python简单的笔记-

笔记本:	zymapple的笔记本		
创建时间:	2018/10/31 23:18	**更新时间:**	2018/11/26 1:26
作者:	519799178@qq.com		
标签:	python		
URL:	http://www.runoob.com/python3/python3-file-writelines.html		

python全栈中的笔记

参数
1.普通参数
2.默认参数
3.关键字参数
4.收集参数 -----> def func(*args):
pass
调用:
func(p1.p2.p3,.....)
收集参数可以和其他参数共存
收集参数后放入了 ‘tuple’
4.1收集参数之关键字收集参数
把关键字参数按字典（dict）格式存入收集参数
def func(**kwargs):
pass
调用:
func(p1=v1,p2=v2,p3=v3,.....)

*收集参数混合调用的顺序问题
普通参数-->关键字参数-->收集参数tuple-->收集参数dict
*收集参数的解包问题
（位置参数，*args，关键字参数，收集参数dict）
func(*a) ------->对list进行解包需要在变量前加一个星号
func(**b) ------>对dict进行解包需要加两个星号

1.函数文档
查看 ---> help(func)

func.__doc__
1.1 --> 在函数内部开始的第一行使用''' ''' 来加上注释

1.变量作用域

1.1 全局（global） 在函数外定义
1.2 局部（local） 在函数内部定义

LEGB原则
L（local）局部作用域
E（Enclosing function locale）外部嵌套函数作用域
G（Global module）函数定义所在模块作用域
B（Buildin）python内置模块的作用域

1.3 提升局部变量为全局变量
.使用global

2.globals,locals函数

3.eval()函数
把一个字符串当成一个表达式来执行，返回表达式执行后的结果

4.exec()函数
与eval（）功能类似，但是不返回结果。

5.递归函数
函数直接或者间接的调用自身

1.内置数据结构（变量类型）
1.list
2.set
3.dict
4.tupie

1.1list 列表
1.1.1一组有顺序的数据的组合
1.1.2创建列表
1.1.2.1 空列表
1.1.3列表的常用操作
访问 （索引下标）
分片操作 （包左不包右）
分片可以控制增长幅度，默认增长幅度为1
[sater:end:1]
下标可以越界，超出后不在考虑多余下标的内容

规定：数组最后一个数字下标为 ‘-1’

默认分片总是从左向右截取

[::-1]为list列表反转排序提供了另一种方法

分片操作时生成了一个新的list

**内置函数 id，

负责显示一个变量或者数据的唯一确定编号
print(id(a))

列表内涵：
通过简单方法创建列表

可以用列表推导式 进行列表的嵌套（嵌套中的列表和列表的双层循环基本等价）

关于列表的函数
append
insert
del
pop
remove
clear
reverse
extend
count
copy 浅拷贝，只拷贝一层内容，对嵌套的列表只拷贝了地址

集合的特征
1. 集合内数据无序，即无法使用索引和分片
2. 集合内数据元素具有唯一性，可以用来排除重复数据
3. 集合内部只能放置可哈希数据

copy
remove
discard

intersection 交集

difference　　差集
union　　并集
issubset　　检查一个集合是否为另一个子集
issuperset　　检查一个集合是否为另一个超集
frozen set　　冰冻集合（不可进行任何修改的集合）
　　是一种特殊的集合

dict 字典
　　字典事一种组合数据，没有顺序的组合数据，数据以键值对形式出现

```
d = {}
d = dict()
d = {'one':1,'two:2','three:3'}
d = dict({'one':1,'two':2,'three':3})
d = dict(one=1,two=2,three=3)
d = dict([('one':1),('two':2),('three':3)])
```
以上列表创建方法基本上事等价的，选择其一来用

字典的特征
　字典事序列类型，但是它是无序序列，所以没有分片和索引
　字典中的数据每个都由键值对组成，即KV对
　　key:必须是可哈希的值，比如，int string,float,tuple,但是 list, set,dict不行~
　　value:任何值

in or not in 检查的是键（K）

```
for k in d :
    print(k.d[k])        #输出的是键和值
for k in d.keys():
    print(k.d[k])        #输出的是键和值
for v in d.values():
    print(v)             #输出的是值
for k,v in d.items():
    print(k,'----',v)     #输出的是键和值
```

字典生成式
```
    dd = {k:v for k,v in d.items() if v % 2 == 0}
```

生成式是为了加入过滤条件IF

字典相关函数
  len,max,min,dict,
    str
    clear
    items   #可迭代类型

  * get 根据指定键返回相应的值，好处是，可以设置默认值（默认值是None）找到值返回，找不到时返回默认值。

    fromkeys: 使用指定的序列作为键，使用一个值，作为字典的所有键的值

OOP 面向对象
    1.面向对象编程
    -基础
    -公有私有
    -继承
    -组合，Mixin
    2.魔法函数
    -魔法函数概述
    -构造类魔法函数
    -运算类魔法函数

面向对象概述
    -OOP思想
      -OO:面向对象
    -OOA:面向对象分析
    -OOD:面向对象设计
    -OOI:xxx的实现
    -OOP:xxx的编程
    -OOA->OOD->OOI:面向对象的实现过程

类和对象的概念
    -类:抽象名词，代表一个集合，共性的事物。
    -对象:具象的事物，单个个体。
    -类和对象的关系
            -一个具体，代表一类事物的某一个个体

-一个是抽象，代表的是一大类事物
类中的内容，应该具有两个方面
-表明事物的特称，叫做属性（变量）
-表明事物功能或动作，称为成员方法（函数）

类的基本实现
-类的命名
-遵守变量命名的规范
-大驼峰（由多个单词构成，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名字

