#倒入自己编写模块，可以把自己编写的模块放到/usr/local/lib/python3.6/site

#还可以设置环境变量PYTHONPATH，指向自己所写的模块路径

#如果模块文件在一个目录中，可以把目录当作包

#mkdir rrr

#cp railway rrr/

#python3

# >>> import.rrr.railway

#md5值： 一种hash哈希算法，单项加密，相同的数据生成相同的乱码，不同的数据，生成的

# 乱码一定不同，不能通过乱码反推回原始数据

搜索路径

搜索路径在sys.path中定义

1. 内置模块
2. hashlib模块

hashlib用来替换md5和sha模块,并使他们的API一致,专门提供hash算法

# import hashlib

# with open('/etc/passwd','rb') as fobj:

# data = fobj.read()

#

# m = hashlib.md5(data)

# print(m.hexdigest())

# import sys

# import hashlib

#

# def check\_md5(fname):

# m = hashlib.md5()

#

# with open(fname,'rb') as fobj:

# while True:

# data = fobj.read(4096)

# if not data:

# break

# m.updata(data) #updata更新

# return m.hexdigest()

#

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

# print(check\_md5(sys.argv[1]))

1. tarfile模块

tarfile模块允许创建、访问tar文件

同时支持gzip、bzip2格式

# tarfile模块

#

# import tarfile

# import os

# os.chdir('/etc')

# tar = tarfile.open('/tmp/anquan.tar.gz','w:gz') #gz可压缩

# tar.add('security')

# tar.add('hosts')

# tar.close()

# os.chdir('/var/tmp')

# tar = tarfile.open('/tmp/xxx.tar.gz')

# tar.extractall() #全压缩

# tar.close()

备份程序

1. 需要支持完全和增量备份

2. 周一执行完全备份

3. 其他时间执行增量备份

4. 备份文件需要打包为tar文件并使用gzip格式压缩

import os

import tarfile

import hashlib

import pickle as p

from time import strftime

def check\_md5(fname):

m = hashlib.md5()

with open(fname, 'rb') as fobj:

while True:

data = fobj.read(4096)

if not data:

break

m.update(data)

return m.hexdigest()

def full\_backup(src\_dir, dst\_dir, md5file):

fname = "%s\_full\_%s.tar.gz" % \

(os.path.basename(src\_dir.rstrip('/')), strftime('%Y%m%d'))

fname = os.path.join(dst\_dir, fname)

tar = tarfile.open(fname, 'w:gz')

tar.add(src\_dir)

tar.close()

md5\_dict = {}

for path, folders, files in os.walk(src\_dir):

for each\_file in files:

key = os.path.join(path, each\_file)

md5\_dict[key] = check\_md5(key)

with open(md5file, 'wb') as fobj:

p.dump(md5\_dict, fobj)

def incr\_backup(src\_dir, dst\_dir, md5file):

fname = "%s\_incr\_%s.tar.gz" % \

(os.path.basename(src\_dir.rstrip('/')), strftime('%Y%m%d'))

fname = os.path.join(dst\_dir, fname)

new\_md5 = {}

with open(md5file, 'rb') as fobj:

old\_md5 = p.load(fobj)

for path, folders, files in os.walk(src\_dir):

for each\_file in files:

key = os.path.join(path, each\_file)

new\_md5[key] = check\_md5(key)

with open(md5file, 'wb') as fobj:

p.dump(new\_md5, fobj)

tar = tarfile.open(fname, 'w:gz')

for key in new\_md5:

if old\_md5.get(key) != new\_md5[key]:

tar.add(key)

tar.close()

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

# cp -r /etc/security /tmp

# mkdir /tmp/backup

src\_dir = '/tmp/security'

dst\_dir = '/tmp/backup/'

md5file = '/tmp/backup/md5.data'

if strftime('%a') == 'Mon':

full\_backup(src\_dir, dst\_dir, md5file)

else:

incr\_backup(src\_dir, dst\_dir, md5file)

