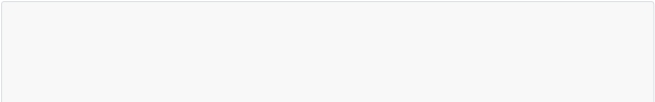
特殊属性

|  |  |
| --- | --- |
| 属性 | 含义 |
| \_\_name\_\_ | 类、函数、方法等的名字 |
| \_\_module\_\_ | 类定义所在的模块名 |
| \_\_class\_\_ | 对象或类所属的类 |
| \_\_bases\_\_ | 类的基类的元组，顺序为它们在基类列表中出现的顺序 |
| \_\_doc\_\_ | 类、函数的文档字符串，如果没有定义则为None |
| \_\_mro\_\_ | 类的mro，class.mro()返回的结果的保存在  \_\_mro\_\_  中 |
| \_\_dict\_\_ | 类或实例的属性，可写的字典 |

|  |  |
| --- | --- |
| 方法 | 意义 |
| \_\_dir\_\_ | 返回类或者对象的所有成员名称列表。  dir()函数操作实例就是调用  \_\_dir\_\_  ()。 |



查看属性

如果dir([obj])参数obj包含方法  \_\_dir\_\_()  ，该方法将被调用。如果参数obj不包含  \_\_dir\_\_()  ，该方法将最大限

度地收集属性信息。

dir(obj)对于不同类型的对象obj具有不同的行为：

如果对象是模块对象，返回的列表包含模块的属性名和变量名

如果对象是类型或者说是类对象，返回的列表包含类的属性名，及它的祖先类的属性名

如果是类的实例

有  \_\_dir\_\_  方法，返回可迭代对象的返回值

没有  \_\_dir\_\_  方法，则尽可能收集实例的属性名、类的属性和祖先类的属性名

如果obj不写，返回列表包含内容不同

在模块中，返回模块的属性和变量名

在函数中，返回本地作用域的变量名

在方法中，返回本地作用域的变量名

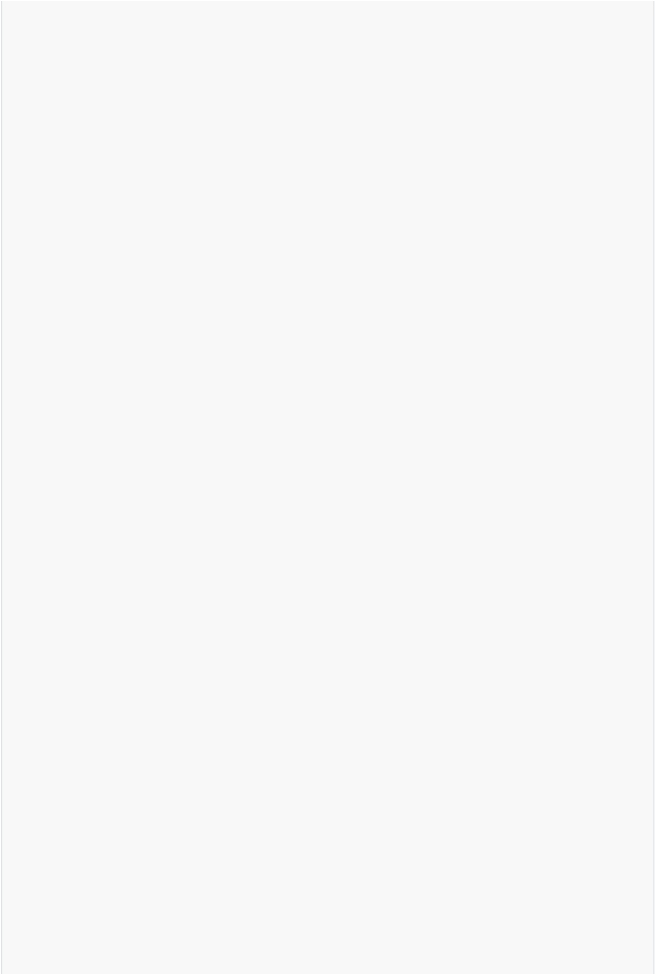
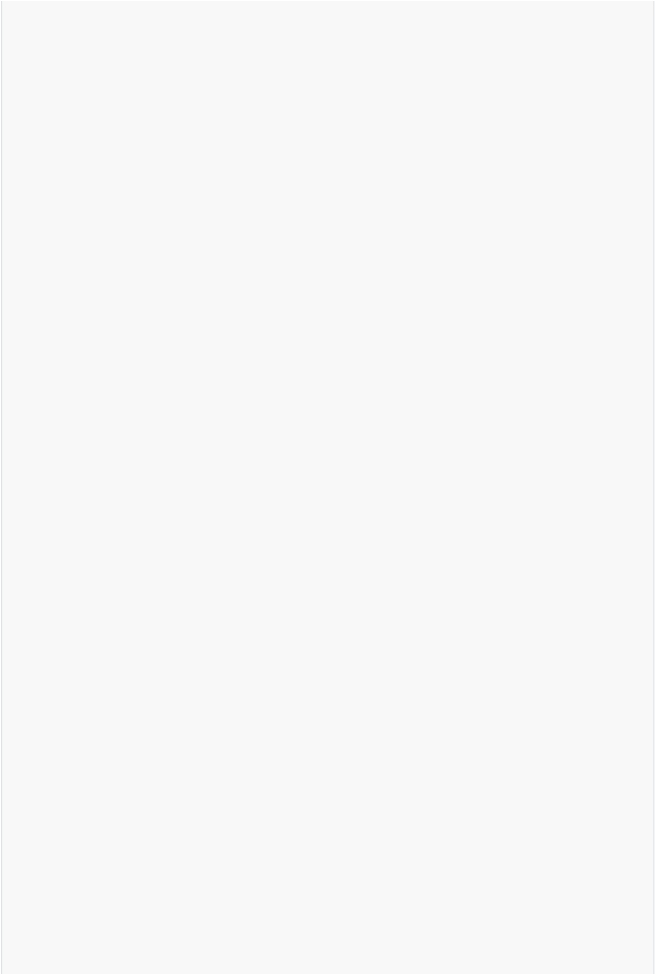
#  animal.py

