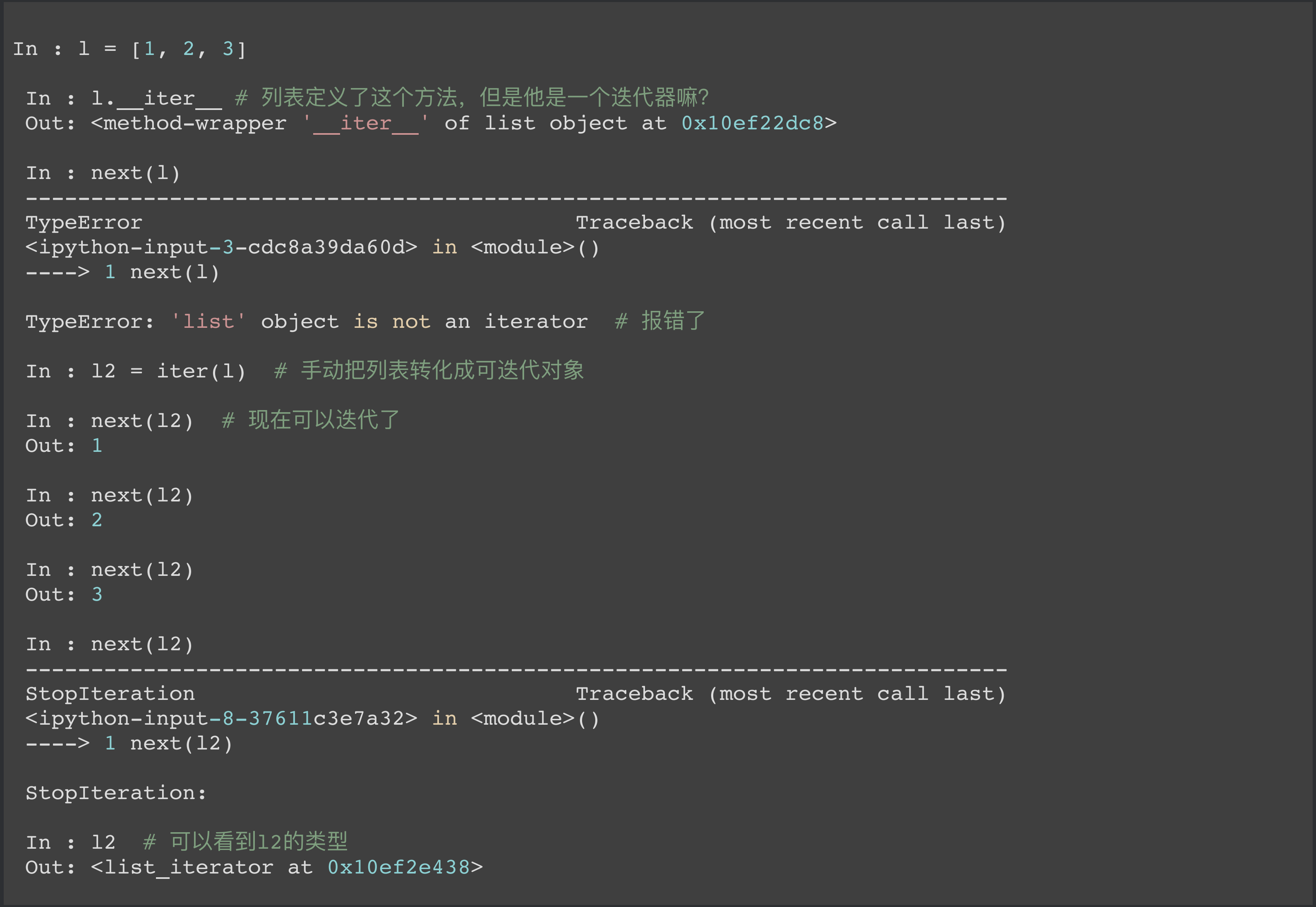
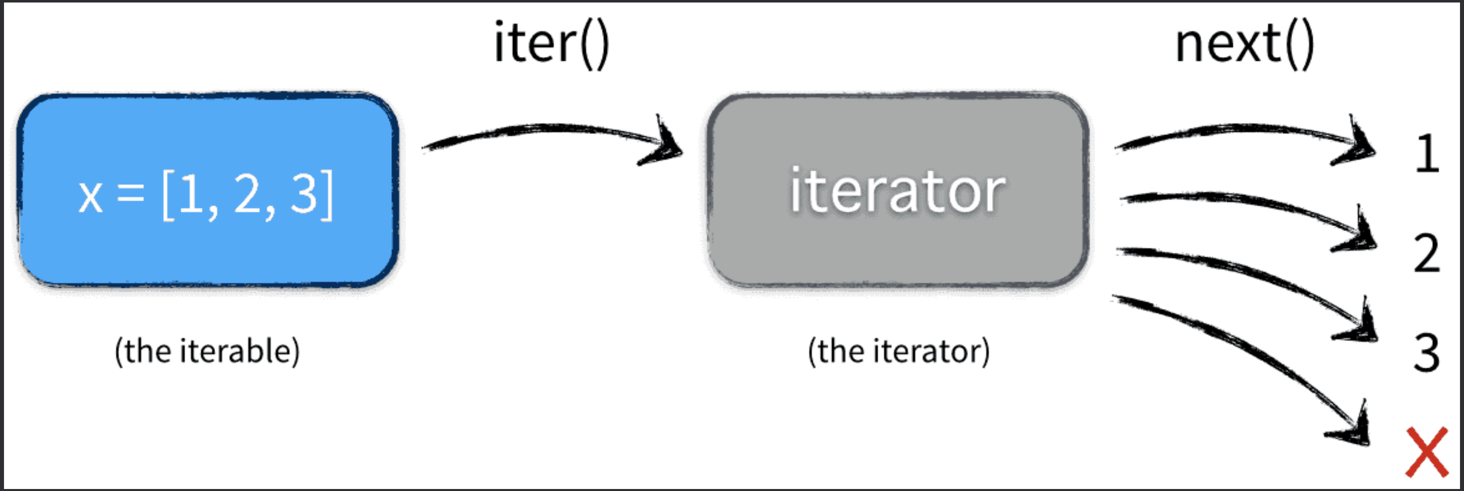
迭代器、生成器和协程

