Python培训班讲义

孔子曰：三人行，则必有我师。是故弟子不必不如师，师不必贤于弟子。闻道有先后，术业有专攻，如是而已。

一、 python基础

# python基础

# 包package和模块module

包就是含有文件：\_\_init\_\_.py的目录，\_\_init\_\_.py本身也是模块，可以是空文件，也可以有代码，当第一次import包或包的其他任何部分时，\_\_init\_\_.py会被执行，通常是一些初始化包的代码

模块就是\*.py、\*.pyw、\*.pyc文件。

**模块的搜索路径：**

当一个模块被导入时，顺序为：内置函数--->sys.path(优先搜索当前路径)，在初始化后，python程序可以修改sys.path,路径放到前面的优先被加载。

**导入模块：**

import XXX 或 from XXX import XXX

**module的一些内置属性：**

**\_\_name\_\_:** 所有的modules都有一个内置的属性\_\_name\_\_，模块的\_\_name\_\_属性的值取决于如何使用这个模块，如果import module，那么\_\_name\_\_属性的值是模块的名字。如果直接执行这个模块的话，那么\_\_name\_\_属性的值就是默认值\_\_main\_\_。

**\_\_file\_\_ :** 当前module的绝对路径

**\_\_doc\_\_ ：**你试一下就知道了。

模块、函数、类的第一条语句会使用文档字符串。

# Pip的使用

Python包安装管理工具，类似于ubuntu的apt-get。在windows平台上使用时请确保用户目录无中文，否则会出错，在linux上需要root权限。

**安装：**

pip install 包名或安装文件名

**卸载：**

pip uninstall 包名

**升级：**

pip install --upgrade 包名

**搜索：**

pip search 包名

**其他请查帮助：**

pip --help

或

https://pypi.python.org

# 数据类型

# 不可变数据类型

1. 整型 int
2. 浮点型 float
3. 字符型 str

单引号、双引号、三个单引号、三个双引号。

\为转义符

split、strip、lower、upper、endswith

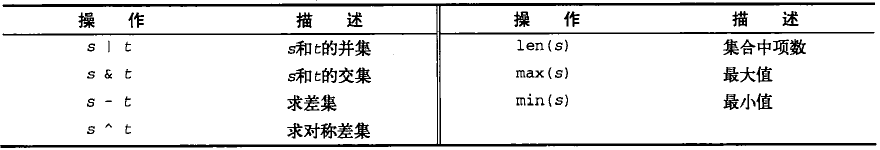
1. 布尔型 bool
2. 字节型
3. 元组 tuple

# 可变数据类型

1. 列表 list

切片、in、del、len、sum、max、min、列表函数

1. 集合 set



1. 字典 dict



Key:任意不可变对象

# 特殊值 None

Python的None可以理解为NULL（空），意思为未知值。

# 字符集问题

字符集问题对中国人来说主要针对的是汉字问题，常见的字符集有：

GBK、GB2312：windows平台上常见字符集，每个汉字2字节，英文1字节。

Unicode：是国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。一个汉字2字节。

UTF-8：Unicode字符集转换格式。一个汉字3字节。

文件缺省字符集受模块第一或第二行：# -\*- coding:utf-8 -\*- 影响。

字符集转换函数：

encode和decode

例如：s=**"**雨辰是个好同志**"**，s数据类型：str转换为：gbk

x = s.encode(**"**gbk**"**)