-如何声明一个类
-必须用cladd关键字
-类由属性和方法构成，其他不允许出现
-成员属性定义可以直接使用变量赋值，如果没有值，可使用None

anaconda基本使用
-anaconda主要是一个虚拟环境管理器
-还是一个安装包管理器
-conda list: 显示anaconda安装的包

实例化类
变量 = 类名（）#实例化了一个对象
访问对象成员
-使用‘.’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__

类和对象的成员分析
-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
-对象储存成员是储存在当前的对象中

-对象访问一个成员是，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中，二十得到一个空对象，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

关于self
-self在对象的方法中表示当前对象本身，如果通拓对象调用一个方法，那么该对象会自动传入到当前方法的第一个参数中
-self并不是关键字，只是一个用于接受对象的普通参数，理论上可以用任何一个普通变量名代替
-方法中有self形参的方法称为非绑定类的方法，可以通过对象访问，没有self的事绑定类的方法，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

面向对象的三大特性
-封装
-继承
-多态

6.1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下划线即可

-python的私有不是真的私有，是一种称为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，但是在外部不可以访问。
-封装方法 在成员名称前添加一个下划线
-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

3.2继承
-继承就是一个类可以获得另外一个类中的成员属性和成员方法
-作用 减少代码，增加代码的复用功能，同时可以设置类与类直接的关系
-继承与被继承的概念
-被继承的类叫做父类，也叫基类，也叫超类
-用于继承的类叫做子类，也叫派生类
-继承与被继承一定存在一个is-a 关系
继承的语法
```
#在python中，任何类都有一个共同的父类叫object
class Person():
    name = 'noname'
    age = 0
    def sleep(self):
        print('sleeping.....')
#父类写在括号内
class Teacher(Person):
    pass
```

-继承的特征
-所有类都继承自object类，即所有类都是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内容
-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

-继承变量函数的查找顺序问题

-任何情况都是优先查找自己的变量
-没有则查找父类
-构造函数如果在本类中没有没有定义，则自动查找调用父类构造函数
-如果本类有定义，则不再继续向上查找

-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，则实例化时使用自己的构造函数，不查找父类的构造函数
-如果没有定义，则自动查找父类构造函数
-如果子类没定义，父类构造函数带参数，则构造对象时参数应该按父类参数构造

-super不是关键字，而是一个类
-super的作用时获取MRO(MethodResolustionOrder)列表中的第一个类
-super与父类没有任何实质性关系，但通过super可以调用到父类
-super使用的两个方法，参见在构造函数中调用父类的构造函数

- .__mro__ 看类的血统

-单继承和多继承

- 单继承: 每个类只能继承一个类
- 多继承: 每个类允许继承多个类

-单继承和多继承的优缺点

- 单继承：
  - 传承有序，逻辑清晰，语法简单，隐患少
  - 缺点：功能不能无限扩展，只能在当前唯一的继承链中扩展
- 多继承
  - 优点：类的功能扩展方便
  - 缺点：继承关系混乱

-菱形继承/钻石继承问题

- 多个子类继承自同一个父类，这些子类又被同一个类继承，于是继承关系图形成一个菱形图形
- MRO
- 关于多继承的MRO

    - MRO就是多继承中。用于保存继承顺序的一个列表
    - python本身采用C3算法对多继承的菱形继承进行计算的结果
    - 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父类的书写顺序存放
        - 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亲的父亲

-构造函数

- 在对象进行实例化的时候，系统自动调用的一个函数叫做构造函数，通常此函数用来对实例对象进行初始化，顾名构造函数
- 构造函数一定要有，如果没有，则自动向上查找，按照MRO顺序，知道找到为止

3.3多态
-多态就是同一个对象在不同情况下以不同的状态出现
-多态不是语法，是一种涉及思想
-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，动物分为人类，哺乳类，鸟类
- ‘多态和多态性’

-Mixin设计模式（它不是语法，是一种思想）

- 主要采用多继承方式对类的功能进行扩展

-我们使用多继承语法来实现Mixin
-使用Mixin实现多继承的时候要非常小心

- 首先它必须表示某一单一功能，而不是某个整体
- 职责必须单一，如果需要有多个功能，则写多个Mixin
- Mixin不能依赖于子类的实现
- 子类即使没有继承这个Mixin类，也能照样工作，只是缺少了某个功能

-优点：

- 使用Mixin可以在不对类进行任何修改的情况下，扩充功能
- 可以方便的组织和维护不同功能组件的划分
- 可以根据需要任意调整功能类的组合
- 可以避免创建很多新的类，导致类的继承混乱

4.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
-gerattr:
-setattr:
-delattr:
-dir:获取对象的成员列表

5.类的成员描述符（属性）
-类的成员描述符是为了在类中对类的成员属性进行相关操作而创建的一种方式

- get:获取属性的操作
- set：修改或者添加属性操作
- delete：删除属性的操作

-如果想使用类的成员描述符，大概有三种方法

- 使用类实现描述器
- 使用属性修饰符
- 使用property函数
    - property（fget，gset，fdel，doc）

-无论哪种修饰符都是为了对成员属性进行相应的控制

- 类的方式：适合多个类中的多个属性共用一个描述符
- property：使用当前类中使用，可以控制一个类中多个属性
- 属性修饰符：适用于当前类中使用，控制一个类中的一个属性

-6.类的内置属性

- __dict__:以指点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，以元祖的方式显示

-7.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划线包裹
-操作类

- ‘__init__’:构造函数
- ‘__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（推荐使用）
- ‘__repr__’:返回字符串，与‘__repr__’具体区别需要百度

-描述符相关

- ‘__set__’
- ‘__get__’
- ‘__delete__’

-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（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)）

