Pytho 全栈文档

第一章 函数式编程

1、生成器

通过列表生成式，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，创建一个包含100万个元素的列表，不仅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。所以，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。

创建生成器，列表生成式，我们知道，只要将最外一层的中括号，改为小括号，

看代码

```
# python交互式环境

>>> y = [x + 1 for x in range(10) if x % 2 == 0]
>>> y
[1, 3, 5, 7, 9]
>>> g = (x + 1 for x in range(10) if x % 2 == 0)
>>> g
<generator object <genexpr> at 0x0000020EF13E79C8>
```

创建 y 和 g 的区别仅在于最外层的 [ ] 和 ( )， y 是一个列表，而 g 是一个生成器。我们可以直接打印出L的每一个元素，但我们怎么打印出g的每一个元素呢？如果要一个一个打印出来，可以通过 next() 函数获得生成器的下一个返回值：

```
>>> next(g)
1
>>> next(g)
3
>>> next(g)
5
>>> next(g)
7
>>> next(g)
9
>>> next(g)
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    next(g)
StopIteration
```

也可以通过循环打印出来

```
>>> g = (x + 1 for x in range(10) if x % 2 == 0)

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

生成器保存的是算法，每次调用 next(g)，就计算出 g 的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出 StopIteration 的异常。当然，这种不断调用 next() 实在是太繁琐了，虽然是点一次出现一次，但正确的方法是使用 for 循环，因为生成器也是可迭代对象。所以，我们创建了一个生成器后，基本上永远不会调用 next()，而是通过 for 循环来迭代它，并且不需要关心StopIteration 异常。

所以，我们创建了一个生成器后，基本上不会调用`next()`，而是通过`for`循环来迭代它，并且不需要关心`StopIteration`的错误。generator非常强大。如果推算的算法比较复杂，用类似列表生成式的`for`循环无法实现的时候，还可以用函数来实现。比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

代码如下

```
# -*- coding: UTF-8 -*-
# 文件名：class_c.py

# 定义一个斐波那契函数
def fib_a(times):
    # 初始化
    n = 0
    a, b = 0, 1
    while n < times:
        print(b)
        a, b = b, a + b
        n += 1
    return 'done'


z = fib_a(5)
print(z)
# 打印结果如下

1
1
2
3
5
done
```

仔细观察，可以看出，fib_a函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。也就是说，上面的函数和generator仅一步之遥。要把`fib`函数变成generator，只需要把`print(b)`改为`yield(b)`就可以了：

```
def fib_a(times):
    # 初始化
    n = 0
    a, b = 0, 1
    while n < times:
        yield(b)
        a, b = b, a + b
        n += 1
    return 'done'
```

那此时调用方法和之前一样，可以使用next试试

```
f = fib_a(5)
>>> next(f)
1
>>> next(f)
1
>>> next(f)
2
>>> next(f)
3
>>> next(f)
5
>>> next(f)
Traceback (most recent call last):
  ......
StopIteration: done
```

在上面fib 的例子，我们在循环过程中不断调用 yield ，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。同样的，把函数改成generator后，我们基本上从来不会用 next() 来获取下一个返回值，而是直接使用 for 循环来迭代：

```
def fib_a(times):
    # 初始化
    n = 0
    a, b = 0, 1
    while n < times:
        yield(b)
        a, b = b, a + b
        n += 1
    return 'done'

for i in fib_a(5):
    print(i)
1
1
2
```

```
3
5
```

但是用for循环调用generator时，发现拿不到generator的return语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration的value中：

异常处理

```
# -*- coding: UTF-8 -*-
# 文件名：class_d.py

def fib_d(times):
    # 初始化
    n = 0
    a, b = 0, 1
    while n < times:
        yield (b)
        a, b = b, a + b
        n += 1
    return 'done'


f = fib_d(5)

while True:
    try:
        x = next(f)
        print("value:%d" % x)
    except StopIteration as e:
        print("生成器返回值:%s" % e.value)
        break
```

这样的到结果，是可以拿到返回值的，我们通过有限的值捕获异常，拿到return的值

那我们看下面的代码

```
def gen():
    i = 0
    while i < 5:
        temp = yield i
        print(temp)
        i += 1

f = gen()

print(next(f))
print(next(f))
print(next(f))
print(next(f))
print(next(f))
print(next(f))
# 打印结果如下
```

