一、生成器交互

def  inc():

base  =  1

while  True:

response  =  yield  base

print(response,  '~~~~~~~')

if  response:

base  =  response

base  +=  1

print('base  =  {}'.format(base))

g  =  inc()

next(g)

next(g)

#  交互

g.send(-1)  #  相当于next同时传入-1

生成器提供了一个send方法，该方法可以和生成器方向沟通。

调用send方法，就可以把send的实参传给yield语句做结果，这个结果可以在等式右边被赋值给其它变量。

send和next一样可以推动生成器启动并执行。

二、\_\_slots\_\_

问题的引出

都是字典惹的祸。

字典为了提升查询效率，必须用空间换时间。

一般来说一个实例，属性多一点，都存储在字典中便于查询，问题不大。

但是如果数百万个实例，那么字典占的总空间就有点大了。

这个时候，能不能把属性字典  \_\_dict\_\_  省了？

Python提供了  \_\_slots\_\_

class  A:

X  =  1

def  \_\_init\_\_(self):

self.y  =  5

self.z  =  6

def  show(self):

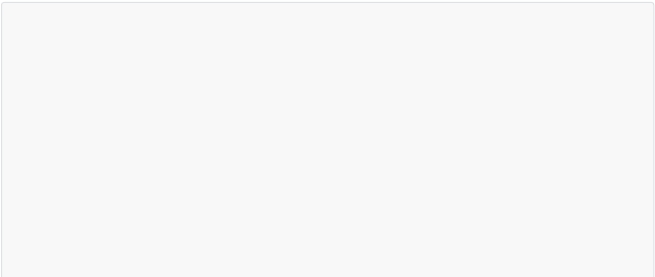
print(self.X,  self.y,  self.z)

a  =  A()

print(A.\_\_dict\_\_)  #  ？

print(a.\_\_dict\_\_)  #  ？

思考



上面2个字典，谁的字典是个问题？

实例多达百万个的时候，这么多存放实例属性的字典是个问题

class  A:

X  =  1

\_\_slots\_\_  =  ('y','z')  #  元组

#  \_\_slots\_\_  =  ['y','z']  #  可以吗

#  \_\_slots\_\_  =  'y','z'  #  可以吗

#  \_\_slots\_\_  =  'y'

def  \_\_init\_\_(self):

self.y  =  5

#self.z  =  6

def  show(self):

print(self.X,  self.y)

a  =  A()

a.show()

#

print('A',  A.\_\_dict\_\_)

#print('obj',  a.\_\_dict\_\_)

print(a.\_\_slots\_\_)

\_\_slots\_\_   告诉解释器，实例的属性都叫什么，一般来说，既然要节约内存，最好还是使用元组比较好。

一旦类提供了  \_\_slots\_\_  ，就阻止实例产生  \_\_dict\_\_  来保存实例的属性。

尝试为实例a动态增加属性

a.newx = 5

返回AttributeError: 'A' object has no attribute 'newx'

说明实例不可以动态增加属性了

A.NEWX = 20，这是可以的，因为这个类属性。

继承

class  A:

X  =  1

\_\_slots\_\_  =  ('y','z')  #  元组

def  \_\_init\_\_(self):

self.y  =  5

#self.z  =  6

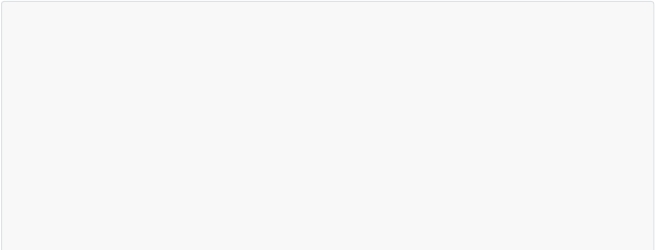
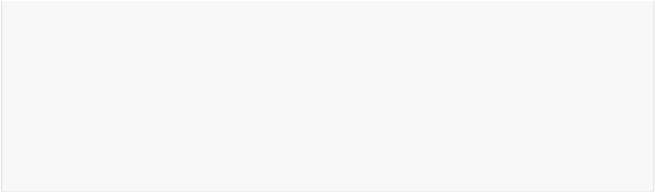
def  show(self):

print(self.X,  self.y)

a  =  A()

a.show()

#



print('A',  A.\_\_dict\_\_)

#print('obj',  a.\_\_dict\_\_)

print(a.\_\_slots\_\_)

class  B(A):

pass

print('B',  B().\_\_dict\_\_)

\_\_slots\_\_  不影响子类实例，不会继承下去，除非子类里面自己也定义了  \_\_slots\_\_  。

应用场景

使用需要构建在数百万以上众多对象，且内存容量较为紧张，实例的属性简单、固定且不用动态增加的场景。

三、未实现和未实现异常

print(type(NotImplemented))

print(type(NotImplementedError))