-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

8.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借助对象的其他对象方法完成

-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

-三个方法的区别需要百度

-属性的三种操作方法

- 赋值
- 读取
- 删除

-类属性property

- 应用场景：
- 对变量除了普通的三种操作外，还想增加一些附加的操作，那么可以通过ptoperty完成

-10.抽象类
-抽象方法：没有具体实现内容的方法称为抽象方法
-抽象方法的主要意义是规范了子类的行为和接口
-抽象类的使用需要借助abc模块
    import abc
-抽象类：包含抽象方法的类叫抽象类，通常称为ABC类

```
#抽象类的实现
import abc

#声明一个类并且制定当前类的元素
class Human(metaclass=abc.ABCMeta):

    #定义一个抽象的方法
    @abc.abstractmethod
    def somking(self):
            pass

    #定义类抽象方法
    @abc.abstractclassmethod
    def drink():
```

```
        pass

    #定义静态抽象方法
    @abc.abstractstaticmethod
    def play():
        pass
```

-抽象类的使用

- 抽象类可以包含抽象方法，也可以包含具体方法
- 抽象类中可以有方法也可以有属性
- 抽象类不允许直接实例化
- 必须继承才可以使用，且继承的子类必须实现所有继承来的抽象方法
- 假定子类没有实现所有继承的抽象方法，这子类也不能实例化
- 抽象类的主要作用是设定类的标准，以便于开发的时候具有统一的规范

-11.自定义类
-类其实是一个类定义和各种方法的只有组合
-可以定义类和函数，然后自己通过类直接赋值（T1）
-可以借助MethodType实现（T2）
-借助于type实现　（T3）
-利用元类实现- MetaClass　（T4）

- 元类是类
- 被用来创造别的类

```
-#自己组装一个类　T2
from types import MethodType

class A():
    pass

#这里注意，下面新建的函数和上面的类没关系
def say(self):
    print('Saying.....')

a = A()
a.say = MethodType(say,A)
```

```
#现在组装在一起了，可以用实例化后的a调用say（）函数
a.say()

结果为:
Saying.....

-#自己组装一个类　T1（这个是直接赋值法）
class A():
    pass

def say(self):
    print('Saying...1...')

class B():
    def say(self):
        print('Saying....2....')

say(9)            #直接调用函数
A.say = say   #赋值

a = A()            #实例化
a.say()            #调用

#B是用来作对比的，表明A通过赋值操作形成的结果和B正常的创建一个类表现的效果是一样的
b = B()
b.say()

#下面是输出结果
Saying...1...
Saying...1...
Saying....2....


-#自己组装一个类　T3 （这个是使用Type方法）
#利用type创造一个类

#想定义类应该具有的成员函数
def say(self):
    print('Saying.....')
```

```
def talk(self):
    print('Talking.....')

#用type来创建一个类
##        help(type)
    ##      type(name,bases，dict)
A = type('AName',(object,),{'class_say':say,'class_talk':talk})

#然后可以像正常访问一样使用类
a = A()
dir(a)  #看一看里面都有什么方法

a.class_say()
a.calss_talk()

#下面是输出结果
Saying.....
Talking.....

-#自己组装一个类   T4   (元类来创建类)

#元类演示

#元类写法是固定的，必须继承自type
#元类一般明明以MetaClass结尾（约定俗成）
class TulingMetaClass(type):
    #注意以下写法
    def __new__(cls,name,bases,attrs):
            #自己的业务处理
            #比如
            print('打印出来了')
            attrs['id'] = '000000'
            attrs['addr'] = '一个奇怪的地址'

            return type.__new__(cls,name,bases,attrs)

#元类定义完就可以使用，使用注意写法
class Teacher(object,metaclass=TulingMetaClass)
        pass

#下面是调用
```

t = Teacher()　　#实例化

t.id
#以下为输出结果

打印出来了
　‘000000’

#1.模块
-一个模块就是一个包含python代码的文件，后缀名称是.py就可以，模块就是python文件
-为什么我们用模块

- 程序太大，编写维护非常不方便，需要拆分
- 模块可以增加代码重复利用的方式
- 当做命名空间使用。避免命名冲突

-如何定义模块

- 模块就是一个普通文件，所以任何代码可以直接书写
- 不过根据模块的规范，最好在模块中编写以下内容
    - 函数（单一功能）（高内聚，低耦合）
    - 类（相似功能的组合，或者类似业务模块）
    - 测试代码

-如何使用模块

- 模块直接导入
    - 加入模块名称直接以数字开头
        - 借助于importlib可以实现导入以数字开头的模块名称
            - import importlib
            - #下面代码是把名称为 ‘01’ 的模块改名为 ‘A’
            - A =importlib.import_module('01')
            - B = A.XXX（）
            - B.CCC（）

        - #上面‘XXX’为‘01（A）’模块里面的类
        - #‘CCC’为调用‘01’模块类里面包含的函数
- 语法
    - import module_name

      module_name.function_name
      module_name.class_name

- import 模块 as 别名　　(！！！！重要)
    - 导入的同时给模块起一个别名
    - 其余用法跟第一种相同

- from module_name import func_name,class_name
    - 按上述方法有选择性的导入
    - 使用的时候可以直接使用导入的内容，不需要前缀
    -

- from module_name import *
    - 导入模块所有内容
        - 使用这个方法与直接import不同的是：　‘不需要前缀了’

重要
**　if __name__ == '__main__':
　　print('我是模块XX呀。')
　　#做一个判断，程序是否是在自己调用自己（‘__main__’）
　　#　'__main__'是模块里主进程的代指
【***此判断语句建议一直作为程序的入口***】

- ‘if __name__ == '__main__’的使用
    - 可以有效避免模块代码在被导入的时候被动执行的问题
    - 建议所有程序的入口都以此代码作为入口

-2.模块的搜索路径和存储

- 什么是模块的搜索路径
    - 加载模块的时候，系统会在哪些地方寻找此模块

- 系统默认的模块搜索路径
    - import sys

      sys.path　属性可以获得路径列表

      type(sys.path)　　#它的类型是list

- 添加搜索路径
    - sys.path.append(dir)