```
0
None
1
None
2
None
3
None
4
None
Traceback (most recent call last):
  ......
StopIteration
```

我们执行到yield时，gen函数作用暂时保存，返回i的值

next的函数，是基于实际对象的`__next__`方法

也可以使用

```
print(f.__next__())
print(f.__next__())
print(f.__next__())
print(f.__next__())
print(f.__next__())
print(f.__next__())

0
None
1
None
2
None
3
None
4
None

Traceback (most recent call last):
 ......
StopIteration
```

使用方法不同，但是实际的next也是基于`__next__`方法调用

send

temp接收下次f.send("不返回空")，send发送过来的值，f.next()等价f.send(None)

```
print(f.__next__())
print(f.send('不返回空')
print(f.__next__())
print(f.send('不返回空'))
```

生成器是这样一个函数，它记住上一次返回时在函数体中的位置。对生成器函数的第二次（或第 n 次）调用跳转至该函数中间，而上次调用的所有局部变量都保持不变。生成器不仅“记住”了它数据状态；生成器还“记住”了它在流控制构造（在命令式编程中，这种构造不只是数据值）中的位置。生成器的特点：

```
1. 节约内存
2. 迭代到下一次的调用时，所使用的参数都是第一次所保留下的，在整个所有函数调用的参数都是第一次所调用时保留的，而不是新创建的
```

2、迭代器

迭代是访问集合元素的一种方式。迭代器是一个可以记住遍历的位置的对象。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

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 `yield` 的generator function。

这些可以直接作用于 `for` 循环的对象统称为可迭代对象：`Iterable`。

那我们怎么判断一组数据或是一组数据对象是不是 `Iterable` 对象尼?

可以使用 isinstance() 判断一个对象是否是 Iterable 对象

```
>>> from collections.abc import Iterable
>>> print(isinstance([], Iterable))
True
>>> isinstance({}, Iterable)
True
>>> isinstance('1', Iterable)
True
>>> isinstance((x for x in range(10) if x % 2 == 0), Iterable)
True
>>> isinstance(10, Iterable)
False
>>>
```

生成器不但可以作用于 for 循环，还可以被 next() 函数不断调用并返回下一个值，直到最后抛出 StopIteration 错误表示无法继续返回下一个值了。

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() 判断一个对象是否是Iterator 对象，这里就产生一个疑问了，生成器都是 `Iterator` 对象，`list`、`dict`、`str` 是不是 `Iterator`？为什么？。

`list`、`dict`、`str` 不是 `Iterator`，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

那我们还可以通过 isinstance() 来判断是否是`Iterator`对象

注意`Iterator`对象和`Iterable`对象，一个是迭代器，一个是可迭代对象

```
>>> from collections.abc import Iterator
>>> isinstance((i for i in range(10) if i % 2 == 0), Iterator)
True
>>> isinstance([], Iterator)
False
>>> isinstance({}, Iterator)
False
>>> isinstance('abc', Iterator)
False
>>>
```

但是可以将 list 、 dict 、 str 等 Iterable 变成 Iterator，这里我们可以使用 iter() 函数

代码：

```
>>> isinstance(iter([]), Iterator)
True
>>> isinstance(iter('abc'), Iterator)
True
>>>
```

- 所以可作用于`for`循环的对象都是`Iterable`类型；
- 可作用于`next()`函数的对象都是`Iterator`类型，它们表示一个惰性计算的序列；
- 集合数据类型list 、 dict 、 str 等是 Iterable 但不是 Iterator ，不过可以通过 iter() 函数获得一个 Iterator 对象。

3、高阶函数

函数式编程

函数是Python内建支持的一种封装，我们通过把大段代码拆成函数，通过一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称之为面向过程的程序设计。函数就是面向过程的程序设计的基本单元。

而函数式编程（请注意多了一个“式”字）——Functional Programming，虽然也可以归结到面向过程的程序设计，但其思想更接近抽象的计算。

我们首先要搞明白计算机（Computer）和计算（Compute）的概念。

在计算机的层次上，CPU执行的是加减乘除的指令代码，以及各种条件判断和跳转指令，所以，汇编语言是最贴近计算机的语言。

而计算则指数学意义上的计算，越是抽象的计算，离计算机硬件越远。

对应到编程语言，就是越低级的语言，越贴近计算机，抽象程度低，执行效率高，比如C语言；越高级的语言，越贴近计算，抽象程度高，执行效率低，比如Python语言。

函数式编程就是一种抽象程度很高的编程范式。

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

Python对函数式编程提供部分支持。由于Python允许使用变量，因此，Python不是纯函数式编程语言。

高阶函数英文叫Higher-order function。什么是高阶函数？我们来看代码

变量可以指向函数

例如Python内置求绝对值函数abs()

