Python迭代

1. 可迭代对象

```
1. 定义
        内部实现了__iter__魔法方法就是可迭代对象,该方法返回一个迭代器
2. 判断
        from collections import Iterable
        a = '123'
        print(isinstance(a, Iterable)) // True,str是可迭代对象.
        print(hasattr(a,'__iter__'))  // True
3. 举例
        列表,元祖,字符串,字典,集合,文件都是.
```

2. 迭代器

```
1. 定义
        内部实现__iter__和__next__魔法方法
2. 自定义迭代器
        使用iter()方法即可
        类中实现__iter__和__next__魔法方法
3. 举例
        文件对象是一个迭代器.
        myList = [11, 10, 1, 2, 9, 3, 8, 4, 5, ]
        obj = reversed(myList)
        print(hasattr(obj, '__next__'))  # True
4. for循环原理
        myList = [1,2,3]
        for i in myList:
            print(i)
        for首先调用list的__iter__方法,将列表变成一个迭代器,然后调用迭代器的__next__方法,取出元素.
        结尾时引发StopIteration异常,for循环捕获break即可.
```

3. 生成器

```
1. 原理
        生成器本质就是一个迭代器,同样拥有__iter__和__next__两个方法,迭代器不同的是,他可以产生延迟操
        作,既不会立即出现结果,而是在你需要的时候才产生.
2. 创建
        生成器表达式:(i for i in rang(10))
        def 函数 yield返回.
3. 解释
        用yield一次返回一个结果,在每个结果之间挂起和继续它们的状态,来自动实现迭代协议.
        yield是一个语法糖,内部实现支持了迭代器协议,同时yield内部是一个状态机,维护着挂起和继续的状态.
```

```
def A(n):
    print ("----生成器开始-----")
    i = 1
```

```
    while n != 0:
        print("开始循环第%d次"%i)
        yield n
        n -= 1
        print ("第%d循环结束"%i)
        i += 1
    print ("----生成器结束----")

a = A(3) # 此时调用生成器函数但是不会执行.
print (a.__next__()) # next(a)
# 调用next才会执行一次,从yield处断开.如果没有找到yield,就会报出StopIteration异常.
```

```
4. 其他方法
        send(value):send()是除next()外另一个恢复生成器的方法,与next不同的是,它需要传入一个参数
                    一般传入None,即send(None)与next()是等效的.
                    该参数传递给yield前面的那个变量,即a.
                    a = yield xxx
        close():关闭生成器,对关闭的生成器后再次调用next或send将抛出StopIteration异常.
```

```
def gen_obj(lst):
    for i in lst:
        a = yield i
        print("a is %s" % a)

myList = [11, 10, 1, 2, 9, 3, 8, 4, 5]
g = gen_obj(myList)
# print(next(g))
g.__next__()
g.send(100)
g.close()
```

4. iter()使用

```
iter(...)
      iter(iterable) -> iterator
      iter(callable, sentinel) -> iterator

      Get an iterator from an object.In the first form,the argument must
      supply its own iterator,or be a sequence.
```

```
1. iter(iterable)
        只有第一个参数的时候,该参数为迭代协议的集合对象(iteration protocol),或者是支持序列协议对象
        (sequence protocol),返回一个迭代对象.
2. iter(callable, sentinel)
        If the second argument,sentinel,is given,then object must be a callable object.
        没有任何参数的可调用对象,当可调用对象的返回值等于sentinel的值时,抛出StopIteration的异常.否
        则返回当前值.
3. 应用场景
        blokc-reader:根据条件中断读取.
        从二进制数据库文件读取固定宽度的块,直到到达文件的末尾.
```

```
from functools import partial

with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        process_line(line)

with open('mydata.db', 'rb') as f:
    for block in iter(partial(f.read, 64), b''):
        process_block(block)
```

5. 序列协议

```
1. 定义
        可变序列:__getitem__(),__len__(),__setitem__(),__delitem__(),列表底层实现了.
        不可变:__getitem__(),__len__(),字符串底层实现了这两个方法.
2. 序列实现取值
```

```
# 序列协议
class Book:
    def __init__(self):
        self.book = ["红楼梦", "西游记", "金瓶梅"]

    def __getitem__(self, item):
        print("start getitem func")
        return self.book[item]


b = Book()
for i in b:
    print(i)
print(book[2])  # 金瓶梅

# for此时会执行getitem方法,索引访问也会执行getitem方法

class Person:
    def __init__(self):
        self.msg = {"name": "lee", "age": 24}

    def __iter__(self):
        for i in self.msg.items():
            yield i

    def __getitem__(self, item):
        print("start Person-getitem func")
        return self.msg[item]

p = Person()
for i in p:
    print(i)
print(p["name"])
```

6. 迭代器实现Iterable

```
# 自定义可迭代对象
class MyIterator(Iterator):
    def __init__(self, lst):
        self.lst = lst
        self.index = 0

    def __next__(self):
        if self.index == len(self.lst):
            raise StopIteration
        city = self.lst[self.index]
        self.index += 1
        return self.getCityMsg(city)

    def getCityMsg(self, city):
        msg = "该%s的信息是...." % city
        return msg


class MyIterable(Iterable):
    def __init__(self, lst):
        self.iterator = MyIterator(lst)

    def __iter__(self):
        return self.iterator


iterable = MyIterable(["北京", "上海", "济南"])
for i in iterable:
    print(i)
```

7. 生成器实现Iterable

```
# 生成器实现可迭代对象
class MyGenerator:
    def __init__(self, lst):
        self.lst = lst

    def __iter__(self):
        for i in self.lst:
            yield self.getCityMsg(i)

    def getCityMsg(self, city):
        msg = "该%s信息是...." % city
        return msg


my_generator = MyGenerator(["上海", "济南", "北京"])
for i in my_generator:
    print(i)
```

8. 关于生成器的思考

1. 存在意义
   比如用列表创建100万个元素，列表容量是有限的，并且只需要访问前几个元素，此时占用了大量的内存空间.
   如果列表可以按照某种算法一个个推算出来，for循环不断推算出后续的元素，从而没有必要创建完整的列表.
   节省空间.
2. 逻辑
   generator保存的是算法，通过算法后续得出结果.

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

for i in fib(10):
    print(i)

# while实现
f = fib(6)
while True:
    try:
        value = next(f)
        print(value)
    except StopIteration as e:
        print(e.value)
        break
```

9. 生成器模拟生消模型

```
def consumer():
    r = ''
    while True:
        n = yield r
        if not n:
            return
        print('[CONSUMER] Consuming %s...' % n)
        r = '200 OK'

def produce(c):
    c.send(None)
    n = 0
    while n < 5:
        n = n + 1
        print('[PRODUCER] Producing %s...' % n)
        r = c.send(n)
        print('[PRODUCER] Consumer return: %s' % r)
    c.close()
```

```
c = consumer()
produce(c)
```