字符串+数字

- 首字母大写title() 全部大写 upper() 全部小写lower() ——name.title() 对变量name执行方法title（），变量name 内的值不变
- 合并字符串 “+” ；制表符 “\t"；换行符 ‘\n’ ， print("\nhello"+'Python') \r——回车
- 剔除字符串两端的空白strip()；剔除字符串末端的空白rstrip()；剔除字符串前端的空白lstrip()（空白泛指任何非打印字符，包括制表符、换行符、空格）
- 将非字符串转换为字符串str()
- s = 'zhang' s[2]为 ‘a' s[1:3]或s[-4:-2] 为 ‘ha’ （切片） 注：[A:B]不包括B
- eval() 函数用来执行一个字符串表达式，并返回表达式的值。 例：eval('3+2')为 5
- 格式化输出：%s， %d， %f

  一种字符串格式化的语法， 基本用法是将值插入到%s占位符的字符串中。

  %s,表示格式化一个对象为字符

  "%±(正负号表示)3(数字表示字符串的长度)s"%(取代s的字符串)

  %s string型 表示格式化一个对象为字符 "%s1"%S2 s1放置的是一个字符串(格式化字符串) S2放置的是一个希望要格式化的值

```
string = "good"
print("string = zzw%+5s"%string) # %+5s 字符串长度小于5时，左侧补空格使得总长度为5
print("string = zzw%-5s"%string) # %-5s 字符串长度小于5时，右侧补空格使得总长度为5
print("string = zzw%.3s"%string) # %.3s 截取前3个字符
print("string = zzw%5.3s"%string) # %.3s 截取前3个字符，字符串长度小于5时，左侧补空格使得总长度为5
print("string = zzw%*.*s"%(5,3,string)) # %.3s 截取前3个字符，字符串长度小于5时，左侧补空格使得总长度为5
print ("He is %d years old"%(25)) #打印整数
print ("His height is %.2f m"%(1.83)) #打印浮点数（指定保留小数点位数）
```

```
"{} ， {}and{} ".format("a","b","c") #输出a,b and c 字符串方法format
"{2} ， {1}and{0} ".format("a","b","c") #输出 c,b and a
```

```
a={'name':'wenshuang',"lover":'zhenwei'}
print("{name} love {lover}".format(**a)) # 字典format应用
```

列表

```
# 格式（方括号）：
familys = ['zhangzhenwei', 'huwenshuang', 'kids']
```

- 列表末尾添加元素 familys.append()；列表中间添加元素，其他右移 familys.insert（1，" kid"）；
- 删除第一个元素（以后不再使用）del familys [0]；poped_familys= familys.pop(2) 将第3个元素弹出(后面可继续使用已删除的元素)，并将弹出变量保存到变量 poped_familys ,pop()默认弹出最后一个元素， familys内永久改变；

```
1 familys = ['zhangzhenwei', 'huwenshuang',  'kids']
2 poped_familys= familys.pop(2)
3 # poped_familys为'kids';
```

familys.remove('kids')——根据元素值删除（注意:remove（）只删除第一次出现的值，后面的需用循环判断）；

- 函数 sorted(familys)——临时顺序；注意方法和函数调用方式区别；

方法familys.sort()——按首字母永久排序（无返回值，c=familys.sort(),c返回None,）；familys.sort(reverse = True)——永久倒序；

函数(函数参数1，函数参数2）； 对象.方法（）

- len(familys)——确认元素个数/列表长度；
- 遍历列表每一个元素

```
1 familys = ['zhangzhenwei', 'huwenshuang', 'kids']
2  for member in familys:
3  print(member)
```

- range(1,4) 表示（1，2，3），numbers = list(range（1，4））创建数字列表 numbers,函数list()——元组转换为列表
- digits = [1,3,5,4,2] min(degits), max(degits), sum(degits): 可直接计算
- 列表解析(将for循环和创建新元素的代码合为1行)

```
1 squares = [value for value in range(1,6)]
2 # squres = [1, 2, 3, 4, 5]
```

- 使用列表一部分 print(squares[2:4]) ——[3,4]（注，不包含：后位置）；

print(squares[2:]-——[3,4,5,6]; print(squares[:4]-——[1,2.3,4];