```
>>> abs(-5)
5
>>> abs
<built-in function abs>
```

可以看见，只写abc只是函数本身，而abs(-5)才是函数调用，实际我们如果需要函数的结果，可以将函数赋值给变量

例如:

```
>>> i = abs(-5)
>>> i
5
>>> f = abs
>>> f
<built-in function abs>
>>> f(-5)
5
```

我们可以看到，我们将调用函数的结果，赋值给变量 i ，这样变量就可以打印出结果，如果将函数本身赋值给变量 f ，那么变量也拥有这个函数的功能，这个变量将指向这个函数，使用变量 f ()来调用函数和直接调用abs()效果是一样的

函数名也可以是变量

函数是由def定义，函数名，括号，括号参数，冒号，函数体组成

那么函数名是什么尼，可以发现，函数名是指向函数的变量，例如abs()这个函数，可以将abs看成变量，它指向一个可以求绝对值的函数，如果把abs指向其他的对象，例如我们给abs赋值，那看看还会指向求绝对值的函数么

```
>>> abs = 5
>>> abs(-5)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    abs(-5)
TypeError: 'int' object is not callable
>>> abs
5
```

TypeError: 'int' object is not callable 提示，类型错误，int类型是不可以被调用的，我们看到，abs这个变量被赋值5，然后使用abs(-5)调动函数，发现异常，此时abs变量指向的不是函数，而是一个int类型的 5 ，实际上，我们工作或是开发中写代码，是不能这么写的，由此可以看出函数名其实就是变量

注意：由于`abs`函数实际上是定义在`import builtins`模块中的，所以要让修改`abs`变量的指向在其它模块也生效可以使用

```
import builtins
builtins.abs = 10
```

我们想到，函数可以传参数，而函数名可以做变量，那我们函数里面的参数，也可以传入函数名，

```
# -*- coding: UTF-8 -*-
# 文件名：fun_a.py

def fun(i):
    return i * 2


def total(x, y, fun):
    return fun(x) + fun(y)


add_sum = total(1, 2, fun)
print(add_sum)
```

那我们可以看到，将函数作为参数，传给另一个参数就是**高阶函数**

map函数

来看一下map函数的参数与返回值

```
map(func, *iterables) --> map object
```

参数详解：

```
func：代表传入参数为函数，这里的函数指定指向函数的函数名，
*iterables：代表参数指定的可迭代的，
返回值：返回处理好的数据
map()函数：是将传入的func函数作用于，可迭代的数据里面每个元素，并将处理好的新的结果返回
```

代码：

```
>>> def fun_a(x):
...     return x * 10
...
>>> list_a = map(fun_a, [1, 2, 3, 4, 5])
>>> list(list_a)
[10, 20, 30, 40, 50]
```

`map()`传入的第一个参数是`fun_a`，即函数对象本身。由于结果`list_a`是一个`Iterator`，`Iterator`是惰性序列，因此通过`list()`函数让它把整个序列都计算出来并返回一个list。

很多情况下，也可以使用for循环也可以解决问题，但实际上map作为高级函数，将运算抽象化，还可计算复杂的函数，例如将列表的元素int类型转换为int类型，只需要一行代码

```
>>> list(map(str, [1, 2, 3, 4, 5]))
['1', '2', '3', '4', '5']
```

reduce函数

我们来看一下map函数的参数与返回值，注意使用reduce函数时需要先导入

reduce函数是在 functools模块里面的

```
from functools import reduce
reduce(function, sequence[, initial]) -> value
```

参数详解

