恶补了 Python 装饰器的八种写法，你随便问~

王炳明
公众号：Python编程时光，致力分享 Python优质干货

对于每一个学习 Python 的同学，想必对 `@` 符号一定不陌生了，正如你所知， @ 符号是装饰器的语法糖，@符号后面的函数就是我们本文的主角：**装饰器。**

装饰器放在一个函数开始定义的地方，它就像一顶帽子一样戴在这个函数的头上。和这个函数绑定在一起。在我们调用这个函数的时候，第一件事并不是执行这个函数，而是将这个函数做为参数传入它头顶上这顶帽子，这顶帽子我们称之为 `装饰器` 。

1. Hello，装饰器

装饰器的使用方法很固定

先定义一个装饰器（帽子）
再定义你的业务函数或者类（人）
最后把这装饰器（帽子）扣在这个函数（人）头上

就像下面这样子

```
def decorator(func):
    def wrapper(*args, **kw):
        return func()
    return wrapper

@decorator
def function():
    print("hello, decorator")
```

实际上，装饰器并不是编码必须性，意思就是说，你不使用装饰器完全可以，它的出现，应该是使我们的代码

更加优雅，代码结构更加清晰
将实现特定的功能代码封装成装饰器，提高代码复用率，增强代码可读性

接下来，我将以实例讲解，如何编写出各种简单及复杂的装饰器。

2. 入门：日志打印器

首先是**日志打印器。** 实现的功能：

在函数执行前，先打印一行日志告知一下主人，我要执行函数了。
在函数执行完，也不能拍拍屁股就走人了，咱可是有礼貌的代码，再打印一行日志告知下主人，我执行完啦。

```
# 这是装饰器函数，参数 func 是被装饰的函数
def logger(func):
    def wrapper(*args, **kw):
        print('主人，我准备开始执行：{} 函数了:'.format(func.__name__))

        # 真正执行的是这行。
        func(*args, **kw)

        print('主人，我执行完啦。')
    return wrapper
```

假如，我的业务函数是，计算两个数之和。写好后，直接给它带上帽子。

```
@logger
def add(x, y):
    print('{} + {} = {}'.format(x, y, x+y))
```

然后执行一下 add 函数。

```
add(200, 50)
```

来看看输出了什么?

```
主人，我准备开始执行：add 函数了:
200 + 50 = 250
主人，我执行完啦。
```

3. 入门：时间计时器

再来看看 **时间计时器** 实现功能：顾名思义，就是计算一个函数的执行时长。

```
# 这是装饰函数
def timer(func):
    def wrapper(*args, **kw):
        t1=time.time()
        # 这是函数真正执行的地方
        func(*args, **kw)
        t2=time.time()
```

```
        # 计算下时长
        cost_time = t2-t1
        print("花费时间：{}秒".format(cost_time))
    return wrapper
```

假如，我们的函数是要睡眠10秒。这样也能更好的看出这个计算时长到底靠不靠谱。

```
import time

@timer
def want_sleep(sleep_time):
    time.sleep(sleep_time)

want_sleep(10)
```

来看看输出，如预期一样，输出10秒。

```
花费时间：10.0073800086975098秒
```

4. 进阶：带参数的函数装饰器

通过上面两个简单的入门示例，你应该能体会到装饰器的工作原理了。

不过，装饰器的用法还远不止如此，深究下去，还大有文章。今天就一起来把这个知识点学透。

回过头去看看上面的例子，装饰器是不能接收参数的。其用法，只能适用于一些简单的场景。不传参的装饰器，只能对被装饰函数，执行固定逻辑。

装饰器本身是一个函数，做为一个函数，如果不能传参，那这个函数的功能就会很受限，只能执行固定的逻辑。这意味着，如果装饰器的逻辑代码的执行需要根据不同场景进行调整，若不能传参的话，我们就要写两个装饰器，这显然是不合理的。

比如我们要实现一个可以定时发送邮件的任务（一分钟发送一封），定时进行时间同步的任务（一天同步一次），就可以自己实现一个 periodic_task （定时任务）的装饰器，这个装饰器可以接收一个时间间隔的参数，间隔多长时间执行一次任务。

可以这样像下面这样写，由于这个功能代码比较复杂，不利于学习，这里就不贴了。

```
@periodic_task(spacing=60)
def send_mail():
     pass
```

```
@periodic_task(spacing=86400)
def ntp()
    pass
```

那我们来自己创造一个伪场景，可以在装饰器里传入一个参数，指明国籍，并在函数执行前，用自己国家的母语打一个招呼。

```
# 小明，中国人
@say_hello("china")
def xiaoming():
    pass

# jack，美国人
@say_hello("america")
def jack():
    pass
```

那我们如果实现这个装饰器，让其可以实现 `传参` 呢?

