第一步：最简单的函数，准备附加额外功能

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | # -\*- coding:gbk -\*-  '''示例1: 最简单的函数,表示调用了两次'''    def myfunc():      print("myfunc() called.")    myfunc()  myfunc() |

第二步：使用装饰函数在函数执行前和执行后分别附加额外功能

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | # -\*- coding:gbk -\*-  '''示例2: 替换函数(装饰)  装饰函数的参数是被装饰的函数对象，返回原函数对象  装饰的实质语句: myfunc = deco(myfunc)'''    def deco(func):      print("before myfunc() called.")      func()      print("  after myfunc() called.")      return func    def myfunc():      print(" myfunc() called.")    myfunc = deco(myfunc)    myfunc()  myfunc() |

第三步：使用语法糖@来装饰函数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | # -\*- coding:gbk -\*-  '''示例3: 使用语法糖@来装饰函数，相当于“myfunc = deco(myfunc)”  但发现新函数只在第一次被调用，且原函数多调用了一次'''    def deco(func):      print("before myfunc() called.")      func()      print("  after myfunc() called.")      return func    @deco  def myfunc():      print(" myfunc() called.")    myfunc()  myfunc() |

第四步：使用内嵌包装函数来确保每次新函数都被调用

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | # -\*- coding:gbk -\*-  '''示例4: 使用内嵌包装函数来确保每次新函数都被调用，  内嵌包装函数的形参和返回值与原函数相同，装饰函数返回内嵌包装函数对象'''    def deco(func):      def \_deco():          print("before myfunc() called.")          func()          print("  after myfunc() called.")          # 不需要返回func，实际上应返回原函数的返回值      return \_deco    @deco  def myfunc():      print(" myfunc() called.")      return 'ok'    myfunc()  myfunc() |

第五步：对带参数的函数进行装饰

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | # -\*- coding:gbk -\*-  '''示例5: 对带参数的函数进行装饰，  内嵌包装函数的形参和返回值与原函数相同，装饰函数返回内嵌包装函数对象'''    def deco(func):      def \_deco(a, b):          print("before myfunc() called.")          ret = func(a, b)          print("  after myfunc() called. result: %s" % ret)          return ret      return \_deco    @deco  def myfunc(a, b):      print(" myfunc(%s,%s) called." % (a, b))      return a + b    myfunc(1, 2)  myfunc(3, 4) |

第六步：对参数数量不确定的函数进行装饰

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | # -\*- coding:gbk -\*-  '''示例6: 对参数数量不确定的函数进行装饰，  参数用(\*args, \*\*kwargs)，自动适应变参和命名参数'''    def deco(func):      def \_deco(\*args, \*\*kwargs):          print("before %s called." % func.\_\_name\_\_)          ret = func(\*args, \*\*kwargs)          print("  after %s called. result: %s" % (func.\_\_name\_\_, ret))          return ret      return \_deco    @deco  def myfunc(a, b):      print(" myfunc(%s,%s) called." % (a, b))      return a+b    @deco  def myfunc2(a, b, c):      print(" myfunc2(%s,%s,%s) called." % (a, b, c))      return a+b+c    myfunc(1, 2)  myfunc(3, 4)  myfunc2(1, 2, 3)  myfunc2(3, 4, 5) |

第七步：让装饰器带参数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | # -\*- coding:gbk -\*-  '''示例7: 在示例4的基础上，让装饰器带参数，  和上一示例相比在外层多了一层包装。  装饰函数名实际上应更有意义些'''    def deco(arg):      def \_deco(func):          def \_\_deco():              print("before %s called [%s]." % (func.\_\_name\_\_, arg))              func()              print("  after %s called [%s]." % (func.\_\_name\_\_, arg))          return \_\_deco      return \_deco    @deco("mymodule")  def myfunc():      print(" myfunc() called.")    @deco("module2")  def myfunc2():      print(" myfunc2() called.")    myfunc()  myfunc2() |

第八步：让装饰器带 类 参数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | # -\*- coding:gbk -\*-  '''示例8: 装饰器带类参数'''    class locker:      def \_\_init\_\_(self):          print("locker.\_\_init\_\_() should be not called.")        @staticmethod      def acquire():          print("locker.acquire() called.（这是静态方法）")        @staticmethod      def release():          print("  locker.release() called.（不需要对象实例）")    def deco(cls):      '''cls 必须实现acquire和release静态方法'''      def \_deco(func):          def \_\_deco():              print("before %s called [%s]." % (func.\_\_name\_\_, cls))              cls.acquire()              try:                  return func()              finally:                  cls.release()          return \_\_deco      return \_deco    @deco(locker)  def myfunc():      print(" myfunc() called.")    myfunc()  myfunc() |

第九步：装饰器带类参数，并分拆公共类到其他py文件中，同时演示了对一个函数应用多个装饰器

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | # -\*- coding:gbk -\*-  '''mylocker.py: 公共类 for 示例9.py'''    class mylocker:      def \_\_init\_\_(self):          print("mylocker.\_\_init\_\_() called.")        @staticmethod      def acquire():          print("mylocker.acquire() called.")        @staticmethod      def unlock():          print("  mylocker.unlock() called.")    class lockerex(mylocker):      @staticmethod      def acquire():          print("lockerex.acquire() called.")        @staticmethod      def unlock():          print("  lockerex.unlock() called.")    def lockhelper(cls):      '''cls 必须实现acquire和release静态方法'''      def \_deco(func):          def \_\_deco(\*args, \*\*kwargs):              print("before %s called." % func.\_\_name\_\_)              cls.acquire()              try:                  return func(\*args, \*\*kwargs)              finally:                  cls.unlock()          return \_\_deco      return \_deco |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | # -\*- coding:gbk -\*-  '''示例9: 装饰器带类参数，并分拆公共类到其他py文件中  同时演示了对一个函数应用多个装饰器'''    from mylocker import \*    class example:      @lockhelper(mylocker)      def myfunc(self):          print(" myfunc() called.")        @lockhelper(mylocker)      @lockhelper(lockerex)      def myfunc2(self, a, b):          print(" myfunc2() called.")          return a + b    if \_\_name\_\_=="\_\_main\_\_":      a = example()      a.myfunc()      print(a.myfunc())      print(a.myfunc2(1, 2))      print(a.myfunc2(3, 4)) |

下面是参考资料，当初有不少地方没看明白，真正练习后才明白些：

1. Python装饰器学习 <http://blog.csdn.net/thy38/article/details/4471421>

2. Python装饰器与面向切面编程 <http://www.cnblogs.com/huxi/archive/2011/03/01/1967600.html>

3. Python装饰器的理解 http://apps.hi.baidu.com/share/detail/17572338