Decorator in Python

Decorator is used a lot in Python. So how to create a Decorator and why you should use it. Let's go find the answer!

Decorator is used a lot in Python. In this article, TipsMake.com will work with you to learn how to create a Decorator and why you should use it. Let's go find the answer!

What is Decorator in Python?

Python has an interesting feature called decorator. Decorator is a function **that takes the input parameter as another function and extends the function to that function without changing its content.**

This is also called metaprogramming, which simply means "Code generates code", meaning that I write a program and this program will generate, control other programs or do a part of the job right away. at the time of compilation.

Conditions for a Decorator

To understand decorators, you first need to review some of the basics in Python.

Functions are a very basic concept in programming in general. However, in Python, **functions are also objects.** The function names we declare are simply binding identifiers with these objects. A function object can also be associated with many different names. For example:

```
def first(msg): print(msg) first("Hello") second = first second("Hello")
```

When you run code, functions `first` and `second` return the same output. Here, `first` and `second` refer to the same function object.

Stay tuned, functions can be passed as parameters to another function (similar to *map*, *filter* and *reduce* in Python).

Functions that take other functions as input parameters are called higher-order functions. For example:

```
def inc(x): return x + 1 def dec(x): return x - 1 def operate(func, x): result =
```

We call the function as follows.

```
>>> operate(inc,3) 4 >>> operate(dec,3) 2
```

Moreover, a function can return a different function.

```
def is_called(): def is_returned(): print("Hello") return is_returned new = is_ca
```

Here, *is_returned ()* is a nested function, this function is accessed and returns the result every time we call *is_called ().*

For more information, you can refer to the lesson How to Use Closure in Python

Going back to Decorator , most fundamentally, Decorator is a function that can accept other functions, allowing you to run some code before or after the main function without changing the result.

```
def make_pretty(func): def inner(): print("I got decorated") func() return inner
```

Run code in Python shell:

```
>>> ordinary() I am ordinary >>> # Th?
hàm decorate trong hàm ordinary >>> pretty = make_pretty(ordinary) >>> pretty()
```

In the above example, *make_pretty ()* is a decorator. When we call

```
pretty = make_pretty(ordinary)
```

then *ordinary ()* is passed by decorator as a parameter and the function returns a *pretty* name .

You can see here, decorator has added a new function to the original function. Imagine it as a one-piece package. The decorators are the outer covering, the nature of the object decorator passed in as the parameter (the inner gift) does not change, but now it has an external decorator cover.

In general, here comes a function decorator and reassigns it:

```
ordinary = make_pretty(ordinary)
```

This is a common structure, so Python has a syntax to simplify this.

You can use the @ symbol along with the name of the decorator function and place it above the definition of the decorator function. For example:

```
@make_pretty def ordinary(): print("I am ordinary")
```

equivalent to:

```
def ordinary(): print("I am ordinary") ordinary = make_pretty(ordinary)
```

This is a special syntax for implementing decorators.

Decorator function parameter

The above decorators are all quite simple and work with functions that don't have any parameters. Now try passing the parameter to the function returned by the decorator as follows:

```
def divide(a, b): return a/b
```

This function has two parameters, a and b, will report an error if `b=0` .

```
>>> divide(2,5) 0.4 >>> divide(2,0) Traceback (most recent call last): . ZeroDivi
```

When we call the function, it actually calls the function returned by decorator, so if it passes the parameter to this function, it will pass to the decorate function.

Now let's create a decorator to check if this is an error.

```
def smart_divide(func): def inner(a,b): print("I am going to divide",a,"and",b) :
```

The program will return `None` if an error arises.

```
>>> divide(2,5) I am going to divide 2 and 5 0.4 >>> divide(2,0) I am going to di
```

This is how to use the decorator function with parameters.

In addition, you can build a decorator with as many different parameters as you want, just use the syntax * *args* and ** *kwargs.*

```
def works_for_all(func): def inner(*args, **kwargs): print("I can decorate any fu
```

When we call the function, we actually call the function returned by decorator, so if we pass the parameter to this function, it will pass to the decorate function.

String Decorator in Python

Many decorators can be made up of decorator strings in Python.

It means that a function can be decorate many times with the same or different decorators, just put the decorator in front of the function you want.

```
def star(func): def inner(*args, **kwargs): print("*" * 30) func(*args, **kwargs
```

The program will return output:

```
****************************** %%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Hello %%%%%%%%%%%%
```

Syntax:

```
@star @percent def printer(msg): print(msg)
```

equivalent to:

```
def printer(msg): print(msg) printer = star(percent(printer))
```

The order of decorator is also important, if you reverse:

```
@percent @star def printer(msg): print(msg)
```

The result will be different:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ****************************** Hello ************
```

Previous article: How to use Closure in Python

Next article: Declare @property in Python