会比较复杂，需要两层嵌套。

```
def say_hello(contry):
    def wrapper(func):
        def deco(*args, **kwargs):
            if contry == "china":
                print("你好!")
            elif contry == "america":
                print('hello.')
            else:
                return

            # 真正执行函数的地方
            func(*args, **kwargs)
        return deco
    return wrapper
```

来执行一下

```
xiaoming()
print("------------")
jack()
```

看看输出结果。

```
你好!
------------
hello.
```

5. 高阶：不带参数的类装饰器

以上都是基于函数实现的装饰器，在阅读别人代码时，还可以时常发现还有基于类实现的装饰器。

基于类装饰器的实现，必须实现 `__call__` 和 `__init__` 两个内置函数。 `__init__` ：接收被装饰函数 `__call__` ：实现装饰逻辑。

还是以日志打印这个简单的例子为例

```
class logger(object):
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print("[INFO]: the function {func}() is running..."\
            .format(func=self.func.__name__))
        return self.func(*args, **kwargs)

@logger
def say(something):
    print("say {}!".format(something))

say("hello")
```

执行一下，看看输出

```
[INFO]: the function say() is running...
say hello!
```

6. 高阶：带参数的类装饰器

上面不带参数的例子，你发现没有，只能打印 `INFO` 级别的日志，正常情况下，我们还需要打印 `DEBUG` `WARNING` 等级别的日志。 这就需要给类装饰器传入参数，给这个函数指定级别了。

带参数和不带参数的类装饰器有很大的不同。

`__init__` ：不再接收被装饰函数，而是接收传入参数。 `__call__` ：接收被装饰函数，实现装饰逻辑。

```
class logger(object):
    def __init__(self, level='INFO'):
        self.level = level

    def __call__(self, func): # 接受函数
        def wrapper(*args, **kwargs):
            print("[{level}]: the function {func}() is running..."\
                .format(level=self.level, func=func.__name__))
            func(*args, **kwargs)
        return wrapper  #返回函数

@logger(level='WARNING')
def say(something):
    print("say {}!".format(something))

say("hello")
```

我们指定 `WARNING` 级别，运行一下，来看看输出。

```
[WARNING]: the function say() is running...
say hello!
```

7. 使用偏函数与类实现装饰器

绝大多数装饰器都是基于函数和闭包实现的，但这并非制造装饰器的唯一方式。

事实上，Python 对某个对象是否能通过装饰器（ `@decorator` ）形式使用只有一个要求：**decorator 必须是一个“可被调用（callable）的对象。**

对于这个 callable 对象，我们最熟悉的就是函数了。

除函数之外，类也可以是 callable 对象，只要实现了 `__call__` 函数（上面几个例子已经接触过了）。

还有容易被人忽略的偏函数其实也是 callable 对象。

接下来就来说说，如何使用 类和偏函数结合实现一个与众不同的装饰器。

如下所示，DelayFunc 是一个实现了 `__call__` 的类，delay 返回一个偏函数，在这里 delay 就可以做为一个装饰器。（以下代码摘自 Python工匠：使用装饰器的小技巧）

```
import time
import functools

class DelayFunc:
```

```
    def __init__(self,  duration, func):
        self.duration = duration
        self.func = func

    def __call__(self, *args, **kwargs):
        print(f'Wait for {self.duration} seconds...')
        time.sleep(self.duration)
        return self.func(*args, **kwargs)

    def eager_call(self, *args, **kwargs):
        print('Call without delay')
        return self.func(*args, **kwargs)

def delay(duration):
    """
    装饰器：推迟某个函数的执行。
    同时提供 .eager_call 方法立即执行
    """
    # 此处为了避免定义额外函数，
    # 直接使用 functools.partial 帮助构造 DelayFunc 实例
    return functools.partial(DelayFunc, duration)
```

我们的业务函数很简单，就是相加

```
@delay(duration=2)
def add(a, b):
    return a+b
```

来看一下执行过程

```
>>> add    # 可见 add 变成了 Delay 的实例
<__main__.DelayFunc object at 0x107bd0be0>
>>>
>>> add(3,5)  # 直接调用实例，进入 __call__
Wait for 2 seconds...
8
>>>
>>> add.func # 实现实例方法
<function add at 0x107bef1e0>
```

8. 如何写能装饰类的装饰器?