可迭代(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，或者支持下标索引的__getitem__方法，那么它就是一个可迭代对象



```
In : l = [1, 2, 3]

In : l.__iter__ # 列表定义了这个方法，但是他是一个迭代器嘛?
Out: <method-wrapper '__iter__' of list object at 0x10ef22dc8>

In : next(l)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-cdc8a39da60d> in <module>()
----> 1 next(l)

TypeError: 'list' object is not an iterator  # 报错了

In : l2 = iter(l)  # 手动把列表转化成可迭代对象

In : next(l2)  # 现在可以迭代了
Out: 1

In : next(l2)
Out: 2

In : next(l2)
Out: 3

In : next(l2)
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-8-37611c3e7a32> in <module>()
----> 1 next(l2)

StopIteration:

In : l2  # 可以看到l2的类型
Out: <list_iterator at 0x10ef2e438>
```


iter()
next()
x = [1, 2, 3]
iterator
1
2
3
X
(the iterable)
(the iterator)

迭代器(Iterators)

实现了 __iter__ 和 next方法的对象就是迭代器，其中，__iter__ 方法返回迭代器对象本身，next方法返回容器的下一个元素，在没有后续元素时抛出StopIteration异常

PS: 在Python2中要定义的next方法名字不同，应该是__next__

```python
class Fib:
    def __init__(self, max):
        self.a = 0
        self.b = 1
        self.max = max

    def __iter__(self):
        return self

    def __next__(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib
```

```
In : f = Fib(100)

In : for i in f:
...:     print(i)
...:
0
1
1
2
3
5
8
13
21
34
55
89

In : list(Fib(100))
Out: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

```
In : l = iter([1, 2, 3])

In : next(l)
Out: 1

In : next(l)
Out: 2

In : l.__next__()
Out: 3

In : l.__next__()
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-20-731686253790> in <module>()
----> 1 l.__next__()

StopIteration:
```

生成器(Generator)

生成器是一种使用普通函数语法定义的迭代器。生成器和普通函数的区别是使用yield，而不是return返回值

```
In : def my_gen():
...:     yield 1
...:     yield 2
...:

In : g = my_gen()

