小强理解三大器练习

201 直观理解 yield

要想通俗理解 `yield`，可结合普通函数的返回值关键字 `return`，yield便是一种特殊的`return`.

说是特殊的 `return`，是因为执行遇到 `yield` 时，立即返回，这是与 `return` 的相似之处。

不同之处在于：下次进入函数时直接到 `yield` 的下一个语句，而 `return` 后再进入函数，还是从函数体的第一行代码开始执行。

带 `yield` 的函数是生成器，通常与 `next函数` 结合用。下次进入函数，意思是使用 `next函数` 进入到函数体内。

举个例子说明 `yield` 的基本用法。

下面是被熟知的普通函数 `f`，`f()` 就会立即执行函数：

复制
```
In [1]: def f():
   ...:     print('enter f...')
   ...:     return 'hello'

In [2]: ret = f()
enter g...

In [3]: ret
Out[3]: 'hello'
```

但是，注意观察下面新定义的函数 `f`，因为带有 `yield`，所以是生成器函数 `f`

复制
```
def f():
    print('enter f...')
    yield 4
    print('i am next sentence of yield')
```

执行 `f`，并未打印任何信息：

复制
```
g = f()
```

只得到一个生成器对象 `g`

再执行 `next(g)` 时，输出下面信息：

复制
```
In [18]: next(g)
enter f...
Out[18]: 4
```

再执行 `next(g)` 时，输出下面信息：

复制
```
In [23]: next(g)
i am next sentence of yield
---------------------------------------------------------------------------
--
StopIteration                             Traceback (most recent call las
t)
<ipython-input-23-e734f8aca5ac> in <module>
----> 1 next(g)

StopIteration:
```

输出信息 `i am next sentence of yield`，表明它直接进入yield的下一句。

同时，抛出一个异常：`StopIteration`，意思是已经迭代结束。

202 yield与生成器

函数带有 `yield`，就是一个生成器，英文 `generator`，它的重要优点：**节省内存。**

可能有些读者不理解怎么做到节省内存的？下面看一个例子。

首先定义一个函数 `myrange`：

复制
```
def myrange(stop):
    start = 0
    while start < stop:
        yield start
        start += 1
```

使用生成器 `myrange`：

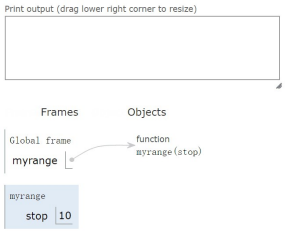
复制
```
for i in myrange(10):
    print(i)
```

返回结果：

复制
```
0
1
2
3
4
5
6
7
8
9
```

整个过程空间复杂度都是 O(1)，这得益于 `yield` 关键字，**遇到就返回且再进入执行下一句**的机制。

以上完整过程，动画演示如下：



如果不使用 `yield`，也就是使用普通函数，如下定义 `myrange`：

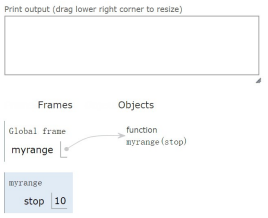
复制

```
def myrange(stop):
    start = 0
    result = []
    while start < stop:
        result.append(start)
        start += 1
    return result

for i in myrange(10):
    print(i)
```

如果不使用 `yield` 关键字，用于创建一个序列的 `myrange` 函数，空间复杂度是 `O(n)`.

以上完整过程，动画演示如下：



203 yield 和 send函数

带 `yield` 的生成器对象里还封装了一个 `send` 方法。下面的例子：

复制

```
def f():
    print('enter f...')
    while True:
        result = yield 4
        if result:
            print('send me a value is:%d'%(result,))
            return
        else:
            print('no send')
```

按照如下调用：

复制

```
g = f()
print(next(g))
print('ready to send')
print(g.send(10))
```

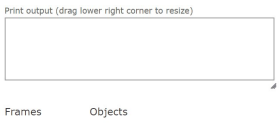
输出结果：

复制

```
enter f...
4
ready to send
send me a value is:10
---------------------------------------------------------------------------
--
StopIteration                             Traceback (most recent call las
t)
<ipython-input-43-0f6c10a3ae1e> in <module>
      2 print(next(g))
      3 print('ready to send')
----> 4 print(g.send(10))

StopIteration:
```

以上完整过程，动画演示如下：



分析输出的结果：

- g = f(): 创建生成器对象，什么都不打印
- print(next(g)): 进入 f，打印 `enter f...`，并 yield 后返回值 4，并打印 4

- print('ready to send')
- print(g.send(10)): send 值 10 赋给 result，执行到上一次 `yield` 语句的后一句打印出 `send me a value is:10`
- 遇到 `return` 后返回，因为f是生成器，同时提示 `StopIteration`.

