目录

[数据类型 2](#_Toc485289332)

[声明变量 2](#_Toc485289333)

[字符串 2](#_Toc485289334)

[列表list: 3](#_Toc485289335)

[元组tuple 4](#_Toc485289336)

[if elif else语句 4](#_Toc485289337)

[循环语句和contuine 以及 break 4](#_Toc485289338)

[字典dict 5](#_Toc485289339)

[List与dict的区别 6](#_Toc485289340)

[set 6](#_Toc485289341)

[切片 6](#_Toc485289342)

[列表生成式 6](#_Toc485289343)

[函数 7](#_Toc485289344)

[高阶函数 7](#_Toc485289345)

[函数返回函数 9](#_Toc485289346)

[闭包 9](#_Toc485289347)

[匿名函数 10](#_Toc485289348)

[装饰器decorator 10](#_Toc485289349)

[自适应任意个数的参数的函数 11](#_Toc485289350)

[在@xxx()中传入参数,在装饰方法中使用参数 11](#_Toc485289351)

[偏函数 12](#_Toc485289352)

[模块和包 12](#_Toc485289353)

[动态导入模块 12](#_Toc485289354)

[导入新特性进行试用 13](#_Toc485289355)

[安装第三方模块 13](#_Toc485289356)

[Python的类 13](#_Toc485289357)

[类的属性 14](#_Toc485289358)

[实例方法 14](#_Toc485289359)

[类方法 15](#_Toc485289360)

[类的继承 15](#_Toc485289361)

[多重继承 16](#_Toc485289362)

[获取对象信息的一些方法 17](#_Toc485289363)

[特殊方法 17](#_Toc485289364)

[对set方法进行校验 17](#_Toc485289365)

[限制类拥有的属性 18](#_Toc485289366)

[将类变为可调用对象 19](#_Toc485289367)

[异常捕获 19](#_Toc485289368)

[传统的try语句 19](#_Toc485289369)

[with语句 19](#_Toc485289370)

[raise语句 20](#_Toc485289371)

[assert语句 20](#_Toc485289372)

## 数据类型

需要注意的就只有:Python把0、空字符串''和None看成 False,其他数值和非空字符串都看成 True.

Python没有++和—运算符, python 中，变量是以内容为基准而不是以变量名为基准，只要数字内容相同，不管变量名是否相同,这个变量的 ID 是相同的,所以如果允许++和—操作,当一个变量改变了另外一个变量也会改变,这样是不正确的.

可以使用库函数的 isinstance(x,type) 判断一个变量是否为对应类型

## 声明变量

Python是一门动态语言,变量本身的类型不固定,在变量名后直接进行赋值即可.

a = 1 b = 1.2

## 字符串

字符串的类是str,表示字符串使用单引号或使用双引号,字符串中的特殊字符可以使用 \ 进行转义.

如果一个字符串包含很多需要转义的字符，可以在字符串前面加个前缀 r ,表示这是一个 raw 字符串,里面的字符不需要转义. 如: r'\(~\_~)/ \(~\_~)/'

r'...'表示法不能表示多行字符串，也不能表示包含 ' 和 " 的字符串

要表示多行字符串，可以用'''...'''表示,可以在多行字符串前面添加r,把多行字符串也变成一个raw字符串,raw的多行字符串中可以包含 ' 和 ".

以Unicode表示的字符串用u'...'表示，比如：print u'中文'

如果中文字符串在Python环境下遇到 UnicodeDecodeError，这是因为.py文件保存的格式有问题,可以在第一行添加注释: # -\*- coding: utf-8 -\*-

代码编辑第一行的注释表示文件中的代码使用utf-8编码，在代码中还使用 u'''...'''会让解释器使用Unicode编码输出导致解码出错.

操作基本数据类型和字符串的一些基本方法:

int(x [,base]) 将x转换为一个整数,当传入第二个参数时第一个参数必须为字符串,表示这个字符串为多少进制.

float(x) 将x转换到一个浮点数

str(x) 将对象x转换为字符串

repr(x) 和str()并没有什么不同,区别在于repr()输出对Python比较友好,而str()的输出对用户比较友好.