class  Animal:

x  =  123

def  \_\_init\_\_(self,  name):

self.\_name  =  name



self.\_\_age  =  10

self.weight  =  20

print('animal  Module\'s  names   =  {}'.format(dir()))  #  模块的属性

#  cat.py

import  animal

from  animal  import  Animal

class  Cat(Animal):

x  =  'cat'

y  =  'abcd'

class  Dog(Animal):

def  \_\_dir\_\_(self):

return  ['dog']  #  必须返回可迭代对象

print('---------')

print('Current  Module\'s  names  =  {}'.format(dir()))  #  模块名词空间内的属性

print('animal  Module\'s  names   =  {}'.format(dir(animal)))  #  指定模块名词空间内的属性

print("object's  \_\_dict\_\_ =  {}".format(sorted(object.\_\_dict\_\_.keys())))  #  object的字典

print("Animal's  dir()  =  {}".format(dir(Animal)))  #  类Animal的dir()

print("Cat's  dir() =  {}".format(dir(Cat)))  #  类Cat的dir()

print('~~~~~~~~~')

tom  =  Cat('tom')

print(sorted(dir(tom)))  #  实例tom的属性、Cat类及所有祖先类的类属性

print(sorted(tom.\_\_dir\_\_()))  #  同上

#  dir()的等价  近似如下，\_\_dict\_\_字典中几乎包括了所有属性

print(sorted(set(tom.\_\_dict\_\_.keys())  |  set(Cat.\_\_dict\_\_.keys())  |  set(object.\_\_dict\_\_.keys())))

print("Dog's  dir  =  {}".format(dir(Dog)))

dog  =  Dog('snoppy')

print(dir(dog))

print(dog.\_\_dict\_\_)

#  执行结果

animal  Module's  names   =  ['Animal',  '\_\_builtins\_\_',  '\_\_cached\_\_',  '\_\_doc\_\_',  '\_\_file\_\_',

'\_\_loader\_\_',  '\_\_name\_\_',  '\_\_package\_\_',  '\_\_spec\_\_']

---------

Current  Module's  names  =  ['Animal',  'Cat',  'Dog',  '\_\_builtins\_\_',  '\_\_cached\_\_',  '\_\_doc\_\_',

'\_\_file\_\_',  '\_\_loader\_\_',  '\_\_name\_\_',  '\_\_package\_\_',  '\_\_spec\_\_',  'animal']

animal  Module's  names   =  ['Animal',  '\_\_builtins\_\_',  '\_\_cached\_\_',  '\_\_doc\_\_',  '\_\_file\_\_',

'\_\_loader\_\_',  '\_\_name\_\_',  '\_\_package\_\_',  '\_\_spec\_\_']

object's  \_\_dict\_\_ =  ['\_\_class\_\_',  '\_\_delattr\_\_',  '\_\_dir\_\_',  '\_\_doc\_\_',  '\_\_eq\_\_',

'\_\_format\_\_',  '\_\_ge\_\_',  '\_\_getattribute\_\_',  '\_\_gt\_\_',  '\_\_hash\_\_',  '\_\_init\_\_',  '\_\_le\_\_',

'\_\_lt\_\_',  '\_\_ne\_\_',  '\_\_new\_\_',  '\_\_reduce\_\_',  '\_\_reduce\_ex\_\_',  '\_\_repr\_\_',  '\_\_setattr\_\_',

'\_\_sizeof\_\_',  '\_\_str\_\_',  '\_\_subclasshook\_\_']

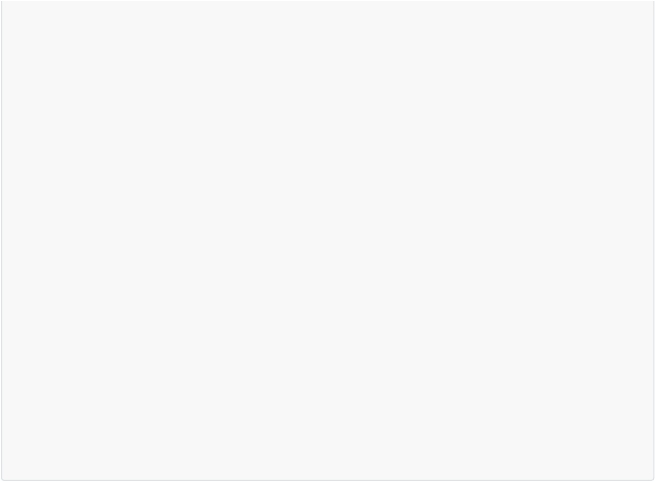
Animal's  dir()  =  ['\_\_class\_\_',  '\_\_delattr\_\_',  '\_\_dict\_\_',  '\_\_dir\_\_',  '\_\_doc\_\_',  '\_\_eq\_\_',