x数据类型：bytes

x.decode(**"**gbk**"**)后字符集为unicode

# 变量

变量名命名规则：以下划线或英文字母开头，可以包含下划线、英文字母及数字。

以\_\_开头和结尾的变量一般为系统变量。

**复制、浅复制、深复制**

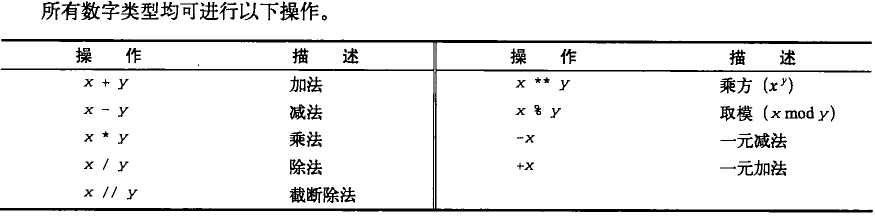
需要import copy，可以参照C语言的指针来帮助理解。

浅拷贝：只拷贝父对象，不会拷贝对象的内部的子对象。

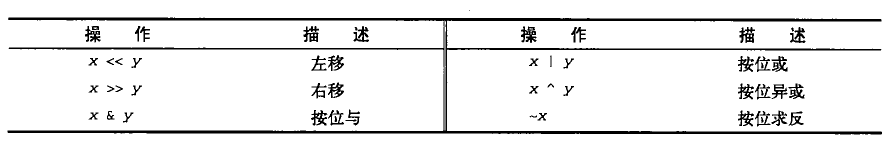
深拷贝：拷贝对象及其子对象。

# 运算符与表达式

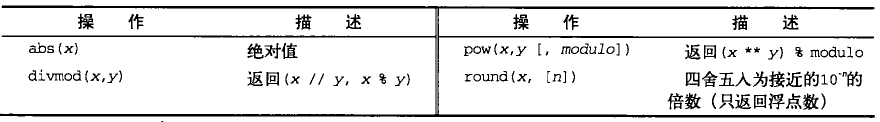
# 数字操作



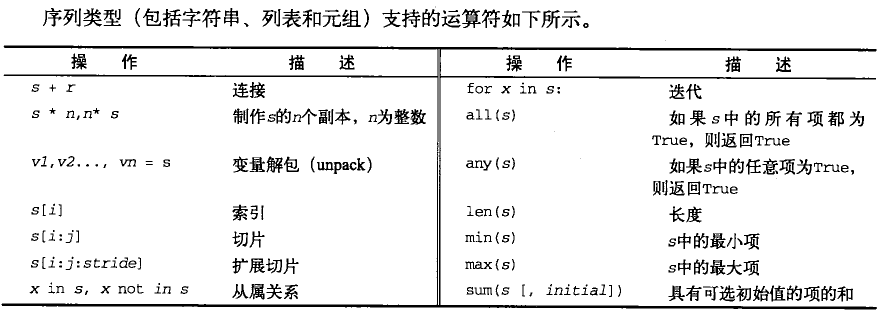
整数位操作：

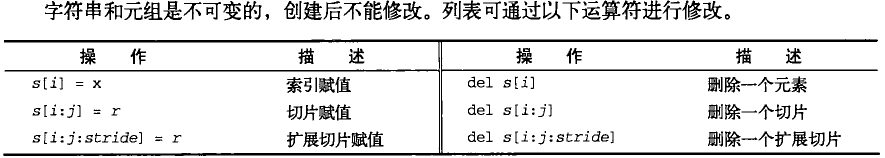


内置函数：



# 序列操作





还有：

append

insert

pop

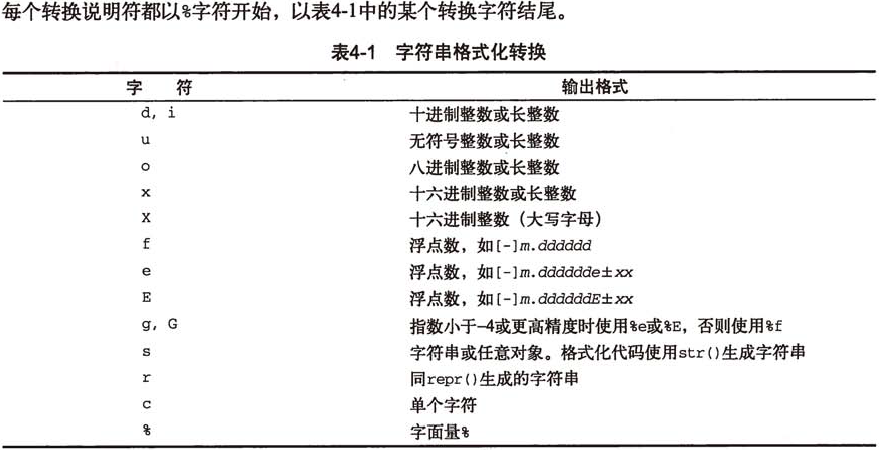
sort

扩展sort：

a=[[1,4,2],[2,6,1],[1,2,3]]

a.sort(key=lambda x:x[1])

# 字符串格式化



高级字符串格式化：

s.format(\*args,\*kwargs)

s的形式如下：

{0}，{1}，{2} ……针对元组

{key}……针对字典

还可以：

{0[index]},{0[key]}

复数：

{0.real},{0.imag}

还可以对格式进行更加精确的控制，可以用冒号（：）给每个占位符添加可选的格式说明：

{place:format\_spec}

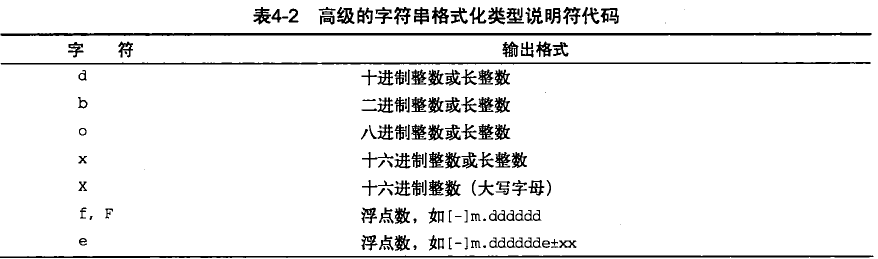
format\_spec： [fill[align]][sign][0|width][.prescision][type]

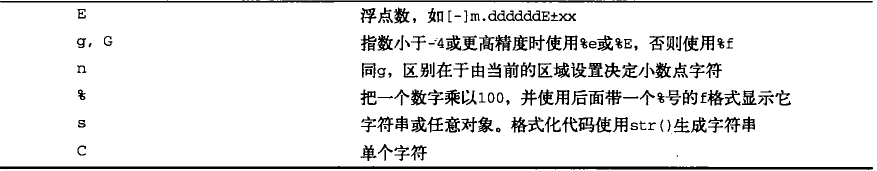
fill：填充空白的字符

align：左对齐 <、右对齐 >、居中 ^

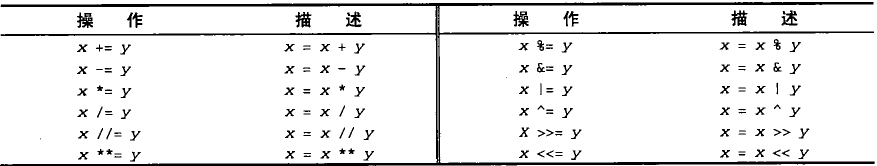
sign：符号，+、-、空格，+正负前都加符号，-负数前加符号，空格 正数前加空格。缺省为：-

type：





# 增量操作



# 测试运算

成员测试：

in, not in

同一性测试：

is, not is

例如：

a is None、a is not None

a is b、a is not b

扩展：

isinstance(object, class\_or\_type)

callable(object)

hasattr(object, name)