eval(str) 将字符串转为数据类型,比如list的字符串转换为list对象

tuple(s) 将序列s转换为一个元组

list(s) 将序列s转换为一个列表

chr(x) 将一个整数转换为一个字符

unichr(x) 将一个整数转换为Unicode字符

ord(x) 将一个字符转换为它的整数值

hex(x) 将一个整数转换为一个十六进制字符串

oct(x) 将一个整数转换为一个八进制字符串

**lstrip([chars]) 从头去掉包含传入参数字符的部分,直到遇到字符不为传入的字符串中的元素,默认为空,即空格.**

rstrip([chars]) 与上面一样,不过是从字符串尾开始

strip([chars]) 去除字符串中包含传入字符串元素的内容,默认为空串,即去掉空格.

join(x) 将传入的序列,字符串,元组或字典根据调用的字符串作为分隔符拼接

## 列表list:

是一种有序的集合(相当于java中的数组)，可以随时添加和删除其中的元素。用 [ ] 把list的所有元素都括起来，就是一个list对象。 比如: L = ['1', '2', '3']

Python是动态语言，list中包含的元素并不要求都必须是同一种数据类型，可以包含各种数据

根据索引访问和更新list中的元素(与java中的数组一样),比如: L[0] = 2 ;索引为负数时代表倒数多少个,比如倒数第一个元素: L[-1]

使用del进行删除,比如: del L[1]

列表对+和\*的操作符与字符串相似,+号用于组合列表，\*号用于重复列表

**print** [1, 2, 3] + [4, 5, 6]; # 输出[1, 2, 3, 4, 5, 6]  
**print** ['Hi!'] \* 4; # 输出 ['Hi!', 'Hi!', 'Hi!', 'Hi!']

list的方法:

list.append(obj) 在列表末尾添加新的对象

list.count(obj) 统计某个元素在列表中出现的次数

list.extend(seq) 在列表末尾一次性追加另一个序列中的多个值

list.index(obj) 从列表中找出某个值第一个匹配项的索引位置，索引从0开始

list.insert(index, obj) 将对象插入列表

list.pop(obj=list[-1]) 移除列表中的一个元素(默认最后一个)，并且返回该元素的值

list.remove(obj) 移除列表中某个值的第一个匹配项

list.reverse() 反向列表中元素，倒转

list.sort([func]) 对原列表进行排序,可以传入一个比较函数、设置是否逆序

## 元组tuple

另一种有序的列表, tuple一旦创建完毕,就不能修改了,创建tuple用( )

比如: t = ('Adam', 'Lisa', 'Bart')

但是实际上元组是由逗号决定的,不是小括号,即便没有了小括号也还是元组:

比如: t = "a", "b", "c", "d"

当tuple中存储的元素为list时,list的内容可变(类似java中被final修饰的集合)

当()内只有一个元素时, ()既可以表示tuple又可以作为括号表示运算时的优先级,()会被Python解释器计算出结果,导致得到的不是tuple,而是其中的内容. 所以单元素 tuple 要多加一个逗号","

比如: (1,)

tuple没有append()方法,也没有insert()和pop()方法,访问元素的方法与list一样,也可以使用+和\*进行运算,元组无法删除元素,可以使用del语句来删除整个元组

+ 和 \* 对于列表和元组都可以产生作用,但是对于后面的字典是无法使用的.

对于有序的集合,其中的元素确定是有索引的,使用库函数的enumerate()函数可以获取其中的索引和元素,这个函数会将每个元素变为存储元组的集合,其中的每个元素都是 (index,value)形式的元组,就可以在for循环中使用解构声明获取到元组的两个值

## if elif else语句

**if** 2 > 1 :  
 **print** '123'  
**elif** 1 > 2:  
 **print** '234'  
**else** :  
 **print** '345'

## 循环语句和contuine 以及 break

L = [1, 2, 3]  
**for** i **in** L:  
 **if** i == 2:  
 **continue  
 print** i

**结构声明**: **for** x, y **in** enumerate(list):

i = 1  
**while** True:  
 **if** i == 2:  
 **break  
 print** i  
 i += 1

## 字典dict

字典dict(相当于java中的Map)使用花括号 {} 表示,然后按照 key: value, 写出来即可,最后一个key: value后的逗号可以省略,存储的key-value序对没有顺序,在一个dict中,key不能重复.

