一　函数式编程

1.1　python 中函数式编程简介

函　数：function
函数式：functional，一种编程范式
函数不等于函数式。

函数式编程的特点：

1、把计算视为函数而非指令
2、纯函数式编程：不需要变量，没有副作用、测试简单
3、支持高阶函数，代码简洁

Python 支持的函数式编程：

1、不是纯函数式编程：允许有变量
2、支持高阶函数：函数也可以作为变量传入
3、支持闭包：有了闭包就能返回函数
4、有限度地支持匿名函数

1.2　python 中高阶函数

1、变量可以指向函数，调用这个变量，和调用函数的结果是一样的。

```
>>> f = abs
>>> f(-20)
20
```

2、函数名其实就是指向函数的变量，和普通的变量没有区别。

```
>>> abs = len
>>> abs( [1, 2, 3] )
3
```

3、高阶函数：能接收函数做参数的函数。

```
>>> def add(x, y, f):
>>>         return f(x) + f(y)
>>> add(-5, 9, abs)
14
```

1.3　python 中 map()函数

map()是 python 内置的高阶函数，它接收一个函数 f 和一个 list，并通过把函数 f 依次作用在 list 的每个元素上，得到一个新的 list 并返回。

```
>>> def f(x):
>>>         return x * x
```

```
>>>     print map(f, [1, 2, 3, 4])
[1, 4, 9, 16]
```

注意：map()函数不改变原有的 list，而是返回一个新的 list。

capitalize()：首字母大写，其余全部小写

title()：标题首字大写

1.4 python 中 reduce()函数

reduce()函数也是 python 内置的一个高阶函数。reduce()函数接收一个函数 f 和一个 list，reduce()传入的函数 f 必须接收两个参数，reduce()对 list 的每个元素反复调用函数 f，并返回最终结果值。

```
>>> def f(x, y):
>>>     return x+y
>>> reduce( f, [1, 3, 5, 7, 9] )
```

reduce()函数将做如下计算：

```
f(1, 3) --> 4
f(4, 5) --> 9
f(9, 7) --> 16
f(16, 9) --> 25
```

计算结束，返回结果 25.

reduce()还可以接收第 3 个可选参数，作为计算的初始值。如果把初始值设为 100：

```
>>> reduce( f, [1, 3, 5, 7, 9], 100 )
```

结果将变成 125，因为第一轮计算是：f(100, 1)

Python3 中，要使用 reduce()，得从 functools 中引入：

```
from functools import reduce
```

1.5 python 中 filter()函数

filter()函数接收一个函数 f 和一个 list，这个函数 f 的作用是对每个元素进行判断，返回 Ture 或 False，filter()根据判断结果自动过滤掉不符合条件的元素，返回由符合条件元素组成的新 list。

删除 None 或者空字符串：

```
>>> def is_not_empty(s):
>>>     return s and len( s.strip() ) > 0
>>> filter( is_not_empty, [ 'test', None, '', 'str', '  ', 'End' ] )
[ 'test', 'str', 'End' ]
```

注意：s.strip(rm)删除 s 字符串中开头、结尾处的 rm 序列的字符。

当 rm 为空时，默认删除空白符（包括'\n'，'\r'，'\t'，' '），如下：

```
a = '    123'
a.strip()
结果：'123'
```

1.6 python 中自定义排序函数

Python 内置的 sorted()函数可对 list 进行排序：

```
>>> sorted( [36, 5, 12, 9, 21 )
[ 5, 9, 12, 21, 36 ]
```

但 sorted()也是一个高阶函数，它可以接收一个比较函数来实现自定义排序，比较函数的定义是，传入两个待比较的元素 x, y，如果 x 应该排在 y 的前面，返回-1，如果 x 应该排在 y 的后面，返回 1。如果 x 和 y 相等，返回 0。

因此，如果我们要实现倒序排序，只需要编写一个 reversed_cmp 函数：

```
>>> def reversed_cmp( x, y ):
>>>     if x > y:
>>>         return -1
>>>     if x < y:
>>>         return 1
>>>     return 0
```