# 关系表达式

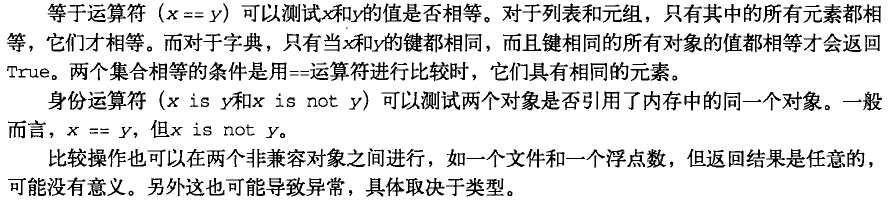
<、<=、>、>=、!=、==

# 逻辑表达式

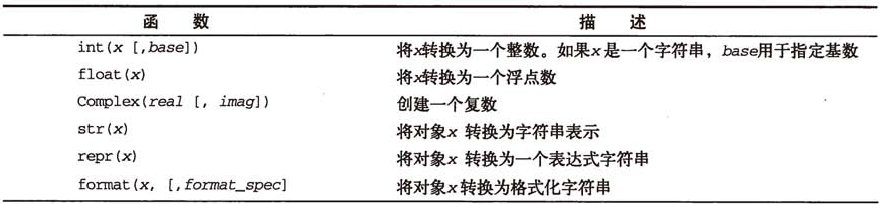
and、or、not

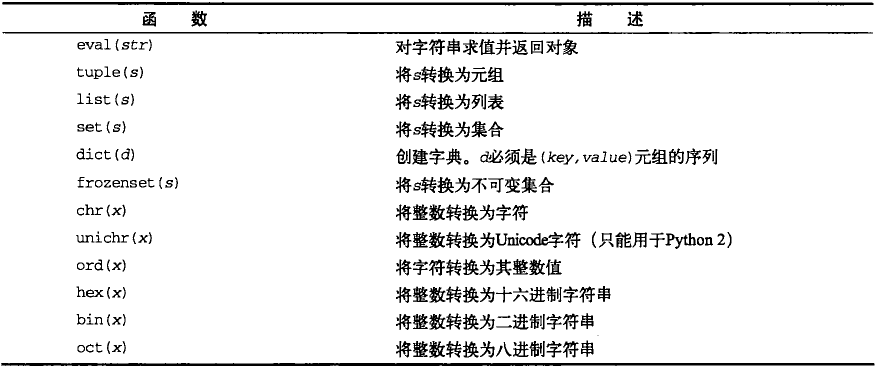
python 中的为False：0、False、[]、{}、()、''、""、None、set()。空列表、空字典、空元组、空集合都是False.

# 对象的比较



# 转换函数





# 日期和日期时间

import datetime

datetime.date.today()

datetime.datetime.now()

datetime.timedelta([days[,seconds[,microseconds]]])

**日期和日期时间的字符串格式化：**

例子：

import datetime

now=datetime.datetime.now()

now.strftime("%Y-%m-%d %H:%M:%S")

**strftime参数：**

python中时间日期格式化符号：

%y 两位数的年份表示（00-99）

%Y 四位数的年份表示（000-9999）

%m 月份（01-12）

%d 月内中的一天（0-31）

%H 24小时制小时数（0-23）

%I 12小时制小时数（01-12）

%M 分钟数（00=59）

%S 秒（00-59）

%a 本地简化星期名称

%A 本地完整星期名称

%b 本地简化的月份名称

%B 本地完整的月份名称

%c 本地相应的日期表示和时间表示

%j 年内的一天（001-366）

%p 本地A.M.或P.M.的等价符

%U 一年中的星期数（00-53）星期天为星期的开始

%w 星期（0-6），星期一为星期的开始

%W 一年中的星期数（00-53）星期一为星期的开始

%x 本地相应的日期表示

%X 本地相应的时间表示

%Z 当前时区的名称

%% %号本身

**将字符串形式的日期时间转换为日期时间：**

例子：

datetime.datetime.strptime(“2018-1-25 01:30:30”, "%Y-%m-%d %H:%M:%S")

# time

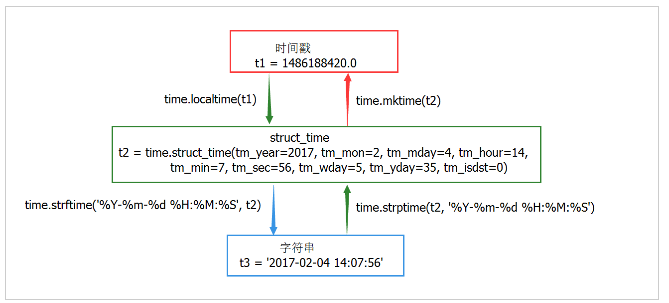
import time

常用函数：

time.clock()：返回当前进程所消耗的处理器运行时间秒数（不包括sleep时间），值为小数；该方法Python3.3改成了time.process\_time()

time.time()：返回时间戳（自格林威治时间1970-1-1 0:00:00 至今所经历的秒数）

time.sleep(n)：休眠n秒，支持小数。



# 选择结构

**单分支：**

if 条件:

语句

**双分支：**

if 条件:

语句

else:

语句

**多分支：**

if 条件:

语句

elif 条件:

语句

elif 条件:

语句

else:

语句

# 循环结构

# 迭代 for

**对于可迭代对象：**

for v in iterable:

语句

for i,v in enumerate(iterable[, start]):

语句

**对于字典：**

for key in dictobject:

语句

for key in dictobject.keys():

语句

for k,v in dictobject.items():

语句

# While

while 条件:

语句

do while 条件{

}

do {

}while 条件

# break和continue

和其他语言中的意义是一致的，break退出循环，continue立即开始新的循环。后续代码均不执行。

# 函数

# 内置函数

部分内置函数：

id()、type()、dir()、help()

len()：字符时python2是计算字节个数，python3是计算字符个数。

filter(function,sequence)：

例如：

a=[1,4,2,10,5,8,9]

filter(lambda x:True if x>=5 else False,a)

map(function,sequence)：函数是批量的对对象进行相同的操作

例如：

b=[2,1,3,4]

map(lambda x:2\*x,b)

zip()：

例如：

x1=(1,2,3,4)

x2=(1,3,5,4)

x3=(1,3,5,6)

zip(x1,x2,x3)