通过以上分析，能体会到 `send` 函数 的用法：它传递给 `yield` 左侧的 `result` 变量。

`return` 后抛出迭代终止的异常，此处可看作是正常的提示。

理解以上后，再去理解 Python 高效的 协程 机制就会容易很多。

204 yield 使用案例之嵌套 list 的完全展开

下面的函数 `deep_flatten` 定义中使用了 `yield` 关键字，实现嵌套 list 的完全展开。

复制
```
def deep_flatten(lst):
    for i in lst:
        if type(i)==list:
            yield from deep_flatten(i)
        else:
            yield i
```

`deep_flatten` 函数，返回结果为生成器类型，如下所示：

复制
```
In [10]: gen = deep_flatten([1,['s',3],4,5])

In [11]: gen
Out[11]: <generator object deep_flatten at 0x000002410D1C66C8>
```

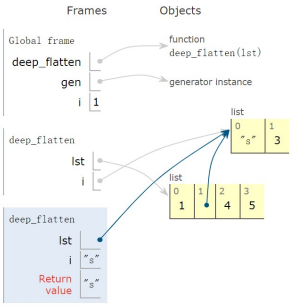
返回的 `gen` 生成器，与 `for` 结合打印出结果：

复制
```
In [14]: for i in gen:
    ...:     print(i)

1
s
3
4
5
```

`yield from` 表示再进入到 `deep_flatten` 生成器。

下面为返回值 `s` 的帧示意图：



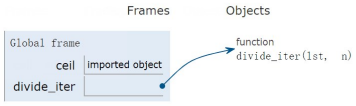
205 yield 使用案例之列表分组

复制
```
from math import ceil

def divide_iter(lst, n):
    if n <= 0:
        yield lst
        return
    i, div = 0, ceil(len(lst) / n)
    while i < n:
        yield lst[i * div: (i + 1) * div]
        i += 1

list(divide_iter([1, 2, 3, 4, 5], 0))  # [[1, 2, 3, 4, 5]]
list(divide_iter([1, 2, 3, 4, 5], 2))  # [[1, 2, 3], [4, 5]]
```

完整过程，演示动画如下：



---

206 可迭代协议之 *_iter_* 方法使用案例

只要 `iterable` 对象支持可迭代协议，即自定义了 `__iter__` 函数，便都能配合 `for` 依次迭代输出其元素。

如下，`TestIter` 类实现了迭代协议， `__iter__` 函数。

复制
```
class TestIter(object):
    def __init__(self):
        self._lst = [1,3,2,3,4,5]

    #支持迭代协议(即定义有__iter__()函数)
```

```
    def __iter__(self):
        print ("__iter__ is called!!")
        return iter(self._lst)
```

所以，对象 `t` 便能结合 `for` ，迭代输出元素。

复制

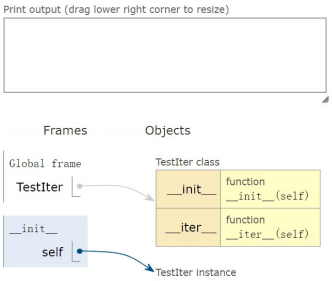
```
t = TestIter()
for e in t: # 因为实现了__iter__方法，所以 t 能被迭代
    print(e)
```

打印结果，如下所示：

复制

```
__iter__ is called!!
1
3
2
3
4
5
```

完整过程，动画演示：



207 迭代器相关方法之 next 函数使用

next(iterator,[, default])

返回可迭代对象的下一个元素

复制

```
In [129]: it = iter([5,3,4,1])

In [130]: next(it)
Out[130]: 5

In [131]: next(it)
Out[131]: 3

In [132]: next(it)
Out[132]: 4

In [133]: next(it)
Out[133]: 1
```

当迭代到最后一个元素 1 时，再执行 `next` ，就会抛出 `StopIteration` ，迭代器终止运行。

复制

```
In [135]: next(it)
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-135-bc1ab118995a> in <module>
----> 1 next(it)

StopIteration:
```

208 定制一个递减迭代器案例

编写一个迭代器，通过循环语句，对某个正整数，依次递减 1，直到 0.

