



# Python程序设计：三器语法



南开大学软件学院

王斌辉

## ■ “三器” 语法

- 装饰器 Decorator
- 迭代器 Iterator
- 生成器 Generator

## ■ 装饰器 Decorator

```
def func():  
    print("Hello world!")
```

```
func()
```

## ■ 装饰器 Decorator

```
import time
```

```
def func():  
    start = time.time()  
    print("Hello world!")  
    end = time.time()  
    return end - start
```

```
func()
```



## ■ 装饰器 Decorator

```
import time
```

```
def func():  
    print("Hello world!")
```

```
def time_counter(fn):  
    start = time.time()  
    fn()  
    end = time.time()  
    return end - start
```

```
time_counter(func)
```

```
def func_1():  
    pass
```

```
def func_2():  
    pass
```

```
def func_3():  
    pass
```

```
def func_4():  
    pass
```



## ■ 装饰器 Decorator



```
import time

def func():
    print("Hello world!")

def time_counter(fn):
    def wrapper():
        start = time.time()
        fn()
        end = time.time()
        return end - start

    return wrapper

func = time_counter(func)
func()
```

## ■ 装饰器 Decorator

**装饰器**是一种常见的设计模式，经常被用于有切面需求的场景，较为经典的应用有**插入日志**、**增加计时逻辑来检测性能**、**加入事务处理**等。装饰器是解决这类问题的绝佳设计，有了装饰器，我们可以抽离出大量函数中与函数功能本身无关的代码并继续重用。一言蔽之，装饰器的作用是为已经存在的对象添加额外的功能。

```
import time
def time_counter(fn):
    def wrapper():
        start = time.time()
        fn()
        end = time.time()
        return end - start

    return wrapper

@time_counter
def func():
    print("Hello world!")

func = time_counter(func)
func()
```

## ■ 装饰器 Decorator

```
def func_1():
    pass
```

```
def func_2():
    pass
```

```
def func_3():
    pass
```

```
def func_4():
    pass
```

**问题：**如何利用装饰器为这些函数加入日志功能？

```
from datetime import datetime, timezone
```

```
def logger(fn):
    def inner(*args, **kwargs):
        called_at = datetime.now()
        to_execute = fn(*args, **kwargs)
        print(f'{fn.__name__} executed. Logged at {called_at}')
        return to_execute
    return inner
```

```
@logger
def func_1():
    pass

def func_1():
    pass

def func_4():
    pass

def func_2():
    pass

def func_3():
    pass
```

```
func_1 executed. Logged at 2022-10-21 14:49:06.637438
func_4 executed. Logged at 2022-10-21 14:49:06.637438
func_2 executed. Logged at 2022-10-21 14:49:06.637438
func_3 executed. Logged at 2022-10-21 14:49:06.637438
```



## ■ 装饰器 Decorator

```
def foo(x):  
    sum_ = 0  
    for i in range(x):  
        sum_ += i ** 2  
    return sum_  
  
def foo(x):  
    return sum([i ** 2 for i in range(x)])
```

```
foo(10000)
```

## ■ 装饰器 Decorator

```
import time

def foo(x):
    start = time.time()
    sum_ = 0
    for i in range(x):
        sum_ += i ** 2
    end = time.time()
    exe_time = end - start
    return sum_, exe_time

foo(10000)
```

## ■ 装饰器 Decorator

```
import time

def foo(x):
    sum_ = 0
    for i in range(x):
        sum_ += i ** 2
    return sum_

def time_counter(func, x):
    start = time.time()
    func(x)
    end = time.time()
    return end - start

time_counter(foo, 10000)
```

## ■ 装饰器 Decorator

```
import time

def foo(x):
    sum_ = 0
    for i in range(x):
        sum_ += i ** 2
    return sum_

def time_counter(func):
    def wrapper(x):
        start = time.time()
        result = func(x)
        end = time.time()
        return result, end - start

    return wrapper

foo = time_counter(foo)
foo(10000)
```