# compile()、eval()和 exec()

compile()：将字符串编译字节代码。

eval()：执行一段代码，有返回值。

exec()：执行一段代码，无返回值。

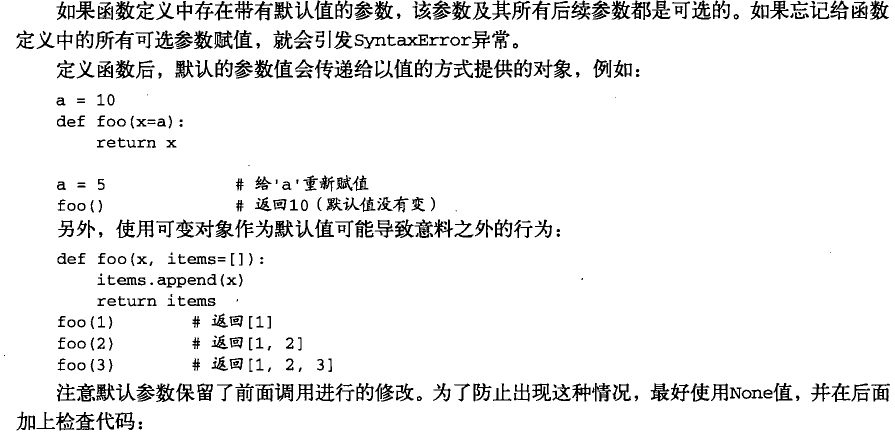
# 自定义函数

使用def定义函数。例如：

def add(x,y):

return(x+y)

如无return则返回None





Python支持嵌套的函数定义：

def **jc**(n):

def **\_jc**(x):

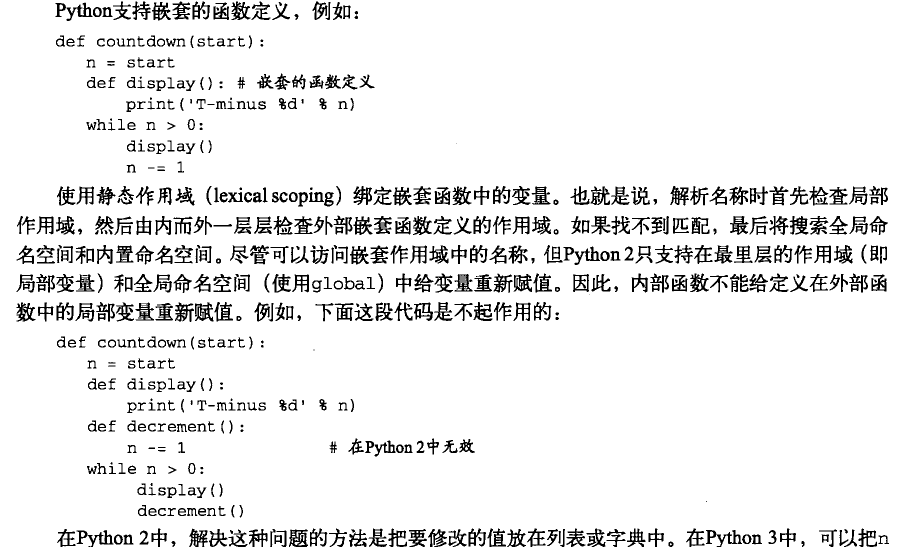
if x <= 0:

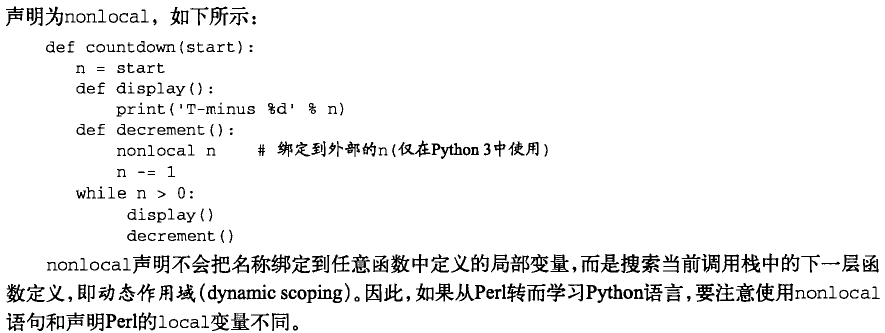
return 1

return x \* \_jc(x - 1)

return \_jc(n)

再看一下书上说的：





# \*args和\*\*kwargs

\*：元组形式的参数，参数个数任意。

\*\*：字典形式的参数，参数个数任意。

请测试如下代码加深理解：

def **fun1**(\*args, \*\*kwargs):

print *"fun1:"*, args

print *"fun1:"*, kwargs

fun2(args, kwargs)

fun2(\*args, \*\*kwargs)

def **fun2**(\*args, \*\*kwargs):

print *"fun2:"*, args

print *"fun2:"*, kwargs

**fun1(1, "abc", 2, name='yuchen', age=18)**

# 生成器

例如：

def **fbnq**():

a = 0

b = 1

while True:

yield b

a, b = b, a + b

if b > 1000:

break

**f = fnbq()**

**print(f.\_\_next\_\_())**

def **myRange**(start, end=None, step=1):

if end is None:

