17. Decorators

Decorators có lẽ là một design pattern đẹp nhất và mạnh mẽ nhất trong Python, nhưng đồng thời khái niệm này được nhiều người cho là phức tạp. Chính xác, việc sử dụng decorators rất dễ dàng, nhưng việc viết decorator có thể trở nên phức tạp nếu bạn không có kinh nghiệm với decorator và một số khái niệm functional program.

Mặc dù khái niệm cơ bản giống nhau, Python có hai loại decorator khác nhau:

Function decorators

Class decorators

Một decorator trong Python là bất kỳ đối tượng Python callable nào được sử dụng để sửa đổi một hàm hay một lớp. Một tham chiếu đến một hàm "func" hoặc một lớp "C" được truyền cho một decorator và decorator sẽ trả về một hàm hay lớp được sửa đổi. Các hàm hoặc lớp sửa đổi thường chứa các lời gọi đến hàm gốc "func" hoặc lớp "C". Chú ý “C” là class.

Bước đầu để decorator

Để giới thiệu decorator dễ hình dung nhất, chúng tôi sẽ giới thiệu decorator bằng cách lặp lại một số khía cạnh quan trọng của các hàm. Đầu tiên bạn phải biết hoặc nhớ rằng các tên hàm là các tham chiếu đến các hàm và chúng ta có thể gán nhiều tên cho cùng một hàm:

```
def add(x, y):
    return x + y

add_negtive = add
add_positive = add
print "add_negtive(-1,-3) = ", add_negtive(-1, -3)
print "add_positive(3,4) = ", add_positive(3, 4)

Output:

add_negtive(-1,-3) =  -4

add_positive(3,4) =  7
```

Điều này có nghĩa là chúng tôi có ba tên, tức là "add", "add_ negtive" và “add_ positive” cho cùng một chức năng. Thực tế, chúng ta có thể xoá "add" hoặc "add_ negtive" hoặc “add_ positive” mà không xoá chức năng của chính nó.

```
def add(x, y):
    return x + y
add_negtive = add
add_positive = add
print "add_negtive(-1,-3) = ", add_negtive(-1, -3)
print "add_positive(3,4) = ", add_positive(3, 4)
del add
print "add() was deleted, add_negtive(-1,-3) = ", add_negtive(-1, -3)

Output:

add_negtive(-1,-3) =  -4

add_positive(3,4) =  7

add() was deleted, add_negtive(-1,-3) = -4
```

Hàm trong hàm

Việc định nghĩa các hàm bên trong của một hàm khác là hoàn toàn mới đối với các lập trình viên C hoặc C++. Nhưng điều này có thể thực hiện trong Python.

Ví dụ sau ta sẽ thấy hàm toVnd() được định nghĩa và sử dụng bên trong hàm money()

```
def money(x):
    def toVnd(m):
        return m * 22000

   print toVnd(x)

money(23)

Output:

506000
```

Hàm như đối số:

Mỗi tham số của một hàm là một tham chiếu đến một đối tượng và các hàm cũng chính là các đối tượng, chúng ta có thể

truyền các hàm như các tham số cho một hàm khác. Chúng ta sẽ chứng minh điều này trong ví dụ đơn giản tiếp theo:

```
def convertMoney(amount, func):
    print "using", func
    return func(amount)


def toVND(x):
    return x * 22000

def toSGD(x):
    return x * 22000 / 16500

print "converting US$40 to SGD$"
print convertMoney(40, toSGD)
print "converting US$40 to VND"
print convertMoney(40, toVND)

Output:

converting US$40 to SGD$

using <function toSGD at 0x000000000277E978>

53

converting US$40 to VND

using <function toVND at 0x000000000277E908>

880000
```

Hàm trả về hàm

Đầu ra của một hàm cũng là một tham chiếu đến một đối tượng. Do đó các hàm có thể trả về tham chiếu đến các đối tượng hàm.

```
def f(x):
    def g(y):
        return x*y**2
   return g

func1 = f(1)
func2 = f(4)
print "func1(3) = ",func1(3)
print "func2(3) = ",func2(3)

Output:

func1(3) =  9

func2(3) =  36
```