比如: d = {'Adam': 95, 'Lisa': 85, 'Bart': 59}

dict作为key的元素必须不可变,Python的基本类型如字符串、整数、浮点数都是不可变的,都可以作为 key,但list是可变的,不能作为 key,这个限制仅作用于key,value是否可变无所谓.在字典中value无论是列表还是元组,都可以修改.(实际上是改掉了key指向的内容)

**访问**: 使用 d[key] 的形式来查找对应的 value 比如 : d['Adam']

通过key访问dict的value,若key存在,dict会返回对应的value,若key不存在,会直接报错:KeyError,若想避免出现该情况:

1. 判断key 是否存在,用 in 操作符或用has\_key(key)方法判断

if 'Paul' in d:

2. 使用dict本身提供的 get 方法,在Key不存在的时候会返回None

**添加**: 使用 d[new key] = value的形式来添加键值对, 比如: d['Paul'] = 72

如果 key 已经存在则赋值会用新的value替换掉原来的value

**删除:** 使用del dict[key]删除dict中键是key的条目

dict.clear() 清空词典所有条目

del dict 删除对应名称的词典.

**迭代dict的key值** : 使用for循环遍历dict时, for a in d: 其中的a为dict中的key值

**迭代dict的value**值 : dict对象有一个 values() 方法,这个方法把dict转换成一个包含所有value的list

除了values()方法还有一个itervalues()方法,它与values()方法迭代效果完全一样,values()方法实际上把一个 dict 转换成了包含 value 的list,但是 itervalues() 方法不会转换,它会在迭代过程中依次从dict中取出value，itervalues()方法比 values()方法节省了生成 list 所需的内存.

**同时迭代dict的key和value** : 使用items()方法,会把dict对象转换成了包含tuple的list，对这个list进行迭代即可以同时获得key和value. 例: for a, b in d.items():

items()也有一个对应的 iteritems()，iteritems() 不把dict转换成list，而是在迭代过程中不断给出 tuple，所以iteritems() 不占用额外的内存.

dict的几个其他方法:

dict.fromkeys() 创建一个新字典,当传入一个序列参数时,以该序列为key,value都为默认值.若传入第二个参数,value都以该参数为默认值

dict.setdefault(key, default=None) 如果键不存在，将会添加键并将值设为default,若键存在不会改变值

dict.update(dict2) 把字典dict2的键值对添加到dict里

### List与dict的区别

dict的查找速度快，无论dict有10个元素还是10万个元素，查找速度都一样。而list的查找速度随着元素增加而逐渐下降。

dict的缺点是占用内存大，还会浪费很多内容，list正好相反，占用内存小，但是查找速度慢。

## set

set(和java中的set一样) 持有一系列元素,没有重复并且是无序的,和 dict 的 key很像.

创建 set 的方式是调用 set() 并传入一个 list，list的元素将作为set的元素. 当传入包含重复元素的 list ,set会自动去掉重复的元素. 比如: 时s = set(['A', 'B', 'C'])

set是无序的,无法通过索引来访问,可使用for循环直接遍历,set的内部结构和dict很像，唯一区别是不存储value，判断一个元素是否在set中速度很快,set存储的元素必须是不变对象.

添加元素: 用set的add()方法：s.add(4),若添加的元素已经存在于set中,add()不会报错删除

移除元素: 用set的remove()方法:s.remove(x), 若删除的元素不存在set中，remove()会报错

## 切片

Python提供了切片（Slice）操作符从集合、元组中获取元素,也可用于字符串,对什么类型进行切片就产生什么类型的切片结果.

切片操作符的格式为是 [x:y] 或 [x:y:z]

x 表示从x索引开始,若x省略表示为头即0,若x为负数表示从倒数第x位置开始

y 表示从x索引到y索引,但不包括y索引位置的元素,若y省略表示为到结尾,若y为负数表示到倒数第y个位置,不包括y位置.

z 表示每z个取一个,若z为负数时表示从末尾处每z个取一个

比如:

L[0:3]表示，从索引0开始取，直到索引3为止，但不包括索引3,可以简写为 L[:3]

L[:]表示从头到尾

L[::-1] 会给出L倒序的对象

## 列表生成式

列表生成式的格式为: 要生成的元素放在前面,后面跟 for 循环

for 循环后面还可以加上 if 判断,表示只有if判断为True时才把循环的当前元素添加到列表中

**比如:** [x \* x for x in range(1, 11) if x % 2 == 0]

表示将从1到10的数中,处于2余数为0的数的平方存入列表

其中的for循环可以嵌套:

比如: b = [m + n **for** m **in** 'ABC' **for** n **in** '123']

结果为: ['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']

使用for循环还可以迭代dict:

比如: tds = ['%s%s' % (name, score) **for** name, score **in** d.iteritems()]

把dict d进行迭代,将name和score格式化为字符串,组成list

## 函数

定义函数使用def语句，依次写出函数名、括号、括号中的参数和冒号,

在缩进块中编写函数体,返回值用 return 语句返回.