- copys = squares[:] 复制列表,不相互影响，同copys = squares.copy() copys = list(squares)；若copys = squares，对squares操作时，copy亦变化
- **元组**（圆括号）——不可变的的列表； dimensions = (100,200) ,若要改变，需对dimensions这个元组变量重新赋值；

```
1 dimensions = (100,200)
2 dimensions = (1200, 600)
```

- if __name__ == '__main__': 可作为判断是否调用其他模块的条件

```
1 # module.py
2 if __name__ == '__main__':
```

```
 print('相等')
print(__name__)
# 运行输出 相等 __main__
```

```
# example.py
import module
# 运行输出 module
```

if __name__=='__main__'就是判断__name__变量是不是等于__main__，当执行文件本身时候__name__变量等于main，此时判断成立并执行判断语句中的代码，当该模块被调用时的时候__name__为模块名，而"__main__"等于当前执行文件的名称（包含了后缀.py），不执行判断下面的预计

- **join()** 方法

用法1：用于将序列中的元素（字符串，不能为数字）以指定的字符连接生成一个新的字符串

```
# 语法格式str.join(sequence)
str = "-";
seq = ("a", "b", "c"); # 字符串序列
print str.join( seq );
# 输出 a-b-c
```

用法2：os.path.join()： 将多个路径组合后返回

```
 import os
os.path.join('/hello/','good/date','datbody')
hello/good/date/datbody
```

- **split()** 通过指定分隔符对字符串进行切片

语法格式str.split(str="", num=string.count(str)).

str -- 分隔符，默认为所有的空字符，包括空格、换行(\n)、制表符(\t)等。num -- 分割次数。默认为 -1， 即分隔所有

```
a='ahs+HKa+kAJ'
print(a.split('+'))
# 输出['ahs', 'HKa', 'kAJ']
```

```
#replace 替换字符串
print('abababababaaba'.replace('a','c'))
#输出cbcbcbcbccbc
```

判断是否符合特定条件

isupper islower isdigit istitle

```
print('abababababaaba'.isupper()) False
```

字典

字典为一系列键一值对，值可以为任何python对象，比如数字、字符串、列表、甚至字典等；通过字典名[键]获取值； 格式： 字典名 = { 键1：值1，键2：值2}

dict() 可通过其他映射创建字典

```
c=dict(a='1',b='2')
print(c)
#输出 {'a': '1', 'b': '2'}
```

```
love={
 'wenshuang':'cute',
 'lip':'red'
 }
print(love)
love['song']='great'
love['song']='well' #修改字典、=中的值
del love['song'] #删除字典中的值
love['song']='cute'
for k,v in love.items(): #遍历字典，方法items（）返回一个键-值对列表，for循环依此将每个键-值对存储到指定两个变量中
  print(k+' '+v) #返回顺序和存储顺序可能不同
for k in love.keys(): #遍历所有键，使用方法keys（），事实上可省略keys(),因为python默认遍历所有键 即 for k in love:
for k in love.values(): #遍历所有值，使用方法values（）,重复值也会显示
for k in set(love.values()): #set（），创建集合，集合类似列表，但每个元素独一无二，可剔除重复项
love={
  'wenshuang':'cute',
  'lip':['red','pink','like'] #嵌套，字典内可嵌套列表，同理列表可嵌套字典，字典可嵌套字典
 }

```

print(love.get('haha')) #访问不存在时不会引发异常，会显示None

print(love.get('haha','N/A')) #访问不存在时不会引发异常，会显示N/A

love.setdefaule('haha','N/A') #访问不存在时会添加指定的键值对 {'haha','N/A'}

"color is {lip}".format_map(love) #字典不能用format设置字符串格式，要用format_map

深拷贝deepcopy（）是将**对象本身复制**给另一个对象。这意味着如果对对象的副本进行更改时**不会影响**原对象。在 Python 中，我们使用 deepcopy（）**函数**进行深拷贝（位于模块 copy），

浅拷贝copy（）是将对象的**引用复制**给另一个对象。因此，如果我们在副本中进行更改，则会**影响原对象**。使用 copy（）**方法**进行浅拷贝

fromkeys（）函数：创建一个新字典，包含对应键

```
dict.fromkeys(['name','age']) # 创建一个新字典，包含对应键，值为None
{'name' : None , 'age' : None}
dict.fromkeys(['name','age'],'(haha)') # 创建一个新字典，包含对应键，所有值为haha
```

while

break:中断循环

```
a = 0
while a<10:
 a+=1
 if a==5:
 break
 print(a)
#1 2 3 4
```

continue:返回循环开头，进入下一循环

```
while a<10:
 a+=1
 if a%2 == 0:
 continue
 print(a)
#1 3 5 7 9
```