```
function:一个有两个参数的函数
sequence:是一个序列，是一些数据的集合，或者是一组数据，可迭代对象
initial：可选，初始参数
返回值：返回函数计算的结果
reduce()函数,使用function函数（有两个参数）先对集合中的sequence第 1、2 个元素进行操作，如果存在initial参数，则将会以sequence中的第一个元素和initial作为参数，用作调用，得到的结果再与sequence中的下一个数据用 function 函数运算，最后得到一个结果。
```

代码：

```
from functools import reduce
list_a = [1, 2, 3, 4, 5]

def fun_b(x, y):
    return x + y
print(reduce(fun_b, list_a))

# 运算结果如下
15
```

依次按照顺序从列表list_a中提取两个元素作为参数，进入fun_b中进行运算，得到的结果，作为下次运算时的其中一个参数，再从列表中取出一个元素，再进行运算。最终得到的结果是总和的计算。

filter函数

Python内建的`filter()`函数用于过滤序列，和`map()`类似，`filter()`也接收一个函数和一个序列

但是不同的是`filter()`把传入的函数依次作用于每个元素，然后根据返回值是`True`还是`False`决定元素的保留与丢弃

看一下filter的参数

```
filter(function, iterable)
```

参数列表

```
function：判断函数。
iterable：序列，（可迭代对象）。
返回值：返回列表
filter函数，序列（可迭代对象）的每个元素作为参数传递给函数进行判断，然后返回 True 或 False，最后将返回 True 的元素放到新列表中
```

代码：

```
def not_odd(num):
    return n % 2 == 0

newlist = filter(not_odd, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(newlist)

# 运算结果
[2, 4, 6, 8, 10]
```

这里定义了一个函数not_odd,顾名思义，不是奇数的函数，这个函数，只有当参数为2的整数倍时返回True，

这里filter函数的两个参数第一个是过滤方法，第二个是需要过滤的列表，将列表里面的元素依次带入函数中进行运算，得到的结果如果为True时，将此结果作为新的filter对象保留，等待函数里面的列表执行完成后，返回最终的值，这里的值为列表，也就是过滤掉了False的数据或元素。

那可以用filter来计算素数，计算素数的其中一个方法是埃氏筛法。

给出要筛数值的范围n，找出以内的素数。先用2去筛，即把2留下，把2的倍数剔除掉；再用下一个质数，也就是3筛，把3留下，把3的倍数剔除掉；接下去用下一个质数5筛，把5留下，把5的倍数剔除掉；不断重复下去……

用Python来实现这个算法，我们先写一个生成器构造一个从3开始的无限奇数序列，首先偶数列先排除

```
def oddnum():
    n = 1
    while True:
        n = n + 2
        yield n
```

写一个筛选的函数，这里使用了匿名函数，这个函数是返回一个判断，判断是否为可整除数

```
def undivisible(n):
    return lambda x: x % n > 0
```

使用filter来过滤，不断返回素数的生成迭代，

```
def primes():
    yield 2
    it = oddnum() # 初始序列
    while True:
        n = next(it) # 返回序列的第一个数
        yield n
        it = filter(undivisible(n), it) # 构造新序列
```

一个基本的判断素数方法就产生了，这里我们需要手动结束一下

```
# 打印100以内的素数

for n in primes():
    if n < 100:
        print(n)
    else:
        break
```

好的我们来解释一下代码，看到我们这里用到了几个知识点

生成器与迭代器、匿名函数、filter函数，其中

第一段代码生成了以3开始的奇数序列

第二段代码自定义过滤函数，包含匿名函数，判断值的取余是否能被整除

第三段代码用来返回素数，这里先返回一个2为素数，因为偶数都被排除了所以整除3为基础进行排除，将数据不断地迭代生成，留下对应的素数序列。

那这里就对应的filter函数就是用来过滤的方法进行返回数据

4、匿名函数

匿名函数，顾名思义没有名字的函数我来看一下语法

```
lambda [list] : 表达式
```

这里我们介绍一下参数

```
[list]：表示参数列表，
注意：参数与表达式之间需要冒号来区分
表达式 ：表达式方法非常多，表达形式也非常多
返回值 ：为表达式的结果value
```

例如，上方的代码这里：

```
lambda x: x % n > 0
```

x为需要传入的参数，而x % n > 0为表达式，之间需要用冒号进行引用，计算的表达式结果为返回值

这里举例说明：如果设计一个求 2 个数之和的函数，使用普通函数的方式，定义如下：