1. OOP简介

类(Class):用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。

实例化:创建一个类的实例,类的具体对象

方法:类中定义的函数。

对象:通过类定义的数据结构实例。对象包括两个数据成员(类变量和实例变量)和方法。

# class BearToy:

# def \_\_init\_\_(self,name,size,color): #在实列化时自动执行

# self.name = name

# self.size = size

# self.color = color

#

# def sing(self):

# print('I am %s, lalala....' % self.name)

#

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

# #把参数传递给\_init\_，实列本身，如tidy，自动作为第一个参数传递

# tidy =BearToy('tidy','middle','yellow')

# print(tidy.size)

# print(tidy.color)

# tidy.sing()

# 编写酒店类

# 1. 用于计算住宿开销

# 2. 酒店有会员卡可以打九折

# 3. 每天早餐15元

# 4. 根据住宿天数返加总费用

# class Hotel:

# def \_\_init\_\_(self,basic=200,cf=1.0,br=15):

# self.basic = basic

# self.cutoff = cf

# self.br = br

#

# def calc(self,days=1):

# return (self.basic \* self.cutoff + self.br) \* days

#

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

# stdroom = Hotel()

# print(stdroom.calc())

# print(stdroom.calc(2))

# bigbed = Hotel(basic=230,cf=0.9)

# print(bigbed.calc())

# print(bigbed.calc(2))

# class Vendor:

# def \_\_init\_\_(self,phone,email):

# self.phone = phone

# self.email = email

# def send\_email(self):

# print('Send email to %s' % self.email)

# def send\_phone(self):

# print('send email to %s' % self.phone)

#

#

# class BearToy:

# def \_\_init\_\_(self,name,size,color,phone,email): #在实列化时自动执行

# self.name = name

# self.size = size

# self.color = color

# self.vendor = Vendor(phone,email)

#

# def sing(self):

# print('I am %s, lalala....' % self.name)

#

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

# #把参数传递给\_init\_，实列本身，如tidy，自动作为第一个参数传递

# tidy =BearToy('tidy','middle','yellow','400-123-8899','admin@tedu.cn')

# tidy.vendor.send\_email()

# tidy.vendor.send\_phone()

创建子类只需要在圆括号中写明从哪个父类继承即可

继承描述了基类的属性如何“遗传”给派生类

子类可以继承它的基类的任何属性,不管是数据属性还是方法

# class BearToy:

# def \_\_init\_\_(self,name,size,color): #在实列化时自动执行

# self.name = name

# self.size = size

# self.color = color

#

# def sing(self):

# print('I am %s, lalala....' % self.name)

#

# class NewBear(BearToy): #父类，也叫基类，是BearToy

# def run(self):

# print('running...')

#

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

# b1 = NewBear('big\_bear','Large','Brown')

# b1.sing()

# b1.run()

#

通过继承覆盖方法

如果子类中有和父类同名的方法,父类方法将被覆盖

如果需要访问父类的方法,则要调用一个未绑定的父类方法,明确给出子类的实例

#

# class BearToy:

# def \_\_init\_\_(self,name,size,color): #在实列化时自动执行

# self.name = name

# self.size = size

# self.color = color

#

# def sing(self):

# print('I am %s, lalala....' % self.name)

#

# class NewBear(BearToy): #父类，也叫基类，是BearToy

# def \_\_init\_\_(self,name,size,color,material):

# 1. #BearToy.\_\_init\_\_(self,name,size,color) #三种方法都可以，效果一样

# 2. #super(NewBear,self).\_\_init\_\_(name,size,color)

# 3. # self.name = name

# # self.size = size

# # self.color = color

# self.material = material

#

#

# def run(self):

# print('running...')

#

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

# b1 = NewBear('big\_bear','Large','Brown','cotton')

# b1.sing()

# b1.run()

多重继承

python允许多重继承,即一个类可以是多个父类的子类,子类可以拥有所有父类的属性

# class A:

# def foo(self):

# print('in A-foo')

# class B:

# def bar(self):

# print('in B-bar')

# class C(A,B):

# pass

#

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

# c =C()

# b =bar()

# a =foo()

1. 特殊方法

1.类方法

使用classmethod装饰器定义

第一个参数cls表示类本身

2.\_\_init\_\_方法

实例化类实例时默认会调用的方法

1. \_\_str\_\_方法

打印/显示实例时调用方法

返回字符串

4.\_\_call\_\_方法

用于创建可调用的实例

# #有双划线的方法，一般都是python自带的方法，被称作magic

# class Book:

# def \_\_init\_\_(self,title,author):

# self.title = title

# self.author = author

# def \_\_str\_\_(self):

# return '《%s》' % self.title

# def \_\_call\_\_(self):

# print('《%s》 is written by %s' % (self.title,self.author))

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

# core\_py = Book('core python','wesley') #调用\_\_init\_\_

# print(core\_py) #调用\_\_str\_\_

# core\_py() #调用\_\_call\_\_

出版商程序

1. 为出版商编写一个Book类

2. Book类有书名、作者、页数等属性

3. 打印实例时,输出书名

4. 调用实例时,显示该书由哪个作者编写

# class Book:

# def \_\_init\_\_(self,shuming,zuozhe,yeshu):

# self.shuming = shuming

# self.zuozhe = zuozhe

# self.yeshu = yeshu

# def \_\_str\_\_(self):

# return '《%s》' % self.shuming

# def \_\_call\_\_(self):

# print('《%s》 is changzhihong')

#

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

# dy = Book('python','chang',500)

# print(dy)

# dy()