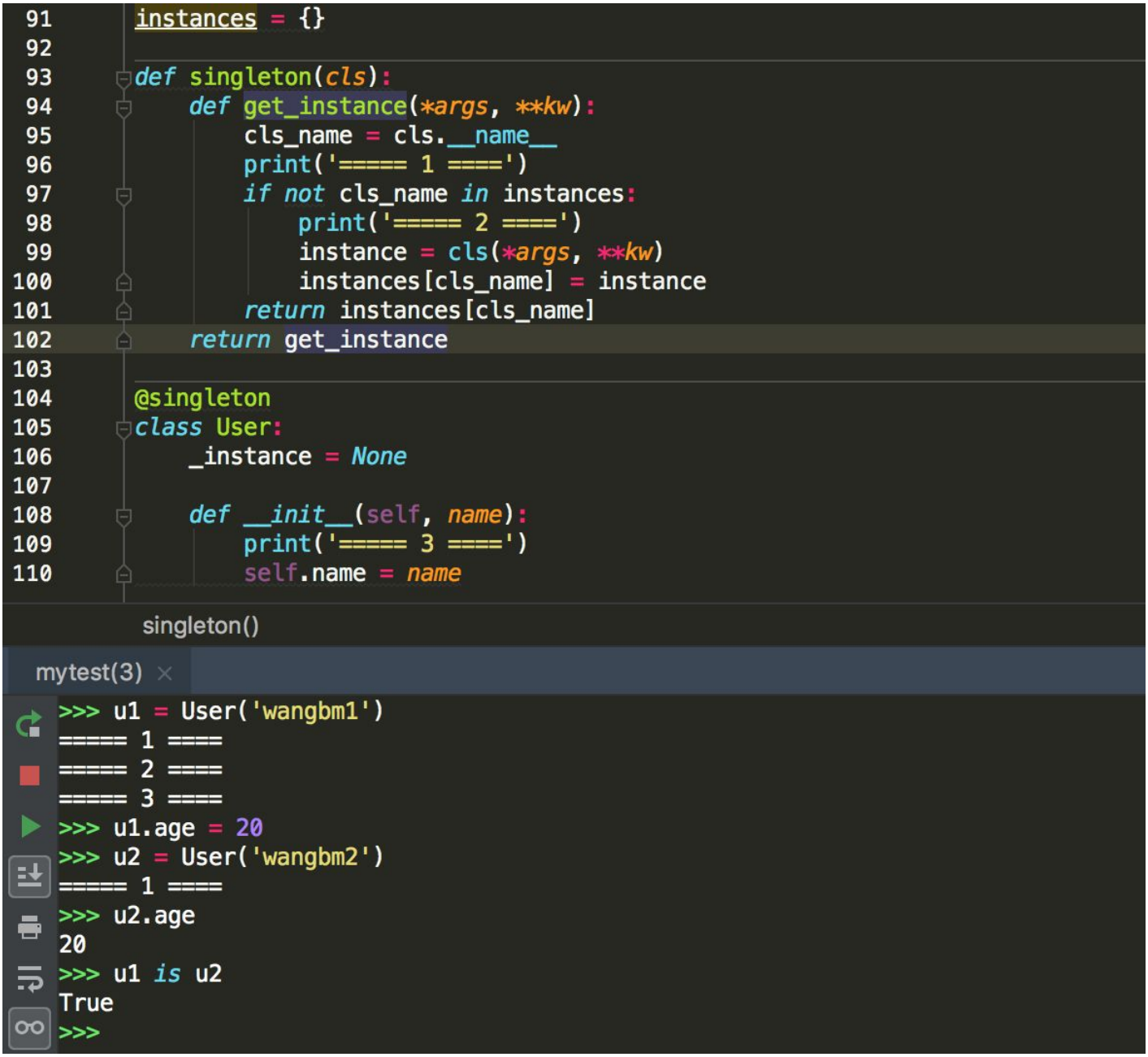
用 Python 写单例模式的时候，常用的有三种写法。其中一种，是用装饰器来实现的。

以下便是我自己写的装饰器版的单例写法。

```
instances = {}

def singleton(cls):
    def get_instance(*args, **kw):
        cls_name = cls.__name__
        print('===== 1 ====')
        if not cls_name in instances:
            print('===== 2 ====')
            instance = cls(*args, **kw)
            instances[cls_name] = instance
        return instances[cls_name]
    return get_instance

@singleton
class User:
    _instance = None

    def __init__(self, name):
        print('===== 3 ====')
        self.name = name
```

可以看到我们用singleton 这个装饰函数来装饰 User 这个类。装饰器用在类上，并不是很常见，但只要熟悉装饰器的实现过程，就不难以实现对类的装饰。在上面这个例子中，装饰器就只是实现对类实例的生成的控制而已。

其实例化的过程，你可以参考我这里的调试过程，加以理解。



9. wraps 装饰器有啥用?