比如: def method(x):  
 print x  
 return

Python的函数可返回多个值,其实返回的是单一值,其返回值是一个tuple,接收时按位置赋给对应的值.

比如: **return** x, y a, b = method()

Python的函数可定义默认参数(类似于java的方法重载),函数的默认参数的作用是简化调用,在不传入默认参数时,将以该默认参数在方法中进行适应,若传入了参数会覆盖默认的参数值.

比如: **def method**(x, y=2):

Python的函数可以定义可变参数,让一个函数能接受任意个参数，格式为在参数名称前加上一个\*,解释器会把传入的一组参数组装成一个tuple在函数内部，直接把变量 args 看成一个 tuple即可.

比如: **def method**(\*a):  
 **return** a[1]

## 高阶函数

变量可以指向一个函数,直接对这个变量进行调用和调用函数效果一样,函数名就是一个指向函数的变量.

f = abs   
**print** f(-12)  
abs = len  
**print** abs([12, 2, 3])

高阶函数就是指能接收函数作为参数的函数:

**def add**(x, y, z):  
 **return** z(x) + z(y)

**print** add(-1, -2, abs)

Python 内置的高阶函数:

**map()** 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.

比如: 将集合中的字符串转为首字母大写其余小写

**def format\_name**(s):  
 **return** s.lstrip()[0].upper() + s.lstrip()[1:].lower()

**print** map(format\_name, [' adam', 'LISA', 'barT'])

**reduce()**  接手一个函数f和一个list,传入的函数 f 必须接收两个参数，开始将集合中的前2个元素传入f,然后将得到的结果和下一个元素作为参数传入f,直到集合结束,可以接收第3个可选参数,作为计算的初始值.

比如: 以100为基数,计算集合中所有元素的和, 最后输出115

**def prod(x, y):** **return** x + y

**print** reduce(prod, [1, 2, 3, 4, 5], 100)

**filter()** 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 True或 False，filter()根据判断结果自动过滤掉不符合条件的元素,组成新list

比如: 选取1-100中的平方根为整数的数(math.sqrt为开平方方法,返回float)

**def is\_sqr**(x):  
 y = int(math.sqrt(x))  
 **return** y \* y == x  
  
**print** filter(is\_sqr, range(1, 101))

**sorted()** 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.

比如: 将字符串忽略首字母大小写根据ASCII进行排序

**def cmp\_ignore\_case**(s1, s2):  
 a = s1[0].upper()  
 b = s2[0].upper()  
 **if** a > b:  
 **return** 1  
 **elif** a < b:  
 **return** -1  
 **elif** a == b:  
 **return** 0  
  
**print** sorted(['bob', 'about', 'Zoo', 'Credit'], cmp\_ignore\_case)

## 函数返回函数

Python的函数除了返回一些数据类型,还可以返回函数,返回函数可以把一些计算延迟执行,直到对返回的函数进行调用时才会进行计算.

比如: 接收一个list,返回一个函数,返回函数可以计算参数的乘积。

**def calc\_prod**(list):  
 **def calc**():  
 **def f**(x, y):  
 **return** x \* y  
 **return** reduce(f, list)  
 **return** calc   
  
f = calc\_prod([1, 2, 3, 4]) #返回了一个函数  
**print** f() #在此处对函数进行了调用

## 闭包

在函数内部定义的函数和外部定义的函数是一样的,只是无法被外部访问, 当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,要正确使用闭包,要确保引用的局部变量在函数返回后不能变, 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.

比如: 在count函数中返回了f函数的集合,f函数中进行对i自乘的运算,但是在返回fs之后,i的值为3,在调用f1 f2 f3时其中的i都为3.

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

可以进行如下修正: 在count中定义f函数,f函数中定义g函数,g函数返回一个函数并使用f函数中的变量,使用i调用f函数,返回了g函数,每次调用f函数都会重新初始化一个g, g函数中的j并不是同一个变量,最终调用时结果正常.

**def count**():  
 fs = []  
 **for** i **in** range(1, 4):  
 **def f**(j):  
 **def g**():  
 **return** j \* j  
 **return** g  
 r = f(i)  
 fs.append(r)  
 **return** fs  
f1, f2, f3 = count()  
**print** f1(), f2(), f3()

## 匿名函数

匿名函数通过lambda实现, 格式为: lambda函数参数 : 计算内容 其中计算内容会被返回

比如: lambda x: x \* x

lambda表达式可以使用变量接收,然后进行调用.