'\_\_format\_\_',  '\_\_ge\_\_',  '\_\_getattribute\_\_',  '\_\_gt\_\_',  '\_\_hash\_\_',  '\_\_init\_\_',  '\_\_le\_\_',

'\_\_lt\_\_',  '\_\_module\_\_',  '\_\_ne\_\_',  '\_\_new\_\_',  '\_\_reduce\_\_',  '\_\_reduce\_ex\_\_',  '\_\_repr\_\_',

'\_\_setattr\_\_',  '\_\_sizeof\_\_',  '\_\_str\_\_',  '\_\_subclasshook\_\_',  '\_\_weakref\_\_',  'x']

Cat's  dir() =  ['\_\_class\_\_',  '\_\_delattr\_\_',  '\_\_dict\_\_',  '\_\_dir\_\_',  '\_\_doc\_\_',  '\_\_eq\_\_',



'\_\_format\_\_',  '\_\_ge\_\_',  '\_\_getattribute\_\_',  '\_\_gt\_\_',  '\_\_hash\_\_',  '\_\_init\_\_',  '\_\_le\_\_',

'\_\_lt\_\_',  '\_\_module\_\_',  '\_\_ne\_\_',  '\_\_new\_\_',  '\_\_reduce\_\_',  '\_\_reduce\_ex\_\_',  '\_\_repr\_\_',

