魔术方法

|  |  |
| --- | --- |
| 方法 | 意义 |
| \_\_enter\_\_ | 进入与此对象相关的上下文。如果存在该方法，with语法会把该方法的返回值作为绑定到as  子句中指定的变量上 |
| \_\_exit\_\_ | 退出与此对象相关的上下文。 |



分类：

创建与销毁

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

上下文管理

文件IO操作可以对文件对象使用上下文管理，使用with...as语法。

with  open('test')  as  f:

pass

仿照上例写一个自己的类，实现上下文管理

class  Point:

pass

with  Point()  as  p:  #  AttributeError:  \_\_exit\_\_

pass

提示属性错误，没有  \_\_exit\_\_  ，看了需要这个属性

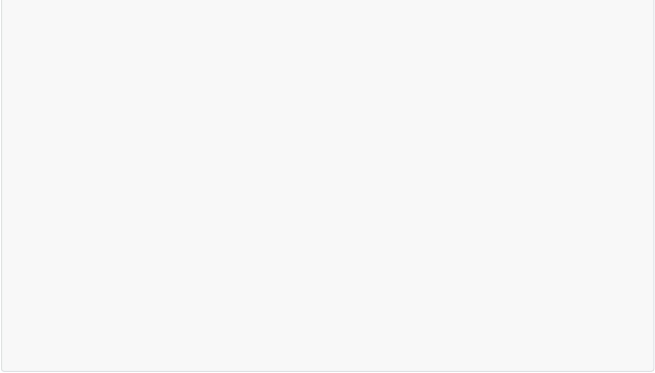
某些版本会显示没有  \_\_enter\_\_

上下文管理对象

当一个对象同时实现了  \_\_enter\_\_  ()和  \_\_exit\_\_  ()方法，它就属于上下文管理的对象

import  time

class  Point:



def  \_\_init\_\_(self):

print('init  ~~~~~~~~')

time.sleep(1)

print('init  over')

def  \_\_enter\_\_(self):

print('enter  ~~~~~~~~')

def  \_\_exit\_\_(self,  exc\_type,  exc\_val,  exc\_tb):

print('exit  ============')

with  Point()  as  p:

print('in  with-------------')

time.sleep(2)

print('with  over')

print('=======end==========')

实例化对象的时候，并不会调用enter，进入with语句块调用  \_\_enter\_\_  方法，然后执行语句体，最后离开with语

句块的时候，调用  \_\_exit\_\_  方法。

with可以开启一个上下文运行环境，在执行前做一些准备工作，执行后做一些收尾工作。

注意，with并不开启一个新的作用域。

上下文管理的安全性

看看异常对上下文的影响。

import  time

class  Point:

def  \_\_init\_\_(self):

print('init  ~~~~~~~~')

time.sleep(1)

print('init  over')

def  \_\_enter\_\_(self):

print('enter  ~~~~~~~~')

def  \_\_exit\_\_(self,  exc\_type,  exc\_val,  exc\_tb):

print('exit  ============')

with  Point()  as  p:

print('in  with-------------')

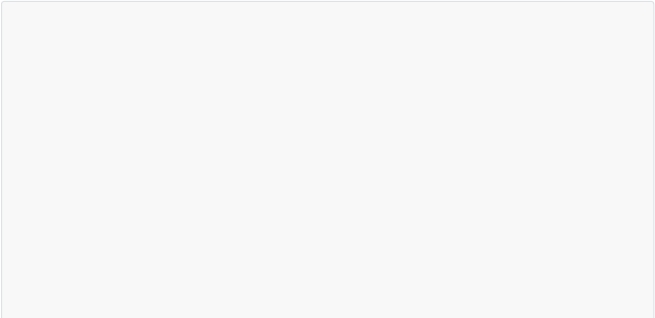
raise  Exception('error')

time.sleep(2)

print('with  over')

print('=======end==========')