实现类 `Descrease` ，继承于 `Iterator` 对象，重写两个方法：

- `__iter__`
- `__next__`

复制

```
from collections.abc import Iterator

class Decrease(Iterator):
    def __init__(self, init):
        self.init = init

    def __iter__(self):
        return self

    def __next__(self):
        while 0 < self.init:
            self.init -= 1
            return self.init
        raise StopIteration
```

调用递减迭代器 `Decrease` ：

复制

```
descend_iter = Decrease(6)
for i in descend_iter:
    print(i)
```

打印结果：

复制

```
5
4
3
```

```
2
1
0
```

核心要点：

(1) `__next__` 名字不能变，实现定制的迭代逻辑

(2) `raise StopIteration` ：通过 `raise` 中断程序。

209 Python 生成枚举对象的方法

enumerate(iterable, start=0)

`enumerate` 是很有用的一个内置函数，尤其要用到列表索引时。

它返回可枚举对象，也是一个迭代器。

复制

```
In [93]: s = ["a","b","c"]

In [95]: for i ,v in enumerate(s):
    ...:     print(i,v)
    ...:
0 a
1 b
2 c
```

也可以手动执行 `next`，依次输出一个 `tuple`.

复制

```
In [96]: enum = enumerate(s)

In [97]: next(enum)
Out[97]: (0, 'a')

In [98]: next(enum)
Out[98]: (1, 'b')

In [99]: next(enum)
Out[99]: (2, 'c')
```

210 列表和迭代器区别

有些读者朋友，区分不开列表、字典、集合等非迭代器对象与迭代器对象，觉得迭代器是多余的。

先探讨它们的区别。首先，创建一个列表 `a`:

复制

```
a = [1,3,5,7]
```

有没有朋友认为，列表就是迭代器的？注意列表 `a` 可不是迭代器类型（`Iterator`）。要想成为迭代器，需要经过内置函数 `iter` 包装：

复制

```
a_iter = iter(a)
```

此时 `a_iter` 就是 `Iterator` ，迭代器。可以验证：

复制

```
In [21]: from collections.abc import Iterator
    ...: isinstance(a_iter,Iterator)
Out[21]: True
```

分别遍历 `a`，`a_iter`:

复制

```
In [22]: for i in a:
    ...:     print(i)
    ...:
1
3
5
7

In [23]: for i in a_iter:
    ...:     print(i)
    ...:
1
3
5
7
```

打印结果一样，但是，再次遍历 `a`，`a_iter` 就会不同，`a` 正常打印，`a_iter` 没有打印出任何信息：

复制

```
In [24]: for i in a:
    ...:     print(i)
    ...:
1
3
5
7

In [25]: for i in a_iter:
    ...:     print(i)
    ...:
```

`

这就是列表 `a` 和迭代器 `a_iter` 的区别：

- 列表不论遍历多少次，表头位置始终是第一个元素；
- 迭代器遍历结束后，不再指向原来的表头位置，而是为最后元素的下一个位置。

只有迭代器对象才能与内置函数 `next` 结合使用，`next` 一次，迭代器就前进一次，指向一个新的元素。

所以，要想迭代器 `a_iter` 重新指向 `a` 的表头，需要重新创建一个新的迭代器 `a_iter_copy`

复制

```
In [27]: a_iter_copy = iter(a)
```

调用 `next` ，输出迭代器指向 `a` 的第一个元素：

复制
```
In [28]: next(a_iter_copy)
Out[28]: 1
```

值得注意，我们无法通过调用 `len` 获得迭代器的长度，只能迭代到最后一个末尾元素时，才知道其长度。

那么，怎么知道迭代到元素末尾呢？我们不妨一直 `next` ，看看会发生什么：

复制
```
In [30]: next(a_iter_copy)
Out[30]: 3

In [31]: next(a_iter_copy)
Out[31]: 5

In [32]: next(a_iter_copy)
Out[32]: 7

In [33]: next(a_iter_copy)

StopIteration:
```