end = start

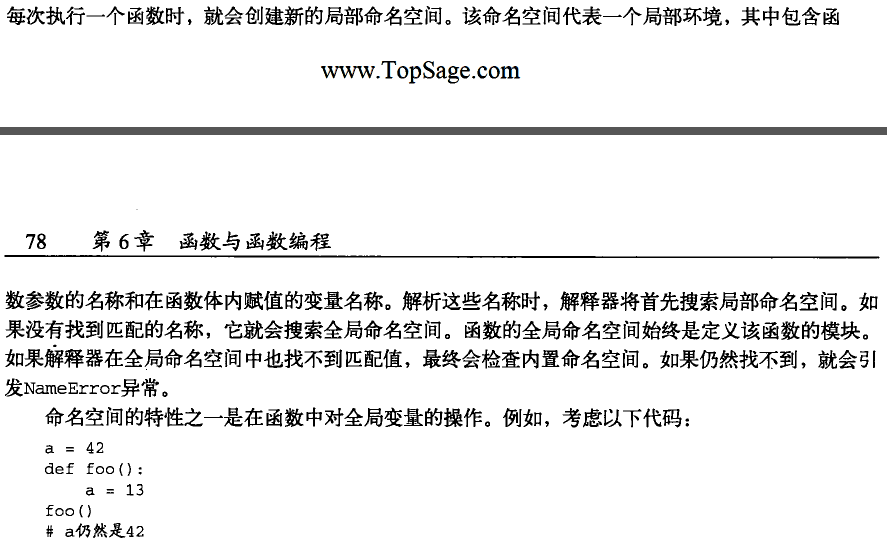
start = 0

while (start < end) if step > 0 else (start > end):

yield start

start += step

# 作用域



可以用关键字global指定变量为全局变量。

python中的命名空间就是一个dict，key是变量的名字，value是变量的值。

可以通过函数：locals()查看局部命名空间，globals()查看全局命名空间。

# 类

# 类的基本实现

* 类的命名
* 遵守变量命名的规范；
* 建议大驼峰（由一个或者多个单词构成，每个单词首字母大写，单词跟单词直接相连）；
* 尽量避开跟系统命名相似的命名。
* 你如何声明一个类
* 必须用class关键字；
* 类由属性和方法构成；
* 成员属性定义可以直接使用变量赋值，如果没有值，可以使用None。
* 实例化类

对象名 = 类名(参数) #实例化了一个对象

* 访问对象成员
* 使用点操作符
  + obj.成员属性名称
  + obj.成员方法()
* 可以通过默认内置变量检查类和对象的所有成员

对象所有成员检查

* # dict前后各有两个下划线
* obj.\_\_dict\_\_

类所有的成员

* # dict前后各有两个下划线

class\_name.\_\_dict\_\_

# 属性

class A():

name = "dana"

age = 18

# 注意say的写法，参数有一个self

def say(self):

self.name = "aaaa"

self.age = 200

类的属性和对象的属性

用id()函数查看一下是不是属于同一个对象。

如何实现共享变量。

public公共变量和private私有变量（双下划线开头），能否实现protected变量？

private的本质是：\_类名\_\_成员。可通过：\_\_dict\_\_查看

# 方法

方法的声明：def 方法名(self)

私有方法（双下划线开头）

静态方法：加@staticmethod修饰符

类方法：加@classmethod修饰符

# 类和对象的成员

* 类和对象都可以存储成员，成员可以归类所有，也可以归对象所有。
* 类存储成员时使用的是与类关联的一个对象。
* 对象存储成员是是存储在当前对象中。
* 对象访问一个成员时，如果对象中没有该成员，尝试访问类中的同名成员。
* 如果对象中有此成员，一定使用对象中的成员
* 创建对象的时候，类中的成员不会放入对象当中，而是得到一个空对象，没有成员
* 通过对象对类中成员重新赋值或者通过对象添加成员时，对应成员会保存在对象中，而不会修改类成员

给对象（实例）绑定一个属性

给对象（实例）绑定一个方法：

**def** **set\_age**(self, age): *# 定义一个函数作为实例方法*

... self.age = age

...

**from** types **import** MethodType

s.set\_age = MethodType(set\_age, s) *# 给实例绑定一个方法*

s.set\_age(25) *# 调用实例方法*

s.age

为了给所有实例都绑定方法，可以给class绑定方法：

**def** **set\_score**(self, score):

... self.score = score

...

Student.set\_score = set\_score

使用type定义类：

type(name, bases, dict)

bases：元组

### 使用\_\_slots\_\_

但是，如果我们想要限制实例的属性怎么办？比如，只允许对Student实例添加name和age属性。

为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的\_\_slots\_\_变量，来限制该class实例能添加的属性：

**class Student(object):**

\_\_slots\_\_ = ('name', 'age') *# 用tuple定义允许绑定的属性名称*

使用\_\_slots\_\_要注意，\_\_slots\_\_定义的属性仅对当前类实例起作用，对继承的子类是不起作用的。除非在子类中也定义\_\_slots\_\_，这样，子类实例允许定义的属性就是自身的\_\_slots\_\_加上父类的\_\_slots\_\_。

# self

self只是一个约定的变量名。

c=MyClass()

c.foo(10)

实际上执行的操作是：MyClass.foo(c,10)

方法中有self形参的方法成为非绑定类的方法，可以通过对象访问， 没有self的是绑定类的方法，只能通过类访问。

使用类访问绑定类的方法时， 如果类方法中需要访问当前类的成员，可以通过 \_\_class\_\_.成员名 来访问。也可以通过 类名.成员名 访问。

# 封装

* 封装就是对对象的成员进行访问限制
* 封装的三个级别：
  + 公开，public
  + 受保护的，protected
  + 私有的，private
  + public，private，protected不是关键字
* 判别对象的位置
  + 对象内部
  + 对象外部
  + 子类中
* 私有
  + 私有成员是最高级别的封装，只能在当前类或对象中访问
  + 在成员前面添加两个两个下划线即可

class Person():

# name是共有的成员

name = "liuying"

# \_\_age就是私有成员

\_\_age = 18

* + Python的私有不是真私有，是一种成为name mangling的改名策略，可以使用对象.\_classname\_\_attributename访问
* 受保护的封装 protected
  + 受保护的封装是将对象成员进行一定级别的封装，然后，在类中或者子类中都可以进行访问，但是在外部不可以
  + 封装方法： 在成员名称前加一个下划线即可（只是一个约定）
* 公开的，公共的 public
  + 公共的封装实际对成员没有任何操作，任何地方都可以访问