函数（尽量让每一个函数只实现**一个功能**）

- 函数定义内参数为形参，函数调用内参数为**实参**；

```
def index(request): #request 形参
 return render(request,'index.html')
```

- 传递实参：

1. 位置实参：基于实参顺序关联至形参；
2. 关键字实参：传递给函数名称-值对；例：

```
Pet(animal_name='dog',pet_name='harry')
```

3.默认值

```
def Pet(animal_name,pet_name='harry'):
 ...
Pet('cat') #默认pet_name='harry'
Pet(animal_name='dog',pet_name='hahaha') #覆盖默认值
```

- 以列表副本操作：切片表示法—— list_name[:] , 创建副本 （尽量避免使用，占内存）

```
a = [1,2,3,4]
b = a[:].pop(2)
print(a) #a = [1,2,3,4]
b = a.pop(2)
print(a) #a = [1,2,4]
```

- 传递**任意数量实参**, 用*形参名 表示，*会创建一个**空元组**

```
def pet(*name)
```

若结合使用位置形参和任意数量形参，任意数量形参必须放在最后

```
def pet(age,*name)
```

- 传递任意数量关键字实参，**形参名，**会创建一个**空字典**。

将函数存储在模块中：

- 导入模块

```
import module_name
module_name.function_name() #需使用句点指明模块
```

- 导入特定函数

```
from module_name import func1,func2
func1() #调用时可直接使用，无需使用句点，已显式导入函数
```

- 使用as 给函数或者模块指定别名

```
from module_name import pet as p
p()
import module_name as m
m.p()
```

- 导入所有函数(最后不要使用这种方法，大型项目可能会出现重名函数)

```
from module_name import *
```

多行注释 ''' '''

习惯：

1. 函数定义后面简要注释函数功能
2. 函数定义过长可 ‘（’ 后两次tab 排列整齐；

类 继承

实例化：根据类创建对象

约定：类的首字母大写

```
class Ws():
  def __init__(self, screen): #形参self不可缺少，必须放在首位，创建实例时，self会自动传递，__init__是用来封装实例化对象的属性，只要是实例化对象就一定会执行__init__方法，如果对象子类中没有则会寻找父类（超类），如果父类（超类）也没有，则直接继承object（python 3.x）类，执行类中的__init__方法。
  self.screen = screen
  self.image = pygame.image.load('images/wenshuang.bmp')
  self.rect = self.image.get_rect() #以self为前缀的变量可供类中所有方法使用，
  def update(self):
  if self.moving_right:
  self.rect.centerx += 5
  if self.moving_left:
   self.rect.centerx -= 5
   if self.moving_up:
   self.rect.centery -= 5
   if self.moving_down:
   self.rect.centery += 5
 class Zw(Ws): #类的继承，类Ws为父类，Zw为子类
   def __init__(self,screen):
   super().__init__(screen) #将父类和子类关联一起，让python调用父类方法_init_(),让子类实例拥有父类所有属性，父类又称为超类superclass
 a = Zw(b)
 print(a.update())
```

可将实例作为属性，可将类一部分提取出来作为一个单独的类

```
class Ws():
  def __init__(self, screen):
  self.screen = screen
  def update(self):
  print("好喜欢" + self.screen)
class Zw():
  def __init__(self,s):
  self.s=s
  self.ws=Ws('haha') #将将实例作为一个属性

 a = Zw()
 a.ws.update()
```

定义私有方法：

双下划线：名称前加双下划线，类外部不能访问_ _ update() ，但类内部可访问；类定义中，会将其转换为" _ classname _ _ update()" ,可通过 instance . _ classname _ _ update() 在类外部访问私有方法，但不应该这样做；

单下划线：

程序员使用名称前的单下划线，用于指定该名称属性为“私有”。这有点类似于惯例，为了使其他人（或你自己）使用这些代码时将会知道以“_”开头的名称只供内部使用。正如Python文档中所述：