等迭代到最后一个元素后，再执行 `next` ，会触发 `StopIteration` 异常。

所以，通过捕获此异常，就能求出迭代器指向 `a` 的长度，如下：

复制
```
a = [1, 3, 5, 7]
a_iter_copy2 = iter(a)
iter_len = 0
try:
    while True:
        i = next(a_iter_copy2)
        print(i)
        iter_len += 1
except StopIteration:
    print('iterator stops')

print('length of iterator is %d' % (iter_len,))
```

打印结果：

复制
```
1
3
5
7
iterator stops
length of iterator is 4
```

`

以上总结：遍历列表，表头位置始终不变；遍历迭代器，表头位置相应改变；next函数执行一次，迭代对象指向就前进一次；`StopIteration` 触发时，意味着已到迭代器尾部。

认识到迭代器和列表等区别后，我们再来说说生成器。

带 `yield` 的函数是生成器，而生成器也是一种迭代器。所以，生成器也有上面那些迭代器的特点。前些天已经讨论过生成器的一些基本知识，今天主要讨论生成器带来哪些好处，实际的使用场景在哪里。

211 节省内存的案例

求一组数据累积乘，比如三个数 [1,2,3]，累积乘后返回：[1,2,6].

一般的方法：

复制
```
def accumulate_div(a):
    if a is None or len(a) == 0:
        return []
    rtn = [a[0]]
    for i in a[1:]:
        rtn.append(i*rtn[-1])
    return rtn

rtn = accumulate_div([1, 2, 3, 4, 5, 6])
print(rtn)
```

打印结果：

复制
```
[1, 2, 6, 24, 120, 720]
```

这个方法开辟一段新内存 `rtn` ，空间复杂度为 `O(n)`

更节省内存的写法：

复制
```
def accumulate_div(a):
    if a is None or len(a) == 0:
        return []
    it = iter(a)
    total = next(it)
    yield total
    for i in it:
        total = total * i
        yield total
```

调用生成器函数 `accumulate_div` ，结合 `for` ，输出结果：

复制
```
acc = accumulate_div([1, 2, 3, 4, 5, 6])
for i in acc:
    print(i)
```

打印结果：

复制
```
1
```

```
2
6
24
120
720
```

也可以，直接转化为 `list`，如下：

复制
```
rtn = list(accumulate_div([1, 2, 3, 4, 5, 6]))
print(rtn)
```

打印结果：

复制
```
[1, 2, 6, 24, 120, 720]
```

这种使用 `yield` 生成器函数的方法，占用内存空间为 `O(1)`.

当输入的数组 [1, 2, 3, 4, 5, 6]，只有 6 个元素时，这种内存浪费可以忽视，但是当处理几个 G 的数据时，这种内存空间的浪费就是致命的，尤其对于单机处理。

所以，要多学会使用 `yield`，写出更加节省内存的代码，这是真正迈向 Python 中高阶的必经之路。

Python 内置的 itertools 模块就是很好的使用 `yield` 生成器的案例，今天我们再来学习几个。

212 拼接迭代器

chain 函数实现元素拼接，原型如下，参数 `*` 表示可变的参数

复制
```
chain(*iterables)
```

应用如下：

复制
```
In [9]: from itertools import *

In [10]: chain_iterator = chain(['I','love'],['python'],['very', 'much'])

In [11]: for i in chain_iterator:
    ...:     print(i)

I
love
python
very
much
```

它有些 join 串联字符串的感觉， join 只是一次串联一个序列对象。而 chain 能串联多个可迭代对象，形成一个更大的可迭代对象。

查看函数返回值 `chain_iterator` ，它是一个迭代器(`Iterator`)。

复制
```
In [6]: from collections.abc import Iterator
In [9]: isinstance(chain_iterator,Iterator)
Out[9]: True
```

chain 如何做到高效节省内存？chain 主要实现代码如下：

复制
```
def chain(*iterables):
    for it in iterables:
        for element in it:
            yield element
```

chain 是一个生成器函数，在迭代时，每次吐出一个元素，所以做到最高效的节省内存。

213 累积迭代器

返回可迭代对象的累积迭代器，函数原型：

复制
```
accumulate(iterable[, func, *, initial=None])
```

应用如下，返回的是迭代器，通过结合 `for` 打印出来。

如果 `func` 不提供，默认求累积和：

复制
```
In [15]: accu_iterator = accumulate([1,2,3,4,5,6])

In [16]: for i in accu_iterator:
    ...:     print(i)
1
3
6
10
15
21
```