In : next(g)
Out: 1

In : g.__next__()
Out: 2

In : for i in my_gen():
...:     print(i)
...:
1
2
```

生成器表达式

```
In : g = (i for i in range(10) if i % 2)

In : g
Out: <generator object <genexpr> at 0x11219e620>

In : for i in g:
...:     print(i)
...:
1
3
5
7
9
```

协程(Coroutine)

```
In : def coroutine():
...:     print('Start')
...:     x = yield
...:     print(f'Received: {x}')
...:

In : coro = coroutine()

In : coro
Out: <generator object coroutine at 0x11219eaf0>

In : next(coro)
Start

In : coro.send(10)
Received: 10
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-18-7a1f101c1ec1> in <module>()
----> 1 coro.send(10)

StopIteration:
```

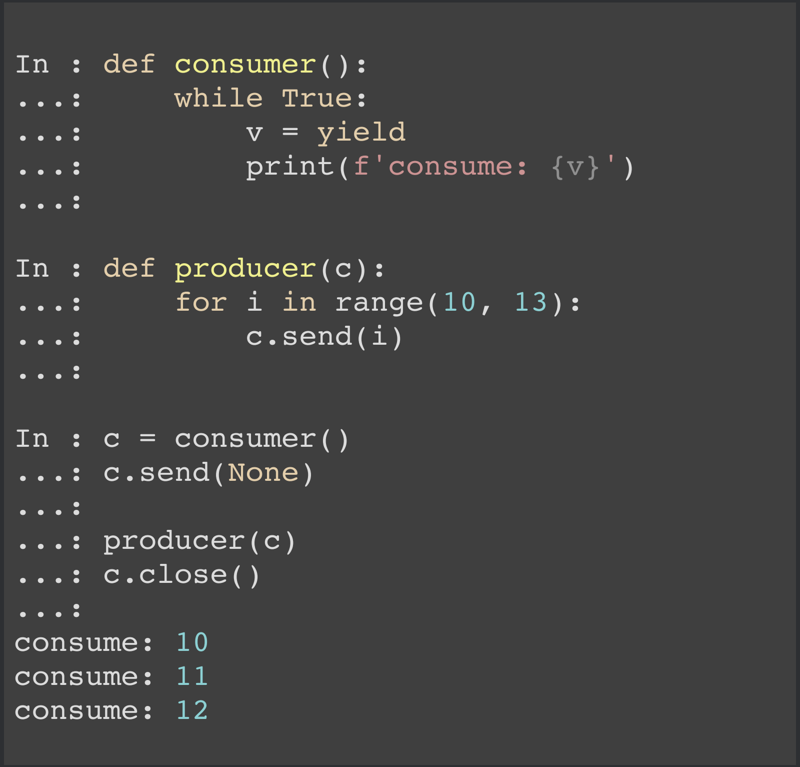
```
In : def coroutine2(a):
...:     print(f'Start: {a}')
...:     b = yield a
...:     print(f'Received: b={b}')
...:     c = yield a + b
...:     print(f'Received: c={c}')
...:

In : coro = coroutine2(1)

In : next(coro)
Start: 1
Out: 1

In : coro.send(2)
Received: b=2
Out: 3

In : coro.send(10)
Received: c=10
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-39-7a1f101c1ec1> in <module>()
----> 1 coro.send(10)