可以看出在enter和exit照样执行，上下文管理是安全的。



极端的例子

调用sys.exit()，它会退出当前解释器。

打开Python解释器，在里面敲入sys.exit()，窗口直接关闭了。也就是说碰到这一句，Python运行环境直接退出

了。

import  time

class  Point:

def  \_\_init\_\_(self):

print('init  ~~~~~~~~')

time.sleep(1)

print('init  over')

def  \_\_enter\_\_(self):

print('enter  ~~~~~~~~')

def  \_\_exit\_\_(self,  exc\_type,  exc\_val,  exc\_tb):

print('exit  ============')

with  Point()  as  p:

print('in  with-------------')

import  sys

sys.exit(1)

time.sleep(2)

print('with  over')

print('=======end==========')

从执行结果来看，依然执行了  \_\_exit\_\_  函数，哪怕是退出Python运行环境。

说明上下文管理很安全。

with语句

class  Point:

def  \_\_init\_\_(self):

print('init')

def  \_\_enter\_\_(self):

print('enter')

def  \_\_exit\_\_(self,  exc\_type,  exc\_val,  exc\_tb):

print('exit')

f  =  open('t3.py')

with  f  as  p:

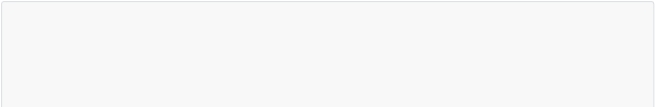
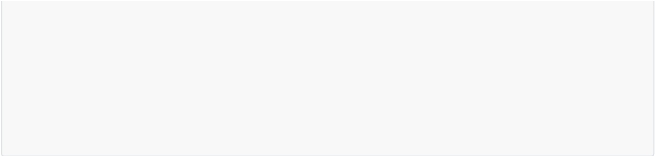
print(f)

print(p)

print(f  is  p)  #  打印什么

print(f  ==  p)  #  打印什么

p  =  Point()



with  p  as  f:

print('in  with-------------')

print(p  ==  f)

print('with  over')

print('=======end==========')

问题在于  \_\_enter\_\_  方法上，它将自己的返回值赋给f。修改上例

class  Point:

def  \_\_init\_\_(self):

print('init  ~~~~~~~~')

def  \_\_enter\_\_(self):

print('enter  ~~~~~~~~')

return  self  #  增加返回值

def  \_\_exit\_\_(self,  exc\_type,  exc\_val,  exc\_tb):

print('exit  ============')

p  =  Point()

with  p  as  f:

print('in  with-------------')

print(p  ==  f)

print('with  over')

print('=======end==========')

with语法，会调用with后的对象的\_\_enter\_\_方法，如果有as，则将该方法的返回值赋给as子句的变量。

方法的参数

\_\_enter\_\_  方法 没有其他参数。

\_\_exit\_\_  方法有3个参数：

\_\_exit\_\_(self,  exc\_type,  exc\_value,  traceback)

