### 函数式编程

* 1. reduce()

reduce(fn,list)

|  |
| --- |
| def fn(x,y):  return x+y |

fn必须有两个参数，计算过程如下

|  |
| --- |
| reduce(f, [1, 3, 5, 7, 9])  先计算头两个元素：f(1, 3)，结果为4；  再把结果和第3个元素计算：f(4, 5)，结果为9；  再把结果和第4个元素计算：f(9, 7)，结果为16；  再把结果和第5个元素计算：f(16, 9)，结果为25；  由于没有更多的元素了，计算结束，返回结果25。 |

reduce(fn, [1, 3, 5, 7, 9],100) 接受第三个参数，初始值，第一步运算取此初始值和list第一个元素。

注意： python3中使用reduce前需要from functools import reduce

* 1. filter(fn,list)

将list中的每一个元素都经过fn处理，返回True则保存到新list中，返回False则过滤掉。

* 1. s.strip(rm) 删除s开头和结尾处rm序列的字符

若rm为空，默认删除空白字符 '\n', '\r', '\t', ' '

* 1. sorted(list[,fn])

若没有第二个参数，默认升序排列

若有第二个参数，可以自定义排序规则

排序：如果希望 x 排在 y 的前面，则返回 -1，如果希望 x 排在 y 的后面，返回 1。如果希望 x 和 y 相等，返回 0。

|  |
| --- |
| 自定义倒序规则：  def reverse(x,y):  if x>y:  return -1  if x<y:  return 1  return 0 |

* 1. 匿名函数

lambda x:x\*x

等同于

|  |
| --- |
| def f(x):  return x\*x |

冒号前面表示参数，后面是返回结果

限制：只能有一个表达式作为结果返回，不能进行其他运算

### 装饰器

装饰器：把一个函数当做参数传入装饰器，并赋值给原函数名，返回一个新函数，但还是使用旧函数名。

* 1. 无参数decorator

使用@语法，相当于 f=decorator(f)

|  |
| --- |
| def log(f):  def fn(x):  print 'call ' + f.\_\_name\_\_ + '()...' #函数增加的内容  return f(x) #还是返回传入的函数的返回值  return fn #返回新函数  def factorial(n):  return reduce(lambda x,y: x\*y, range(1, n+1))  传统方法：  factorial = log(factorial)  使用@语法：（在就函数前面一行使用@log）  @log  def factorial(n):  return reduce(lambda x,y: x\*y, range(1, n+1)) |

|  |
| --- |
| 但是若函数是多个参数，上面方法就不行了，因为定义log时，里面传的是一个参数（上面绿色部分），所以定义log时，fn传入的参数可以用\*args,\*\*kw来代替。  def log(f):  def fn(\*args,\*\*kw):  print ('call ' + f.\_\_name\_\_ + '()...')  return f(\*args,\*\*kw)  return fn |

* 1. 详细了解python装饰器

<http://python.jobbole.com/85056/>

* 1. 带参数的装饰器

上面没带参数的装饰器，log返回的函数是不能变的，若有需求，有时需要打印出【结果1】 ，有时候需要打印出【结果2】，所以需要给log传参数。

|  |
| --- |
| def log(prefix):  def log\_decorator(f):  def wrapper(\*args, \*\*kw):  print '[%s] %s()...' % (prefix, f.\_\_name\_\_)  return f(\*args, \*\*kw)  return wrapper  return log\_decorator  @log(‘结果1’)  def test():  pass  print test() |

即增加一个prefix参数，可以通过装饰器改变增加的内容。

解释： 1）log\_decorator = log(‘结果1’)

2）@log\_decorator 后面就跟不带参数的装饰器一样了

* 1. @functools.wraps(f) 被装饰器改造过的函数的一些属性复制给新函数

|  |
| --- |
| import functools  def log(f):  @functools.wraps(f)  def wrapper(\*args, \*\*kw):  print( 'call...')  return f(\*args, \*\*kw)  return wrapper  @log  def fn():  print('fn')  print(fn.\_\_name\_\_)  若没有@functools.wraps(f)  print(fn.\_\_name\_\_)结果是wrapper，即新函数名。 |

* 1. functools.partial()

例如int2 = function.partial(int,base=2)

这个函数可以把参数多的函数变成参数少的，在创建的时候传入默认值即可。

1. 模块
   1. 导入模块

import math

math.pi #可以使用math模块里的所有函数

import os.path 模块也有层级

使用import引入的模块调用时，都要写上完整的模块名

若只想引入math中的部分函数：

from math import pow , log

调用时，可以不带模块名，直接使用pow(1,2);

这样调用势必会冲突，所以可以为函数设置别名：

from logging import log as logger 这样就避免冲突了

* 1. 错误捕捉

利用错误捕捉，动态导入模块：

|  |
| --- |
| try:  from cStringIO import StringIO  catch ImportError:  from StringIO import StringIO |

先尝试从cStringIO里引入，若失败，则从StringIO里引入

* 1. 从旧版本的python中测试新特性

from \_\_future\_\_ import XXX