StopIteration:
```

回调例子 👇

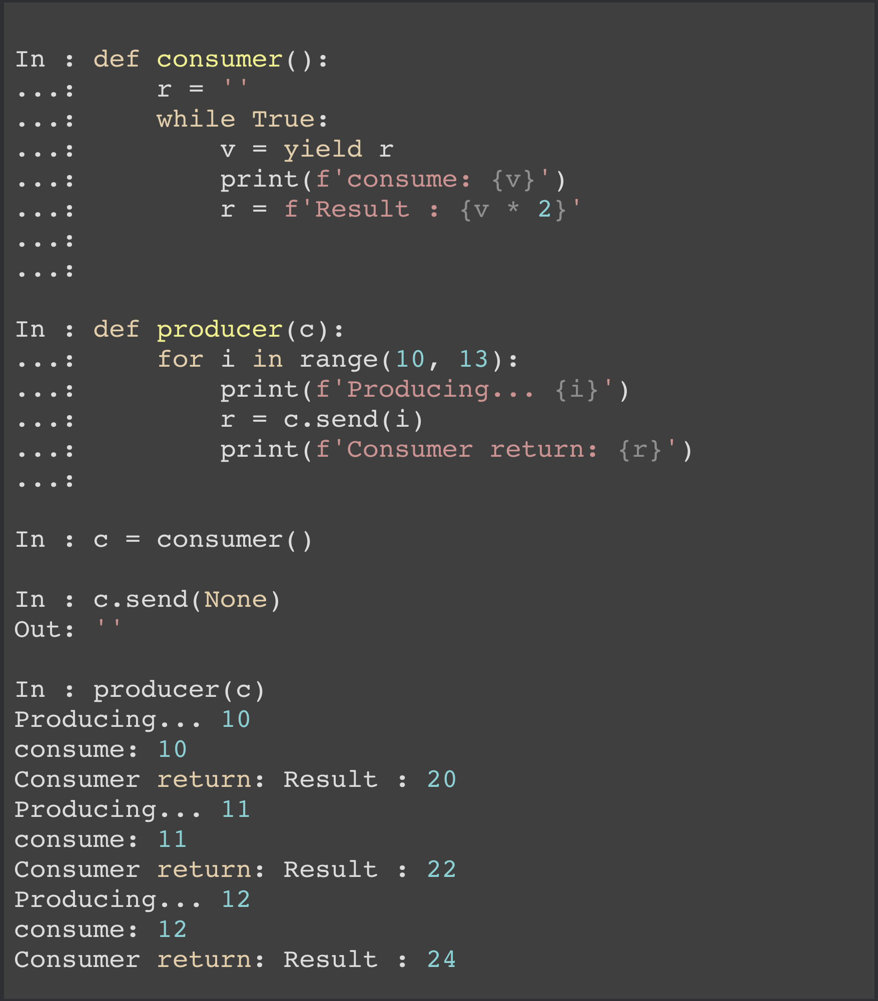
```
In : def framework(logic, callback):
...:     s = logic()
...:     print(f'[FX] logic: {s}')
...:     print(f'[FX] do something...')
...:     callback(f'async: {s}')
...:

In : def logic():
...:     return 'Logic'
...:

In : def callback(s):
...:     print(s)
...:

In : framework(logic, callback)
[FX] logic: Logic
[FX] do something...
async: Logic
```

使用yield改善程序的结构设计

```
In : def framework(logic):
...:     try:
...:         it = logic()
...:         s = next(it)
...:         print(f'[FX] logic: {s}')
...:         print(f'[FX] do something...')
...:         it.send(f'async: {s}')
...:     except StopIteration:
...:         pass
...:

In : def logic():
...:     s = 'Logic'
...:     r = yield s
...:     print(r)
...:

In : framework(logic)
[FX] logic: Logic
[FX] do something...
async: Logic
```



```
In : def consumer():
...:     while True:
...:         v = yield
...:         print(f'consume: {v}')
...:

In : def producer(c):
...:     for i in range(10, 13):
...:         c.send(i)
...:

In : c = consumer()
...: c.send(None)
...:
...: producer(c)
...: c.close()
...:
consume: 10
consume: 11
consume: 12
```



```
In : def consumer():
...:     r = ''
...:     while True:
...:         v = yield r
...:         print(f'consume: {v}')
...:         r = f'Result : {v * 2}'
...:
...:

In : def producer(c):
...:     for i in range(10, 13):
...:         print(f'Producing... {i}')
...:         r = c.send(i)
...:         print(f'Consumer return: {r}')
...:

In : c = consumer()

In : c.send(None)
Out: ''

In : producer(c)
Producing... 10
consume: 10
Consumer return: Result : 20
Producing... 11
consume: 11
Consumer return: Result : 22
Producing... 12
consume: 12
Consumer return: Result : 24
```

延伸阅读

1. http://www.dabeaz.com/coroutines/Coroutines.pdf
2. http://dongweiming.github.io/Expert-Python/#16
3. https://github.com/qyuhen/book/
4. https://zhuanlan.zhihu.com/p/34142963
5. https://nvie.com/posts/iterators-vs-generators/ (迭代器的定义是有问题的)
6. https://docs.python.org/2/library/stdtypes.html#iterator-types