这三个参数都与异常有关。

如果该上下文退出时没有异常，这3个参数都为None。

如果有异常，参数意义如下

exc\_type  ，异常类型

exc\_value  ，异常的值

traceback  ，异常的追踪信息

\_\_exit\_\_  方法返回一个等效True的值，则压制异常；否则，继续抛出异常

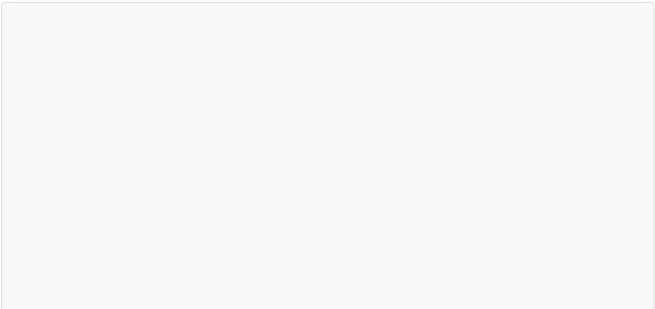
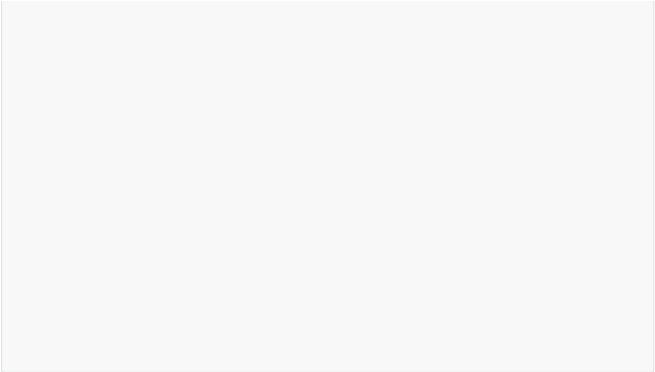
class  Point:

def  \_\_init\_\_(self):

print('init  ~~~~~~~~')

def  \_\_enter\_\_(self):

print('enter  ~~~~~~~~')



return  self  #  增加返回值

def  \_\_exit\_\_(self,  exc\_type,  exc\_val,  exc\_tb):

print(1,  exc\_type)

print(2,  exc\_val)

print(3,  exc\_tb)

print('exit  ============')

#return  'abc'

#return  None

#return  0  #  1

#return  'a'  #  ''  []

p  =  Point()

with  p  as  f:

print('in  with-------------')

raise  Exception('Error')

print('with  over')

print('=======end==========')

练习

为加法函数计时

方法1、使用装饰器显示该函数的执行时长

方法2、使用上下文管理方法来显示该函数的执行时长

import  time

def  add(x,  y):

time.sleep(2)

return  x  +  y

装饰器实现

import  time

import  datetime

from  functools  import  wraps

def  timeit(fn):

@wraps(fn)

def  wrapper(\*args,  \*\*kwargs):

start  =  datetime.datetime.now()

ret  =  fn(\*args,  \*\*kwargs)

delta  =  (datetime.datetime.now()  -  start).total\_seconds()

print('{}  took  {}s  '.format(fn.\_\_name\_\_,delta))

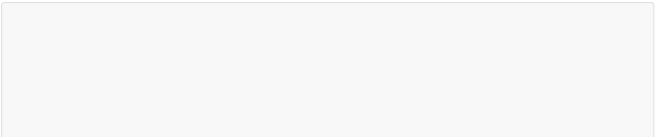
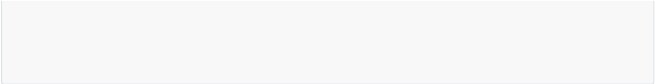
return  ret

return  wrapper

@timeit

def  add(x,  y):

time.sleep(2)



return  x  +  y

print(add(4,  5))

上下文实现

import  time

import  datetime

from  functools  import  wraps

def  timeit(fn):

@wraps(fn)

def  wrapper(\*args,  \*\*kwargs):

start  =  datetime.datetime.now()

ret  =  fn(\*args,  \*\*kwargs)

delta  =  (datetime.datetime.now()  -  start).total\_seconds()

print('{}  took  {}s  '.format(fn.\_\_name\_\_,delta))

return  ret

return  wrapper

@timeit

def  add(x,  y):

time.sleep(2)

return  x  +  y

class  Timeit:

def  \_\_init\_\_(self,  fn):

self.fn  =  fn

def  \_\_enter\_\_(self):

self.start  =  datetime.datetime.now()

return  self.fn

def  \_\_exit\_\_(self,  exc\_type,  exc\_val,  exc\_tb):

delta  =  (datetime.datetime.now()  -  self.start).total\_seconds()

print("{}  took  {}s".format(self.fn.\_\_name\_\_,  delta))

with  Timeit(add)  as  fn:

#print(fn(4,  6))

print(add(4,  7))

