# python装饰器以及一些常用的装饰器介绍

这里这个整理一下python中的装饰器的用法,以及在看代码时经常会看到的一些常用装饰器.

## 一. 装饰器

必须记住的几点:

- 装饰器能将被装饰的函数替换为其他的函数(下边例子的**第三行的输出结果**显示)
- **装饰器在加载模块的时候立即执行** (下边例子的**前两行的输出**结果)
- 装饰器一般在一个模块中定义,在另外一个模块中使用 (下方例子的简单示例所示的组织方式)
- 装饰器可以带有参数,这就是**参数化装饰器**,也就是装饰器的工厂函数 (**第二个例子**所示的参数化装饰器)

## 一个简单的装饰器的例子:

```
# 装饰器的定义文件clock_register.py, 装饰器的作用为打印函数运行时间,传入的参数以及调用的结果返回.
import time

def clock(func):
    print('Decroate the function:', func.__name__)
```

```
    def clocked(*args):
        t0 = time.perf_counter()  # 作为初始时间,构成下文中的elapsed(函数运行的消耗时间.)

        result = func(*args)# 被装饰函数的执行

        elapsed = time.perf_counter() - t0

        name = func.__name__  # 装饰器所修饰的函数的名称

        arg_str = ', '.join(repr(arg) for arg in args)  # 传入参数的字符格式

        # 格式化输出,分别打印函数的运行时间, 函数名称, 传入的参数, 函数计算结果
        print('[%0.8fs] %s(%s) -> %r' % (elapsed, name, arg_str, result))
        return result

    return clocked



###### clock装饰器的调用模块
from clock_register import clock
import time

@clock
def snooze(seconds):
    time.sleep(seconds)

@clock
def factorial(n):
    if n < 2:
        return 1
    return  n * factorial(n-1)

if __name__ == '__main__':
    print(snooze)
```

```
    print('*' * 40, 'calling snooze(.123)')
    snooze(.123)
    print('*' * 40, 'calling factorial(6)')
    print('6!=', factorial(6))
```

输出结果:

前两行的输出结果说明:  **装饰器在加载模块时立即执行**

第三行的输出结果显示: 装饰器能将被装饰的函数替换为其他的函数, 此时的snooze函数为clocked函数,传入的`args=0.123`为clock的参数.

```
Decroate the function: snooze
Decroate the function: factorial
<function clock.<locals>.clocked at 0x7f770f855b00>
**************************************** calling snooze(.123)
[0.12307592s] snooze(0.123) -> None
**************************************** calling factorial(6)
[0.00000098s] factorial(1) -> 1
[0.00002348s] factorial(2) -> 2
[0.00003814s] factorial(3) -> 6
[0.00005152s] factorial(4) -> 24
[0.00006555s] factorial(5) -> 120
[0.00008150s] factorial(6) -> 720
6!= 720
```

# 参数化的装饰器的例子

一个简单的例子

```
### 参数化的装饰器

DEFAULT_FMT = '[{elapsed}s] {name}({args}) -> {result}'
def clock_args(fmt = DEFAULT_FMT):
```

```
    def decroate(func):
        print('Decroate the function:', func.__name__)

        def clocked(*args):

            t0 = time.perf_counter()  # 作为初始时间,构成下文中的
elapsed(函数运行的消耗时间.)

            result = func(*args)# 被装饰函数的执行

            elapsed = time.perf_counter() - t0

            name = func.__name__  # 装饰器所修饰的函数的名称

            arg_str = ', '.join(repr(arg) for arg in args)  # 传入参数的字符格式

            # 格式化输出,按照参数化,打印指定的输出格式, 这里采用
**locals()获取局部参数
            print(fmt.format(**locals()))

            return result

        return clocked
    return decroate



####### 参数化装饰器的调用

from clock_register import clock, DEFAULT_FMT, clock_args
import time


@clock_args()
def snooze(seconds):
    time.sleep(seconds)
```

```
@clock_args(fmt='{name}: {elapsed}s')
def snooze1(seconds):
    time.sleep(seconds)


if __name__ == '__main__':
    print(snooze1)
    print('*' * 40, 'calling snooze(.123)')
    snooze(.123)
    print('*' * 40, 'calling snooze(.123)')
    snooze1(.123)
```

最终的输出结果:

```
Decroate the function: snooze
Decroate the function: snooze1
<function clock_args.<locals>.decroate.<locals>.clocked at
0x7f770f84f0e0>
**************************************** calling snooze(.123)
[0.1231747239944525s] snooze((0.123,)) -> None
(base) walle@walle-All-Series:~/123$ python clock_test.py
Decroate the function: snooze
Decroate the function: snooze1
**************************************** calling snooze(.123)
[0.12314573698677123s] snooze((0.123,)) -> None
**************************************** calling snooze(.123)
snooze1: 0.12315590702928603s
```

# 二. 常用的装饰器

# @property

参考这篇文章:@property装饰器的简单理解

# @classmethod与@staticmethod

这两个装饰器在很多的代码中经常会看到,@classmethod叫类方法,@staticmethod叫静态方法,下边以一个简单的例子分析这两个迭代器的用法以及一些区别.