6 以下划线“_”为前缀的名称（如_spam）应该被视为API中非公开的部分（不管是函数、方法还是数据成员）。此时，应该将它们看作是一种实现细节，在修改它们时无需对外部通知。它对解释器来说确实有一定的意义，如果你写了代码“from <模块/包名> import *”，那么以“_”开头的名称都不会被导入，除非模块或包中的“__all__”列表显式地包含了它们。

```
class Person:
 print('ws ')
class Boy(Person):
 print("zw")
m=Boy()
print(issubclass(Boy,Person)) #判断是否为子类
print(Boy.__bases__) #想知道其父类
print(isinstance(m,Boy)) #想知道m是否为类Boy的实例
print(isinstance(m,Person)) #想知道m是否为类Person的实例
 print(m.__class__) #想知道m对象属于哪个类
 >>
 ws
 zw
 True
 (<class '__main__.Person'>,)
 True
 True
 <class '__main__.Boy'>
```

hasattr(instance，'方法name') 检查实例中是否有该方法；

多重继承：class A（classname B , classname C）,应避免使用，若多个父类以不同方式实现同一个方法，即有同名方法，要注意父类顺序B C，前面会覆盖后面；

_ _dict_ _

类的静态函数、类函数、普通函数、全局变量以及一些内置的属性都是放在类__dict__里的,以字典

形式存储；

抽象类（包含抽象方法的类）：最重要的特征是不能被实例化；
Python本身不提供抽象类和接口机制，要想实现抽象类，可以借助abc模块。

待定

生成器

在Python中，一边循环一边计算的机制，称为生成器：generator。
方法一，只要把一个列表生成式的[]改成()，就创建了一个generator：
方法二， 如果一个函数中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator。调用函数就是创建了一个生成器（generator）对象。yield相当于 return 返回一个值，并且记住这个返回的位置，下次迭代时，代码从yield的下一条语句开始执行。

```
1 #encoding:UTF-8
2 def yield_test(n):
3 for i in range(n):
4 yield call(i)
5 print("i=",i)
6 print("Done.")
7
8 def call(i):
9 return i*2
10
11 for i in yield_test(5):
12 print(i,",")
13 结果
14 >>>
15 0 ,
16 i= 0
17 2 ,
18 i= 1
19 4 ,
20 i= 2
21 6 ,
22 i= 3
```

```
8 ,
i= 4
Done.
>>>
```

迭代器

迭代器是访问集合元素的一种方式。迭代器对象从集合的第一个元素开始访问，迭代器只能往前不会后退。
迭代器不要求你事先准备好整个迭代过程中所有的元素。仅仅是在迭代至某个元素时才计算该元素，而在这之前或之后，元素可以不存在或者被销毁。这个特点使得它特别适合用于遍历一些巨大的或是无限的集合。

任何实现了__iter__和__next__()方法的对象都是迭代器，__iter__返回迭代器自身，__next__返回容器中的下一个值，如果容器中没有更多元素了，则抛出StopIteration异常;
凡是可作用于for循环的对象都是可迭代类型;
凡是可作用于next()函数的对象都是迭代器，它们表示一个惰性计算的序列;
集合数据类型如list、dict、str等是Iterable但不是Iterator，不过可以通过iter()函数获得一个迭代器。iter()函数会返回一个定义了next()方法的迭代器对象;
Python的for循环本质上就是通过不断调用next()函数实现的，

```
1 使用对象内置的__iter__()方法生成迭代器
2 >>>L1 = [1,2,3,4,5,6]
3 >>>I1 = L1.__iter__() #使用对象内置的__iter__()方法生成迭代器
#也可通过iter()函数生成迭代器 I2 = iter(L1)
4 >>>print I1
5 <listiterator object at 0x7fe4fd0ef550>
6 >>> I1.next()
7 1
8 >>> I1.next()
 9 2
10 >>> I1.next()
11 3
```

区别：生成器能做到迭代器能做的所有事,而且因为自动创建了__iter__()和next()方法,生成器显得特别简洁,而且生成器也是高效的，使用生成器表达式取代列表解析可以同时节省内存

文件和异常：

```
with open('p.txt') as file_object:
 contents = file_object.read() #打开并读取文件
 print(content.rstrip()) #删除末尾空行
```

```
with open('files\p.txt') as file_object: #相对文件路径打开文件
 for line in file_object: #逐行读取
 print(line)
```

```
with open('files\p.txt') as file_object:
 line= file_object.readlines()#创建包含各行的列表
 for line in lines:
 print(line)
```