比如: myabs = lambda x: -x if x < 0 else x

myabs(-1)

## 装饰器decorator

当需要在一个函数运行时增加一些功能又不想改变之前函数本身的代码时就可以使用装饰器.(本质上decorator就是一个高阶函数)

经过decorator装饰后的函数,函数名等其他属性都被改变了,在最里层函数上使用functools的wrap()方法可以将原函数除参数信息以外的内容都复制到新函数上.

比如: 拥有一个函数a ,需要在a中添加打印日志的内容

**def a**(x):  
 **return** x \* 2

使用函数b来对其进行装饰:

**def b**(f):

@functools.wraps(f)  
 **def c**(x):  
 # 这里写装饰的内容  
 **return** f(x) # 返回了原函数的调用结果  
  
 **return** c

最后调用:

b(a)(2) # 这里实际上是调用的c

或者

a = b(a) # 将a变量指向c方法  
a(2)

Python提供了@语法来简化装饰: 在旧方法上使用@新方法名 , 旧函数就会指向新方法中返回的函数

@b  
**def a**(x):  
 **return** x \* 2

### 自适应任意个数的参数的函数

上面的例子中,f(x)处写死了参数为x,所以只支持方法参数为一个的方法,当参数为多个时会报错.

如果要让装饰方法自适应各种不同参数数量的方法,需要利用Python的 \*args 和 \*\*kw ,就可以让任意个数的参数正常调用.

\*args 用来[解包list]将参数打包成tuple给函数体调用

\*\*kw 打包关键字参数成dict给函数体调用

比如:

**def b**(f):

@functools.wraps(f)  
 **def c**(\*args, \*\*kw):  
 # 装饰的内容  
 **return** f(\*args, \*\*kw)  
  
 **return** c

### 在@xxx()中传入参数,在装饰方法中使用参数

如果要在@xxx()中传入参数,需要在装饰方法外套一层方法接收该参数,这个参数就可以在里面的方法中使用,之前的返回值为最里层的方法,在套上之后,将套入的方法返回即可.(最里层的方法返回了需要装饰的方法,第二层返回了最里层的方法,最外层继续返回下一层的方法即可,最终返回的即需要装饰的方法)

比如:

**def d**(x):  
 **def b**(f):

@functools.wraps(f)  
 **def c**(\*args, \*\*kw):  
 # 装饰的内容  
 **print** x  
 **return** f(\*args, \*\*kw) #这里返回了传入的方法  
 **return** c #这里返回了c也就相当于返回了传入的方法  
 **return** b #返回b等于返回c等于返回传入的方法  
  
  
@d('1') #就相当于把参数传入d,并将a方法传入b,把x传入了c的参数  
**def a**(x):  
 **return** x \* 2

## 偏函数

如果一个包含默认参数的方法需要覆盖其默认参数为并同一值调用多次时,可以定义一个新的方法将该需要覆盖的参数传入,这个方法就叫做偏函数

比如:

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

Python中提供了functools.partial()方法对偏函数进行简单实现:

int2 = functools.partial(int, base=2)

## 模块和包

每一个.py文件就是一个模块,在一个模块中调用另一个模块使用import进行引用,既可引用到其他模块中的方法和变量. 比如: **import** math

使用导入的模块的函数和变量时格式为: 模块名.函数名/变量名

模块可以放在不同的包中,在import要写入完整的包名,包即文件夹,可以有多级,在Python的每一个包的每一个目录下必须有一个\_init\_.py文件,即使该文件内容为空.

若只需要导入模块中的某几个函数和变量,可以使用: from 模块名 import 函数名/包名

这样导入的函数和变量使用时前面不需要加上 模块名. 所以多个模块中导入的函数为重名时会产生冲突,可以使用 as关键字将函数名更改为别名.

比如: **from** os.path **import** isdir, isfile **as** a

**import** os.path **as** b

### 动态导入模块

当导入的模块不存在是会导入失败,导入失败时会报错,可以利用try except语句对捕获异常并进行处理.

由于版本和库的一些情况就需要使用该方式进行处理.

