装饰器

概念：是一个闭包，把一个函数作为参数然后返回一个替代版函数，本质上就是一个返回函数的函数

作用：不修改原函数的前提下增加函数的功能

简单装饰器

```
def func():
    print("sunck is a good man")

def wrapper(f):
    def inner():
        print("***********")
        f()
    return inner
```

复杂装饰器

```
def wrapper(f):
    def inner(name, age):
        #增加功能
    if age <= 0:
            age = 0
        return f(name, age)
    return inner
```

通用装饰器

```
def wrapper(f):
    def inner(*args, **kwargs):
        #在这增加功能
    print("no zuo no die")
        res = f(*args, **kwargs)
#如果要修改原函数的返回值，在这修改
    return res
    return inner
```

带参数的装饰器

作用：可以控制被装饰函数执行的次数

多嵌套一层外函数便形成了带参数的装饰器

例子：

```
def wrapper(count=3):
    def deco(f):
        def inner(*args, **kwargs):
            for i in range(count):
                f(*args, **kwargs)
        return inner
    return deco
```

多个装饰器

原理：其实就是层层嵌套的关系

多个装饰器同时装饰同一个函数时

装饰时：从距离近的装饰器开始装饰
执行时：从距离远的装饰器内部函数开始执行

@ python2.4支持使用@将装饰器应用在函数上，只需要再函数定义前加上 @装饰器的名称即可

装饰器的使用场景

1、参数、结果检查
2、缓存
3、计数
4、日志
5、统计
6、权限管理
7、重试

也可以计算一个函数的执行次数

例子：

```
def count(f):
    index = 0
    def inner(*args, **kwargs):
        nonlocal index
        index += 1
        res = f(*args, **kwargs)
        print("第%d次执行"%index)
        return res
    return inner
```

retry装饰器

需求：数据库遇到异常后需要重新连接，三次后若还不成功，提示无法完成连接

可以使用装饰器来实现这个需求

例子：

```
def retry(count=3, wait=0,exceptions=(Exception,)):
    import time
    def wrapper(f):
        def inner(*args, **kwargs):
            for i in range(count):
                try:
                    print("-------------------")
                    res = f(*args, **kwargs)
                except exceptions as e:
                    time.sleep(wait)
                    continue
                else:
                    return res
        return inner
    return wrapper

import random

@retry(5)
def connetSQL(ip, port, dbName, passwd):
    num = random.choice([1,2,3,4])
    print("***************", num)
    if num <= 4:
        10 / 0

connetSQL("", "", "", "")
```