### 函数式编程

#### map()

map(fn,list)

fn：函数

list：有序集合

map将list里的每个元素都经过fn函数处理，返回一个新的list，并不改变原来的list

python 3 中需要用list（）处理下才能返回list

#### 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

#### 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。

python 3 中返回的是个filter对象，需要用list()处理下才能显示

#### 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', ' '

#### 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 |

根据key排序：

L = [('b',2),('a',1),('c',3),('d',4)]

sorted(L,key=lambda x:x[1]) 按照value排序

sorted(L,key=lambda x:x[0]) 按照key排序

是否逆序：

sorted([5, 2, 3, 1, 4], reverse=True) #逆序，按从大到小排列

输出 [5, 4, 3, 2, 1]

sorted([5, 2, 3, 1, 4], reverse=False) #不逆序，按从小到大排列

[1, 2, 3, 4, 5]

#### 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。

#### 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 |

#### 详细了解python装饰器

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

#### 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

#### @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。 |

#### 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。

### 模块

#### 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

注意：一个目录要想作为python的模块包必须含有\_\_init\_\_.py

#### 模块搜索机制

1)程序的当前目录

2)PYTHONPATH目录，也就是环境变量里面设置的PYTHON目录

3)标准库的目录

#### 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

#### 从旧版本的python中测试新特性

from \_\_future\_\_ import XXX

#### \_\_name\_\_

这个系统变量显示了当前模块执行过程中的名称。

如果我们直接执行该文件，那么\_\_name\_\_就等于\_\_main\_\_；

如果另一个py文件通过import引入该文件，这时\_\_name\_\_的值就是这个文件的名字。

### 面向对象编程

#### 类

类的定义 class Person（object）:

pass

实例化 xiaoming = Person()

#### \_\_init\_\_() 给实例对象设置初始属性

\_\_init\_\_(self,name,age,gender):

self.name = name

self.age = age

self.gender = gender

第一个参数必须是self（指的是对象自身，并不是类class），（可以用其他名，但习惯用self）

注意：\_\_init\_\_方法并不是类的构造方法。

另外还有析构方法：

def \_\_del\_\_(self):当对象不再使用时自动调用，也即当对象的所有引用都不存在时

#### \_\_new\_\_() 返回一个实例对象

\_\_new\_\_是实例化一个类时，第一个调用的方法，一般不需要重新定义，除非我们需要继承一个不可变的类，比如int，str，set等。

下面的类继承了不可变类型int，因为int是不可变类型，所以要重写他的时候，必须在创建的时候重写\_\_new\_\_方法。

|  |
| --- |
| class Nint(int):  def \_\_new\_\_(cls, num): #第一个参数是cls  if type(num)==str:  sum=0  for i in num:  sum +=ord(i)  return int.\_\_new\_\_(cls,sum)  else:  return int.\_\_new\_\_(cls,num)  print(Nint('ABC')) |

#### 为对象设置属性

setattr(self,name,value)

#### 私有属性

class Person(object):

def \_\_init\_\_(self, name):

self.name = name

self.\_title = 'Mr'

self.\_\_job = 'Student' #外部访问不到

但是，其实在外部也可以访问到（伪私有）：

person = Person()

person.\_Person\_\_job 即可访问到

#### 类属性（所有实例化的对象都共享的属性）

|  |
| --- |
| class Person(object):  address = 'Earth'  def \_\_init\_\_(self, name):  self.name = name |

类属性只有一份，所有实例访问的类属性都是同一个。

不用实例化就可以访问到，无论在类里面还是外面，都是 类名.属性名

例 ：Person.address

#### 实例属性

class Person(object):

def \_\_init\_\_(self, name):

self.\_\_name = name

def get\_name(self): #第一个参数必须是self

return self.\_\_name

调用： xiaoming = Person(‘xiaoming’)

xiaoming.get\_name() ##调用时不用self参数

#### 类方法

|  |
| --- |
| class Person(object):  count = 0 类属性  @classmethod 类方法  def how\_many(cls): cls指类本身，此处相当于Person  return cls.count  def \_\_init\_\_(self, name):  self.name = name 实例属性  Person.count = Person.count + 1  def whatName(self): 实例方法（第一个参数是self，实例对象本身）  return self.name  print (Person.how\_many())  # print(Person.whatName()) 实例方法不能用类来调用  print(Person.count)  p1 = Person('Bob')  实例化类之后，除了\_\_init\_\_初始化的属性和方法，实例化对象会继承类属性和类方法，所以也可以用实例化对象调用类方法和类属性。但是类名不能用来调用实例对象和方法。  print(p1.whatName())  print(p1.count)  print(p1.how\_many()) |

#### 类的静态方法

@staticmethod

不需要表示自身对象的self和自身类的cls参数，就跟使用函数一样

只能直接类名.属性名或类名.方法名来访问类的属性和方法

而@classmethod因为持有cls参数，可以来调用类的属性，类的方法，实例化对象等（cls参数表示当前类，在子类继承的时候，他可以表示当前的子类，这时候就不能用staticmethod）

|  |
| --- |
| class Person(object):  id = 'person'  @staticmethod  def speak():  print(Person.id)  @classmethod  def walk(cls):  print(cls.id)  class Student(Person):  id = 'student'  Student.speak()  Student.walk() #这里调用的walk，里面的参数cls指的就是Student了  输出：  person  student |