另一种实现，使用可调用对象实现。

import  time

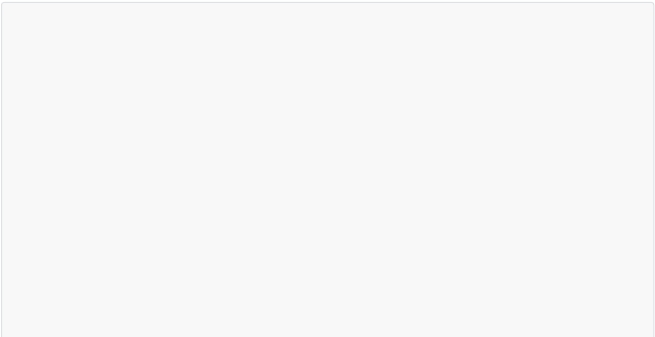
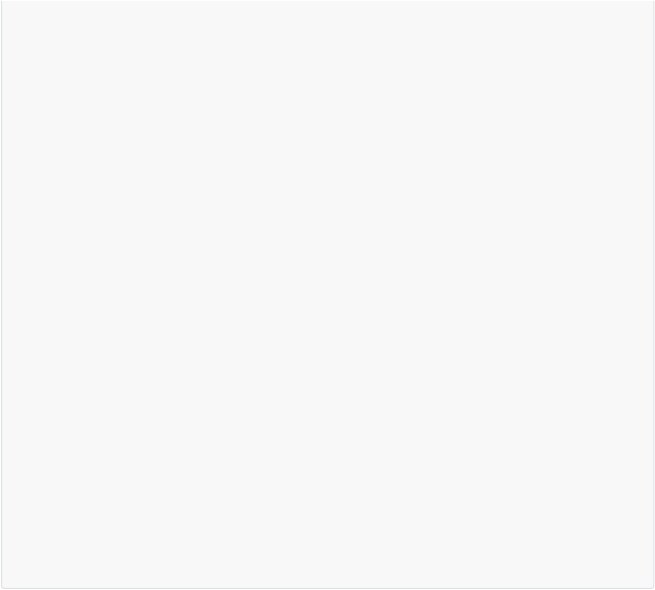
import  datetime

from  functools  import  wraps

def  timeit(fn):

@wraps(fn)

def  wrapper(\*args,  \*\*kwargs):



start  =  datetime.datetime.now()

ret  =  fn(\*args,  \*\*kwargs)

delta  =  (datetime.datetime.now()  -  start).total\_seconds()

print('{}  took  {}s  '.format(fn.\_\_name\_\_,delta))

return  ret

return  wrapper

@timeit

def  add(x,  y):

time.sleep(2)

return  x  +  y

class  Timeit:

def  \_\_init\_\_(self,  fn):

self.fn  =  fn

def  \_\_enter\_\_(self):

self.start  =  datetime.datetime.now()

return  self

def  \_\_exit\_\_(self,  exc\_type,  exc\_val,  exc\_tb):

delta  =  (datetime.datetime.now()  -  self.start).total\_seconds()

print("{}  took  {}s".format(self.fn.\_\_name\_\_,  delta))

def  \_\_call\_\_(self,  x,  y):

print(x,  y)

return  self.fn(x,  y)

with  Timeit(add)  as  timeitobj:

print(timeitobj(5,  6))

根据上面的代码，能不能把类当做装饰器用？

import  time

import  datetime

from  functools  import  wraps

class  TimeIt:

def  \_\_init\_\_(self,  fn):

self.fn  =  fn

def  \_\_enter\_\_(self):

self.start  =  datetime.datetime.now()