如果 `func` 提供，`func` 的参数个数要求为 2，根据 `func` 的累积行为返回结果。

复制
```
In [13]: accu_iterator = accumulate([1,2,3,4,5,6],lambda x,y: x*y)

In [14]: for i in accu_iterator:
    ...:     print(i)
1
2
6
24
120
720
```

accumulate 主要的实现代码，如下：

复制
```
def accumulate(iterable, func=operator.add, *, initial=None):
    it = iter(iterable)
    total = initial
    if initial is None:
```

```
        try:
            total = next(it)
        except StopIteration:
            return
    yield total
    for element in it:
        total = func(total, element)
        yield total
```

accumulate 函数工作过程如下：

- 1) 包装 `iterable` 为迭代器
- 2) 初始值 `initial` 很重要，
- 如果它的初始值为 `None`，迭代器向前移动求出下一个元素，并赋值给 `total`，然后 `yield`
- 如果初始值被赋值，直接 `yield`

3) 此时迭代器 `it` 已经指向 `iterable` 的第二个元素，遍历迭代器 `it` ，`func(total, element)` 后，求出 `total` 的第二个取值， `yield` 后，得到返回结果的第二个元素。

4) 直到 `it` 迭代结束

214 漏斗迭代器

compress 函数，功能类似于漏斗功能，所以称它为漏斗迭代器，原型：

compress(data, selectors)

经过 `selectors` 过滤后，返回一个更小的迭代器。

复制
```
In [20]: compress_iter = compress('abcdefg',[1,1,0,1])

In [21]: for i in compress_iter:
    ...:     print(i)

a
b
d
```

compress 返回元素个数，等于两个参数中较短序列的长度。

它的主要实现代码，如下：

复制
```
def compress(data, selectors):
    return (d for d, s in zip(data, selectors) if s)
```

215 drop 迭代器

扫描可迭代对象 `iterable`，从不满足条件处往后全部保留，返回一个更小的迭代器。

原型如下：

复制
```
dropwhile(predicate, iterable)
```

应用举例：

复制
```
In [48]: drop_iterator = dropwhile(lambda x: x<3,[1,0,2,4,1,1,3,5,-5])

In [49]: for i in drop_iterator:
    ...:     print(i)
    ...:
4
1
1
3
5
-5
```

主要实现逻辑，如下：

复制
```
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

函数工作过程，如下：

- `iterable` 包装为迭代器
- 迭代 `iterable`
- 如果不满足条件 `predicate`，`yield x`，然后 跳出迭代，迭代 完 `iterable` 剩余所有元素。
- 如果满足条件 `predicate` ，就继续迭代，如果所有都满足，则返回空的迭代器。

216 take 迭代器

扫描列表，只要满足条件就从可迭代对象中返回元素，直到不满足条件为止，原型如下：

复制
```
takewhile(predicate, iterable)
```

应用举例：

复制
```
In [50]: take_iterator = takewhile(lambda x: x<5, [1,4,6,4,1])

In [51]: for i in take_iterator:
    ...:     print(i)
    ...:
1
4
```

主要实现代码，如下：

```
def takewhile(predicate, iterable):
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break #立即返回
```

函数工作过程：

- 遍历 `iterable`
- 符合条件 `predicate` ，`yield` x
- 否则跳出循环

217 克隆迭代器

tee 实现对原迭代器的复制，原型如下：

```
tee(iterable, n=2)
```

应用如下，克隆出 2 个迭代器，以元组结构返回

```
In [52]: a = tee([1,4,6,4,1],2)

In [53]: a
Out[53]: (<itertools._tee at 0x223ec30cfc8>, <itertools._tee at 0x223ec30
c548>)

In [54]: a[0]
Out[54]: <itertools._tee at 0x223ec30cfc8>

In [55]: a[1]
Out[55]: <itertools._tee at 0x223ec30c548>
```

并且，复制出的两个迭代器，相互独立，如下，迭代器 `a[0]` 向前迭代一次：