'\_\_setattr\_\_',  '\_\_sizeof\_\_',  '\_\_str\_\_',  '\_\_subclasshook\_\_',  '\_\_weakref\_\_',  'x',  'y']

~~~~~~~~~

['\_Animal\_\_age',  '\_\_class\_\_',  '\_\_delattr\_\_',  '\_\_dict\_\_',  '\_\_dir\_\_',  '\_\_doc\_\_',  '\_\_eq\_\_',

'\_\_format\_\_',  '\_\_ge\_\_',  '\_\_getattribute\_\_',  '\_\_gt\_\_',  '\_\_hash\_\_',  '\_\_init\_\_',  '\_\_le\_\_',

'\_\_lt\_\_',  '\_\_module\_\_',  '\_\_ne\_\_',  '\_\_new\_\_',  '\_\_reduce\_\_',  '\_\_reduce\_ex\_\_',  '\_\_repr\_\_',

'\_\_setattr\_\_',  '\_\_sizeof\_\_',  '\_\_str\_\_',  '\_\_subclasshook\_\_',  '\_\_weakref\_\_',  '\_name',  'weight',

'x',  'y']

['\_Animal\_\_age',  '\_\_class\_\_',  '\_\_delattr\_\_',  '\_\_dict\_\_',  '\_\_dir\_\_',  '\_\_doc\_\_',  '\_\_eq\_\_',

'\_\_format\_\_',  '\_\_ge\_\_',  '\_\_getattribute\_\_',  '\_\_gt\_\_',  '\_\_hash\_\_',  '\_\_init\_\_',  '\_\_le\_\_',

'\_\_lt\_\_',  '\_\_module\_\_',  '\_\_ne\_\_',  '\_\_new\_\_',  '\_\_reduce\_\_',  '\_\_reduce\_ex\_\_',  '\_\_repr\_\_',

'\_\_setattr\_\_',  '\_\_sizeof\_\_',  '\_\_str\_\_',  '\_\_subclasshook\_\_',  '\_\_weakref\_\_',  '\_name',  'weight',

'x',  'y']

['\_Animal\_\_age',  '\_\_class\_\_',  '\_\_delattr\_\_',  '\_\_dir\_\_',  '\_\_doc\_\_',  '\_\_eq\_\_',  '\_\_format\_\_',

'\_\_ge\_\_',  '\_\_getattribute\_\_',  '\_\_gt\_\_',  '\_\_hash\_\_',  '\_\_init\_\_',  '\_\_le\_\_',  '\_\_lt\_\_',

'\_\_module\_\_',  '\_\_ne\_\_',  '\_\_new\_\_',  '\_\_reduce\_\_',  '\_\_reduce\_ex\_\_',  '\_\_repr\_\_',  '\_\_setattr\_\_',

'\_\_sizeof\_\_',  '\_\_str\_\_',  '\_\_subclasshook\_\_',  '\_name',  'weight',  'x',  'y']

Dog's  dir  =  ['\_\_class\_\_',  '\_\_delattr\_\_',  '\_\_dict\_\_',  '\_\_dir\_\_',  '\_\_doc\_\_',  '\_\_eq\_\_',

'\_\_format\_\_',  '\_\_ge\_\_',  '\_\_getattribute\_\_',  '\_\_gt\_\_',  '\_\_hash\_\_',  '\_\_init\_\_',  '\_\_le\_\_',

'\_\_lt\_\_',  '\_\_module\_\_',  '\_\_ne\_\_',  '\_\_new\_\_',  '\_\_reduce\_\_',  '\_\_reduce\_ex\_\_',  '\_\_repr\_\_',

'\_\_setattr\_\_',  '\_\_sizeof\_\_',  '\_\_str\_\_',  '\_\_subclasshook\_\_',  '\_\_weakref\_\_',  'x']

['dog']

{'\_Animal\_\_age':  10,  'weight':  20,  '\_name':  'snoppy'}

dir()的测试如下

class  Person:

def  show(self):  #  方法中

a  =  100

t  =  int(a)

print(dir())

def  test(a=50,  b=100):  #  函数中

print(dir())

Person().show()

test()

#  显示结果如下

['a',  'self',  't']

['a',  'b']

魔术方法 \*\*\*

分类：

创建、初始化与销毁

\_\_init\_\_   与   \_\_del\_\_

|  |  |
| --- | --- |
| 方法 | 意义 |
| \_\_new\_\_ | 实例化一个对象  该方法需要返回一个值，如果该值不是cls的实例，则不会调用  \_\_init\_\_  该方法永远都是静态方法 |

hash

bool

可视化

运算符重载

容器和大小

可调用对象

上下文管理

反射

描述器

其他杂项

实例化

class  A:

def  \_\_new\_\_(cls,  \*args,  \*\*kwargs):

print(cls)

print(\*args)

print(\*\*kwargs)

#return  super().\_\_new\_\_(cls)

#return  1

return  None

def  \_\_init\_\_(self,  name):

self.name  =  name

a  =  A()

print(a)