```
def add(x, y):
    return x+ y
print(add(3,4))
```

我们看到只有一行表达式，直接运算结果返回值，那这是时我们如果使用匿名函数一行代码即可完成

```
add = lambda x,y:x+y
print(add)
```

这里我们将直接写出我们的结果，我们可以看到对于比较单行返回的函数，使用 lambda 表达式可以省去定义函数的过程，让代码更加简洁，针对不需要多次复用的函数，使用 lambda 表达式可以在用完之后立即释放，提高程序执行的性能。而且还能配合其他的一些高阶函数配合使用

5、返回函数

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。我们在操作函数的时候，如果不需要立刻求和，而是在后面的代码中，根据需要再计算

例如下面：

```
def sum_fun_a(*args):
    a = 0
    for n in args:
        a = a + n
    return a
```

这是我不需要立即计算我的结果sum_fun方法，不返回求和的结果，而是返回求和的函数，例如下方

```
def sum_fun_b(*args):
    def sum_a():
        a = 0
        for n in args:
            a = a + n
        return a
    return sum_a
```

当我们调用`sum_fun_b()`时，返回的并不是求和结果，而是求和函数`sum_a`，当我们在调用sum_fun_b函数时将他赋值给变量

```
>>> f1 = sum_fun_b(1, 2, 3, 4, 5)
#  此时f为一个对象实例化，并不会直接生成值，
>>> f1()
15
>>> f2 = sum_fun_b(1, 2, 3, 4, 5)
>>> f3 = sum_fun_b(1, 2, 3, 4, 5)

>>> print(f2, f3)
<function sum_fun_b.<locals>.sum_a at 0x0000029D496C8168>
<function sum_fun_b.<locals>.sum_a at 0x0000029D496D0EE8>

>>> print(id(f2), id(f3))
1920197821496 1920197820776
```

此时我们直接拿到的值就是15，那可以想一想，此时 f = sum_a，那这里存在一个疑问参数去哪里了?

而且我们看到创建的两个方法相互不影响的，地址及值是不相同的

在函数 `sum_fun_b` 中又定义了函数 `sum_a`，并且，内部函数 `sum_a` 可以引用外部函数 `sum_fun_b` 的参数和局部变量，当 `sum_fun_b` 返回函数 `sum_a` 时，而对应的参数和变量都保存在返回的函数中，这里称为 `闭包`。

6、闭包

先看一段代码:

```
# -*- coding: UTF-8 -*-
# 文件名: clos_a.py

# 定义一个函数
def fun_a(num_a):
    # 在函数内部再定义一个函数，并且这个内部函数用到了外部的变量，这个函数以及用到外部函数的变量及参数叫闭包
    def fun_b(num_b):
        print("in test_in 函数, number_in is %d" % num_b)
        return num_a + num_b

    # 这里返回的就是闭包的结果
    return fun_b


# 给fun_a函数赋值，这个10就是给参数num
ret = fun_a(10)
# 注意这里的10其实给参数fun_b
print(ret(10))
# 注意这里的90其实给参数fun_b
print(ret(90))
```

运行结果

```
in test_in 函数, number_in is 10
20
in test_in 函数, number_in is 90
100
```

此时，内部函数对外部函数作用域里变量的引用（非全局变量），则称内部函数为闭包。这里闭包需要有三个条件

```
三个条件，缺一不可：
1)必须有一个内嵌函数(函数里定义的函数）——这对应函数之间的嵌套
2)内嵌函数必须引用一个定义在闭合范围内(外部函数里)的变量——内部函数引用外部变量
3)外部函数必须返回内嵌函数——必须返回那个内部函数
```

```
# python交互环境编辑器

>>> def counter(start=0):
    count = [start]
    def incr():
        count[0] += 1
        return count[0]
    return incr

>>> c1 = counter(5)
>>> print(c1())
6
>>> print(c1())
7
>>> c2=counter(50)
>>> print(c2())
51
>>> print(c2())
52
>>>
```

当一个函数在本地作用域找不到变量申明时会向外层函数寻找，这在函数闭包中很常见

但是在本地作用域中使用的变量后，还想对此变量进行更改赋值就会报错

```
def test():
    count = 1
    def add():
        print(count)
        count += 1
    return add

a = test()
a()
```

报错信息

```
Traceback (most recent call last):