## ■ 装饰器 Decorator

```
import time
def time_counter(func):
    def wrapper(x):
        start = time.time()
        result = func(x)
        end = time.time()
        return result, end - start

    return wrapper

@time_counter
def foo(x):
    sum_ = 0
    for i in range(x):
        sum_ += i ** 2
    return sum_

foo = time_counter(foo)
foo(10000)
```



## ■ 装饰器 Decorator

### — 带参数装饰器

```
@decorator(args)
def func():
    pass
```

```
func = decorator(arg)(func)
```

带有参数的装饰器，其实是在装饰器函数的外面又包裹了一个函数，使用该函数接收参数，返回的是装饰器函数。

# 添加输出日志的功能

```
def logging(flag):
    def decorator(fn):
        def inner(num1, num2):
            if flag == "+":
                print("--正在努力加法计算--")
            elif flag == "-":
                print("--正在努力减法计算--")
            result = fn(num1, num2)
            return result

        return inner

    return decorator
```

```
@logging("+")
def add(a, b):
    result = a + b
    return result
```

```
@logging("-")
def sub(a, b):
    result = a - b
    return result
```

```
add(1, 2)
sub(1, 2)
```



## ■ 装饰器 Decorator

### — 多个装饰器

```
@decorator_one
@decorator_two
def func():
    pass
```

```
func = decorator_one(decorator_two(func))
```

```
def make_bold(fn):
    def wrapped():
        return "<b>" + fn() + "</b>"

    return wrapped
```

```
def make_italic(fn):
    def wrapped():
        return "<i>" + fn() + "</i>"

    return wrapped
```

```
@make_bold
@make_italic
def hello():
    return "hello world"
```

```
hello() # returns <b><i>hello world</i></b>
```

## ■ 装饰器 Decorator

### — 多个带参数函数装饰器

```
def make_html_tag(tag, *args, **kwargs):  
    def real_decorator(fn):  
        css_class = " class='{0}'".format(kwargs["css_class"]) if "css_class" in kwargs else ""  
  
        def wrapped(*args, **kwargs):  
            return "<" + tag + css_class + ">" + fn(*args, **kwargs) + "</" + tag + ">"  
  
        return wrapped  
  
    return real_decorator
```

```
@make_html_tag(tag="b", css_class="bold_css")  
@make_html_tag(tag="i", css_class="italic_css")  
def func(x):  
    return x
```

```
func("Hello world!")    <b class='bold_css'><i class='italic_css'>Hello world!</i></b>
```



## ■ 装饰器 Decorator

### — 类装饰器

```
inside Decorator.__init__()
Finished decorating func()
inside func()
inside Decorator.__call__()
```

```
class Decorator:
```

```
    def __init__(self, fn):
        print("inside Decorator.__init__()")
        self.fn = fn
```

```
    def __call__(self):
        self.fn()
        print("inside Decorator.__call__()")
```

```
@Decorator
```

```
def func():
    print("inside func()")
```

```
print("Finished decorating func()")
func()
```

## ■ 装饰器 Decorator

### — 多个带参数类装饰器

```
class MakeHtmlTag:
```

```
    def __init__(self, tag, css_class=""):
        self._tag = tag
        self._css_class = " class='{0}'".format(css_class) \
            if css_class != "" else ""

    def __call__(self, fn):
        def wrapped(*args, **kwargs):
            return "<" + self._tag + self._css_class + ">" \
                + fn(*args, **kwargs) + "</" + self._tag + ">"

        return wrapped
```

```
@MakeHtmlTag(tag="b", css_class="bold_css")
@MakeHtmlTag(tag="i", css_class="italic_css")
def func(name):
    return "Hello, {}".format(name)
```

```
func("Wang")
```

```
<b class='bold_css'><i class='italic_css'>Hello, Wang</i></b>
```