比如: Python 2.6/2.7提供了json 模块，但Python 2.5以及更早版本没有json模块，不过可以安装一个simplejson模块，这两个模块提供的函数签名和功能都一模一样。

**try**:  
 **import** json  
**except** ImportError:  
 **import** simplejson **as** json  
  
**print** json.dumps({'python': 2.7})

### 导入新特性进行试用

Python的新版本会引入新的功能,实际上这些功能在上一个老版本中就已经存在了,若要试用某一新的特性,可以通过导入 \_\_future\_\_ 模块的某些功能来实现,该语句必须是模块或程序的第一个语句,否则会报错

比如: Python中2.7的 / 的结果为整数,3.0以后 / 结果为浮点数, // 才为整数,若要在2.7中使用3.0的该特性可以导入 \_\_future\_\_的division

导入unicode\_literals模块可以使用3.0以后的: 字符串统一为unicode，不需要加前缀 u，而以字节存储的str则必须加前缀 b的新特性.

**from** \_\_future\_\_ **import** division, unicode\_literals

## 安装第三方模块

工具: easy\_install 、 pip(官方推荐,内置于2.7.9)

pip使用时在cmd控制台输入 pip install 模块名.py ,在文件中即可import该模块.

pypi.python.org提供Python模块的搜索

## Python的类

Python中使用class定义类,Python 的编程习惯为类名以大写字母开头,后面的括号中的名称表示该类是从哪个类继承.

在创建对象实例时采用 类名() 进行创建, 对每一个实例，都可以直接给他们的属性赋值

类可以使用 \_\_init\_\_ 方法进行初始化(类似于java的构造方法,不写时表示无参构造,写的时候可以加入其他属性),在该方法中可以定义多个属性,不过第一个属性一定需要为self(Python中self类似于java的this)

比如:

**class Person**(object):  
 **def** \_\_init\_\_(self, name):  
 self.name = name  
  
zhangSan = Person("zhangSan")

若想在\_\_init\_\_ 方法中可以添加任意关键字参数,可以使用\*\*kw,在进行初始化时,需要在初始化时传入参数名 = 参数值,解释器会将传入的内容解释为dict,使用setattr方法可方便的对当前对象的该参数进行赋值

比如:

**class Person**(object):  
 **def** \_\_init\_\_(self, name, gender, birth, \*\*kw):  
 self.name = name  
 self.gender = gender  
 self.birth = birth  
 **for** k, v **in** kw.iteritems():  
 setattr(self, k, v) #相当于 self.k = v  
  
xiaoming = Person('Xiao Ming', 'Male', '1990-1-1', job='Student')

### 类的属性

绑定在一个实例上的属性不会影响其他实例,但是,类本身也是一个对象,如果在类上绑定一个属性,则所有实例都可以访问类的属性, 访问类属性不需要创建实例，就可以直接访问;并且,所有实例访问的类属性都是同一个,也就是说,实例属性每个实例各自拥有,互相独立,而类属性有且只有一份.(类的属性类似于java的被static修饰的变量) 类的属性可以动态添加和修改.

若类属性和实例属性重名,实例属性优先级高,它将屏蔽掉对类属性的访问,当没有实例属性将查找类属性.(在\_\_init\_\_中使用self访问将定义实例属性)

### 实例方法

当属性被双下划线修饰时,该属性无法被外部访问,可以在类中进行访问.(类似java的private)

比如:

**class Person**(object):  
 **def** \_\_init\_\_(self, name, score):  
 self.name = name  
 self.\_\_score = score

实例的方法就是在类中定义的函数,它的第一个参数永远是 self,指向调用该方法的实例本身,其他参数和一个普通函数是完全一样的, 调用实例方法必须在实例上调用.

比如:

**class Person**(object):  
 **def** \_\_init\_\_(self, name, score):  
 self.name = name  
 self.\_\_score = score  
  
 **def get\_grade**(self):  
 **if** self.\_\_score >= 80:  
 **return** 'A'  
 **elif** self.\_\_score >= 60:  
 **return** 'B'  
 **else**:  
 **return** 'C'  
  
p1 = Person('Bob', 90)  
  
**print** p1.get\_grade()

在 class 中定义的实例方法其实也是属性,它实际上是一个函数对象,该返回的是一个函数对象,但这个函数是一个绑定到实例的函数,最后利用实例将方法调用.