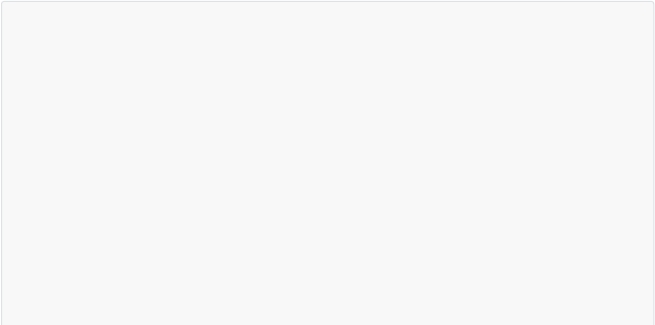
\_\_new\_\_  方法很少使用，即使创建了该方法，也会使用  return  super().\_\_new\_\_(cls)  基类object的  \_\_new\_\_  方

法来创建实例并返回。

hash

print(hash(1))

|  |  |
| --- | --- |
| 方法 | 意义 |
| \_\_hash\_\_ | 内建函数  hash()  调用的返回值，返回一个整数。如果定义这个方法该类的实例就可hash。 |



print(hash('tom'))

print(hash(('tom',)))

class  A:

def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

def  \_\_hash\_\_(self):

return  1

def  \_\_repr\_\_(self):

return  self.name

print(hash(A('tom')))

print((A('tom'),  A('tom')))

print([A('tom'),  A('tom')])

print('~~~~~~~~~~~~~~~~~~~~')

print({1,  1})

s  =  {A('tom'),  A('tom')}  #  set

print(s)  #  去重了吗

print({tuple('t'),  tuple('t')})

print({('tom',),  ('tom',)})

print({'tom',  'tom'})

上例中set为什么不能剔除相同的key？

hash值相同就会去重吗？

class  A:

def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

def  \_\_hash\_\_(self):

return  1

def  \_\_eq\_\_(self,  other):  #  这个函数作用？

return  self.name  ==  other.name

def  \_\_repr\_\_(self):

return  self.name

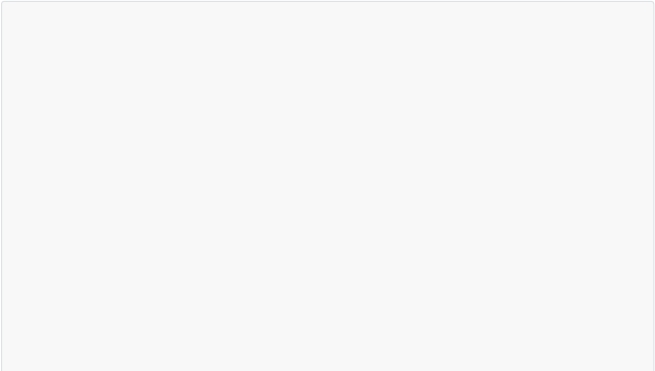
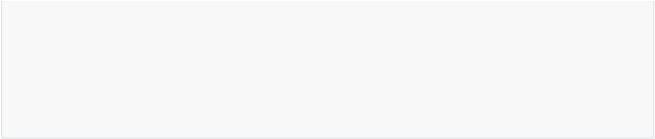
print(hash(A('tom')))

print((A('tom'),  A('tom')))

print([A('tom'),  A('tom')])

print('~~~~~~~~~~~~~~~~~~~~')

|  |  |
| --- | --- |
| 方法 | 意义 |
| \_\_eq\_\_ | 对应==操作符，判断2个对象是否相等，返回bool值 |



s  =  {A('tom'),  A('tom')}  #  set

print(s)

print({tuple('t'),  tuple('t')})

print({('tom',),  ('tom',)})

print({'tom',  'tom'})

\_\_hash\_\_  方法只是返回一个hash值作为set的key，但是  去重  ，还需要  \_\_eq\_\_  来判断2个对象是否相等。

hash值相等，只是hash冲突，不能说明两个对象是相等的。

因此，一般来说提供  \_\_hash\_\_  方法是为了作为set或者dict的key，所以  去重  要同时提供  \_\_eq\_\_  方法。

不可hash对象isinstance(p1, collections.Hashable)一定为False。

去重  需要提供  \_\_eq\_\_  方法。

思考：

list类实例为什么不可hash？

练习

设计二维坐标类Point，使其成为可hash类型，并比较2个坐标的实例是否相等？

list类实例为什么不可hash

源码中有一句  \_\_hash\_\_   = None，也就是如果调用  \_\_hash\_\_  ()相当于None()，一定报错。

所有类都继承object，而这个类是具有  \_\_hash\_\_  ()方法的，如果一个类不能被hash，就把  \_\_hash\_\_  设置为

None。

练习参考

from  collections  import  Hashable

class  Point:

def  \_\_init\_\_(self,  x,  y):

self.x  =  x

self.y  =  y

def  \_\_hash\_\_(self):

return  hash((self.x,  self.y))

def  \_\_eq\_\_(self,  other):

return  self.x  ==  other.x  and  self.y  ==  other.y

p1  =  Point(4,  5)

p2  =  Point(4,  5)

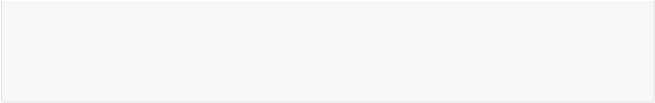
print(hash(p1))

print(hash(p2))

print(p1  is  p2)

print(p1  ==  p2)  #  True  使用\_\_eq\_\_

|  |  |
| --- | --- |
| 方法 | 意义 |
| \_\_bool\_\_ | 内建函数bool()，或者对象放在逻辑表达式的位置，调用这个函数返回布尔值。  没有定义  \_\_bool\_\_  ()，就找  \_\_len\_\_  ()返回长度，非0为真。  如果  \_\_len\_\_  ()也没有定义，那么所有实例都返回真 |



print(hex(id(p1)),  hex(id(p2)))

print(set((p1,  p2)))

print(isinstance(p1,  Hashable))

bool

class  A:  pass

print(bool(A()))

if  A():

print('Real  A')

class  B:

def  \_\_bool\_\_(self):

return  False

print(bool(B))

print(bool(B()))

if  B():

print('Real  B')