  ......
UnboundLocalError: local variable 'count' referenced before assignment
```

如果我在函数内加一行nonlocal count就可解决这个问题

代码

```
# -*- coding: UTF-8 -*-
# 文件名  : nonlocal_a.py

def test():
    count = 1
    def add():
        nonlocal count
        print(count)
        count += 1
        return count
    return add

a = test()
a()
# 1
a()
# 2
```

nonlocal声明的变量不是局部变量,也不是全局变量,而是外部嵌套函数内的变量。

如果从另一个角度来看我们给此函数增加了记录函数状态的功能。当然，这也可以通过申明全局变量来实现增加函数状态的功能。当这样会出现以下问题：

1. 每次调用函数时，都得在全局作用域申明变量。别人调用函数时还得查看函数内部代码。
2. 当函数在多个地方被调用并且同时记录着很多状态时，会造成非常地混乱。

使用**nonlocal**的好处是，在为函数添加状态时不用额外地添加全局变量，因此可以大量地调用此函数并同时记录着多个函数状态，每个函数都是独立、独特的。针对此项功能其实还个一个方法，就是使用类，通过定义`__call__`可实现在一个实例上直接像函数一样调用

代码如下：

```
# -*- coding: UTF-8 -*-
# 文件名: clos_c.py

def line_conf(a, b):
    def line(x):
        return a * x + b

    return line
```

```
line1 = line_conf(1, 1)
line2 = line_conf(4, 5)
print(line1(5))
print(line2(5))
```

运行结果为

```
6
25
```

从这段代码中，函数line与变量a,b构成闭包。在创建闭包的时候，我们通过line_conf的参数a,b说明了这两个变量的取值，这样，我们就确定了函数的最终形式(y = x + 1和y = 4x + 5)。我们只需要变换参数a,b，就可以获得不同的直线表达函数。由此，我们可以看到，闭包也具有提高代码可复用性的作用。如果没有闭包，我们需要每次创建函数的时候同时说明a,b,x。这样，我们就需要更多的参数传递，也减少了代码的可移植性。

```
1.闭包似优化了变量，原来需要类对象完成的工作，闭包也可以完成
2.由于闭包引用了外部函数的局部变量，则外部函数的局部变量没有及时释放，消耗内存
```

但是还没有结束，我们知道，函数内部函数，引用外部函数参数或值，进行内部函数运算执行，并不是完全返回一个函数，也有可能是一个在外部函数的值，我们还需要知道返回的函数不会立刻执行，而是直到调用了函数才会执行。

看代码:

```
def fun_a():
    fun_list = []
    for i in range(1, 4):
        def fun_b():
            return i * i

        fun_list.append(fun_b)
    return fun_list


f1, f2, f3 = fun_a()

print(f1(), f2(), f3())

# 结果: 9, 9, 9
```

这里创建了一个fun_a函数，外部函数的参数fun_list定义了一个列表，在进行遍历，循环函数fun_b，引用外部变量i 计算返回结果，加入列表，每次循环，都创建了一个新的函数，然后，把创建的3个函数都返回了

但是实际结果并不是我们想要的**1,4,9**,而是9，9，9，这是为什么尼?

这是因为，返回的函数引用了变量 i ，但不是立刻执行。等到3个函数都返回时，它们所引用的变量i已经变成了3，

每一个独立的函数引用的对象是相同的变量，但是返回的值时候，3个函数都返回时，此时值已经完整了运算，并存储，当调用函数，产生值不会达成想要的，返回函数不要引用任何循环变量，或者将来会发生变化的变量，但是如果一定需要尼，如何修改这个函数尼?

可以再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变，那我们就可以完成下面的代码

```
# -*- coding: UTF-8 -*-
# 文件名  : closure_a.py

def fun_a():
    def fun_c(i):
        def fun_b():
            return i * i

        return fun_b

    fun_list = []
    for i in range(1, 4):
        # f(i)立刻被执行，因此i的当前值被传入f()
        fun_list.append(fun_c(i))
    return fun_list


f1, f2, f3 = fun_a()
print(f1(), f2(), f3())