Python3 中有六个标准的数据类型：

- Number（数字）
- String（字符串）
- List（列表）
- Tuple（元组）
- Set（集合）
- Dictionary（字典）

装饰器：

python装饰器本质上就是一个函数，它可以让其他函数在不需要做任何代码变动的前提下增加额外的功能，装饰器的返回值也是一个函数对象（函数的指针）。

作用和功能：

1.可以在外层函数加上时间计算函数，计算函数运行时间；

2.计算函数运行次数；

3.可以用在框架的路由传参上；

4.插入日志，作为函数的运行日志；

5.事务处理，可以让函数实现事务的一致性，让函数要么一起运行成功，要么一起运行失败；

6.缓存，实现缓存处理；

7.权限的校验，在函数外层套上权限校验的代码，实现权限校验；

```
def decorator(func):
 @wraps（func） #functool.wraps可把原函数的原信息拷贝到装饰器里的func函数中
 def wrapper(*args, **kwargs):
 start_time = time.time()
 func()
 end_time = time.time()
 print(end_time - start_time)

 return wrapper #重要，必须写上
```

```
10
11 @decorator #相当于 func = decorator(func)
12 def func():
13  time.sleep(0.8)
```

lambda 函数：匿名函数

lambda的语法是唯一的。其形式如下： lambda argument_list: expression

lambda函数有**如下特性**：

1.lambda函数是匿名的：所谓匿名函数，通俗地说就是没有名字的函数。lambda函数没有名字。

2.lambda函数有输入和输出：输入是传入到参数列表argument_list的值，输出是根据表达式expression计算得到的值。

3.lambda函数一般功能简单：单行expression决定了lambda函数不可能完成复杂的逻辑，只能完成非常简单的功能。由于其实现的功能一目了然，甚至不需要专门的名字来说明。

作用：

1.将lambda函数赋值给一个变量，通过这个变量间接调用该lambda函数。

例如，执行语句add=lambda x, y: x+y，定义了加法函数lambda x, y: x+y，并将其赋值给变量add，这样变量add便成为具有加法功能的函数。例如，执行add(1,2)，输出为3。

filter函数。此时lambda函数用于指定过滤列表元素的条件。例如filter(lambda x: x % 3 == 0, [1, 2, 3])指定将列表[1,2,3]中能够被3整除的元素过滤出来，其结果是[3]。

2.sorted函数。此时lambda函数用于指定对列表中所有元素进行排序的准则。例如sorted([1, 2, 3, 4, 5, 6, 7, 8, 9], key=lambda x: abs(5-x))将列表[1, 2, 3, 4, 5, 6, 7, 8, 9]按照元素与5距离从小到大进行排序，其结果是[5, 4, 6, 3, 7, 2, 8, 1, 9]。

3.map函数。此时lambda函数用于指定对列表中每一个元素的共同操作。例如map(lambda x: x+1, [1, 2,3])将列表[1, 2, 3]中的元素分别加1，其结果[2, 3, 4]。

4.reduce函数。此时lambda函数用于指定列表中两两相邻元素的结合条件。例如reduce(lambda a, b: '{}, {}'.format(a, b), [1, 2, 3, 4, 5, 6, 7, 8, 9])将列表 [1, 2, 3, 4, 5, 6, 7, 8, 9]中的元素从左往右两两以逗号分隔的字符的形式依次结合起来，其结果是'1, 2, 3, 4, 5, 6, 7, 8, 9'。

延迟绑定

运行嵌套函数时才会引用外部变量，不运行时不会去找i的值，

```
1 def a（）：
2  return [lamabda x : i*x for i in range（4） print]
```

```
print [m（2） for m in a（）]

# [6,6,6,6]
```