class  C:

def  \_\_len\_\_(self):

return  0

print(bool(C()))

if  C():

print('Real  C')

可视化

class  A:

|  |  |
| --- | --- |
| 方法 | 意义 |
| \_\_str\_\_ | str()函数、format()函数、print()函数调用，需要返回对象的字符串表达。如果没有定义，就  去调用  \_\_repr\_\_  方法返回字符串表达，如果  \_\_repr\_\_  没有定义，就直接返回对象的内存地  址信息 |
| \_\_repr\_\_ | 内建函数repr()对一个对象获取字符串表达。  调用  \_\_repr\_\_  方法返回字符串表达，如果  \_\_repr\_\_  也没有定义，就直接返回object的定义  就是显示内存地址信息 |
| \_\_bytes\_\_ | bytes()函数调用，返回一个对象的bytes表达，即返回bytes对象 |

def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

self.age  =  age

def  \_\_repr\_\_(self):

return  'repr:  {},{}'.format(self.name,  self.age)

def  \_\_str\_\_(self):

return  'str:   {},{}'.format(self.name,  self.age)

def  \_\_bytes\_\_(self):

#return  "{}  is  {}".format(self.name,  self.age).encode()

import  json

return  json.dumps(self.\_\_dict\_\_).encode()

print(A('tom'))  #  print函数使用\_\_str\_\_

print('{}'.format(A('tom')))

print([A('tom')])  #  []使用\_\_str\_\_，但其内部使用\_\_repr\_\_

print([str(A('tom'))])  #  []使用\_\_str\_\_，其中的元素使用str()函数也调用\_\_str\_\_

print('str:a,1')  #  字符串直接输出没有引号

s  =  '1'

print(s)

s1  =  'a'

print(s1)

print([s1],(s,))  #  字符串在基本数据类型内部输出有引号

print({s,  'a'})

print(bytes(A('tom')))

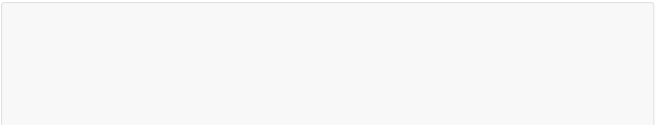
注意不能通过判断是否带引号来判断输出值的类型，类型判断要使用type或isinstance

运算符重载

operator模块提供以下的特殊方法，可以将类的实例使用下面的操作符来操作

实现A类的2个实例相减

|  |  |  |
| --- | --- | --- |
| 运算符 | 特殊方法 | 含义 |
| <, <=, ==, >,  >=, != | \_\_lt\_\_  ,   \_\_le\_\_  ,   \_\_eq\_\_  ,   \_\_gt\_\_  ,   \_\_ge\_\_  ,   \_\_ne\_\_ | 比较运算符 |
| +, -, \*, /, %, //,  \*\*, divmod | \_\_add\_\_  ,   \_\_sub\_\_  ,   \_\_mul\_\_  ,   \_\_truediv\_\_  ,   \_\_mod\_\_  ,  \_\_floordiv\_\_  ,   \_\_pow\_\_  ,   \_\_divmod\_\_ | 算数运算符，移位、位运  算也有对应的方法 |
| +=, -=, \*=, /=,  %=, //=, \*\*= | \_\_iadd\_\_  ,   \_\_isub\_\_  ,   \_\_imul\_\_  ,   \_\_itruediv\_\_  ,  \_\_imod\_\_  ,   \_\_ifloordiv\_\_  ,   \_\_ipow\_\_ |  |



class  A:

def  \_\_init\_\_(self,  name,  age=18):

self.name  =  name

self.age  =  age

def  \_\_sub\_\_(self,  other):

return  self.age  -  other.age

def  \_\_isub\_\_(self,  other):  #  如果没有定义\_\_isub\_\_，则会调用\_\_sub\_\_

return  A(self.name,  self  -  other)

tom  =  A('tom')

jerry  =  A('jerry',  16)

print(tom  -  jerry)

print(jerry  -  tom,  jerry.\_\_sub\_\_(tom))

print(id(tom))

tom  -=  jerry

print(tom.age,  id(tom))

\_\_isub\_\_  方法定义，一般会in-place就地来修改自身

如果没有定义  \_\_isub\_\_  方法，则会调用  \_\_sub\_\_

练习：

完成Point类设计，实现判断点相等的方法，并完成向量的加法

在直角坐标系里面,定义原点为向量的起点.两个向量和与差的坐标分别等于这两个向量相应坐标的和与差若向量的

表示为(x,y)形式,

A(X1,Y1)  B(X2,Y2)，则A+B=（X1+X2，Y1+Y2），A-B=（X1-X2，Y1-Y2）

class  Point:

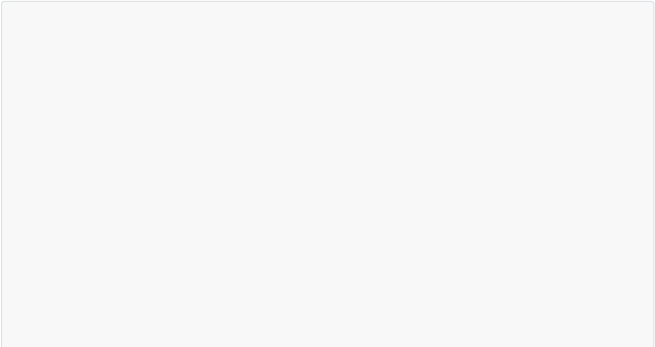
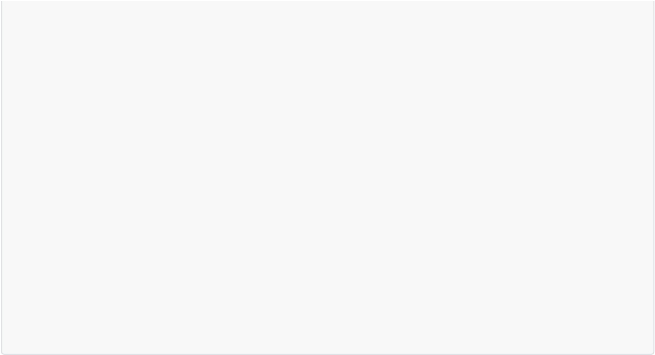
def  \_\_init\_\_(self,  x,  y):

self.x  =  x

self.y  =  y

def  \_\_eq\_\_(self,  other):

return  self.x  ==  other.x  and  self.y  ==  other.y