- 模块的加载顺序
    - 1.先搜索内存中已经加载好的模块
    - 2.搜索python的内置模块
    - 3.搜索sys.path路径

-包

- 包是一种组织管理代码的方式，包里面存放的是模块
- 用于将模块包含在一起的文件夹就是包
- 自定义包的结构

    包里面要有‘__init__.py’文件，这个文件是包的标志性文件

-包的导入操作

- import package_name
    - 直接导入一个包，可以使用__init__.py中的内容
    - 使用方式是：

      package_name.func_name
      package_name.class_name.func_name()

    - 此种方式的访问内容是
    - 案例。。。
- import package_name as 别名
    - 具体用法和作用方式，和上述简单导入一直
    - 注意的是此种方法是默认对__init__.py内容的导入

- import package.module
  - 导入包中某一个具体的模块
  - 使用方法

```
package.module.func_name
package.module.class.fun()
package.module.class.var
```

- import package.module as 别名

-from.....import　导入

- from package import module,module2,
- 此种导入方法不执行__init__的内容
- from package import *
  - 导入当前包‘__init__py’文件中所有的函数和类
  - 使用方法

```
fun_name()
class_name.func_name()
class_name.var
```

-from package.module import *

- 导入保重制定的模块的所有内容
- 使用方法

```
func_name()
class_name.func_name()
```

-在开发环境中经常会使用其他模块，可以在当前包中直接导入其他模块的内容

- import 完整的包或者模块的路径

-'__all__'的用法

- 在使用from package import * 的时候，*可以导入的内容
- ‘__init__.py’中如果文件为空，或者没有‘__all__’那么只可以把__init__中的内容导入

- ‘__init__’如果设置了‘__all__’的值，那么这按照‘__all__’制定的子包或者模块进行加载，如此这不会导入‘__init__’中的内容
- ‘__all__’ = ['module','module2','package1',.....]

-命名空间

- 用于区分不同位置不同功能但相同名称的函数或者变量的一个特定前缀
- 作用是防止命名冲突

1.异常
-广义上的错误分为错误和异常
-错误指的是可以人为避免
-异常是指在语法逻辑正确的前提下，出现的问题
-在python里，异常是一个类，可以处理和使用

2.异常处理
-不能保证程序永远正确运行
-但是，必须保证程序在最坏的情况下得到的问题呗妥善的处理
-python的异常处理模块全部语法为：

```
try:
    尝试实现某个操作，
    如果没出现异常，任务就可以完成
    如果出现异常，将异常从当前代码块扔出去尝试解决异常
except   异常类型1：
    解决方案1：用于尝试在此处处理异常，解决问题
except   异常类型2：
    解决方案2： .....
except    （异常类型1，异常类型2....）
    解决方案：针对多个异常使用相同的处理方式
except:
    解决方案：所有异常的解决方案
else：
    如果没有出现任何异常，将会执行此处代码
finally：
    不管有没有异常都要执行的代码
```

-流程

- 1.执行try下面的语句
- 2.如果出现异常，则在except语句里面查找对应异常并进行处理
- 3.如果没有出现异常，则执行else语句内容
- 4.最后，不管是否出现异常，都要执行finally语句

-除了except（最少一个）以外，else和finally都是可选的

-如果是多种error的情况

- 需要把更具体指向的error 往更前放
- 在异常继承关系中，越是子类的异常，越要往前放
- 越是父类的异常，越需要往后（下）放

-在处理异常的时候，一旦拦截到某一个异常，则不在继续向下进行查找，直接进行下一个代码，即有finally这执行finally语句块，否则就执行下一个大的语句块

-*所有异常都继承自Excepton

-用户手动引发异常
-当某些情况，用户希望自己引发一个异常的时候，可以使用
raise 关键字来引发异常
-语法 raise ErrorClassName

-自己定义异常

- 需要注意：自定义异常必须是系统异常的子类
- class yongshierror（NameError）：

pass
-关于自定义异常

- 只要是raise异常，则推荐自定义异常
- 在自定义异常的时候，一般包含以下内容：
  - 自定义发生异常的异常代码
  - 自定义发生异常后的问题提示
  - 自定义发生异常的行数
- 最终的目的是，一旦发生异常，方便快速定位错误现场
-

-else语句

-常用模块

- celendar
- time
- datetime
- timeit
- os
- shutil
- zip
- math
- string
- 上述模块使用理论上都应该先导入，string是特例
- calendar,time,datetime的区别参考中文意思

-calendar模块

- 跟日历相关的模块
- isleap：判断某一年是否闰年
- leapdays：获取制定年份之间的闰年的个数
- month：获取某个月的日历字符串
- monthrange：获取一个月从周几开始和总天数
- monthcalendar：返回一个月每天的矩阵列表
- prcal：直接打印日历
- prmonth：直接打印整个月的日历
- weekday：获取周几

-time模块
--时间戳

- 一个时间表示，根据不同语言，可以是整数或者浮点数
- 是从1970年1月1日0时0分0秒到现在经历的秒数
- 如果表示的时间是1970年以前或者太遥远的未来，可能出现异常
- 32位操作系统能够支持到2038年

-UTC时间

- 世界协调时间，也叫做世界标准时间
  - 中国　utc+8　东八区

-夏令时

- 夏令时就是在夏天的时候将时间调快一小时

-时间元组

- 一个包含时间内容的普通元组

-需要单独导入
-时间模块的属性

- timezone：当前时区和UTC时间相差的秒数，在没有夏令时的情况下的间隔（一个固定值）
- altzone：获取当前时区与UTC时间相差的秒数，包含夏令时
- daylight：测试当前是否是夏令时时间状态，0表示true（linux）