在 functools 标准库中有提供一个 wraps 装饰器，你应该也经常见过，那他有啥用呢?

先来看一个例子

```
def wrapper(func):
    def inner_function():
        pass
    return inner_function

@wrapper
def wrapped():
    pass
```

```
print(wrapped.__name__)
#inner_function
```

为什么会这样子？不是应该返回 `func` 吗？

这也不难理解，因为上边执行 `func` 和下边 `decorator(func)` 是等价的，所以上面 `func.__name__` 是等价于下面 `decorator(func).__name__` 的，那当然名字是 `inner_function`

```
def wrapper(func):
    def inner_function():
        pass
    return inner_function

def wrapped():
    pass

print(wrapper(wrapped).__name__)
#inner_function
```

那如何避免这种情况的产生？方法是使用 functools .wraps 装饰器，它的作用就是将 **被修饰的函数(wrapped)** 的一些属性值赋值给 **修饰器函数(wrapper)**，最终让属性的显示更符合我们的直觉。

```
from functools import wraps

def wrapper(func):
    @wraps(func)
    def inner_function():
        pass
    return inner_function

@wrapper
def wrapped():
    pass

print(wrapped.__name__)
# wrapped
```

准确点说，wraps 其实是一个偏函数对象（partial），源码如下

```
def wraps(wrapped,
          assigned = WRAPPER_ASSIGNMENTS,
          updated = WRAPPER_UPDATES):
    return partial(update_wrapper, wrapped=wrapped,
                   assigned=assigned, updated=updated)
```

可以看到wraps其实就是调用了一个函数 `update_wrapper` ，知道原理后，我们改写上面的代码，在不使用wraps的情况下，也可以让 `wrapped.__name__` 打印出 wrapped，代码如下：

```
from functools import update_wrapper

WRAPPER_ASSIGNMENTS = ('__module__', '__name__', '__qualname__', '__doc__',
                       '__annotations__')

def wrapper(func):
    def inner_function():
        pass

    update_wrapper(inner_function, func, assigned=WRAPPER_ASSIGNMENTS)
    return inner_function

@wrapper
def wrapped():
    pass

print(wrapped.__name__)
```

10. 内置装饰器：property

以上，我们介绍的都是自定义的装饰器。

其实Python语言本身也有一些装饰器。比如 `property` 这个内建装饰器，我们再熟悉不过了。

它通常存在于类中，可以将一个函数定义成一个属性，属性的值就是该函数return的内容。

通常我们给实例绑定属性是这样的

```
class Student(object):
    def __init__(self, name, age=None):
        self.name = name
        self.age = age

# 实例化
xiaoming = Student("小明")

# 添加属性
xiaoming.age=25

# 查询属性
xiaoming.age
```

```
# 删除属性
del xiaoming.age
```

但是稍有经验的开发人员，一下就可以看出，这样直接把属性暴露出去，虽然写起来很简单，但是并不能对属性的值做合法性限制。为了实现这个功能，我们可以这样写。

```
class Student(object):
    def __init__(self, name):
        self.name = name
        self.name = None

    def set_age(self, age):
        if not isinstance(age, int):
            raise ValueError('输入不合法：年龄必须为数值!')
        if not 0 < age < 100:
            raise ValueError('输入不合法：年龄范围必须0-100')
        self._age=age

    def get_age(self):
        return self._age

    def del_age(self):
        self._age = None


xiaoming = Student("小明")

# 添加属性
xiaoming.set_age(25)

# 查询属性
xiaoming.get_age()

# 删除属性
xiaoming.del_age()
```

上面的代码设计虽然可以变量的定义，但是可以发现不管是获取还是赋值（通过函数）都和我们平时见到的不一样。 按照我们思维习惯应该是这样的。

```
# 赋值
xiaoming.age = 25

# 获取
xiaoming.age
```

那么这样的方式我们如何实现呢。请看下面的代码。

```
class Student(object):
    def __init__(self, name):
        self.name = name
        self.name = None

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if not isinstance(value, int):
            raise ValueError('输入不合法：年龄必须为数值!')
        if not 0 < value < 100:
            raise ValueError('输入不合法：年龄范围必须0-100')
        self._age=value

    @age.deleter
    def age(self):
        del self._age

xiaoming = Student("小明")

# 设置属性
xiaoming.age = 25

# 查询属性
xiaoming.age

# 删除属性
del xiaoming.age
```

用 `@property` 装饰过的函数，会将一个函数定义成一个属性，属性的值就是该函数return的内容。**同时**，会将这个函数变成另外一个装饰器。就像后面我们使用的 `@age.setter` 和 `@age.deleter` 。