def  \_\_add\_\_(self,  other):

return  Point(self.x  +  other.x,  self.y  +  other.y)

def  add(self,  other):

return  (self.x  +  other.x,  self.y  +  other.y)

def  \_\_str\_\_(self):

return  '<Point:  {},{}>'.format(self.x,  self.y)

p1  =  Point(1,1)

p2  =  Point(1,1)

points  =  (p1,  p2)

print(points[0].add(points[1]))

#  运算符重载

print(points[0]  +  points[1])

print(p1  ==  p2)

运算符重载应用场景

往往是用面向对象实现的类，需要做大量的运算，而运算符是这种运算在数学上最常见的表达方式。例如，上例中

的对+进行了运算符重载，实现了Point类的二元操作，重新定义为Point + Point。

提供运算符重载，比直接提供加法方法要更加适合该领域内使用者的习惯。

int类，几乎实现了所有操作符，可以作为参考。

@functools.total\_ordering 装饰器

\_\_lt\_\_  ,   \_\_le\_\_  ,   \_\_eq\_\_  ,   \_\_gt\_\_  ,   \_\_ge\_\_  是比较大小必须实现的方法，但是全部写完太麻烦，使用

@functools.total\_ordering 装饰器就可以大大简化代码。

但是要求  \_\_eq\_\_  必须实现，其它方法  \_\_lt\_\_  ,   \_\_le\_\_  ,   \_\_gt\_\_  ,   \_\_ge\_\_ 实现其一

from  functools  import  total\_ordering

@total\_ordering

class  Person:

def  \_\_init\_\_(self,  name,  age):

self.name  =  name

self.age  =  age

def  \_\_eq\_\_(self,  other):

return  self.age  ==  other.age

def  \_\_gt\_\_(self,  other):

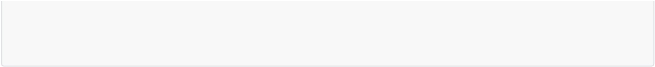
return  self.age  >  other.age

tom  =  Person('tom',  20)

jerry  =  Person('jerry',  16)

print(tom  >  jerry)

print(tom  <  jerry)



print(tom  >=  jerry)  #

print(tom  <=  jerry)  #

上例中大大简化代码，但是一般来说比较实现等于或者小于方法也就够了，其它可以不实现，所以这个装饰器只是

看着很美好，且可能会带来性能问题，建议需要什么方法就自己创建，少用这个装饰器。

class  Person:

def  \_\_init\_\_(self,  name,  age):

self.name  =  name

self.age  =  age

def  \_\_eq\_\_(self,  other):

return  self.age  ==  other.age

def  \_\_gt\_\_(self,  other):

return  self.age  >  other.age

def  \_\_ge\_\_(self,  other):

return  self.age  >=  other.age

tom  =  Person('tom',  20)

jerry  =  Person('jerry',  16)

print(tom  >  jerry)

print(tom  <  jerry)

print(tom  >=  jerry)  #

print(tom  <=  jerry)

print(tom  ==  jerry)

print(tom  !=  jerry)

\_\_eq\_\_  等于可以推断不等于

\_\_gt\_\_  大于可以推断小于

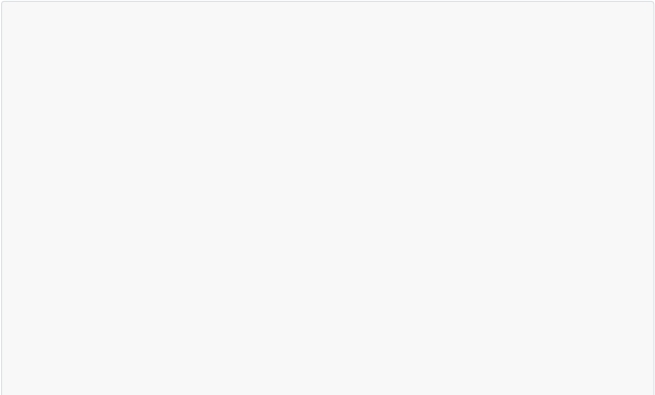
\_\_ge\_\_  大于等于可以推断小于等于

也就是用3个方法，就可以把所有比较解决了，所以total\_ordering可以不使用

容器相关方法

class  A(dict):

|  |  |
| --- | --- |
| 方法 | 意义 |
| \_\_len\_\_ | 内建函数len()，返回对象的长度（>=0的整数），如果把对象当做容器类型看，就如同list  或者dict。bool()函数调用的时候，如果没有  \_\_bool\_\_()  方法，则会看  \_\_len\_\_()  方法  是否存在，存在返回非0为真 |
| \_\_iter\_\_ | 迭代容器时，调用，返回一个新的迭代器对象 |
| \_\_contains\_\_ | in 成员运算符，没有实现，就调用  \_\_iter\_\_  方法遍历 |
| \_\_getitem\_\_ | 实现self[key]访问。序列对象，key接受整数为索引，或者切片。对于set和dict，key为  hashable。key不存在引发KeyError异常 |
| \_\_setitem\_\_ | 和  \_\_getitem\_\_  的访问类似，是设置值的方法 |
| \_\_missing\_\_ | 字典或其子类使用  \_\_getitem\_\_()  调用时，key不存在执行该方法 |



def  \_\_missing\_\_(self,  key):

print('Missing  key  :  ',  key)

return  0

a  =  A()

print(a['k'])

思考

为什么空字典、空字符串、空元组、空集合、空列表等可以等效为False？

练习

将购物车类改造成方便操作的容器类

class  Cart:

def  \_\_init\_\_(self):

self.items  =  []