# 继承及多重继承

* 继承与被继承的概念：
  + 被继承的类叫父类，也叫基类，也叫超类
  + 用于继承的类，叫子类，也叫派生类
  + 继承与被继承一定存在一个 is-a 关系

语法：

class 类名(父类s):

任何类都有一个共同的父类：object

继承的特征

* 所有的类都继承自object类，即所有的类都是object类的子类
* 子类一旦继承父类，则可以使用父类中除私有成员外的所有成员
* 子类继承父类后并没有将父类成员完全赋值到子类中，而是通过引用关系访问调用
* 子类中可以定义独有的成员属性和方法
* 子类中定义的成员和父类成员如果相同，则优先使用子类成员
* 子类如果想扩充父类的方法，可以在定义新方法的同时访问父类成员来进行代码重用，可以使用 [父类名.父类成员] 的格式来调用父类成员，也可以使用super().父类成员的格式来调用。

变量、函数的查找顺序问题

* 优先查找自己的变量
* 没有则查找父类
* 构造函数如果本类中没有定义，则自动查找调用父类构造函数
* 如果本类有定义，则不在继续向上查找

构造函数

* 是一类特殊的函数，在类进行实例化之前进行调用
* 如果定义了构造函数，则实例化时使用构造函数，不查找父类构造函数
* 如果没定义，则自动查找父类构造函数
* 如果子类没定义，父类的构造函数带参数，则构造对象时的参数应该按父类参数构造

super

* super不是关键字， 而是一个类
* super的作用是获取MRO（Method Resolustion Order）列表中的第一个类
* super于父类直接没任何实质性关系，但通过super可以调用到父类
* super使用两个方法,super()和super(类,对象)

多重继承及MRO

* MRO就是多继承中，用于保存继承顺序的一个列表
* MRO列表的计算原则：
  + 子类永远在父类前面
  + 如果多个父类，则根据继承语法中括号内类的书写顺序存放
  + 如果多个类继承了同一个父类，孙子类中只会选取继承语法括号中第一个父类的父类
* 多重继承时构造函数问题

# 多态

* 多态就是同一个对象在不同情况下有不同的状态出现
* 多态不是语法，是一种设计思想
* 多态性： 一种调用方式，不同的执行效果
* 多态： 同一事物的多种形态
* Mixin设计模式（有兴趣请自学）

# 类相关函数

* issubclass:检测一个类是否是另一个类的子类
* isinstance:检测一个对象是否是一个类的实例
* hasattr:检测一个对象是否有成员XXX
* getattr: get attribute
* setattr: set attribute
* delattr: delete attribute
* dir: 获取对象的成员列表

# 类的属性修饰符

* 使用property函数：

属性=property(fget,fset,fdel,doc)

* 使用@property修饰符

class C(object):

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

self.\_x = None

@property

def x(self):

"""I'm the 'x' property."""

return self.\_x

@x.setter

def x(self, value):

self.\_x = value

@x.deleter

def x(self):

del self.\_x

# 类的内置属性

\_\_dict\_\_:以字典的方式显示类的成员组成

\_\_doc\_\_: 获取类的文档信息

\_\_name\_\_:获取类的名称，如果在模块中使用，获取模块的名称

\_\_bases\_\_: 获取某个类的所有父类，以元组的方式显示

# 类的常用魔术方法

魔术方法就是不需要人为调用的方法，基本是在特定的时刻自动触发

魔术方法的统一的特征，方法名被前后各两个下滑线包裹

操作类：

\_\_init\_\_: 构造函数

\_\_del\_\_: 析构函数

\_\_new\_\_: 对象实例化方法，此函数较特殊，一般不需要使用

\_\_call\_\_: 对象当函数使用的时候触发

\_\_str\_\_: 当对象被当做字符串使用的时候调用

\_\_repr\_\_: 返回自我描述字符串

描述符相关：

\_\_set\_\_

\_\_get\_\_

\_\_delete\_\_

例子：

class RevealAccess(object):

"""A data descriptor that sets and returns values

normally and prints a message logging their access.

"""

def \_\_init\_\_(self, initval=None, name='var'):

self.val = initval

self.name = name

def \_\_get\_\_(self, obj, objtype):

print 'Retrieving', self.name

return self.val

def \_\_set\_\_(self, obj, val):

print 'Updating', self.name

self.val = val

>>> class MyClass(object):

... x = RevealAccess(10, 'var "x"')

... y = 5

...

>>> m = MyClass()

>>> m.x

Retrieving var "x"

>>> m.x = 20

Updating var "x"

>>> m.x

Retrieving var "x"

>>> m.y

属性操作相关：

\_\_getattr\_\_: 访问一个不存在的属性时触发

\_\_setattr\_\_: 对成员属性进行设置的时候触发

参数：

self用来获取当前对象

被设置的属性名称，以字符串形式出现

需要对属性名称设置的值

作用：进行属性设置的时候进行验证或者修改

注意： 在该方法中不能对属性直接进行赋值操作，否则死循环

参看案例

class Person():

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

pass

def \_\_setattr\_\_(self, name, value):

print("设置属性： {0}".format(name))

# 下面语句会导致问题，死循环

#self.name = value

# 此种情况，为了避免死循环，规定统一调用父类魔法函数

super().\_\_setattr\_\_(name, value)

p = Person()

print(p.\_\_dict\_\_)

p.age = 18

运算分类相关魔术方法：

\_\_gt\_\_: 进行大于判断的时候触发的函数

参数：

self

第二个参数是第二个对象

返回值可以是任意值，推荐返回布尔值

\_\_ge\_\_

\_\_lt\_\_

\_\_le\_\_

\_\_eq\_\_

\_\_ne\_\_

# 类和对象的三种方法

* 实例方法

需要实例化对象才能使用的方法，使用过程中可能需要截止对象的其他对象的方法完成

* 静态方法

不需要实例化，通过类直接访问

* 类方法

不需要实例化

# 使用枚举类

# 使用元类

# 装饰器

# 入门

函数的返回值为函数：

日志打印器

时间计数器

改造函数：比如将返回值修改为绝对值

# 带参数的函数装饰器

例子：

def say\_hello(contry):

def wrapper(func):

def deco(\*args, \*\*kwargs):

if contry == "china":

print("你好!")