-得到一个时间戳

- time.time（）

-time.localtime（）

- 得到当前时间的时间结构
- 可以通过点号操作符得到相应的属性元素的内容

-asctime（）

- 返回元组的正常字符串化之后的时间格式
- 返回值：字符串 Rue Jun　31　11:11:11　2011

-ctime（）

- 获取字符串化的当前时间

-mktime（）

- 使用时间元组获取对应的时间戳
- 浮点数时间戳

- 格式：time.mktime（时间元组）

-clock（）

- 获取CPU时间　　3.0——3.3版本直接使用，3.6调用有问题

-sleep（）

- 使程序进入睡眠，n秒后继续

-strftime（）********重要

- 将时间元组转化为自定义的字符串格式
  - t=time.localtime()
  - ft=time.strftime('%y年m%月d%日 H%:M%'，t)
  - print（ft)

-datetime模块

- datetime提供日期和时间的运算和表示
- datetime常见属性
- datetime.date ： 提供year，month，day属性
  - print（datetime.date（2011,11,11））
  - 2011-11-11
- datetime.time : 提供一个日期和时间，hour，minute。sec，microsec
- datetime.datetime：提供日期和时间的组合
- datetime.timedelta：提供一个时间差，时间长度
  - datetime.datetime常用类方法：
    - today
    - now
    - utcnow
    - fromtimestamp：从时间戳中返回本地时间
  - datetime.timedelta
    - 表示一个时间间隔
    - 先导入

-timeit-时间测量工具

*-OS操作系统相关

- 跟操作系统相关，主要是对文件进行操作
- 与系统相关的操作，主要包含在三个模块里
  - os，操作系统目录相关
  - os.path，系统路径相关操作
  - shutil，高级文件操作，目录树的操作，文件赋值，删除，移动
- 路径：
  - 绝对路径：总是从根目录上开始
  - 相对路径：基本以当前环境为开始的一个相对的地方

*-os模块

- getcwd（）获取当前的工作目录
- chdir（）改变当前的工作目录
- listdir（）获取一个目录中所有子目录和文件的名称列表
- makedirs（）递归创建文件夹
- system（）运行系统shell命令
  - 一般推荐使用subprocess代替
- getenv（）获取指定的系统环境变量值
- putenv（）添加环境变量
- exit（）退出当前程序

-值部分

- os.curdir: curretn dir,当前目录
- os.pardir: parent dir,父亲目录
- os.sep: 当前目录的路径分隔符
  - windows: '\'
  - linux: '/'
- os.linesep: 当前系统的换行符号
  - windows: ‘\r\n’
  - unix,linux,macos: '\n'
- 上面两个符号不要手动拼写，要用函数，否则代码不具有移植性*
- os.name: 当前系统名称
  - windows: nt
  - mac,unix,linux: posix

-os.path模块跟路径相关的模块

- abspath() 将路径转化我绝对路径

- abselute 绝对
- ‘.’代表当前目录

  ‘..’代表父目录

- basename（）获取路径中的文件名部分
- join（）将多个路径拼合成一个路径
- split（）将路径切割为文件夹部分和当前文件部分
- isdir（）检测是否是目录
- exists（）检测文件或者目录是否存在

*-shutil模块

- copy（）复制文件/　拷贝的同时，可以给文件重命名
- copy2（）赋值文件，保留元数据（文件信息）　copy和copy2的唯一区别在于copy2复制文件时尽量保留元数据
- copyfile（）将一个文件中的内容复制到另一个文件当中
- move（）移动文件/文件夹

-归档和压缩

- 归档：把多个文件或者文件夹合并到一个文件当中
- 压缩：用算法把多个文件或者文件夹无损或者有损合并到一个文件当中

- make_archive()　归档操作
- unpack_archive（）解包操作

*-ZIP压缩包

- 模块名称 zipfile
- ZipFile（）
- getinfo()
- namelist()　获取ZIP文档内所有文件的名称列表
- extractall()　解压缩ZIP文档中的所有文件到当前目录

-random

- 随机数

- 所有的随机数模块都是伪随机

- random（）获取0-1之间的随机小数
- choice（）随机返回序列中的某个值
- shuffle（）随机打乱列表
- randint（）返回一个a到b之间的随机整数，包含a和b

-LOG模块

  - www.cnblogs.com/yyds/p/6901864.html

-函数式编程

-Python语言的高级特征

- 函数式编程（FunctionalProgramming）
  - 基于lambda演算的一种编程方式
    - 程序中只有函数
    - 函数可以作为参数，同样可以作为返回值
    - 纯函数式编程语言：LISP, Haskell
- Python函数式编程只是借鉴函数式编程的一些特点，可以理解成一半函数式一半Python
- 需要讲
  - 高阶函数
  - 返回函数
  - 匿名函数
  - 装饰器
  - 偏函数

-lambda表达式

- 函数：最大程度的复用代码
  - 存在问题：如果函数很小，很短，则会造成啰嗦
  - 如果函数被调用次数少，则会造成浪费
  - 对于阅读者来说，造成阅读流程的被迫中断

- lambda表达式（匿名函数）
  - 一个表达式，函数体相对简单
  - 不是一个代码块，仅仅是一个表达式
  - 可以有参数，有多个参数时，用逗号隔开
- lambda表达式的用法
  - 1.以lambda开头
  - 2.紧跟一定的参数（如果有的话）
  - 3.参数后用冒号和表达式主题隔开
  - 4.只是一个表达式，所以没有return