```
In [56]: next(a[0])
Out[56]: 1
```

克隆出的迭代器 `a[1]` 向前迭代一次，还是输出元素 1 ，表明它们之间是相互独立的：

```
In [57]: next(a[1])
Out[57]: 1
```

这种应用场景，需要用到迭代器至少两次的场合，一次迭代器用完后，再使用另一个克隆出的迭代器。

实现它的主要逻辑，如下：

```
from collections import deque

def tee(iterable, n=2):
    it = iter(iterable)
    deques = [deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:
                try:
                    newval = next(it)
                except StopIteration:
                    return
                for d in deques:
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

218 复制元素

repeat 实现复制元素 n 次，原型如下：

```
repeat(object[, times])
```

应用如下：

```
In [66]: list(repeat(6,3))
Out[66]: [6, 6, 6]

In [67]: list(repeat([1,2,3],2))
Out[67]: [[1, 2, 3], [1, 2, 3]]
```

它的主要实现逻辑，如下，有趣的是 repeat 函数如果 `times` 不设置，将会一直 `repeat` 下去。

```
def repeat(object, times=None):
    if times is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

219 笛卡尔积

笛卡尔积实现的效果，同下：

```
 ((x,y) for x in A for y in B)
```

举例，如下：

```
In [68]: list(product('ABCD', 'xy'))
Out[68]:
[('A', 'x'),
```

```
 ('A', 'y'),
 ('B', 'x'),
 ('B', 'y'),
 ('C', 'x'),
 ('C', 'y'),
 ('D', 'x'),
 ('D', 'y')]
```

它的主要实现代码，包括：

复制

```
def product(*args, repeat=1):
    pools = [tuple(pool) for pool in args] * repeat
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

220 加强版zip

若可迭代对象的长度未对齐，将根据 `fillvalue` 填充缺失值，返回结果的长度等于更长的序列长度。

复制

```
In [69]: list(zip_longest('ABCD', 'xy', fillvalue='-'))
Out[69]: [('A', 'x'), ('B', 'y'), ('C', '-'), ('D', '-')]
```

它的主要实现逻辑：

复制

```
def zip_longest(*args, fillvalue=None):
    iterators = [iter(it) for it in args]
    num_active = len(iterators)
    if not num_active:
        return
    while True:
        values = []
        for i, it in enumerate(iterators):
            try:
                value = next(it)
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
                iterators[i] = repeat(fillvalue)
                value = fillvalue
            values.append(value)
        yield tuple(values)
```

它里面使用 repeat，也就是在可迭代对象的长度未对齐时，根据 fillvalue 填充缺失值。

221 自定义装饰器之call_print

为了帮助大家更容易理解装饰器，以下函数包装，可能会让某些函数丢掉一些函数属性等信息。

记住，我们的主要目标：理解装饰器。

首先，定义函数 `call_print`

它的入参 `f` 为一个函数

它里面内嵌一个函数 `g` ，并返回函数 `g`

复制

```
def call_print(f):
    def g():
        print('you\'re calling %s function' % (f.__name__,))
    return g
```

Python 中，@call_print 函数，放在函数上面，函数 `call_print` 就变为装饰器。

变为装饰器后，我们不必自己去调用函数 `call_print`。

复制

```
@call_print
def myfun():
    pass


@call_print
def myfun2():
    pass
```

直接调用被装饰的函数，就能调用到 `call_print` ，观察输出结果：

复制

```
In [27]: myfun()
you're calling myfun function

In [28]: myfun2()
you're calling myfun2 function
```

使用 `call_print` ，`@call_print` 放置在任何一个新定义的函数上面。都会默认输出一行，输出信息：正在调用这个函数的名称。

222 装饰器本质

`call_print` 装饰器实现效果，与下面调用方式，实现的效果是等效的。

复制

```
def myfun():
    pass


def myfun2():
    pass


def call_print(f):
    def g():
        print('you\'re calling %s function' % (f.__name__,))
    return g
```

下面两行代码，对于理解装饰器，非常代码：

```
myfun = call_print(myfun)

myfun2 = call_print(myfun2)
```

`call_print(myfun)` 后不是返回一个函数吗，然后，我们再赋值给被传入的函数 `myfun`.