这样，调用 sorted()并传入 reversed_cmp 就可以实现倒序排序：

```
>>> sorted( [36, 5, 12, 9, 21], reversed_cmp )
[ 36, 21, 12, 9, 5 ]
```

sorted()也可以对字符串进行排序，字符串默认按照 ASCII 大小来比较。

```
>>> sorted( [ ‘bob’, ‘about’, ‘Zoo’, ‘Credit’ ] )
[ ‘Credit’, ‘Zoo’, ‘about’, ‘bob’ ]
```

‘Zoo’排在’about’之前是因为’Z’的 ASCII 码比’a’小。

1.7 python 中返回函数

Python 的函数不但可以返回 int、str、list、dict 等数据类型，还可以返回函数！

```
>>> def calc_prod(lst):
>>>     def g():
>>>         p = 1
>>>         for x in lst:
>>>             p = p * x
>>>         return p
>>>     return g
>>> f = calc_prod([1, 2, 3, 4])
>>> print f()
24
```

1.8 python 中闭包

类似 1.7 中的示例函数，这种内层函数引用了外层函数的变量（参数也算变量），然后返回内层函数的情况，称为闭包（Closure）。

闭包的特点是返回的函数还引用了外层函数的局部变量，所以，要正确使用闭包，就要

确保引用的局部变量在函数返回后不能变。

```
>>> def count():
>>>     fs = []
>>>     for i in range(1, 4):
>>>         def f():
>>>             return i*i
>>>         fs.append(f)
>>>     return fs
>>> f1, f2, f3 = count()
```

调用 f1()，f2()，f3()的结果全部是 9，原因是当 count()函数返回了 3 个函数时，这 3 个函数所引用的变量 i 的值已经变成了 3。由于 f1、f2、f3 并没有被调用，所以，此时他们并未计算 i*i，当 f1 被调用时：

```
>>> f1()
9      #因为 f1 现在才计算 i*i，但现在 i 的值已经变成 3
```

因此，返回函数不要引用任何循环变量，或者后续会发生变化的变量。

要得到结果 1、4、9，正确的 count()函数为：

```
>>> def count():
>>>     fs = []
>>>     for i in range(1, 4):
>>>         def f( m=i ):
>>>             return m*m
>>>         fs.append(f)
>>>     return fs
>>> f1, f2, f3 = count()
```

问题的产生是因为函数只在执行时才去获取外层函数 i，若函数在定义时就获取到 i，问题就得以解决。

1.9 python 中匿名函数

高阶函数可以接收函数做参数，有些时候，我们不需要显示地定义函数，直接传入匿名函数更方便。

函数 $f(x)=x^2$，除了定义一个 f(x)的函数外，用匿名函数可以定义为：

```
>>> lambda x: x * x
```

实际上相当于：

```
>>> def f(x):
>>>     return x * x
```

关键字 lambda 表示匿名函数，冒号前面的 x 表示函数参数。

匿名函数有个限制，就是**只能有一个表达式，不写 return**，返回值就是该表达式的结果。

返回函数的时候，也可以返回匿名函数：

```
>>> myabs = lambda x : -x if x<0 else x
>>> myabs(-1)
1
>>> myabs(1)
1
```

1.10 python 中 decorator 装饰器

当定义了一个函数，想要动态的为函数增加功能，但不想改动函数代码。如：定义了函数 f1()、f2()、f3()，想要在每个函数调用时，打印"call 函数名()"。第一种方法是为每个函数添加代码“pring fn()”。但是比较繁琐。

高阶函数可以接收函数做参数，也可以返回函数。我们可以定义一个高阶函数，传入一个函数，然后对该函数进行包装，返回包装后的函数。

所以，第二种方法：通过高阶函数返回新函数：

```
>>> def f1(x):
>>>     return x*2
>>> def new_fn(f):        #装饰器函数
>>>     def fn(x):
>>>         print 'call ' + f._name_ + '()'
>>>         return f(x)
>>>     return fn
```

调用 new_fn(f1)，并仍以 f1 接收返回函数，f1 的原始定义函数被彻底隐藏。

```
>>> f1 = new_fn(f1)
>>> print f1(5)
```

Python 内置的@语法就是为了简化装饰器调用。

```
>>> @new_fn                          >>> def f1(x):
>>> def f1(x):          相当于：      >>>     return x*2
>>>     return x*2                    >>> f1 = new_fn(f1)
```