断言方法

```
assert len(lists) >=5,'列表元素个数小于5'
```

```
assertEqual(a,b，[msg='测试失败时打印的信息'])： 断言a和b是否相等，相等则测试用例通过。

assertNotEqual(a,b，[msg='测试失败时打印的信息'])： 断言a和b是否相等，不相等则测试用例通过。

assertTrue(x，[msg='测试失败时打印的信息'])： 断言x是否True，是True则测试用例通过。

assertFalse(x，[msg='测试失败时打印的信息'])： 断言x是否False，是False则测试用例通过。

assertIs(a,b，[msg='测试失败时打印的信息'])： 断言a是否是b，是则测试用例通过。

assertNotIs(a,b，[msg='测试失败时打印的信息'])： 断言a是否是b，不是则测试用例通过。

assertIsNone(x，[msg='测试失败时打印的信息'])： 断言x是否None，是None则测试用例通过。

assertIsNotNone(x，[msg='测试失败时打印的信息'])： 断言x是否None，不是None则测试用例通过。

assertIn(a,b，[msg='测试失败时打印的信息'])： 断言a是否在b中，在b中则测试用例通过。

assertNotIn(a,b，[msg='测试失败时打印的信息'])： 断言a是否在b中，不在b中则测试用例通过。

assertIsInstance(a,b，[msg='测试失败时打印的信息'])： 断言a是是b的一个实例，是则测试用例通过。

assertNotIsInstance(a,b，[msg='测试失败时打印的信息'])： 断言a是是b的一个实例，不是则测试用例通过。
```

当程序出现错误，python会自动引发异常，也可以通过raise显示地引发异常。一旦执行了raise语句，raise后面的语句将不能执行。

try—— except——else: 处理异常

try代码块运行错误时，python将运行except内代码,try正常运行时，将运行else代码块代码，finally语句保证无论try子句发生什么异常，都会执行finally子句;

```
try:
 s = None
 if s is None:
 print "s 是空对象"
 raise NameError #如果引发NameError异常，后面的代码将不能执行
 print len(s) #这句不会执行，但是后面的except还是会走到
```

```
except TypeError:
 print "空对象没有长度"
```

内置异常类

BaseException # 所有异常的基类
+-- SystemExit # 解释器请求退出
+-- KeyboardInterrupt # 用户中断执行(通常是输入^C)
+-- GeneratorExit # 生成器(generator)发生异常来通知退出
+-- Exception # 常规异常的基类
    +-- StopIteration # 迭代器没有更多的值
    +-- StopAsyncIteration # 必须通过异步迭代器对象的__anext__()方法引发以停止迭代
    +-- ArithmeticError # 各种算术错误引发的内置异常的基类
    | +-- FloatingPointError # 浮点计算错误
    | +-- OverflowError # 数值运算结果太大无法表示
    | +-- ZeroDivisionError # 除(或取模)零 (所有数据类型)
    +-- AssertionError # 当assert语句失败时引发
    +-- AttributeError # 属性引用或赋值失败
    +-- BufferError # 无法执行与缓冲区相关的操作时引发
    +-- EOFError # 当input()函数在没有读取任何数据的情况下达到文件结束条件(EOF)时引发
    +-- ImportError # 导入模块/对象失败
    | +-- ModuleNotFoundError # 无法找到模块或在在sys.modules中找到None
    +-- LookupError # 映射或序列上使用的键或索引无效时引发的异常的基类
    | +-- IndexError # 序列中没有此索引(index)
    | +-- KeyError # 映射中没有这个键
    +-- MemoryError # 内存溢出错误(对于Python 解释器不是致命的)
    +-- NameError # 未声明/初始化对象 (没有属性)
    | +-- UnboundLocalError # 访问未初始化的本地变量
    +-- OSError # 操作系统错误，EnvironmentError，IOError，WindowsError，socket.error，select.error和mmap.error已合并到OSError中，构造函数可能返回子类
    | +-- BlockingIOError # 操作将阻塞对象(e.g. socket)设置为非阻塞操作
    | +-- ChildProcessError # 在子进程上的操作失败
    | +-- ConnectionError # 与连接相关的异常的基类
    | | +-- BrokenPipeError # 另一端关闭时尝试写入管道或试图在已关闭写入的套接字上写入

```
|     |     +-- ConnectionAbortedError   # 连接尝试被对等方中止
|     |     +-- ConnectionRefusedError   # 连接尝试被对等方拒绝
|     |     +-- ConnectionResetError     # 连接由对等方重置
|     +-- FileExistsError   # 创建已存在的文件或目录
|     +-- FileNotFoundError   # 请求不存在的文件或目录
|     +-- InterruptedError   # 系统调用被输入信号中断
|     +-- IsADirectoryError   # 在目录上请求文件操作(例如 os.remove())
|     +-- NotADirectoryError   # 在不是目录的事物上请求目录操作(例如 os.listdir())
|     +-- PermissionError   # 尝试在没有足够访问权限的情况下运行操作
|     +-- ProcessLookupError   # 给定进程不存在
|     +-- TimeoutError   # 系统函数在系统级别超时
+-- ReferenceError   # weakref.proxy()函数创建的弱引用试图访问已经垃圾回收了的对象
+-- RuntimeError   # 在检测到不属于任何其他类别的错误时触发
|     +-- NotImplementedError   # 在用户定义的基类中，抽象方法要求派生类重写该方法或者正在开发的类指示仍然需要添加实际实现
|     +-- RecursionError   # 解释器检测到超出最大递归深度
+-- SyntaxError   # Python 语法错误
|     +-- IndentationError   # 缩进错误
|           +-- TabError   # Tab和空格混用
+-- SystemError   # 解释器发现内部错误
+-- TypeError   # 操作或函数应用于不适当类型的对象
+-- ValueError   # 操作或函数接收到具有正确类型但值不合适的参数
|     +-- UnicodeError   # 发生与Unicode相关的编码或解码错误
|           +-- UnicodeDecodeError   # Unicode解码错误
|           +-- UnicodeEncodeError   # Unicode编码错误
|           +-- UnicodeTranslateError   # Unicode转码错误
+-- Warning   # 警告的基类
      +-- DeprecationWarning   # 有关已弃用功能的警告的基类
      +-- PendingDeprecationWarning   # 有关不推荐使用功能的警告的基类
      +-- RuntimeWarning   # 有关可疑的运行时行为的警告的基类
      +-- SyntaxWarning   # 关于可疑语法警告的基类
      +-- UserWarning   # 用户代码生成警告的基类
      +-- FutureWarning   # 有关已弃用功能的警告的基类
```