Một ví dụ khác, tính đa thức với số bậc tự do tùy ý.

```
def polynomial_creator(*coefficients):
    """ coefficients are in the form a_0, a_1, ... a_n
    """
   def polynomial(x):
       res = 0
      for index, coeff in enumerate(coefficients):
          res += coeff * x** index
       return res
    return polynomial

p1 = polynomial_creator(4)
p2 = polynomial_creator(2, 4)
p3 = polynomial_creator(2, 3, -1, 8, 1)
p4  = polynomial_creator(-1, 2, 1)


for x in range(-2, 2, 1):
    print(x, p1(x), p2(x), p3(x), p4(x))

Output:

(-2, 4, -6, -56, -1)

(-1, 4, -2, -9, -2)

(0, 4, 2, 2, -1)

(1, 4, 6, 13, 2)
```

Tôi giải thích thêm về ý nghĩa hàm enumerate(), qua ví dụ sau:

```
for index, value in enumerate((1,2,3,4)):
    print index, value


Output:


0 1

1 2

2 3

3 4
```

Tạo một decorator đơn giản:Nó sẽ hoạt động như sau, sau phép gán foo = our_decorator(foo), foo đã tham chiếu tới func_wrappe() bên trong our_decorator(). Vậy là bằng cách decorate đơn giản ta đã có khoác lên hàm foo() một diện mạo mới, diện mạo của func_wrapper().

```
def our_decorator(func):
   def func_wrapper(x):
      print("Before calling "+ func.__name__)
      func(x)
      print("After calling "+ func.__name__)
   return func_wrapper

def foo(x):
   print("Hi, foo called with " + str(x))

print "We call foo before decoration:"
foo("Hello")
print "---------------------------------"
print "We now decorate foo with f: "
foo = our_decorator(foo)
print "We call foo after decoration:"
foo(120)

Output:

We call foo before decoration:

Hi, foo called with Hello

---------------------------------

We now decorate foo with f:

We call foo after decoration:

Before calling foo

Hi, foo called with 120

After calling foo
```

Cú pháp thường dùng cho decorator trong python

“decorator” trong Python thường không được thực hiện như cách chúng ta đã làm nó trong ví dụ trước mặc dù các ký hiệu foo = our_decorator (foo) là dễ nhớ và dễ nắm bắt. Đây là lý do tại sao chúng tôi không sử dụng nó! Bạn cũng có thể nhìn thấy một vấn đề thiết kế trong cách tiếp cận trước đây của chúng tôi. "foo" tồn tại trong cùng một chương trình với hai phiên bản, trước khi decorator và sau khi decorator.

Chúng tôi sẽ làm một decorator phù hợp ngay bây giờ. Các decorator xảy ra tại dòng trước định nghĩa hàm. Kí tự "@" được theo sau bởi tên hàm decorator.

Chúng ta sẽ viết lại ví dụ ban đầu của chúng ta. Thay vì viết foo = our_decorator(foo), ta viết @our_decorator.

```
def our_decorator(func):
    def func_wrapper(x):
        print("Befor calling "+ func.__name__)
        func(x)
        print("After calling "+ func.__name__)
    return func_wrapper
@our_decorator
def foo(x):
    print("Hi, foo has been called with " + str(x))

foo("Hi")

Output:
```

```
Befor calling foo

Hi, foo has been called with Hi

After calling foo
```

Ta cũng hoàn toàn có thể decorate cả những hàm có sẵn (build-in) qua cách sau: ví dụ hai hàm sẵn có là sin và cos.

```
from math import sin, cos
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        res = func(x)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

sin = our_decorator(sin)
cos = our_decorator(cos)
for f in [sin, cos]:
    f(3.1415)

Output:

Before calling sin

9.26535896605e-05

After calling sin

Before calling cos

-0.999999995708

After calling cos
```

Tổng kết:

chúng ta có thể nói rằng một decorator trong Python là một đối tượng “callable Python object” được sử dụng để sửa đổi một hàm, phương thức hoặc lớp. Các đối tượng ban đầu, sẽ được sửa đổi, được chuyển đến một decorator như một đối số. Decorator sẽ trả về đối tượng được sửa đổi, ví dụ: hàm sửa đổi, nó ràng buộc với tên được sử dụng trong định nghĩa.