return  self

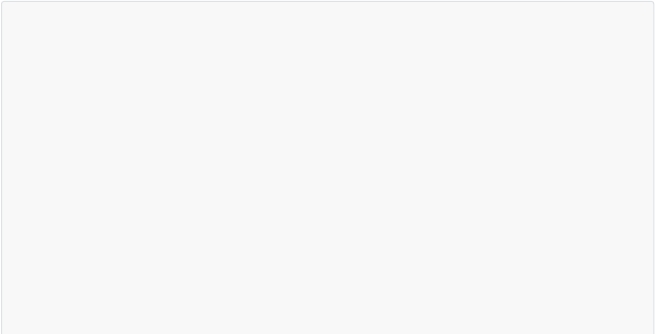
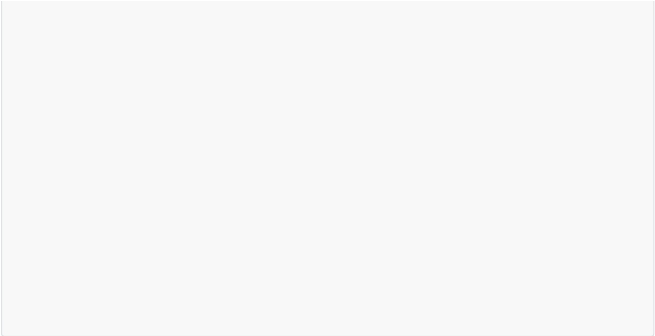
def  \_\_exit\_\_(self,  exc\_type,  exc\_val,  exc\_tb):

self.delta  =  (datetime.datetime.now()  -  self.start).total\_seconds()

print('{}  took  {}s.  context'.format(self.fn.\_\_name\_\_,  self.delta))

pass

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



self.start  =  datetime.datetime.now()

ret  =  self.fn(\*args,  \*\*kwargs)

self.delta  =  (datetime.datetime.now()  -  self.start).total\_seconds()

print('{}  took  {}s.  call'.format(self.fn.\_\_name\_\_,  self.delta))

return  ret

@TimeIt

def  add(x,  y):

"""This  is  add  function."""

time.sleep(2)

return  x  +  y

add(4,  5)

print(add.\_\_doc\_\_)

思考

如何解决文档字符串问题？

方法一

直接修改  \_\_doc\_\_

class  TimeIt:

def  \_\_init\_\_(self,  fn=None):

self.fn  =  fn

#  把函数对象的文档字符串赋给类

self.\_\_doc\_\_  =  fn.\_\_doc\_\_

方法二

使用functools.wraps函数

import  time

import  datetime

from  functools  import  wraps,  update\_wrapper

class  Timeit:

"""This  is  A  Class"""

def  \_\_init\_\_(self,  fn):

self.fn  =  fn

#  把函数对象的文档字符串赋给类

#  self.\_\_doc\_\_  =  fn.\_\_doc\_\_

#  update\_wrapper(self,  fn)

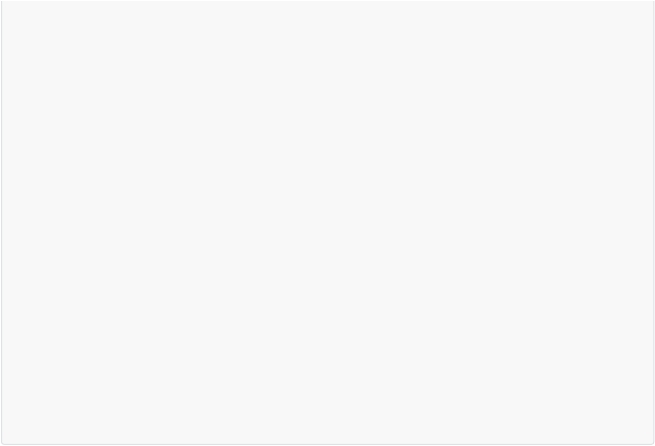
wraps(fn)(self)

def  \_\_enter\_\_(self):

self.start  =  datetime.datetime.now()

return  self

def  \_\_exit\_\_(self,  exc\_type,  exc\_val,  exc\_tb):



delta  =  (datetime.datetime.now()  -  self.start).total\_seconds()

print("{}  took  {}s.  context".format(self.fn.\_\_name\_\_,  delta))