## ■ 装饰器 Decorator

### — 装饰类

```
def refac_str(cls):  
    def __str__(self):  
        return str(self.__dict__)  
  
    cls.__str__ = __str__  
    return cls
```

```
@refac_str  
class MyClass:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

`MyClass(1, 2)`

`{'x': 1, 'y': 2}`

## ■ 装饰器 Decorator

### — 装饰器的副作用

func.__name__	函数的名字
func.__dict__	函数的属性字典
func.__defaults__	函数的默认参数元组
func.__module__	函数所属模块名
func.__doc__	函数的文档（字符串）
.....	

使用装饰器后，再通过函数名访问上述属性时，原来函数的属性数据变为了装饰器函数的属性数据。如何解决属性丢失的问题？

```
import time

def time_counter(func):
    def wrapper(x):
        start = time.time()
        result = func(x)
        end = time.time()
        return result, end - start

    return wrapper

@time_counter
def foo(x):
    sum_ = 0
    for i in range(x):
        sum_ += i ** 2
    return sum_

print(foo.__name__)          wrapper
```

## ■ 装饰器 Decorator

### — 装饰器的副作用

functools 的 wraps, 可用于装饰器的内层函数, 抵消装饰器的副作用, 因为使用装饰器后, 原函数被内层函数赋值覆盖, 函数名称等信息丢失了(装饰器仅仅是不改变函数原有功能)。然即使wraps, 也无法完全消除装饰器的副作用。

```
import time
from functools import wraps
```

```
def time_counter(func):
    @wraps(func)
    def wrapper(x):
        start = time.time()
        result = func(x)
        end = time.time()
        return result, end - start

    return wrapper
```

```
@time_counter
def foo(x):
    sum_ = 0
    for i in range(x):
        sum_ += i ** 2
    return sum_
```

```
print(foo.__name__)
```

```
import time
from functools import update_wrapper
```

```
def time_counter(func):
    def wrapper(x):
        start = time.time()
        result = func(x)
        end = time.time()
        return result, end - start

    return update_wrapper(wrapper, func)
```

```
@time_counter
def foo(x):
    sum_ = 0
    for i in range(x):
        sum_ += i ** 2
    return sum_
```

```
print(foo.__name__)
```

## ■ 装饰器 Decorator

### — 装饰器的应用案例

- ✧ Decorators are extensively used in **access control & authentication** in Django using @login\_required decorators.
- ✧ Decorators are used in **timing** function
- ✧ Decorators are very applicable in **Logging**
- ✧ Decorators are used for **Memoization while Caching**
- ✧ Decorators are used in **rate limiting of an API** (Applicable Programming Interface)
- ✧ Python decorators are used in designing **single-dispatch generic function**

## ■ 装饰器 Decorator

### — 装饰器的应用案例

✧ Decorators are used for **Memoization while Caching**

```
def fib(n):  
    print(f'calculating term - {n}')
```

```
    if n < 3:  
        return 1  
    else:  
        return fib(n - 1) + fib(n - 2)
```

```
print('6th term is: ', fib(6))
```

```
calculating term - 6  
calculating term - 5  
calculating term - 4  
calculating term - 3  
calculating term - 2  
calculating term - 1  
calculating term - 2  
calculating term - 3  
calculating term - 2  
calculating term - 1  
calculating term - 4  
calculating term - 3  
calculating term - 2  
calculating term - 1  
calculating term - 2  
6th term is: 8
```



## ■ 装饰器 Decorator

```
def memoizer(fn):
    # first and second term of Fibonacci series
    cache = {}
```

```
    def inner(n):
        if n not in cache:
            cache[n] = fn(n)
        return cache[n]
```

```
    return inner
```

```
@memoizer
```

```
def fib(n):
    print(f'calculating - {n}')
    if n < 3:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

```
print('-' * 16)
print('5th term: ', fib(5))
print('-' * 16)
print('10th term: ', fib(10))
print('-' * 16)
print('12th term: ', fib(12))
print('-' * 16)
print('8th term: ', fib(8))
```

```
-----
calculating - 5
calculating - 4
calculating - 3
calculating - 2
calculating - 1
5th term:  5
```

```
-----
calculating - 10
calculating - 9
calculating - 8
calculating - 7
calculating - 6
10th term:  55
```

```
-----
calculating - 12
calculating - 11
12th term:  144
-----
8th term:   21
```