#  <class  'NotImplementedType'>

#  <class  'type'>

#raise  NotImplemented

raise  NotImplementedError

NotImplemented是个值，单值，是NotImplementedType的实例

NotImplementedError 是类型，是异常类，返回type

四、运算符重载中的反向方法

前面学习过运算符重载的方法，例如\_\_add\_\_和\_\_iadd\_\_

class  A:

def  \_\_init\_\_(self,  x):

self.x  =  x

def  \_\_add\_\_(self,  other):

print(self,  'add')

return  self.x  +  other.x

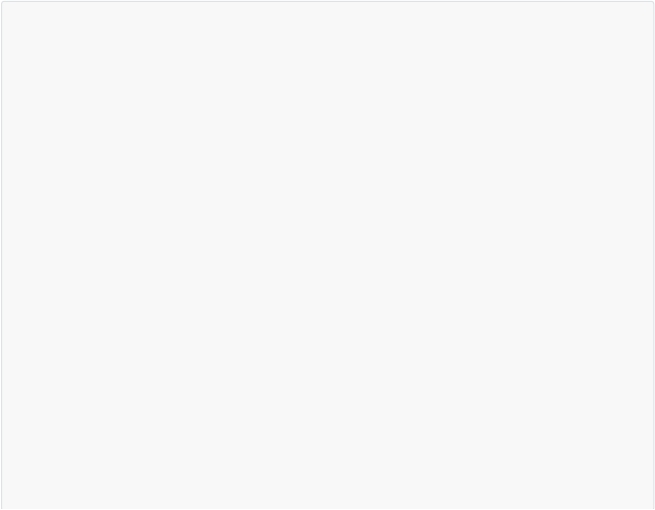
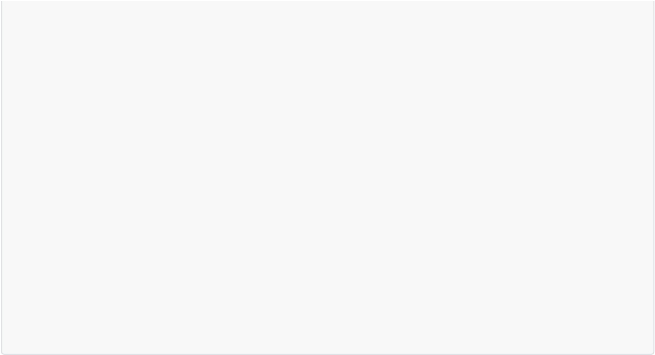
def  \_\_iadd\_\_(self,  other):

print(self,  'iadd')

return  A(self.x  +  other.x)

def  \_\_radd\_\_(self,  other):

print(self,  'radd')



return  self.x  +  other.x

a  =  A(4)

b  =  A(5)

print(a,  b)

print(a  +  b)

print(b  +  a)

b  +=  a

a  +=  b

#  运行结果

<\_\_main\_\_.A  object  at  0x0000000000B8C550>  <\_\_main\_\_.A  object  at  0x0000000000B8C240>

<\_\_main\_\_.A  object  at  0x0000000000B8C550>  add

9

<\_\_main\_\_.A  object  at  0x0000000000B8C240>  add

9

<\_\_main\_\_.A  object  at  0x0000000000B8C240>  iadd

<\_\_main\_\_.A  object  at  0x0000000000B8C550>  iadd

\_\_radd\_\_  方法根本没有执行过，为什么？

因为都是A的实例，都是调用的  \_\_add\_\_  ，无非就是实例a还是b调用而已。

测试一下 a + 1

class  A:

def  \_\_init\_\_(self,  x):

self.x  =  x

def  \_\_add\_\_(self,  other):

print(self,  'add')

return  self.x  +  other.x

def  \_\_iadd\_\_(self,  other):

print(self,  'iadd')

return  A(self.x  +  other.x)

def  \_\_radd\_\_(self,  other):

print(self,  'radd')

return  self.x  +  other.x

a  =  A(4)

a  +  1

#  运行结果

<\_\_main\_\_.A  object  at  0x0000000000B7DA58>  add

Traceback  (most  recent  call  last):

File  "test.py",  line  22,  in  <module>

a  +  1

File  "test.py",  line  10,  in  \_\_add\_\_

return  self.x  +  other.x

AttributeError:  'int'  object  has  no  attribute  'x'

出现了AttributeError，因为1是int类型，没有x这个属性，还是  \_\_add\_\_  被执行了。



测试1 + a，运行结果如下

<\_\_main\_\_.A  object  at  0x0000000000DADA58>  radd

Traceback  (most  recent  call  last):

File  "test.py",  line  22,  in  <module>

1  +  a

File  "test.py",  line  18,  in  \_\_radd\_\_

return  self.x  +  other.x

AttributeError:  'int'  object  has  no  attribute  'x'

这次执行的是实例a的  \_\_radd\_\_  方法。

1 + a 等价于   1.\_\_add\_\_(a)  ，也就是  int.\_\_add\_\_(1,  a)  ，而int类型实现了  \_\_add\_\_  方法的，为什么却不抛出异