装饰器的作用：可以极大地简化代码，避免每个函数编写重复性代码。

打印日志： @log

检测性能： @performance

数据库事务：@transaction

URL 路由： @post('/register')

1.11 python 中编写无参数 decorator

Python 的 decorator 本质上就是一个高阶函数，它接收一个函数作为参数，然后返回一个新函数。

使用 decorator 用 python 提供的@语法，这样可以避免手动编写 f = decorate(f)这样的代码。

当装饰器函数限定了返回函数的参数个数，被装饰函数的参数个数与其不同，则会报错。

要让装饰器函数自适应任何参数定义的函数，可以利用 python 的 *args（用于元组） 和 **kw（用于字典），保证任意个数的参数总是能正常调用。

1.12 python 中编写带参数 decorator

```
>>> def log(f):
>>>     def fn(x):
```

```
>>>         print 'call ' + f.__name__ + '()...'
>>>         return f(x)
>>>     return fn
```

考察@log 装饰器，对于被装饰的函数，log 打印的语句是不能变的（除了函数名）。

如果希望打印的语句有不同，log 函数本身就需要传入参数。类似这样：

```
>>> @log('DEBUG')
>>> def my_func():
>>>     pass
```

打印函数执行的时间，可传入 s 或者 ms：

```
1  import time
2
3  def performance(unit):
4      def per_dec(f):
5          def fn(*args, **kw):
6              t1 = time.time()
7              r = f(*args, **kw)
8              t2 = time.time()
9              t = (t2-t1) * 1000 if unit=='ms' else (t2
                   -t1)
10             print 'call %s() in %f %s' % (f.__name__,
                   t, unit)
11             return r
12         return fn
13     return per_dec
14
15 @performance('ms')
16 def factorial(n):
17     return reduce(lambda x,y: x*y, range(1, n+1))
18
19 print factorial(10)
```

1.13 python 中完善 decorator

在使用 decorator 的情况下，打印函数名：

```
def log(f):
    def wrapper(*args, **kw):
        print 'call...'
        return f(*args, **kw)
    return wrapper
@log
def f2(x):
    pass
print f2.__name__
```

输出：wrapper

可见，由于 decorator 返回的新函数函数名已经不是‘f2’，而是@log 内部定义的’wrapper‘。这对于那些依赖函数名的代码就会失效。Decorator 还改变了函数的__doc__等其他属性。如果要让调用者看不出一个函数经过了@decorator 的“改造”，就需要把原函数的一些属性复制到新函数中：

```
def log(f):
    def wrapper(*args, **kw):
        print 'call...'
        return f(*args, **kw)
    wrapper.__name__ = f.__name__
    wrapper.__doc__ = f.__doc__
    return wrapper
```

这样写 decorator 很不方便，因为我们也很难把原函数的所有必要属性都一个一个复制到新函数上，所以 python 内置的 functools 可以用来自动化完成这个复制任务：

```
import functools
def log(f):
    @functools.wraps(f)
    def wrapper(*args, **kw):
        print 'call...'
        return f(*args, **kw)
    return wrapper
```

注意，由于把原函数签名改成了（*args, **kw），因此，无法获得原函数的原始参数信息。即便采用固定参数来装饰只有一个参数的函数，也可能改变原函数的参数名。

@functools.wraps(f)

1.14 python 中偏函数

int()函数可以把字符串转换为整数，当仅传入字符时，默认按十进制转换。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，就可以按传入的 base 参数进制转换。

```
>>> int('12345' ,base=8)
5349
>>> int('12345', 16)
74565
```

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

```
>>> def int2(x, base=2):
>>>     return int(x, base)
```

这样，转换二进制就非常方便了，只用传入字符串。

functools.partial 就是帮助我们创建一个偏函数的，不需要自己定义 int2()，可直接使用下面的代码创建一个新的函数 int2()：

```
>>> import functools
>>> int2 = functools.partial( int, base=2 )
>>> int2('1000000')
64
```

所以，functools.partial 可以把一个参数多的函数变成一个参数少的新函数，少的函数需要在创建的时候指定默认值，这样，新函数调用的难度就降低了。

二　模块

2.1　python 中模块和包的概念