也就是 `myfun` 指向了被包装后的函数 `g`，并移花接木到函数 `g`，使得 `myfun` 额外具备了函数 `g` 的一切功能，变得更强大。

以上就是装饰器的本质。

再次调用 `myfun`, `myfun2` 时，与使用装饰器的效果完全一致。

```
In [32]: myfun()
you're calling myfun function

In [33]: myfun2()
you're calling myfun2 function
```

223 wraps 装饰器

自定义装饰器 `call_print` ，

```
def call_print(f):
    def g():
        print('you\'re calling %s function' % (f.__name__,))
    return g

@call_print
def myfun():
    pass

@call_print
def myfun2():
    pass
```

我们打印 `myfun` ：

```
myfun()

myfun2()

print(myfun)
```

打印结果：

```
you're calling myfun function
you're calling myfun2 function
<function call_print.<locals>.g at 0x00000215D90679D8>
```

发现，被装饰的函数 `myfun` 名字竟然变为 `g`，也就是定义装饰器 `call_print` 时使用的内嵌函数 `g`

Python 的 `functools` 模块，wraps 函数能解决这个问题。

解决方法，如下，只需重新定义 `call_print` ，并在 `g` 内嵌函数上，使用 `wraps` 装饰器。

```
from functools import wraps

def call_print(f):
    @wraps(f)
    def g():
        print('you\'re calling %s function' % (f.__name__,))
    return g
```

当再次调用打印 `myfun` 时，函数 `myfun` 打印信息正常：

```
print(myfun)
# 结果
<function myfun at 0x000002A34A8479D8>
```

224 统计程序异常出现次数的迭代器

这个装饰器，能统计出某个异常重复出现到指定次数时，历经的时长。

```
import time
import math

def excepter(f):
    i = 0
    t1 = time.time()
    def wrapper():
        try:
            f()
        except Exception as e:
            nonlocal i
            i += 1
            print(f'{e.args[0]}: {i}')
            t2 = time.time()
            if i == n:
                print(f'spending time:{round(t2-t1,2)}')
    return wrapper
```

关键词 `nonlocal` 在前面我们讲到了，今天就是它的一个应用。

它声明变量 `i` 为非局部变量；

如果不声明，根据 Python 的 *LEGB* 变量搜寻规则（这个规则我们在前面的专栏中也讲到过），`i+=1` 表明 `i` 为函数 `wrapper` 内的局部变量，因为在 `i+=1` 引用(`reference`)时，`i` 未被声明，所以会报 `unreferenced variable` 的错误。

使用创建的装饰函数 `excepter`, `n` 是异常出现的次数。

共测试了两类常见的异常：被零除和数组越界。

```
n = 10# except count

@excepter
def divide_zero_except():
    time.sleep(0.1)
    j = 1/(40-20*2)

# test zero divived except
for _ in range(n):
    divide_zero_except()


@excepter
def outof_range_except():
    a = [1,3,5]
    time.sleep(0.1)
    print(a[3])

# test out of range except
for _ in range(n):
    outof_range_except()
```

打印出来的结果如下：

复制

```
division by zero: 1
division by zero: 2
division by zero: 3
division by zero: 4
division by zero: 5
division by zero: 6
division by zero: 7
division by zero: 8
division by zero: 9
division by zero: 10
spending time:1.01
list index out of range: 1
list index out of range: 2
list index out of range: 3
list index out of range: 4
list index out of range: 5
list index out of range: 6
list index out of range: 7
list index out of range: 8
list index out of range: 9
list index out of range: 10
spending time:1.01
```

225 实现一个按照 2*i+1 自增的迭代器

实现类 `AutoIncrease` ，继承于 `Iterator` 对象，重写两个方法：

- `__iter__`
- `__next__`

复制

```
from collections.abc import Iterator

class AutoIncrease(Iterator):
    def __init__(self, init, n):
        self.init = init
        self.n = n
        self.__cal = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.__cal == 0:
            self.__cal += 1
            return self.init
        while self.__cal < self.n:
            self.init *= 2
            self.init += 1
            self.__cal += 1
            return self.init
        raise StopIteration
```

调用递减迭代器 `Decrease` ：

复制

```
iter = AutoIncrease(1,10)
for i in iter:
    print(i)
```

打印结果：

复制

```
1
3
7
15
31
63
127
255
511
1023
```


互动评论

说点什么

评论

The Scrapper 1 个月前

学习了

鼓掌