def  \_\_len\_\_(self):

return  len(self.items)

def  additem(self,  item):

self.items.append(item)

def  \_\_iter\_\_(self):

#  yield  from  self.items

return  iter(self.items)

def  \_\_getitem\_\_(self,  index):  #  索引访问

return  self.items[index]

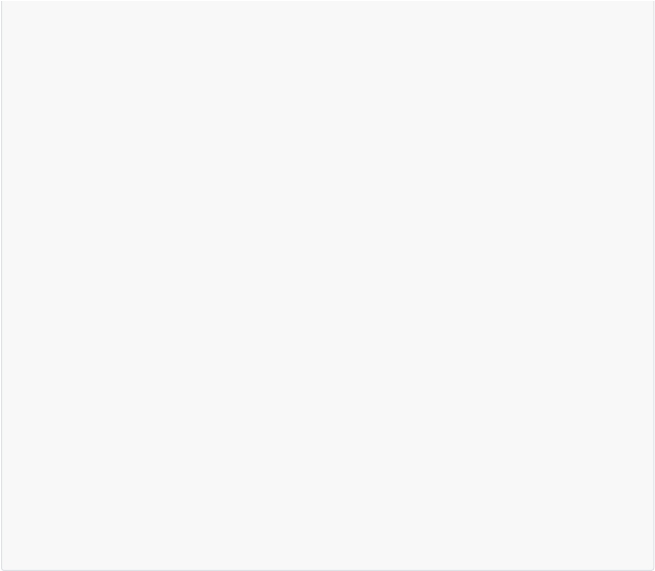
def  \_\_setitem\_\_(self,  key,  value):  #  索引赋值

self.items[key]  =  value

def  \_\_str\_\_(self):

|  |  |
| --- | --- |
| 方法 | 意义 |
| \_\_call\_\_ | 类中定义一个该方法，实例就可以像函数一样调用 |

return  str(self.items)



def  \_\_add\_\_(self,  other):  #  +

self.items.append(other)

return  self

cart  =  Cart()

cart.additem(1)

cart.additem('abc')

cart.additem(3)

#  长度、bool

print(len(cart))

print(bool(cart))

#  迭代

for  x  in  cart:

print(x)

#  in

print(3  in  cart)

print(2  in  cart)

#  索引操作

print(cart[1])

cart[1]  =  'xyz'

print(cart)

#  链式编程实现加法

print(cart  +  4  +  5  +  6)

print(cart.\_\_add\_\_(17).\_\_add\_\_(18))

可调用对象

Python中一切皆对象，函数也不例外。

def  foo():

print(foo.\_\_module\_\_,  foo.\_\_name\_\_)

foo()

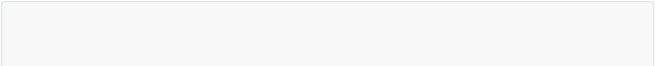
#  等价于

foo.\_\_call\_\_()

函数即对象，对象foo加上()，就是调用此函数对象的  \_\_call\_\_()  方法

可调用对象

可调用对象：定义一个类，并实例化得到其实例，将实例像函数一样调用。



class  Point:

def  \_\_init\_\_(self,  x,  y):

self.x  =  x

self.y  =  y

def  \_\_call\_\_(self,  \*args,  \*\*kwargs):

return  "<Point  {}:{}>".format(self.x,  self.y)

p  =  Point(4,  5)

print(p)

print(p())

class  Adder:

def  \_\_call\_\_(self,  \*args):

ret  =  0

for  x  in  args:

ret  +=  x

self.ret  =  ret

return  ret

adder  =  Adder()

print(adder(4,  5,  6))

print(adder.ret)

练习：

定义一个斐波那契数列的类，方便调用，计算第n项。

增加迭代的方法、返回容器长度、支持索引的方法。

class  Fib:

def  \_\_init\_\_(self):

self.items  =  [0,  1,  1]

def  \_\_call\_\_(self,  index):

if  index  <  0:

raise  IndexError('Wrong  Index')

if  index  <  len(self.items):

return  self.items[index]

for  i  in  range(len(self.items),  index+1):

self.items.append(self.items[i-1]  +  self.items[i-2])

return  self.items[index]

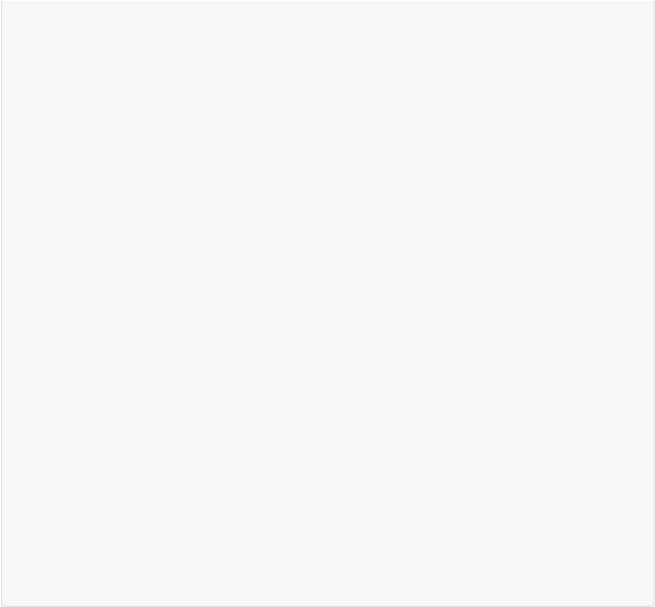
print(Fib()(101))

上例中，增加迭代的方法、返回容器长度、支持索引的方法

class  Fib:

def  \_\_init\_\_(self):

self.items  =  [0,  1,  1]



def  \_\_call\_\_(self,  index):

return  self[index]

def  \_\_iter\_\_(self):

return  iter(self.items)

def  \_\_len\_\_(self):

return  len(self.items)

def  \_\_getitem\_\_(self,  index):

if  index  <  0:

raise  IndexError('Wrong  Index')

if  index  <  len(self.items):

return  self.items[index]

for  i  in  range(len(self),  index+1):

self.items.append(self.items[i-1]  +  self.items[i-2])

return  self.items[index]

def  \_\_str\_\_(self):

return  str(self.items)

\_\_repr\_\_  =  \_\_str\_\_

fib  =  Fib()

print(fib(5),  len(fib))  #  全部计算

print(fib(10),  len(fib))  #  部分计算

for  x  in  fib:

print(x)

print(fib[5],  fib[6])  #  索引访问，不计算

可以看出使用类来实现斐波那契数列也是非常好的实现，还可以缓存数据，便于检索。