elif contry == "america":

print('hello.')

else:

return

# 真正执行函数的地方

func(\*args, \*\*kwargs)

return deco

return wrapper

# 小明，中国人

@say\_hello("china")

def xiaoming():

pass

# jack，美国人

@say\_hello("america")

def jack():

pass

xiaoming()

print("------------")

jack()

# 不带参数的类装饰器

基于类装饰器的实现，必须实现 \_\_**call**\_\_ 和 \_\_init\_\_两个内置函数。  
\_\_**init**\_\_：接收被装饰函数  
\_\_**call**\_\_：实现装饰逻辑。

例子：

class logger(object):

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

self.func = func

def \_\_call\_\_(self, \*args, \*\*kwargs):

print("[INFO]: the function {func}() is running..."

.format(func=self.func.\_\_name\_\_))

return self.func(\*args, \*\*kwargs)

@logger

def say(something):

print("say {}!".format(something))

say("hello")

# 带参数的类装饰器

**\_\_init\_\_**：不再接收被装饰函数，而是接收传入参数。  
**\_\_call\_\_**：接收被装饰函数，实现装饰逻辑。

例子：

class logger(object):

def \_\_init\_\_(self, level='INFO'):

self.level = level

def \_\_call\_\_(self, func): # 接受函数

def wrapper(\*args, \*\*kwargs):

print("[{level}]: the function {func}() is running..."

.format(level=self.level, func=func.\_\_name\_\_))

func(\*args, \*\*kwargs)

return wrapper #返回函数

@logger(level='WARNING')

def say(something):

print("say {}!".format(something))

say("hello")

# 使用偏函数与类实现装饰器

什么是偏函数：

偏函数是functools模块中的一个方法,即functools.partial，偏函数就是为了固定某些参数的传入,作为默认参数传入,下面我们以int为例子来解释用法

from functools import partial

bin2dec = partial( int, base=2 )

print bin2dec( '0b10001' ) # 17

print bin2dec( '10001' ) # 17

hex2dec = partial( int, base=16 )

print hex2dec( '0x67' ) # 103

print hex2dec( '67' ) # 103

例子：

import time

import functools

class DelayFunc:

def \_\_init\_\_(self, duration, func):

self.duration = duration

self.func = func

def \_\_call\_\_(self, \*args, \*\*kwargs):

print('Wait for {self.duration} seconds...')

time.sleep(self.duration)

return self.func(\*args, \*\*kwargs)

def eager\_call(self, \*args, \*\*kwargs):

print('Call without delay')

return self.func(\*args, \*\*kwargs)

def delay(duration):

return functools.partial(DelayFunc, duration)

@delay(duration=2)

def add(a, b):

return a+b

# 创建装饰器时保留函数元信息

装饰器作用在某个函数上，但是这个函数的重要的元信息比如名字、文档字符串、注解和参数签名都丢失了。

解决方案：任何时候你定义装饰器的时候，都应该使用 functools 库中的 @wraps 装饰器来注解底层包装函数。例如：

**import** **time**

**from** **functools** **import** wraps

**def** timethis(func):

*'''*

*Decorator that reports the execution time.*

*'''*

@wraps(func)

**def** wrapper(\*args, \*\*kwargs):

start = time.time()

result = func(\*args, \*\*kwargs)

end = time.time()

**print**(func.\_\_name\_\_, end-start)

**return** result

**return** wrapper

下面我们使用这个被包装后的函数并检查它的元信息：

**>>>** @timethis

**... def** countdown(n):

**...**  *'''*

**...**  *Counts down*

**...**  *'''*

**...**  **while** n > 0:

**...**  n -= 1

**...**

**>>>** countdown(100000)

countdown 0.008917808532714844

**>>>** countdown.\_\_name\_\_

'countdown'

**>>>** countdown.\_\_doc\_\_

'\n\tCounts down\n\t'

**>>>** countdown.\_\_annotations\_\_

{'n': <class 'int'>}

>>>

# 异常

https://www.runoob.com/python3/python3-errors-execptions.html

# 文件操作

# 普通文件操作

<https://www.runoob.com/python3/python3-file-methods.html>

try:

f = open('/path/to/file', 'r')

print(f.read())

finally:

if f:

f.close()

with open('/path/to/file', 'r') as f:

print(f.read())

使用with：

with 语句实质是上下文管理。

1、上下文管理协议。包含方法\_\_enter\_\_() 和 \_\_exit\_\_()，支持该协议对象要实现这两个方法。

2、上下文管理器，定义执行with语句时要建立的运行时上下文，负责执行with语句块上下文中的进入与退出操作。

3、进入上下文的时候执行\_\_enter\_\_方法，如果设置as var语句，var变量接受\_\_enter\_\_()方法返回值。

4、如果运行时发生了异常，就退出上下文管理器。调用管理器\_\_exit\_\_方法。

例子：

class Mycontex(object):

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

self.name=name

def \_\_enter\_\_(self):

print("进入enter")

return self

def do\_self(self):

print(self.name)

def \_\_exit\_\_(self,exc\_type,exc\_value,traceback):

print("退出exit")

print(exc\_type,exc\_value)

if \_\_name\_\_ == '\_\_main\_\_':

with Mycontex('test') as mc:

mc.do\_self()

可以尝试出现异常，看输出什么？

# INI（conf）文件操作

标准库：configparser

# 创建管理对象

conf = configparser.ConfigParser()

