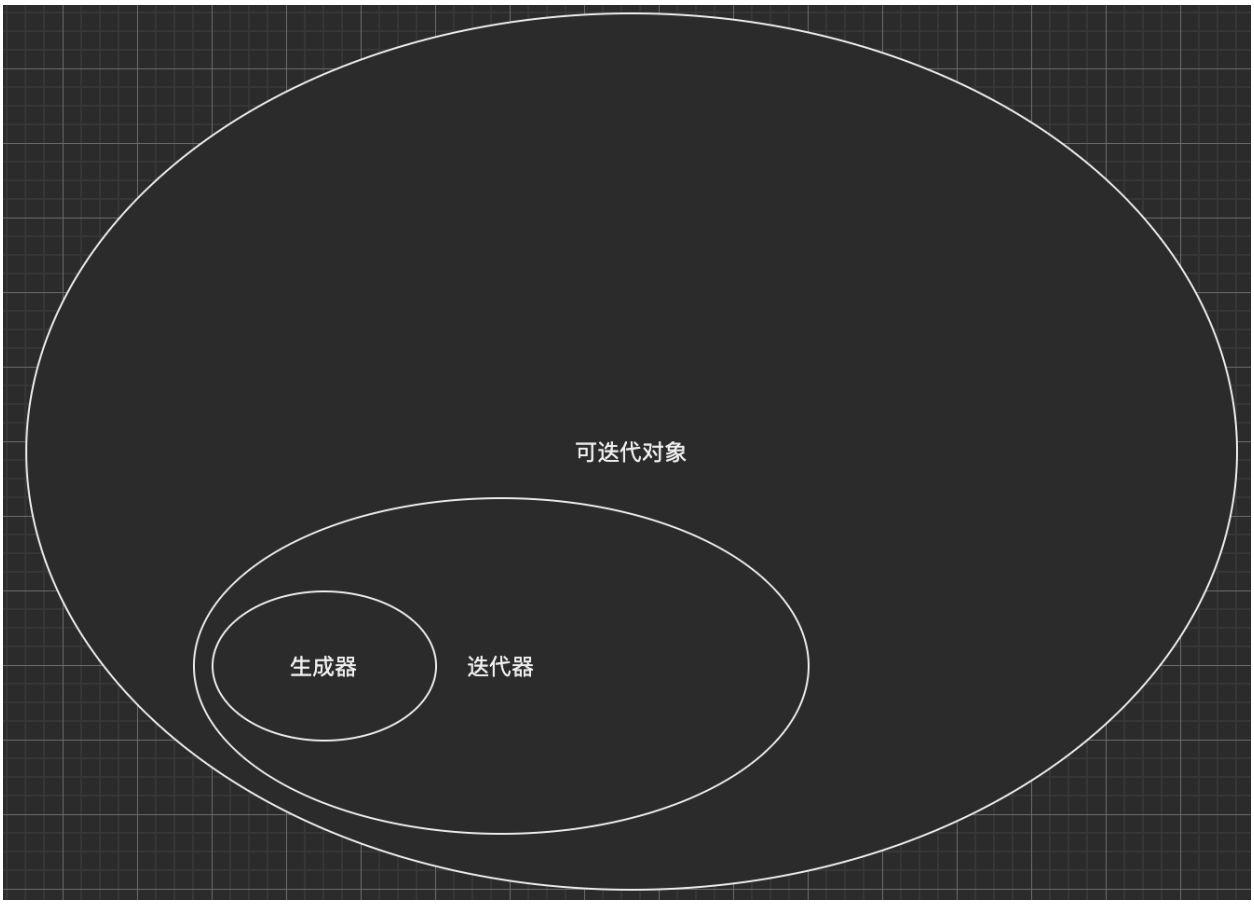
可迭代对象&迭代器小结：

1. 凡是可作用于`for`循环的对象都是可迭代对象（`Iterable`）类型。
2. 凡是可作用于`next()`函数的对象都是迭代器（`Iterator`）类型，它们表示一个惰性计算的序列；
3. 集合数据类型如`list`、`dict`、`str`等是可迭代对象（`Iterable`）但不是迭代器（`Iterator`），不过可以通过`iter()`函数获得一个`Iterator`对象。

对`yield`的小结：

1. 带有`yield`的函数不再是一个普通的函数，而是一个生成器`generator`，可用于迭代。
2. `yield`就是返回一个值，并且记住这个返回的位置。下一次迭代就从这个位置开始。

迭代器、可迭代对象、生成器的关系：




```
yield
  一次计算出一个元素返回，是生成器。
  可以使用next()函数调用，是迭代器
  可以使用for循环遍历，是可迭代对象

l = [1,2,3,4,5]
  可以使用for循环遍历，是可迭代对象
  不可以使用next()函数调用，不是迭代器
```