可以使用types.MethodType()将一个普通方法添加到一个对象和类,作为其实例方法(不常见):

**import** types  
  
**def fn\_get\_grade**(self):  
 **if** self.score >= 80:  
 **return** 'A'  
 **if** self.score >= 60:  
 **return** 'B'  
 **return** 'C'  
  
**class Person**(object):  
 **def** \_\_init\_\_(self, name, score):  
 self.name = name  
 self.score = score  
  
p1 = Person('Bob', 90)  
p1.get\_grade = types.MethodType(fn\_get\_grade, p1, Person)  
**print** p1.get\_grade()

### 类方法

通过标记一个 @classmethod,该方法将绑定到类上,而非类的实例.类方法的第一个参数将传入类本身，通常将参数名命名为 cls,因为是在类上调用,而非实例上调用,因此类方法无法获得任何实例变量，只能获得类的引用.(类似于java的static修饰的方法)

@classmethod  
**def** how\_many(cls):  
 **return** cls.\_\_count

### 类的继承

效果与java中的继承类似,但是在写\_\_init\_\_方法时,方法上的参数还是需要加上父类中的参数,并且必须在方法中使用super对父类的属性进行初始化

**class Person**(object):  
 **def** \_\_init\_\_(self, name, gender):  
 self.name = name  
 self.gender = gender  
  
**class Student**(Person):  
 **def** \_\_init\_\_(self, name, gender, score):  
 super(Student, self).\_\_init\_\_(name, gender)   
 self.score = score

super(Student, self)也可以写为: Person.\_\_init\_\_(self, name, gender)

super(Student, self)将返回当前类继承的父类,然后调用\_\_init\_\_()方法,self参数已在super()中传入,在\_\_init\_\_()中将隐式传递.不需要写出（也不能写）, 多继承的时候这样写无法初始化所有的父类,不能使用super,需要使用 类名.\_\_init\_\_() ,建议都写 类名.\_\_init\_\_()

在类的继承中,子类的实例可以是自己的实例也可以是父类的实例,父类的实例只能是自己的实例不是子类的实例.可以使用isinstance(变量名,类名)判断该变量是否为该类的实例(包括str,list,dict).

类继承后,调用类中的实例方法,会总是先查找它自身的定义,如果没有定义,则顺着继承链向上查找,直到在某个父类中找到为止.

当一个方法是接收一个对象,在方法中会去调用该对象的一个方法时,会自动去找到这个对象的该方法进行调用.

比如: json.load()需要传入一个File对象,会调用其read()方法,但是若传入的不是File对象,也回去找到该对象的read()方法去运行,当传入的对象不包含该方法时会报错

**import** json  
  
**class Students**(object):  
 **def read**(self):  
 **return** r'["Tim", "Bob", "Alice"]'

s = Students()  
**print** json.load(s)

### 多重继承

Python允许从多个父类继承. 多重继承时要分别初始化两个父类的\_\_init\_\_()方法,这样就无法使用super,需要使用类名.\_\_init\_\_()

比如: Python的网络服务器有TCPServer、UDPServer、UnixStreamServer、UnixDatagramServer,而服务器运行模式有多进程ForkingMixin 和多线程ThreadingMixin两种.

要创建多进程模式的 TCPServer： class MyTCPServer(TCPServer, ForkingMixin):

要创建多线程模式的 UDPServer： class MyUDPServer(UDPServer, ThreadingMixin):

比如:

**class A**(object):  
 **def** \_\_init\_\_(self, name):  
 self.name = name  
  
**class B**(A):  
 **def** \_\_init\_\_(self, name, age):  
 A.\_\_init\_\_(self, name)  
 self.age = age  
  
**class C**(A):  
 **def** \_\_init\_\_(self, name, sex):  
 A.\_\_init\_\_(self, name)  
 self.sex = sex  
  
**class D**(B, C):  
 **def** \_\_init\_\_(self, name, age, sex, country):  
 B.\_\_init\_\_(self, name, age)  
 C.\_\_init\_\_(self, name, sex)  
 self.country = country  
  
d = D('1', '2', '3', '4')

## 获取对象信息的一些方法

type(变量名) 获取变量类型

dir(变量名) 获取变量的所有属性

getattr(变量名,属性名称) 获取变量属性值,会报错,可以传入第三个参数,表示未获取到时默认为多少,不会报错

setattr(变量名,属性名称,属性值) 设置变量属性值

## 特殊方法

特殊方法一般命名方式为: \_\_方法名\_\_() , 特殊方法定义在class中,不需要直接调用,某些函数或操作符会调用对应的特殊方法.

比如当进行打印变量时, list会打印list中的内容,打印类实例会打印地址值, 这是因为打印时调用了\_\_str\_\_()(类似于java的toString()),还有用于len的\_\_len\_\_等.

比较函数的\_\_cmp\_\_ : self 应该排在前面，就返回 -1，如果 s 应该排在前面，就返回1，如果两者相当，返回 0.

