1. 模块

**1、导入模块**

**1）搜索路径**

-sys.path下的路径

自定义模块一般放在：/usr/local/lib/python3.6/site-packages

-环境变量PYTHONPATH定义的路径

export PYTHONPATH = '/root/PycharmProjects/gjq\_day07'

1. **导入模块的方法**

>>> import os

>>> from random import choice, randint #导入多个模块

>>> import pickle as p #取别名

1. **pycharm规范化代码的方法**

Code -> Reformat Code #使用PEP增强规范优化代码

Code -> Optimize Imports #优化导入模块

**2、包**

1）包：把目录当成一个特殊模块

[root@room9pc01 gjq\_day07]# mkdir mydemo

[root@room9pc01 gjq\_day07]# vim mydemo/foo.py

hi = 'hello world'

[root@room9pc01 gjq\_day07]# vim mydemo/\_\_init\_\_.py

star = '\*‘ \* 30

>>> from mydemo import foo, .. , ..

>>> demo.foo.hi

>>> import mydemo

>>> mydemo.star

#导入mydemo，相当于导入\_\_init\_\_.py模块

#使用mydemo.foo或from-import的方式导入目录下的其他模块

**3、内置模块**

**1）hashlib模块**

>>> import hashlib

>>> m = hashlib.md5(b'1234') #得到二进制数据的md5对象

>>> m.hexdigest() #获取二进制数据的md5值

>>> n = hashlib.md5() #分批次计算一部分，得到最终结果

>>> n.update(b'12')

>>> n.update(b'34')

############################

**案例：使用md5检测文件完整性**

import hashlib

import sys

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()

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

m = check\_md5(sys.argv[1])

print(m)

1. **tarfile模块**

##创建tar文件

>>> import tarfile

>>> tar = tarfile.open('/tmp/security.tar.gz', 'w:gz')

>>> tar.add('/etc/hosts')

>>> os.chdir('/etc')

>>> tar.add('security')

>>> tar.close()

##解压缩tar文件

>>> os.mkdir('/tmp/demo')

>>> os.chdir('/tmp/demo')

>>> tar = tarfile.open('/tmp/security.tar.gz', 'r:gz')

>>> tar.extractall()

>>> tar.close()

**案例：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

思路：

1、判断文件是否需要增量备份（周一/其他time.）

2、完全备份：把目录打包，计算每个文件的md5值（字典记录）

3、增量备份：判断有哪些新增文件和改动的文件，打包这些文件；更新md5

4、方法：递归列出目录中的所有文件

>>> for path, folders, files in os.walk('/root/test'):

... for file in files:

... os.path.join(path, file)

二、OOP基础

**1、OOP简介**

class PigToy: #定义类

def \_\_init\_\_(self, name, col): #绑定方法

self.name = name

self.color = col

def show\_me(self):

print('Hi, my name is %s. I am %s' % (self.name, self.color))

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

piggy = PigToy('Piggy', 'pink') #创建实例

#创建实例时默认调用\_\_init\_\_方法

#piggy -> self，'Piggy' -> name，'pink' -> color

piggy.show\_me()

george = PigToy('George', 'blue')

george.show\_me()

三、OOP进阶

**1、组合和派生**

**1）组合**

class Vendor:

def \_\_init\_\_(self, company, ph):

self.company = company

self.phone = ph

def call(self):

print('Calling %s ...' % self.phone)

class PigToy:

def \_\_init\_\_(self, name, col, company, ph):

self.name = name

self.color = col

self.vendor = Vendor(company, ph)

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

piggy = PigToy('Piggy', 'pink', 'Fans-tech', '400-123456')

print(piggy.vendor.company)

piggy.vendor.call()

**2）派生（继承）**

class PigToy:

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

self.name = name

self.color = color

class NewPigToy(PigToy): #子类继承它父亲的所有属性

def \_\_init\_\_(self, name, color, size): #覆盖父类方法

# PigToy.\_\_init\_\_(self, name, color)

super(NewPigToy, self).\_\_init\_\_(name, color)

#推荐写法，继承父类的属性

self.size = size

def walk(self): #派生其他方法

print('walking ...')

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

a = NewPigToy('piggy', 'pink', 'Middle')

a.walk()

print(a.size)

1. **多重派生（继承）**

class A:

def foo(self):

print('foo method')

def hi(self):

print('hi , A')

class B:

def bar(self):

print('bar method')

def hi(self):

print('hi , B')

class C(A, B): #继承A，B类的所有属性、方法

def hi(self): #若方法冲突，优先级自下向上，自左向右(C>A>B)

print('hi , C')

A

B

C

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

c = C()

c.foo()

c.bar()

c.hi()

**2、特殊方法**

**1）\_\_init\_\_、\_\_str\_\_、\_\_call\_\_方法**

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》是%s写的' % (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\_\_

**2）类方法、静态方法**

-在OOP中，一般只有创建了实例，才能通过实例执行方法；

-类方法：可以没有实例就执行方法。

-静态方法：把一个函数硬生生塞到函数

class Date:

def \_\_init\_\_(self, year, month, day):

self.year = year

self.month = month

self.day = day

@classmethod #声明类方法，返回Date类的实例

def create\_date(cls, date\_str):

year, month, day = map(int, date\_str.split('-'))

#获得map对象的3个int值

return cls(year, month, day)

@staticmethod #声明静态方法，函数可以无cls/self参数

def is\_date\_valid(date\_str):

year, month, day = map(int, date\_str.split('-'))

return year < 4000 and 1 <= month <= 12 and 1 <= day <= 31

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

d1 = Date(2019, 1, 20)

print(d1.month)

d2 = Date.create\_date('2019-1-21') #相当于创建一个实例d2

print(d2.year)

print(Date.is\_date\_valid('2019-13-21')) #直接执行类的静态方法

Tips：注意self，cls等参数是可以自定义的，但编程习惯一般不改变。