- 高阶函数
  - 把函数作为参数使用的函数，叫做高阶函数
    - 变量可以赋值
    - 函数名称就是一个变量
    - A 变量 / A（）函数
    - 函数A（）可以用 X = A 进行赋值操作 使X（） == A （）
    - 既然函数名称是变量，则应该可以被当做参数传入另一个函数
    - 举个例子
      - def funA(n):

        return n*100
        def funB(n):
        return funA(n)*3
        def funC(n, f):
        return f(n)*3
        比较funC与funB，显然funC的写法要 优于funB, 因为改变输出只用传入不同的 参数就行了，不用更改代码本来的 rutern。

-系统高阶函数-map

- 愿意就是映射，即把集合或者列表的元素，每一个元素都按照一定规则进行操作，生成一个新的列表或者集合
- map函数式系统提供的具有映射功能的函数，返回值是一个对待对象
- map类型是一个可迭代的结构，所以可以使用for遍历

-reduce

- 原意是归并，缩减
- 把一个可迭代对象最后归并成一个结果
- 对于 参数有要求：必须有两个参数，必须有返回结果
- ruduce([1.2.3.4.5]) == f(f(f(f(1.2).3).4).5)
- reduce 需要导入functools包

-filter函数

- 过滤函数：对一组数据进行过滤，符合条件的数据会生成一个新的列表并返回
- 跟map相比较：
  - 相同点：都对列表的每一个元素逐一进行操作
  - 不同点：
    - map会生成一个跟原来数据相对应的新队列
    - filter不一定，只要符合条件的才会进入新的数据集合
  - filter函数怎么写：
    - 利用给定函数进行判断
    - 返回值一定是个布尔值
    - 调用格式：filter(f,data)，f是过滤函数，data是数据

-高阶函数-排序

- 把一个序列按照给定算法进行排序
- key: 在排序前对每一个元素进行key函数运算，可以理解成按照key函数定义的逻辑进行排序
- python2和python3相差巨大
- sorted（）

-返回函数

- 函数可以返回具体的值
- 也可以返回一个函数作为结果

-闭包（closure）

- 当一个函数在内部定义函数，并且内部的函数应用外部函数的参数或者局部变量，但内部函数被当做返回值的时候，相关参数和亮亮保存在返回的函数中，这种结果，叫闭包
- 闭包里的常见大坑
  - 返回闭包时，返回函数不能饮用任何循环变量

- 解决办法：在创建一个函数，用该函数的参数绑定循环变量的当前值无论该循环变量以后如何改变，已经绑定的函数参数值不在改变
- 下图一为在闭包中出现的错误。i不能实时传入f()中

```
def count():
    # 定义列表，列表里存放的是定义的函数
    fs = []
    for i in range(1,4):
        # 定义了一个函数f
        # f是一个闭包结构
        def f():
            return i*i
        fs.append(f)
    return fs

f1,f2,f3 = count()
print(f1())
print(f2())
print(f3())
```

```
9
9
9
```

- 下图二为多新建一个函数来接受需要实时改变的变量，作为参数来传入

In [24]:

```
# 修改上述函数
def count2():
    def f(j):
        def g():
            return j*j
        return g
    fs = []
    for i in range(1,4):
        fs.append(f(i))
    return fs

f1,f2,f3 = count2()
print(f1())
print(f2())
print(f3())
```

```
1
4
9
```

-装饰器

- 在不改动函数代码的基础上无限制扩展函数功能的一种机制，本质上讲，装饰器是一个返回函数的高阶函数
- 装饰器的使用：使用@语法。即在每次要扩展到函数定义前使用@+函数名

*-下图为装饰器的用法，调用hello（）函数时，同时打印出时间。

```python
# 任务:
# 对hello函数进行功能扩展，每次执行hello万打印当前时间

import time

# 高阶函数，以函数作为参数
def printTime(f):
    def wrapper(*args, **kwargs):
        print("Time: ", time.ctime())
        return f(*args, **kwargs)
    return wrapper
```

```python
# 上面定义了装饰器，使用的时候需要用到@， 此符号是python的语法糖
@printTime
def hello():
    print("Hello world")
```

- 装饰器的好处是，一旦定义。这可以装饰任意函数
- 一旦被其装饰，则把装饰器的功能直接添加到定义函数的功能上

-上面对函数的装饰，使用了系统定义的语法糖
-下面开始手动执行装饰器

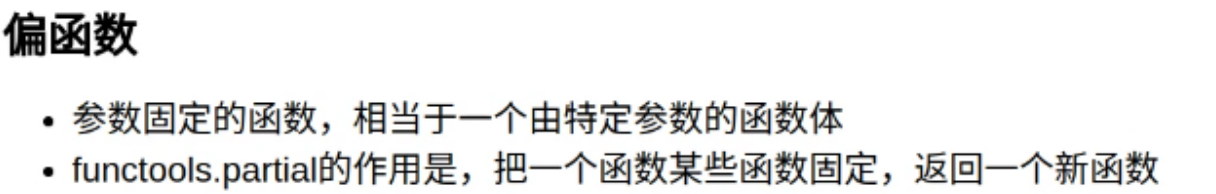
In [42]:
```python
# 上面对函数的装饰使用了系统定义的语法糖
# 下面开始手动执行下装饰器
# 先定义函数

def hello3():
    print("我是手动执行的")

hello3()

hello3 = printTime(hello3)
hello3()
```

```
我是手动执行的
Time:  Mon Apr  2 21:22:10 2018
我是手动执行的
```

*-上图 手动调用了装饰器，其中f 就是需要传入的函数（也就是，谁来调用这个装饰器的函数）与前面两个图连起来看，可以更容易明白装饰器的用法。

-偏函数

- 参数固定的函数，相当于一个有特定参数的函数体
- functools.partial的作用是，把一个函数某些函数固定，返回一个新函数




```
In [48]: import functools
         #实现上面int16的功能
         int16 = functools.partial(int, base=16)

         int16("12345")
```

```
Out[48]: 74565
```