#  1 4 9
```

7、装饰器

看一段代码：

```
# -*- coding: UTF-8 -*-
# 文件名  : decor_b.py

def decor_a(func):
    def decor_b():
        print("I am decor_b")

        func()

        print("I am func")

    return decor_b


def decor_c():
    print("I am decor_c")


decor_c()
# outputs: "I am decor_c"
```

```
decor_c = decor_a(decor_c)

decor_c()
# outputs:   I am decor_b
#            I am decor_c
#            I am func
```

由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。也可以将函数赋值变量，做参传入另一个函数。

那什么是装饰器

> 装饰器本质上是一个Python函数，它可以让其他函数在不需要做任何代码变动的前提下增加额外功能，装饰器的返回值也是一个函数对象。
> 它经常用于有以下场景，比如：插入日志、性能测试、事务处理、缓存、权限校验等场景。装饰器是解决这类问题的绝佳设计

装饰器的作用就是为已经存在的对象添加额外的功能

先看代码：

```
# -*- coding: UTF-8 -*-
# 文件名  : decor_b.py


# 装饰器
def decor_a(fun):
    def decor_b():
        print('I am decor_b')
        fun()
        print('I am fun')

    return decor_b


@decor_a
def decor_c():
    print('I am decor_c')


decor_c()
#outputs:   I am decor_b
#           I am decor_c
#           I am fun
```

我们没有直接将decor_c函数作为参数传入decor_a中，只是将decor_a函数以@方式装饰在decor_c函数上。

也就是说，被装饰的函数，函数名作为参数，传入到装饰器函数上，不影响decor_c函数的功能，再次基础上可以根据业务或者功能增加条件或者信息。

(注意：@在装饰器这里是作为Python语法里面的语法糖写法，用来做修饰。)

但是我们这里就存在一个问题这里引入魔术方法 `__name__` 这是属于 python 中的内置类属性，就是它会天生就存在与一个 python 程序中，代表对应程序名称，一般一段程序作为主线运行程序时其内置名称就是 `__main__`,当自己作为模块被调用时就是自己的名字

代码:

```
print(decor_c.__name__)

#outputs:   decor_b
```

这并不是我们想要的！输出应该是" decor_c "。这里的函数被decor_b替代了。它重写了我们函数的名字和注释文档，那怎么阻止变化尼，Python提供functools模块里面的wraps函数解决了问题

代码:

```
# -*- coding: UTF-8 -*-
# 文件名  : decor_c.py

from functools import wraps


# 装饰器
def decor_a(fun):
    @wraps(fun)
    def decor_b():
        print('I am decor_b')
        fun()
        print('I am fun()')

    return decor_b


@decor_a
def decor_c():
    print('I am decor_c')


print(decor_c.__name__)
#outputs:   decor_c
```

我们在装饰器函数内，作用decor_c的decor_b函数上也增加了一个装饰器wraps还是带参数的。

这个装饰器的功能就是不改变使用装饰器原有函数的结构。

我们熟悉了操作，拿来熟悉一下具体的功能实现，我们可以写一个打印日志的功能

```
# -*- coding: UTF-8 -*-
# 文件名  : decor_d.py

from functools import wraps
```

```
def logs(fun):
    @wraps(fun)
    def with_logging(*args, **kwargs):
        print(fun.__name__ + " Debug")
        return fun(*args, **kwargs)

    return with_logging


@logs
def decor(x):
    return x + x


result = decor(4)
#outputs:   decor Debug
```

带参装饰器

我们也看到装饰器wraps也是带参数的，那我们是不是也可以定义带参数的装饰器尼，

我们可以使用一个函数来包裹装饰器，调入这个参数。

```
# -*- coding: UTF-8 -*-
# 文件名  : log_b.py

from functools import wraps


def logs(logfile='out.log'):
    def logging_decorator(fun):
        @wraps(fun)
        def wrapped_function(*args, **kwargs):
            log_string = fun.__name__ + " Debug"
            print(log_string)
            # 打开logfile，并写入内容
            with open(logfile, 'a') as opened_file:
                # 现在将日志打到指定的logfile
                opened_file.write(log_string + '\n')
            return fun(*args, **kwargs)