`@age.setter` 使得我们可以使用 `XiaoMing.age = 25` 这样的方式直接赋值。 `@age.deleter` 使得我们可以使用 `del XiaoMing.age` 这样的方式来删除属性。

property 的底层实现机制是「描述符」，为此我还写过一篇文章。

这里也介绍一下吧，正好将这些看似零散的文章全部串起来。

如下，我写了一个类，里面使用了 property 将 math 变成了类实例的属性

```
class Student:
    def __init__(self, name):
        self.name = name
```

```
    @property
    def math(self):
        return self._math

    @math.setter
    def math(self, value):
        if 0 <= value <= 100:
            self._math = value
        else:
            raise ValueError("Valid value must be in [0, 100]")
```

为什么说 property 底层是基于描述符协议的呢？通过 PyCharm 点击进入 property 的源码，很可惜，只是一份类似文档一样的伪源码，并没有其具体的实现逻辑。

不过，从这份伪源码的魔法函数结构组成，可以大体知道其实现逻辑。

这里我自己通过模仿其函数结构，结合「描述符协议」来自己实现类 `property` 特性。

代码如下：

```
class TestProperty(object):

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        print("in __get__")
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError
        return self.fget(obj)

    def __set__(self, obj, value):
        print("in __set__")
        if self.fset is None:
            raise AttributeError
        self.fset(obj, value)

    def __delete__(self, obj):
        print("in __delete__")
        if self.fdel is None:
            raise AttributeError
```

```
        self.fdel(obj)


    def getter(self, fget):
        print("in getter")
        return type(self)(fget, self.fset, self.fdel, self.__doc__)

    def setter(self, fset):
        print("in setter")
        return type(self)(self.fget, fset, self.fdel, self.__doc__)

    def deleter(self, fdel):
        print("in deleter")
        return type(self)(self.fget, self.fset, fdel, self.__doc__)
```

然后 Student 类，我们也相应改成如下

```
class Student:
    def __init__(self, name):
        self.name = name

    # 其实只有这里改变
    @TestProperty
    def math(self):
        return self._math

    @math.setter
    def math(self, value):
        if 0 <= value <= 100:
            self._math = value
        else:
            raise ValueError("Valid value must be in [0, 100]")
```

为了尽量让你少产生一点疑惑，我这里做两点说明：

使用 `TestProperty` 装饰后， `math` 不再是一个函数，而是 `TestProperty` 类的一个实例。所以第二个math函数可以使用 `math.setter` 来装饰，本质是调用 `TestProperty.setter` 来产生一个新的 `TestProperty` 实例赋值给第二个 `math` 。
第一个 `math` 和第二个 `math` 是两个不同 `TestProperty` 实例。但他们都属于同一个描述符类（TestProperty），当对 math 对于赋值时，就会进入 `TestProperty.__set__` ，当对math 进行取值里，就会进入 `TestProperty.__get__` 。仔细一看，其实最终访问的还是Student实例的 `_math` 属性。

说了这么多，还是运行一下，更加直观一点。

```
# 运行后，会直接打印这一行，这是在实例化 TestProperty 并赋值给第二个math
in setter
```

```
>>>
>>> s1.math = 90
in __set__
>>> s1.math
in __get__
90
```

如对上面代码的运行原理，有疑问的同学，请务必结合上面两点说明加以理解，那两点相当关键。

11. 其他装饰器：装饰器实战

读完并理解了上面的内容，你可以说是Python高手了。别怀疑，自信点，因为很多人都不知道装饰器有这么多用法呢。

在我看来，使用装饰器，可以达到如下目的：

使代码可读性更高，逼格更高；
代码结构更加清晰，代码冗余度更低；

刚好我在最近也有一个场景，可以用装饰器很好的实现，暂且放上来看看。

这是一个实现控制函数运行超时的装饰器。如果超时，则会抛出超时异常。

有兴趣的可以看看。

```
import signal

class TimeoutException(Exception):
    def __init__(self, error='Timeout waiting for response from Cloud'):
        Exception.__init__(self, error)


def timeout_limit(timeout_time):
    def wraps(func):
        def handler(signum, frame):
            raise TimeoutException()

        def deco(*args, **kwargs):
            signal.signal(signal.SIGALRM, handler)
            signal.alarm(timeout_time)
            func(*args, **kwargs)
            signal.alarm(0)
        return deco
    return wraps
```

非常感谢你能阅读到这里，这篇文章我写了很久，算是比较干货的那种，文章有些长，但还是希望花点时间把这些知识点都搞明白，而不要只是收藏。