Một số trường hợp cho decorator:

Dùng decorator để kiểm tra đối số, trong ví dụ về tính dãy số Fibonacci, nếu ta truyền vào một số âm thì hàm fib(n) ta viết bị LẶP VÔ HẠN. Bằng cách nào ta có thể ngăn chặn việc này mà không cần thay đổi hàm đó trực tiếp?

Đó chính là lý do cho sự có mặt của decorator (phần code bên phải). ta raise lỗi trước, để không xảy ra lỗi lặp vô hạn như phía trên.

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

print "fib(2) = ", fib(2)
print "fib(20) = ", fib(20)
print "fib(-20) = ", fib(-1)


Output:


fib(2) =  1

fib(20) =  6765

…

    return fib(n-1) + fib(n-2)

RuntimeError: maximum recursion depth exceeded
```

```
def argument_test_fib(f):
    def helper(x):
        if x > 0:
            return f(x)
        else:
            raise Exception("Argument is < 0")
    return helper


@argument_test_fib
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

fib(-20)

Output:

Traceback (most recent call last):

    fib(-20)

    raise Exception("Argument is < 0")
```

	Exception: Argument is < 0

Đếm số lần gọi hàm

Ví dụ sau sử dụng một decorator, ta cần đếm số lần một hàm đã được gọi. Chúng ta có thể sử dụng decorator này chỉ cho các hàm với chính xác một tham số:

```
def call_counter(func):
    def helper(x):
        helper.calls += 1
       return func(x)
    helper.calls = 0
   return helper


@call_counter
def succ(x):
    return x + 1

print "Before calling succ, number of calling time should be 0"
print "is checking right? ",0== succ.calls, ", because succ.calls = %d " % succ.calls

times = 10
print "Trying to call succ %d times" % times
for i in range(times):
    succ(i)

print "is checking right? " , times == succ.calls, ", because succ.calls = %d " % succ.calls

Output:

Before calling succ, number of calling time should be 0

is checking right?  True , because succ.calls = 0

Trying to call succ 10 times

is checking right?  True , because succ.calls = 10
```

Với hàm phía trên succ(i) ta chỉ có một đối số. Vậy ta muốn xây dựng một decorator có thể dùng cho nhiều đối số ta sẽ làm thế nào? Các bạn tự giải quyết bài toán này nhé! Gợi ý của tôi là dùng *args và **kwargs

Decorators với các tham số

Nếu chúng ta muốn thêm tham số cho decorator để có thể thực hiện một chức năng lớn hơn, chúng ta phải dùng hàm khác wrap hàm decorator trước đó. Trong ví dụ sau mô tả cách mà tôi muốn thêm một lời chào vào hàm greeting_decorator().

```
def greeting(expr):
    def greeting_decorator(func):
        def function_wrapper(x):
            print(expr + ", " + func.__name__ + " returns:")
            func(x)
        return function_wrapper
    return greeting_decorator

@greeting("Xin Chao")
def foo(x):
    print(42)

foo("Hi")

Output:

Xin Chao, foo returns:

42
```

Kết luận

Tóm lại, Có 2 loại decorator trong Python là Function decorator và Class decorator. Nhưng trong khuôn khổ của bài học này, chúng tôi mới chỉ tập trung đề cập đến Function decorator. Để tiếp cận dễ dàng hơn với decorator trước hết các bạn cần nắm được một số khía cạnh quan trọng của hàm trong Python. Chúng đều khá xa lạ với các ngôn ngữ lập trình cơ bản như C/C++. Điểm đầu tiên là chúng ta có thể gán nhiều tên cho cùng một hàm và khi xóa một trong số những cái tên đó, hàm không mất đi chức năng của nó. Kế đến là có thể khai báo hàm bên trong của một hàm khác, truyền hàm như đối số cho một hàm khác. Cuối cùng là hàm trả về hàm. Cú pháp khai báo decorator cũng khá dễ nhớ với kí tự "@" được theo sau bởi tên

hàm decorator và đặt trước định nghĩa hàm. Decorator thực sự là công cụ hữu ích trong việc sửa đổi, tùy biến một hàm hay một lớp mà không làm mất đi chức năng vốn có của nó. Trong bài tiếp theo chúng tôi sẽ tiếp tục đề cập đến decorator trong memorization.