#### 类的继承

|  |
| --- |
| class Person(object):  def \_\_init\_\_(self, name, gender):  self.name = name  self.gender = gender  class Student(Person):  def \_\_init\_\_(self, name, gender, score):  #函数super(Student, self)将返回当前类继承的父类，即 Person ，然后调用\_\_init\_\_()方法；  #self参数已在super()中传入，在\_\_init\_\_()中将隐式传递，必须要传递了  super(Student, self).\_\_init\_\_(name, gender) #初始化父类，继承父类的属性  self.score = score  xiao = Student('xiaoming','male',90)  print(xiao.name) |

#### 类的组合

组合用于“有一个”的场景，继承用于“是一个”的场景

在类定义时，直接把需要的类放进去实例化就行了

|  |
| --- |
|  |

#### 判断类型

isinstance() 判断一个变量的类型，可以用在python内置的数据类型，如str,int,list，也可以用于自定义的类

用法：isinstance（对象，类名） 返回True或False

#### 多继承

一个子类可以同时继承多个父类

class A（B,C）:

pass

#### 类的内置函数

type(s) 获取变量的类型

dir(s) 获取变量的所有属性

getattr(s,’name’) #获取name属性，若属性不存在就报错

getattr(s,’name’,’john’) #获取name属性，若属性不存在就返回默认值john

setattr(s, 'name', 'Adam') #设置属性值

hasattr(object,attrname) #判断对象是否含有此属性

issubclass(subclass,class) subclass是否是class的子类，是返回True，否则返回False；第二个参数可以是tuple，只要里面有一个他的父类，就返回True

#### 魔术方法

http://bbs.fishc.com/thread-48793-1-1.html

print时，会调用类中的\_\_str\_\_方法

直接敲变量 a，会调用\_\_repr\_\_,不会调用\_\_str\_\_;

一般偷懒，这样定义：当\_\_str\_\_定义之后，直接\_\_repr\_\_=\_\_str\_\_即可

### python中的正则

#### re模块

re.match(reg,string) 尝试从字符串的起始位置匹配一个模式，如果不是起始位置匹配成功的话，match()就返回None。

分组：

|  |
| --- |
| str='010-12345'  reg= r'(\d{3})-(\d{5})'  res = re.match(reg,str)  if res:  print(res.groups()) 所有子组的一个tuple集合  print(res.group(0)) group(0)永远是原始字符串  print(res.group(1)) group(1)表示第一个子串  print(res.group(2)) group(2)表示第二个子串，依次类推  输出：  ('010', '12345')  010-12345  010  12345 |

re.split(reg,string) 根据正则，分割字符串

re.search(reg,string) 扫描整个字符串并返回第一个成功的匹配。用法同match

re.match和re.search区别：

re.match只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回None；而re.search匹配整个字符串，直到找到一个匹配。

re.sub(reg,replace,string)检索或替换

re.findall(reg,string) 返回全部能匹配的子串的列表

re.finditer(reg,string) 返回一个顺序访问每一个匹配结果的迭代器

#### 编译

python使用正则，re模块内会做两件事：

1. 编译正则表达式，如果正则本身不合法，会报错
2. 用编译后的正则去匹配字符串

若一个正则要重复使用上千次，处于效率考虑，我们可以预编译正则，接下来重复使用时就不需要编译这个步骤了，直接匹配。

regpre=re.compile(reg)

regpre.match(string)

编译后生成Regular Expression对象，由于该对象自己包含了正则表达式，所以调用对应的方法时不用给出正则字符串。

#### \b单词分界

\b匹配这样的位置：它的前一个字符和后一个字符不全是(一个是,一个不是或不存在)\w（匹配字母或数字或下划线或汉字）

### 操作数据库

##### Python DB API

包含数据库连接对象connection，数据库交互对象cursor，数据库异常对象exceptions

安装PyMySQL： cmd 下pip install PyMySQL

|  |
| --- |
| import pymysql  # 打开数据库连接  db = pymysql.connect(host='localhost',port=3307,user='root',passwd='',db='testdb')  # 使用 cursor() 方法创建一个游标对象 cursor  cursor = db.cursor()  # 使用 execute() 方法执行 SQL 查询  cursor.execute("SELECT \* from employee")  # 使用 fetchone() 方法获取单条数据.  data = cursor.fetchone()  #输出，是所有字段组成的一个tuple  print(data)  #关闭数据库连接  db.close() |

connection对象的方法：

cursor()

commit()

rollback()

close()

##### 查询

fetchone(): 该方法获取下一个查询结果集。结果集是一个对象

fetchall(): 接收全部的返回结果行.