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

self.start  =  datetime.datetime.now()

ret  =  self.fn(\*args,  \*\*kwargs)

delta  =  (datetime.datetime.now()  -  self.start).total\_seconds()

print("{}  took  {}s.  call".format(self.fn.\_\_name\_\_,  delta))

return  ret

@Timeit

def  add(x,  y):

"""This  is  add  function."""

time.sleep(2)

return  x  +  y

print(add(10,  5))

print(add.\_\_doc\_\_)

print(Timeit(add).\_\_doc\_\_)

上面的类即可以用在上下文管理，又可以用做装饰器

上下文应用场景

1. 增强功能

在代码执行的前后增加代码，以增强其功能。类似装饰器的功能。

2. 资源管理

打开了资源需要关闭，例如文件对象、网络连接、数据库连接等

3. 权限验证

在执行代码之前，做权限的验证，在  \_\_enter\_\_  中处理

contextlib.contextmanager

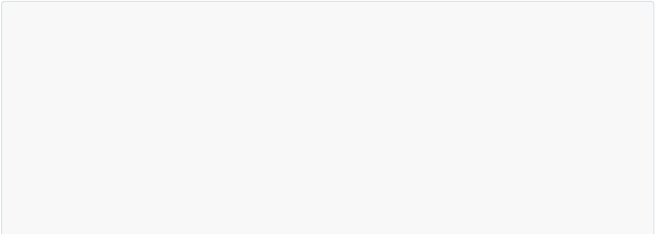
contextlib.contextmanager

它是一个装饰器实现上下文管理，装饰一个函数，而不用像类一样实现  \_\_enter\_\_  和  \_\_exit\_\_  方法。

对下面的函数有要求：必须有yield，也就是这个函数必须返回一个生成器，且只有yield一个值。

也就是这个装饰器接收一个生成器对象作为参数。

import  contextlib



@contextlib.contextmanager

def  foo():  #

print('enter')   #  相当于\_\_enter\_\_()

yield  #  yield  5，yield的值只能有一个，作为\_\_enter\_\_方法的返回值

print('exit')  #  相当于\_\_exit\_\_()

with  foo()  as  f:

#raise  Exception()

print(f)

f接收yield语句的返回值。

上面的程序看似不错但是，增加一个异常试一试，发现不能保证exit的执行，怎么办？

增加try ﬁnally。

import  contextlib

@contextlib.contextmanager

def  foo():

print('enter')

try:

yield  #  yield  5，yield的值只能有一个，作为\_\_enter\_\_方法的返回值

finally:

print('exit')

with  foo()  as  f:

raise  Exception()

print(f)

上例这么做有什么意义呢？

当yield发生处为生成器函数增加了上下文管理。这是为函数增加上下文机制的方式。

把yield之前的当做\_\_enter\_\_方法执行

把yield之后的当做\_\_exit\_\_方法执行

把yield的值作为\_\_enter\_\_的返回值

import  contextlib

import  datetime

import  time

@contextlib.contextmanager

def  add(x,  y):  #  为生成器函数增加了上下文管理

start  =  datetime.datetime.now()

try:

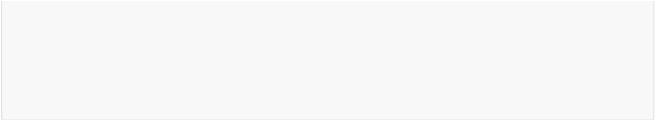
yield  x  +  y#  yield  5，yield的值只能有一个，作为\_\_enter\_\_方法的返回值

finally:

delta  =  (datetime.datetime.now()  -  start).total\_seconds()

print(delta)

with  add(4,  5)  as  f:



#raise  Exception()

time.sleep(2)

print(f)

总结

如果业务逻辑简单可以使用函数加contextlib.contextmanager装饰器方式，如果业务复杂，用类的方式加

\_\_enter\_\_  和  \_\_exit\_\_  方法方便。