+-- ImportWarning # 关于模块导入时可能出错的警告的基类

+-- UnicodeWarning # 与Unicode相关的警告的基类

+-- BytesWarning # 与bytes和bytearray相关的警告的基类

+-- ResourceWarning # 与资源使用相关的警告的基类。被默认警告过滤器忽略。

Python2和Python3中新式类、经典类(旧式类)的区别

他们最明显的区别在于继承搜索的顺序发生了改变，即经典类多继承搜索顺序(深度优先)：先深入继承树左侧查找，然后再返回，开始查找右侧，新式类多继承搜索顺序(广度优先)：先在水平方向查找，然后再向上查找

Python -m django --version

```
-m 将库中的python模块用作脚本去运行，将当前路径放入sys.path中，可避免引用问题；不加m为直接启动，将要运行文件的目录放入sys.path路径中
python -m django --version # 查看django 版本
```

instance 和 type函数

isinstance（A,B）判断实例对象A的类型是否和B相同,B可以为直接或间接类名、基本类型、或者他们组成的元组

```
a=2
isinstance（a,int）
输出 True
```

type（A）==B

区别：type（）不会认为子类是一种父类类型，不考虑继承关系，isinstance（）会认为子类是一种父类类型，考虑继承关系

__repr__（），定义输出对象格式

deepcopy和copy

deepcopy:创建完全独立复制对象，原对象为不可变对象时，id相同；为可变对象时，分配新id，id不同

copy:

1.浅复制对象为不可变对象时，id相同，原对象改变不会影响新对象

2.浅复制对象为可变对象，有复杂子对象（列表中有列表），且改变原对象复杂子对象时，新对象会受影响，其他情况（原对象无复杂子对象过改变原对象非复杂子对象）新对象不会受影响，id不同

=赋值：不会产生独立的对象存在，只是在原有数据块打上一个新标签，当一个标签被改变时，数据块发生变化，另一个标签所指内容会随之改变

JSON

是存储和交换文本信息的语法,数据在名称/值对中; 数据由逗号分隔;

```
import json
a='{"name":"wenshuang","lover":"zhenwei"}'
print(type(a))
b=json.loads(a) #JSON转化为字典
print(type(b))
c=json.dumps(b) #字典转化为JSON
print(type(c))
输出:
#<class 'str'>
 #<class 'dict'>
 #<class 'str'>


 employees ={
"employees": [ # 方括号保存数组
{ "firstName":"John" , "lastName":"Doe" }, # 花括号保存对象;
{ "firstName":"Anna" , "lastName":"Smith" },
{ "firstName":"Peter" , "lastName":"Jones" }
]
}
```