*-上图中，相当于在不改变原函数的情况下，把系统提供的int（）函数中的base默认值固定为16（以前默认为10）

-高阶函数补充

-zip

- 把两个可迭代内容生成成一个可迭代的tuple元素类型组成的内容

-enumerate

- 与zip功能比较像
- 对可迭代对象里的每一元素，配上一个索引，然后索引和内容构成tuple类型

-collections模块

- namedtuple
- deque

- namedtuple
  - tuple类型
  - 是一个可命名的tuple
  - 没有函数的类。相当C语言中的结构

```
import collections
Point = collections.namedtuple("Point", ['x', 'y'])
p = Point(11, 22)
print(p.x)
print(p[0])
```

*-如上图中。可以当成类来用，也能当成tuple来使用索引功能

-deque

- 比较方便的解决了频繁删除插入带来的效率问题

-defaultdict

- 当直接读取dict不存在的属性时，直接返回默认值

-counter

- 统计字符串个数

-调试技术
-调试流程：单元测试-->集成测试-->交测试部

- 分类：
  - 静态调试：
  - 动态调试：

-pdb调试

-pycharm调试

- run/debug模式
- 断点: 程序的某一行吗，在debug模式下，遇到断点就会暂停

-单元测试

- pyunit

-持久化-文件
-文件

- 长久保存信息的一种数据信息集合
- 常用操作
  - 打开关闭（文件一旦打开。需要关闭操作）
  - 读写内容
  - 查找
- open函数
  - open函数负责打开文件，带有很多参数
  - 第一个参数：文件的路径和名称。必须有
  - mode：表明文件用什么方式打开
    - r：以只读方式打开
    - w： 以写方式打开，但会覆盖以前的内容
    - x： 创建方式打开，如文件已经存在，会报错
    - a： append方式，以最佳的方式对文件内容进行写入
    - b： binary方式，二进制方式写入
    - t： 文本方式打开
    - +： 可读写

** f称之为文件句柄
r表示后面字符串内容不需要转义

- f = open(r'test.txt','w')

-以写方式打开文件，默认是如果没有文件。则创建文件

*-with语句

- with语句使用的技术是一种称为上下文管理协议的技术（ContextManagementProtocal）
- 自动判断文件的作用域，自动关闭不再使用的打开文件句柄

- with open(r'text.txt','r') as f:

  pass

- 上面语句块开始对文件f进行操作
- strline = f.readline()
- 上面语句为按行读取内容
- while strline:
- print(strline)
- strline = f.readline()
- 在本模块中不需要再使用close关闭文件f

-list能用打开的文件作为参数，把文件内每一行内容作为一个元素

```
with open(r'text.txt','r') as f:
    # 以打开的文件f作为参数，创建列表
    l = list(f)
    for line in l:
        print(line)
```

-read是按字符读取文件内容

- 允许输入参数决定读取几个字符。没有指定，从当前位置读取到结尾。否则，从当前位置读取指定个字符数

In [9]:
```
# read是按字符读取文件内容
# 允许输入参数决定读取几个字符，如果没有制定，从当前位置读取到结尾
# 否则，从当前位置读取指定个数字符

with open(r'test01.txt', 'r') as f:
    strChar = f.read(1)
    print(len(strChar))
    print(strChar)
```

-seek（offset，from）

- 移动文件的读取位置，也叫读取指针
- from的取值范围：
  - 0： 从文件头开始偏移
  - 1： 从文件当前位置开始偏移
  - 2： 从文件末尾开始偏移

- 移动的单位是字节（byte）
- 一个汉字由若干个字节构成
- 返回文件指针的当前位置

-tell函数

- 用来显示文件读写指针的当前位置
- tell的返回数字的单位是byte
- read是以字符为单位的

-write文件的写操作

- write(str):把字符串写入文件
- writelines(str): 方法用于向文件中写入一序列的字符串。（可以是由迭代对象产生的）
-

-持久化-pickle

- 序列化（持久化，落地）：把程序运行中的信息保存到磁盘上
- 反序列化：序列化的逆过程
- pickle：python提供的序列化模块
- pickle.dump：序列化
- pickle.load：反序列化

-持久化-shelve

- 持久化工具
- 类似字典，用kv对保存数据，存取方式和字典也很类似
- open，close

```python
# 使用shelve创建文件并使用
import shelve

# 打开文件
# shv相当于一个字典
shv = shelve.open(r'shv.db')

shv['one'] = 1
shv['two'] = 2
shv['three'] = 3

shv.close()

# 通过以上案例发现，shelve自动创建的不仅仅是一个shv.db文件，还包括其他
```

-shelve特性

- 不支持多个应用并行写入
  - 为解决这个问题，open的时候可以使用flag=r
- 写回问题
  - shelv情况下不会等待持久化对象进行任何修改
  - 解决办法：强制写回：writeback=true

In [34]:
```python
# shelve忘记写回，需要使用强制写回
shv = shelve.open(r'shv.db')
try:
    k1 = shv['one']
    print(k1)
    # 此时，一旦shelve关闭，则内容还是存在于内存中，没有写回数据库
    k1["eins"] =100
finally:
    shv.close()


shv = shelve.open(r'shv.db')
try:
    k1 = shv['one']
    print(k1)
finally:
    shv.close()
```

```
{'eins': 1, 'zwei': 2, 'drei': 3}
{'eins': 1, 'zwei': 2, 'drei': 3}
```

-log模块（logging）
http://www.cnblogs.com/yyds/p/6901864.html

-logging模块提供模块级别的函数记录日志
-包括四大组件

-1.日志相关概念
-日志的级别（level）

- 不同的用户关注不同的程序信息
  - DEBUG
  - INFO
  - NOTICE
  - WARNING
  - ERROR
  - CRITICAL
  - ALERT
  - EMERGENCY