常，而是执行了实例a的  \_\_radd\_\_  方法？

再看一个例子

class  A:

def  \_\_init\_\_(self,  x):

self.x  =  x

def  \_\_add\_\_(self,  other):

print(self,  'add')

return  self.x  +  other.x

def  \_\_iadd\_\_(self,  other):

print(self,  'iadd')

return  A(self.x  +  other.x)

def  \_\_radd\_\_(self,  other):

print(self,  'radd')

return  self.x  +  other.x

class  B:  #  未实现\_\_add\_\_

def  \_\_init\_\_(self,  x):

self.x  =  x

a  =  A(4)

b  =  B(10)

print(a  +  b)

print(b  +  a)

#  运行结果

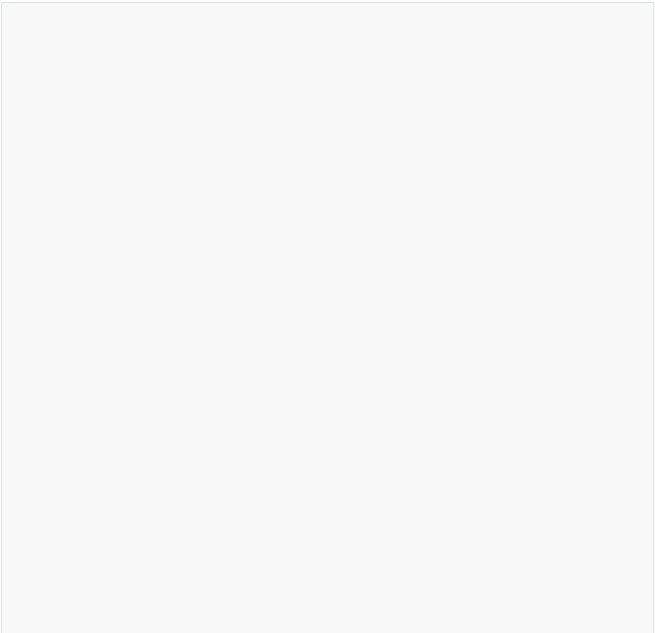
<\_\_main\_\_.A  object  at  0x0000000000B6C240>  add

14

<\_\_main\_\_.A  object  at  0x0000000000B6C240>  radd

14

b + a 等价于   b.\_\_add\_\_(a)  ，但是类B没有实现  \_\_add\_\_  方法，就去找a的  \_\_radd\_\_  方法



1 + a 等价于   1.\_\_add\_\_(a)  ，而int类型实现了  \_\_add\_\_  方法的，不过这个方法对于这种加法的返回值是

NotImplemented ，解释器发现是这个值，就会发起对第二操作对象的  \_\_radd\_\_  方法的调用。

B类也等价于下面的实现

class  B:

def  \_\_init\_\_(self,  x):

self.x  =  x

def  \_\_add\_\_(self,  other):

if  isinstance(other,  type(self)):

return  self.x  +  other.x

else:

return  NotImplemented

1 + a能解决吗？

class  A:

def  \_\_init\_\_(self,  x):

self.x  =  x

def  \_\_add\_\_(self,  other):

print(self,  'add')

if  hasattr(other,  'x'):

return  self.x  +  other.x

else:

try:

x  =  int(other)

except:

x  =  0

return  self.x  +  x

def  \_\_iadd\_\_(self,  other):

print(self,  'iadd')

return  A(self.x  +  other.x)

def  \_\_radd\_\_(self,  other):

print(self,  'radd')

return  self  +  other

class  B:

def  \_\_init\_\_(self,  x):

self.x  =  x

a  =  A(4)

b  =  B(10)

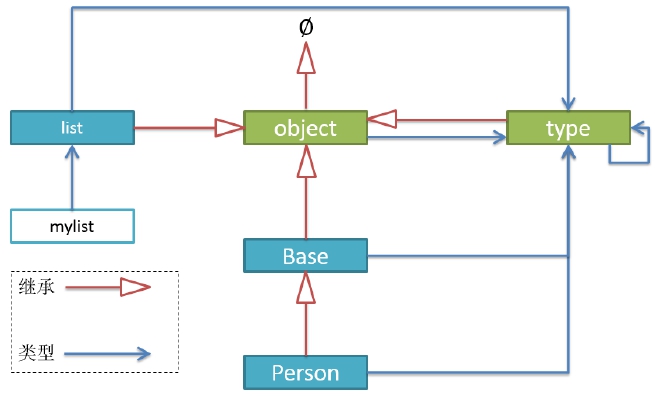
print(a  +  b)

print(b  +  a)

print(a  +  2)

print(2  +  a)

print(a  +  'abc')



print('abc'  +  a)

'abc' + a，字符串也实现了  \_\_add\_\_  方法，不过默认是处理不了和其他类型的加法，就返回NotImplemented。

五、Python的对象模型

在Python中，任何对象都有类型，可以使用type()或者  \_\_class\_\_  查看。

但是类型也是对象即类对象，它也有自己的类型

所有新类型的缺省类型是type（可以使用元类来改变）

内建类型例如list和用户自定义类

特殊类型type是所有对象的缺省类型，也包括type自己。但它又是一个对象，因此从object继承

特殊类型object是继承树的顶层，它是python所有类型的最终基类

也就是说，继承都来自object，类型都看type。type也是对象继承自object，object也有类型是type。

这俩又特殊，type类型是它自己，object没有基类。