        return wrapped_function

    return logging_decorator


@logs()
def decor_a():
    pass


decor_a()
```

```
# Output: decor_a Debug
# 现在一个叫做 out.log 的文件出现了，里面的内容就是上面的字符串

@logs(logfile='out1.log')
def decor_b():
    pass


decor_b()
# Output: decor_b Debug
# 现在一个叫做 func2.log 的文件出现了，里面的内容就是上面的字符串
```

这里我们将带参数的带入进去根据代码流程执行生成了两个文件并将文件打印进去

现在我们有了能用于正式环境的logs装饰器，但当我们的应用的某些部分还比较脆弱时，异常也许是需要更紧急关注的事情。比方说有时你只想打日志到一个文件。而有时你想把引起你注意的问题发送到一个email，同时也保留日志，留个记录。这是一个使用继承的场景，但目前为止我们只看到过用来构建装饰器的函数。

```
# -*- coding: UTF-8 -*-
# 文件名  : log_c.py

from functools import wraps


class Logs(object):
    def __init__(self, logfile='out.log'):
        self.logfile = logfile

    def __call__(self, fun):
        @wraps(fun)
        def wrapped_function(*args, **kwargs):
            log_string = fun.__name__ + " was called"
            print(log_string)
            # 打开logfile并写入
            with open(self.logfile, 'a') as opened_file:
                # 现在将日志打到指定的文件
                opened_file.write(log_string + '\n')
            # 现在，发送一个通知
            self.notify()
            return fun(*args, **kwargs)

        return wrapped_function

    def notify(self):
        # Logs只打日志，不做别的
        pass


@Logs()
def decor_a():
    pass
```

这个实现有一个优势，在于比嵌套函数的方式更加整洁，而且包裹一个函数还是使用跟以前一样的语法

现在，我们给 logit 创建子类，来添加 email 的功能，当然这个功能不在这里详细赘述

```
# -*- coding: UTF-8 -*-
# 文件名  : log_c.py


from mysit.text4.log_c import Logs


class EmailLogs(Logs):
    '''
    一个logit的实现版本，可以在函数调用时发送email给管理员
    '''

    def __init__(self, email='admin@myproject.com', *args, **kwargs):
        self.email = email
        super(EmailLogs, self).__init__(*args, **kwargs)

    def notify(self):
        # 发送一封email到self.email
        # 这里就不做实现了
        pass
```

这里我们继续继承并重写notify方法，完成发送邮件的功能

此时@EmailLogs 将会和 @Logs 产生同样的效果，但是在打日志的基础上，还会多发送一封邮件给管理员。

8、偏函数

Python的`functools`模块提供了很多有用的功能，其中一个就是偏函数（Partial function）。要注意，这里的偏函数和数学意义上的偏函数不一样。

在介绍函数参数的时候，我们讲到，通过设定参数的默认值，可以降低函数调用的难度。而偏函数也可以做到这一点。

例如：int()`函数可以把字符串转换为整数，当仅传入字符串时，`int() 函数默认按十进制转换

```
>>> int('123')
123
```

但`int()`函数还提供额外的`base`参数，默认值为`10`。如果传入`base`参数，就可以做进制的转换

```
>>> int('12345', base=8)
5349

>>> int('12345', 16)
74565
```

如果要转换大量的二进制字符串，每次都传入`int(x, base=2)`非常麻烦，于是，我们想到，可以定义一个`int2()`的函数，默认把`base=2`传进去：

代码：

```
# 定一个转换义函数
>>> def int_1(num, base=2):
    return int(num, base)

>>> int_1('1000000')
64

>>> int_1('1010101')
85
```

把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单

继续优化，functools.partial `就是帮助我们创建一个偏函数的，不需要我们自己定义` int_1()`，可以直接使用下面的代码创建一个新的函数` int_1

```
# 导入
>>> import functools

# 偏函数处理
>>> int_2 = functools.partial(int, base=2)

>>> int_2('1000000')
64


>>> int_2('1010101')
85
```

理清了`functools.partial`的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

注意到上面的新的`int_2`函数，仅仅是把`base`参数重新设定默认值为`2`，但也可以在函数调用时传入其他值

实际上固定了int()函数的关键字参数`base`

```
int2('10010')
```

相当于是：

```
kw = { base: 2 }
int('10010', **kw)
```

当函数的参数个数太多，需要简化时，使用`functools.partial`可以创建一个新的函数，这个新函数可以固定住原函数的部分参数，从而在调用时更简单