rowcount: 这是一个只读属性，并返回执行execute()方法后影响的行数。

##### 增删改

对于支持事务的数据库， 在Python数据库编程中，当游标建立之时，就自动开始了一个隐形的数据库事务。

|  |
| --- |
| try:  # 执行SQL语句  cursor.execute(sql)  # 向数据库提交  db.commit()  except:  # 发生错误时回滚  db.rollback() |

增删改操作都需要执行commit()，否则无效

### 第三方模块

##### Pillow

|  |
| --- |
| from PIL import Image  #打开一个图片  im = Image.open('test.jpg')  #进行图片缩放  new\_size=(20,10)  im.thumbnail(new\_size)  #将缩放后的图像另存  im.save('thumbnail.jpg') |

画验证码：

|  |
| --- |
| *from* PIL *import* Image, ImageFont, ImageDraw *import* random #随机字母 *def* radChar():  *return* chr(random.randint(65,90)) #随机颜色 *def* radColor():  *return* (random.randint(64,255),random.randint(64,255),random.randint(64,255)) #随机颜色2 *def* radColor2():  *return* (random.randint(32,127),random.randint(32,127),random.randint(32,127)) #创建一个图片对象 im = Image.new('RGB',(300,60),(255,255,255)) #创建字体对象 font = ImageFont.truetype('C:\\Windows\\winsxs\\amd64\_microsoft-windows-font-truetype-arial\_31bf3856ad364e35\_6.1.7601.22739\_none\_d1225f21dd240b54\\Arial.ttf',36) #创建绘图对象 draw = ImageDraw.Draw(im)  #在图片上每个像素点上色 *for* x *in* range(300):  *for* y *in* range(100):  draw.point((x,y),fill=radColor()) #画随机线段 *for* i *in* range(300):  x = random.randint(0, 300)  y = random.randint(0, 60)  draw.line((x,y,x+random.randint(-100,100),y+random.randint(-100,100)),fill=radColor()) #四个验证码字符 *for* t *in* range(4):  draw.text((60\*t+30,10),radChar(),fill=radColor2(),font=font) #图片保存 im.save('char.jpg') |

im = Image.new(mode,size,color=0) 创建图片对象

font = ImageFont.truetype(font,size) 加载并创建字体对象

参数：font，字体文件路径

size, 字体大小

draw = ImageDraw.Draw(im) 创建一个能在图片上绘画的对象 参数：im，图片对象

draw.point(xy,fill) 画点 xy:坐标点的集合，fill：颜色

draw.line(xy,fill,width=0) 画线 参数同上类似

draw.text(xy,text,fill=None,font=None)

xy：（left,top）左边距，上边距的一个tuple

text：需要画的字

font：上面创建的字体对象

##### random

random.randint(a,b) 返回a-b之间的一个随机整数

random.choice(list) 随机返回list中的一个元素

##### time模块

time.time() 当前时间戳

time.localtime([secs]) 可接受时间戳，返回当前时间的时间元祖（无参数时）

time.sleep(secs) 休眠几秒

time.mktime(t) 接受一个时间元祖，返回时间戳

t参数为(年，月，日，时，分，秒，周几，年天，0)

time.perf\_counter()  
返回计时器的精准时间（系统的运行时间），包含整个系统的睡眠时间。由于返回值的基准点是未定义的，所以，只有连续调用的结果之间的差才是有效的。

time.process\_time()   
返回当前进程执行 CPU 的时间总和，不包含睡眠时间。由于返回值的基准点是未定义的，所以，只有连续调用的结果之间的差才是有效的。

time.strftime(format[, t]) 把一个时间元祖，格式为时间字符串，不传t，默认传入time.localtime()

format 常用格式：%Y-%m-%d %H:%M:%S

|  |  |
| --- | --- |
| **格式** | **含义** |
| %a | 本地（locale）简化星期名称 |
| %A | 本地完整星期名称 |
| %b | 本地简化月份名称 |
| %B | 本地完整月份名称 |
| %c | 本地相应的日期和时间表示 |
| %d | 一个月中的第几天（01 - 31） |
| %H | 一天中的第几个小时（24 小时制，00 - 23） |
| %l | 一天中的第几个小时（12 小时制，01 - 12） |
| %j | 一年中的第几天（001 - 366） |
| %m | 月份（01 - 12） |
| %M | 分钟数（00 - 59） |
| %p | 本地 am 或者 pm 的相应符 |
| %S | 秒（01 - 61） |
| %U | 一年中的星期数（00 - 53 星期天是一个星期的开始）第一个星期天之前的所有天数都放在第 0 周 |
| %w | 一个星期中的第几天（0 - 6，0 是星期天） |
| %W | 和 %U 基本相同，不同的是 %W 以星期一为一个星期的开始 |
| %x | 本地相应日期 |
| %X | 本地相应时间 |
| %y | 去掉世纪的年份（00 - 99） |
| %Y | 完整的年份 |
| %z | 用 +HHMM 或 -HHMM 表示距离格林威治的时区偏移（H 代表十进制的小时数，M 代表十进制的分钟数） |
| %Z | 时区的名字（如果不存在为空字符） |
| %% | %号本身 |

##### timeit模块 准确测量小段代码的执行时间

http://bbs.fishc.com/thread-55593-1-1.html