-IO操作=>不要频繁操作
-LOG的作用

- 调试
- 了解软件的运行情况
- 分析定位问题

-日志信息

- time
- 地点
- level
- 内容

-成熟的第三方日志

- log4j
- log4php
- logging

-2. logging模块

- 日志级别

    - 级别可自定义
    - DEBUG
    - INFO
    - WARNING
    - ERROR
    - CRITICAL
- 初始化/写日志 实例需要制定级别：只有当级别等于或高于指定级别才被记录
- 单例模式
- 使用方式
    - 直接使用logging（里面封装了其他组件）
    - logging四大组件直接定制

-2.1 logging模块级别的日志

- 使用以下几个函数
    - logging.debug()
    - logging.info()
    - logging.warning()
    - logging.error()
    - logging.critical()
    - logging.log()
    - logging.basicConfig() 对root logger 进行一次性配置
- logging.basicconfig()
    - 只在第一次调用的时候起作用
    - 不配置logger则使用默认值
        - 输出：sys.stderr
        - 级别：WARNING
        - 格式：level：log_name:content

*-

- Ctrl+左键
    - -在pycharm中直接显示函数的文档文件

- filename 日志保存的文件名字和路径
- filemode 模式
- format 格式

- datefmt 时间，日期
- style 分隔符
- level 级别
- stream 流
- handlers

- LOG_FORMAT = '%(asctime)s*****%(levelname)s*****%(message)s'
- logging.basicConfig(filename='111'. level=logging.DEBUG. format= LOG_FORMAT)
- 上面的LOG_FORMAT是自定义了输出格式

```
1   import logging
2
3   LOG_FORMAT = "%(asctime)s*****%(levelname)s*****%(message)s"
4
5   logging.basicConfig(filename="tulingxueyuan.log", level=logging.DEBUG, format=LOG_FORMAT)
6
7   logging.debug("This is a debug log.")
8   logging.info("This is a info log.")
9   logging.warning("This is a warning log.")
10  logging.error("This is a error log.")
11  logging.critical("This is a critical log.")
12
13
14
```


-format参数

- asctime %()s
- created %()f
- relativeCreated %()d
- msecs %()d
- levelname %()s
- levelno %()s
- name %()s
- message %()s
- pathname %()s
- filename %()s
- module %()s
- lineno %()d
- funcName %()s
- process %()d
- processName %()s

- thread %()d
- threadName %(thread)s

-2.2 logging模块的处理流程

- 四大组件
  - 日志器（logger）：产生日志的一个接口
  - 处理器（Handler）：把产生的日志发送到相应的目的地
  - 过滤器（Filter）：精细的控制哪些日志输出
  - 格式器（Formatter）：对输出信息进行格式化

- logger
  - 产生一个日志
  - 操作
    - logger.setLevel()
    - logger.addHandler()
      - logger.removeHandler()
    - logger.addFiter()
      - logger.removeFiter()
    - logger.debug()
    - logger.exception()
    - logger.log()

  - 如何得到一个logger对象
    - 实例化
    - logging.getLogger()

- Handler(处理器)
  - 把log发送到指定位置
  - 方法
    - setLevel
    - setFormat
    - addFilter, removeFilter
  - 不需要直接使用， Handler是基类
    - logging.StreamHandler
    - logging.FileHandler

        - logging.Handlers.RotatingFileHandler
        - logging.Hanlders.TimedRotatingFileHandler
        - logging.Hadlers.HTTPHandler
        - logging.Handlers.SMTPHandler
        - logging.NullHandler

- Format类
    - 直接实例化
    - 可以继承Format添加特殊内容
    - 三个参数
        - fmt: 指定消息格式化字符串，如果不指定该参数则默认使用message的原始值
        - datefmt:指定日期格式字符串，如果不指定该参数则默认使用 '%Y-%M-%D %H:%M:%S'
        - style: Python 3.2新增的参数，可以取值为 '%' '{' 和 '$' ，如果不指定该参数则默认使用 '%'

- Filter类（过滤器）
    - 可以被Handler和logger使用
    - 控制传递过来的信息的具体内容

-多线程

- 环境
- www.cnblogs.com/jokerbj/p/7460260.html
- www.dabeaz.com/python/UnderstandingGIL.PDF
- 多线程VS多进程
    - 程序：存入文件的一堆代码
    - 进程：程序运行的一个状态
        - 包含地址空间，内存，数据栈
        - 每个进程有自己完全独立的运行环境，多进程共享数据是一个问题
    - 线程
        - 一个进程的独立运行片段，一个进程可以有多个线程
        - 轻量化的进程
        - 一个进程的多个线程间共享数据和上下文运行环境
        - 共享互斥问题

- 全局解释器锁（GIL）
  - python代码的执行是由python虚拟机进行控制
  - 在主循环中只能有一个控制线程在执行
- python包
  - thread：有问题，不好用，python3改成了_thread
  - threading: 通行的包
- threading的使用
  - 直接利用threading.Thread生成Thread实例
    - 1. t = threading.Thread(target=xxx, args=(xxx,))
    - 2. t.start(): 启动多线程
    - 3. t.join(): 等待多线程执行完成
  - 守护线程
    - .daemon = true
    - 上下命令等价
    - .setDeamon(true)
    - 如果在程序中将子线程设置成守护线程，则子线程会在主线程结束的手自动结束
    - 一般认为，守护线程不重要或者不允许离开主线程独立运行
    - 守护线程能否有效果和环境有关
    - 守护线程要在多线程启动前进行设置（要在'.start'之间进行设置）
  - 线程常用属性
    - threading.currenThread
    - threading.enumerate
    - threading.activeCount
    - thr.setName
    - thr.gerName
- 直接继承自threading.Thread
  - 直接继承Thread
  - 重写run函数
  - 类实例可以直接运行
  -

‘python’ 中下划线的使用