简单的例子:

```
class A:

    def __init__(self):
        pass

    # 实例方法, 约定俗成第一个参数为self, 与实例化对象绑定
    def m1(self, n):
        print("self: ", self)

    # 类方法, 约定俗成第一个参数为 cls, 与类绑定
    @classmethod
    def m2(cls, n):
        print('cls: ', cls)

    # 静态方法(其实就是一个普通的函数,只是位于类中,与类和实例均没有绑定关系)
    @staticmethod
    def m3(n):
        print('this is a static method.')
```

```
a = A()
a.m1(1)
a.m2(1)
A.m2(1)
A.m3(1)
a.m3(1)
m3(1)
```

输出结果为:

```
self:  <__main__.A object at 0x7f9a9eb50e90>
cls:  <class '__main__.A'>
cls:  <class '__main__.A'>
this is a static method.
this is a static method.
Traceback (most recent call last):
  File "classmethod_staticmethod.py", line 24, in <module>
    m3(1)
NameError: name 'm3' is not defined
```

依据以上结果可以看出:

```
a = A()   # 实例化一个实例对象
a.m1(1)   # m1是一个实例方法,通过实例a调用
a.m2(1)  # m2是一个类方法,通过实例a找到类A,然后通过A调用m2
A.m2(1)  # 利用类直接调用类方法.
A.m3(1)  # 静态方法,既可以直接使用类A调用,也可以使用实例a调用
a.m3(1)
m3(1)   # 不可直接调用
```

# @functools.lru_cache

学操作系统的时候lru这个东西经常出现,在leetcode上也有实现lru的这个题目,这个装饰器可以实现这种功能.

例如一个求斐波那契数列的函数:

```
@clock
def fibonacci(n):
   if n < 2:
      return n
   return  fibonacci(n-2) + fibonacci(n-1)


if __name__ == '__main__':
   print(fibonacci(6))
```

结果为:

```
[0.00000133s] fibonacci(0) -> 0
[0.00000093s] fibonacci(1) -> 1
[0.00004139s] fibonacci(2) -> 1
[0.00000071s] fibonacci(1) -> 1
[0.00000082s] fibonacci(0) -> 0
[0.00000069s] fibonacci(1) -> 1
[0.00002819s] fibonacci(2) -> 1
[0.00005552s] fibonacci(3) -> 2
[0.00012448s] fibonacci(4) -> 3
[0.00000067s] fibonacci(1) -> 1
[0.00000065s] fibonacci(0) -> 0
[0.00000071s] fibonacci(1) -> 1
[0.00002738s] fibonacci(2) -> 1
[0.00005426s] fibonacci(3) -> 2
[0.00000063s] fibonacci(0) -> 0
[0.00000068s] fibonacci(1) -> 1
[0.00002711s] fibonacci(2) -> 1
[0.00000063s] fibonacci(1) -> 1
```

```
[0.00000079s] fibonacci(0) -> 0
[0.00000068s] fibonacci(1) -> 1
[0.00002817s] fibonacci(2) -> 1
[0.00005480s] fibonacci(3) -> 2
[0.00010871s] fibonacci(4) -> 3
[0.00018944s] fibonacci(5) -> 5
[0.00034368s] fibonacci(6) -> 8
8
```

如果使用lru装饰器:

```
@functools.lru_cache()
@clock
def fibonacci(n):
   if n < 2:
      return n
   return  fibonacci(n-2) + fibonacci(n-1)
```

这个装饰器实现了LRU(备忘录)的功能,减少了重复计算.

结果为:

```
[0.00000118s] fibonacci(0) -> 0
[0.00000085s] fibonacci(1) -> 1
[0.00004799s] fibonacci(2) -> 1
[0.00000145s] fibonacci(3) -> 2
[0.00007905s] fibonacci(4) -> 3
[0.00000122s] fibonacci(5) -> 5
[0.00011034s] fibonacci(6) -> 8
8
```

其中有**装饰器的叠放**的操作,装饰器的叠放的简单例子:

```
@d1
@d2
def f():
    print('f')


等价于:
def f():
    print('f')
f = d1(d2(f))
```