# 读ini文件

conf.read(文件名, encoding="utf-8")

# json

https://www.runoob.com/python3/python3-json.html

# re

https://www.runoob.com/python/python-reg-expressions.html

# struct

https://blog.csdn.net/qq\_30638831/article/details/80421019

二、wxpython基础

三、python多线程

四、python数据库访问

关系型（sql）

IBM（DB2、informix）、oracle甲骨文、sybase(赛贝斯powerbuilder、powerdesigner)

MS SQL Server、MySQL、postgres、sqllite

wampserver

select、

select 字段名列表（函数）

from 表的列表

where 条件

order by 字段名

group by ……

having ……

select xh,xm,xsb.bjbm,bjmc

from xsb,bjb

where xsb.bjbm=bjb.bjbm

order by xb desc,xh asc

select count(\*) from xsb where bjbm=”0101001”

select bjmc,xb,1.0\*count(\*)/(select count(\*) from xsb where bjbm=bjb.bjbm)

from xsb,bjb

where xsb.bjbm=bjb.bjbm

order by bjb.bjbm asc

group by xsb.bjbm,bjmc,xb

having 1.0\*count(\*)/(select count(\*) from xsb where bjbm=bjb.bjbm)>0.5

insert、

insert into 表名(字段名列表) values(值列表)

insert into bjb(bjbm,nj,bjmc) values(“001”,2018,”1班”)

update、

update 表名 set 字段名1=值1, 字段名2=值2……

where 条件

delete、

delete from 表名 where 条件

五、python网络编程

1、TCP编程

1.1、TCP server

#!/usr/bin/env python3

# -\*- coding:utf-8 -\*-

u'''

Created on 2018年11月13日

@author: yuchen

'''

import socket

import threading

\_\_author\_\_ = 'fatyuchen <fatyuchen@qq.com>'

\_\_version\_\_ = '1.0.0'

\_\_company\_\_ = u'重庆交大'

\_\_updated\_\_ = '2018-11-27 08:55'

def recv(skt, addr):

data = skt.recv(8192)

senddata = b'hello ' + data

data = data.decode(encoding='utf-8')

print('从：%s 接收到：%s\n' % (addr, data))

skt.sendall(senddata)

if \_\_name\_\_ == "\_\_main\_\_":

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.setsockopt(socket.SOL\_SOCKET,

socket.SO\_REUSEADDR,

1)

s.settimeout(1)

s.bind(('', 5000))

s.listen(1000)

print('服务器已就绪\n')

try:

while True:

try:

skt, addr = s.accept()

th = threading.Thread(target=recv, args=(skt, addr))

th.setDaemon(True)

th.start()

print('接受连接：%s\n' % (addr,))

except socket.timeout:

continue

finally:

s.close()

1.2、TCP Client

#!/usr/bin/env python3

# -\*- coding:utf-8 -\*-

u'''

Created on 2018年11月13日

@author: yuchen

'''

import socket

import sys

\_\_author\_\_ = 'fatyuchen <fatyuchen@qq.com>'

\_\_version\_\_ = '1.0.0'

\_\_company\_\_ = u'重庆交大'

\_\_updated\_\_ = '2018-11-27 08:54'

if \_\_name\_\_ == "\_\_main\_\_":

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.setsockopt(socket.SOL\_SOCKET,

socket.SO\_REUSEADDR,

1)

s.settimeout(1)

try:

s.connect(('127.0.0.1', 5000))

except:

print('连接服务器失败！！')

sys.exit()

try:

msg = '雨辰'

msg = msg.encode(encoding='utf\_8', errors='strict')

s.sendall(msg)

try:

data = s.recv(8192)

print('服务器回复：%s' % data.decode(encoding='utf-8'))

except socket.timeout:

print('超时')

finally:

s.close()

2、UDP编程

2.1、UDP Server

#!/usr/bin/env python3

# -\*- coding:utf-8 -\*-

u'''

Created on 2018年11月13日

@author: yuchen

'''

import socket

\_\_author\_\_ = 'fatyuchen <fatyuchen@qq.com>'

\_\_version\_\_ = '1.0.0'

\_\_company\_\_ = u'重庆交大'

\_\_updated\_\_ = '2018-11-13 09:15'

if \_\_name\_\_ == "\_\_main\_\_":

s = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

s.setsockopt(socket.SOL\_SOCKET,

socket.SO\_REUSEADDR,

1)

s.settimeout(1)

s.bind(('', 5000))

print('服务器已就绪\n')

try:

while True:

try:

data, addr = s.recvfrom(8192)

senddata = b'hello ' + data

data = data.decode(encoding='utf-8')

print('从：%s 接收到：%s\n' % (addr, data))

s.sendto(senddata, addr)

except socket.timeout:

continue

finally:

s.close()

2.2、UDP Client

#!/usr/bin/env python3

# -\*- coding:utf-8 -\*-

u'''

Created on 2018年11月13日

@author: yuchen

'''

import socket

\_\_author\_\_ = 'fatyuchen <fatyuchen@qq.com>'

\_\_version\_\_ = '1.0.0'

\_\_company\_\_ = u'重庆交大'

\_\_updated\_\_ = '2018-11-20 09:52'

if \_\_name\_\_ == "\_\_main\_\_":

s = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

s.setsockopt(socket.SOL\_SOCKET,

socket.SO\_REUSEADDR,

1)

s.settimeout(1)

try:

msg = '雨辰'

msg = msg.encode(encoding='utf\_8', errors='strict')

s.sendto(msg, ('10.1.160.10', 5000))

try:

data, addr = s.recvfrom(8192)

print('服务器回复：%s' % data.decode(encoding='utf-8'))

except socket.timeout:

print('超时')

finally:

s.close()

六、python串口操作

七、python设计COM+组件

八、python高级话题