对象进行相加时是调用了对象的\_\_add\_\_()方法,减法：\_\_sub\_\_ 乘法：\_\_mul\_\_ 除法：\_\_div\_\_

int(对象名)函数会调用对象的类中的\_\_int\_\_()方法, float(对象名)调用 \_\_float\_\_()

### 对set方法进行校验

若将对象的属性用\_\_修饰声明为私有的,在类中提供get和set方法,可以在set方法中进行校验,不过这样在书写代码时需要使用变量名调用方法,Python提供了@property和 @被property修饰的方法名称.setter 修饰符来对get和set并校验进行简便处理.

被@property修饰的方法1可以直接用 变量名.方法1 获取在其中返回的值,

被@方法1.setter修饰的方法2在赋值时用 变量名.方法2 = 值 进行赋值

方法1和方法2的名称可以相同也可以不同,相同时解释器能够找到正确的方法.

比如:

**class Student**(object):  
 **def** \_\_init\_\_(self, name, score):  
 self.name = name  
 self.\_\_score = score  
  
 @property  
 **def score**(self): # 方法1  
 **return** self.\_\_score  
  
 @score.setter # setter前必须为 方法1. 表示对方法1中的返回值对应的属性进行赋值  
 **def score**(self, score): # 方法2  
 **if** score < 0 **or** score > 100:  
 **raise** ValueError('invalid score') # 这里是抛出异常  
 self.\_\_score = score  
  
 @property  
 **def grade**(self):  
 **if** self.\_\_score >= 80:  
 **return** 'A'  
 **elif** self.\_\_score < 60:  
 **return** 'C'  
 **else**:  
 **return** 'B'  
  
s = Student('Bob', 59)  
s.score = 60 # 使用 变量名.方法2 = 进行赋值  
**print** s.score # 使用 变量名.方法1 直接获取  
**print** s.grade # 这里也是直接获取

### 限制类拥有的属性

Python是动态语言,任何实例在运行期都可以动态地添加属性,Python提供了\_\_slots\_\_可以限制类拥有的属性,继续添加非\_\_slots\_\_中不包含的属性会抛出异常.

比如:

**class Student**(Person):  
 \_\_slots\_\_ = ('name', 'gender', 'score')  
  
 **def** \_\_init\_\_(self, name, gender, score):  
 Person.\_\_init\_\_(self, name, gender)  
 self.score = score

### 将类变为可调用对象

函数是一个对象,称为可调用对象,所有的函数都是可调用对象.

一个类实例也可以变成一个可调用对象，只需要实现一个特殊方法\_\_call\_\_()

比如: 直接使用类实例进行调用即会调用类中的特殊方法\_\_call\_\_()

**class Person**(object):  
 **def** \_\_init\_\_(self, name, gender):  
 self.name = name  
 self.gender = gender  
  
 **def** \_\_call\_\_(self, friend):  
 **print** 'My name is %s' % self.name  
 **print** 'My friend is %s' % friend  
  
p = Person('123', 123)  
p('1234')

## 异常捕获

### 传统的try语句

**try**:  
 ...

**except**异常类型1, e:

...  
**except**异常类型2, e2:

...

**except**异常类型3, e3:

...

**else** :

...

**finally**:

....

如果try中未出现异常,运行完运行else中的语句,然后执行finally中的语句,如果出现异常,运行except中的语句,然后在解释器处理异常前运行finally中的语句.

其中 except语句中的异常类型和异常变量名以及else语句和finally语句都是可选的.

### with语句

**with** 返回一个对象 **as** 变量名:

执行语句

将with后执行的语句返回的对象赋值给变量名,然后即可在执行语句中使用该变量.一般使用在文件操作和进程之间互斥对象等情况中.

with语句是一个自动进行上下文管理的语句,若对象要支持with语句,需要根据上下文管理协议包含\_\_enter\_\_()和\_\_exit\_\_()这2个特殊方法,进入时执行\_\_enter\_\_方法,将返回值赋值给as后面的变量,在出现异常退出时执行\_\_exit\_\_.

比如:

**with** open("1.txt") **as** a:  
 **for** line **in** a.readlines():  
 **print** line

### raise语句

类似于java的throw语句: **raise** 异常类型, "提示语句"

### assert语句

断言语句,检测表达式是否为真,若为假时会抛出AssertionError,提示语句为断言语句中的提示语句

**assert** 返回一个boolean : "提示语句"

可以用于当对函数参数类型有要求对参数类型进行校验或比较重要的判断条件