Python Decorator

Decorators are one of the most helpful and powerful tools of Python. These are used to modify the behavior of the function. Decorators provide the flexibility to wrap another function to expand the working of wrapped function, without permanently modifying it.

In Decorators, functions are passed as an argument into another function and then called inside the wrapper function.

It is also called **meta programming** where a part of the program attempts to change another part of program at compile time.

Before understanding the **Decorator**, we need to know some important concepts of Python.

What are the functions in Python?

Python has the most interesting feature that everything is treated as an object even classes or any variable we define in Python is also assumed as an object. Functions are **first-class** objects in the Python because they can reference to, passed to a variable and returned from other functions as well. The example is given below:

Example:

```
def func1(msg):    # here, we are creating a function and passing the parameter
    print(msg)
func1("Hii, welcome to function ")   # Here, we are printing the data of function 1
func2 = func1      # Here, we are copying the function 1 data to function 2
func2("Hii, welcome to function ")   # Here, we are printing the data of function 2
```

Output:

```
Hii, welcome to function
Hii, welcome to function
```

In the above program, when we run the code it give the same output for both functions. The **func2**referred to function **func1** and act as function. We need to understand the following concept of the function:

- The function can be referenced and passed to a variable and returned from other functions as well.
- The functions can be declared inside another function and passed as an argument to another function.

Inner Function

Python provides the facility to define the function inside another function. These types of functions are called inner functions. Consider the following example:

Example:

```
def func():    # here, we are creating a function and passing the parameter
     print("We are in first function")      # Here, we are printing the data of function
     def func1():      # here, we are creating a function and passing the parameter
           print("This is first child function")  # Here, we are printing the data of function 1
     def func2():      # here, we are creating a function and passing the parameter
           print("This is second child function")      # Here, we are printing the data of
 # function 2
     func1()
     func2()
func()
```

Output:

```
We are in first function
This is first child function
This is second child function
```

In the above program, it doesn't matter how the child functions are declared. The execution of the child function makes effect on the output. These child functions are locally bounded with the **func()** so they cannot be called separately.

A function that accepts other function as an argument is also called **higher order function**. Consider the following example:

Example:

```
def add(x):         # here, we are creating a function add and passing the parameter
    return x+1       # here, we are returning the passed value by adding 1
def sub(x):         # here, we are creating a function sub and passing the parameter
    return x-1       # here, we are returning the passed value by subtracting 1
def operator(func, x):    # here, we are creating a function and passing the parameter

    temp = func(x)
    return temp
print(operator(sub,10))  # here, we are printing the operation subtraction with 10
print(operator(add,20))   # here, we are printing the operation addition with 20
```

Output:

21

In the above program, we have passed the **sub()** function and **add()** function as argument in **operator()** function.

A function can return another function. Consider the below example:

Example:

```
def hello():      # here, we are creating a function named hello
    def hi():       # here, we are creating a function named hi
        print("Hello")          # here, we are printing the output of the function
    return hi       # here, we are returning the output of the function
new = hello()
new()
```

Output:

Hello

In the above program, the **hi()** function is nested inside the **hello()** function. It will return each time we call **hi()**.

Decorating functions with parameters

Let's have an example to understand the parameterized decorator function:

Example:

```
def divide(x,y):      # here, we are creating a function and passing the parameter
    print(x/y)        # Here, we are printing the result of the expression
def outer_div(func):      # here, we are creating a function and passing the parameter

    def inner(x,y):      # here, we are creating a function and passing the parameter
        if(x<y):
            x,y = y,x
           return func(x,y)
# here, we are returning a function with some passed parameters
     return inner
divide1 = outer_div(divide)
divide1(2,4)
```

Output:

2.0

Syntactic Decorator

Advertisement

In the above program, we have decorated **out_div()** that is little bit bulky. Instead of using above method, Python allows to **use decorator in easy way with @symbol**. Sometimes it is called "pie" syntax.

```
def outer_div(func):     # here, we are creating a function and passing the parameter
    def inner(x,y):       # here, we are creating a function and passing the parameter
        if(x<y):
           x,y = y,x
          return func(x,y)      # here, we are returning the function with the parameters
     return inner
# Here, the below is the syntax of generator
@outer_div
def divide(x,y):      # here, we are creating a function and passing the parameter
     print(x/y)
```

Output:

2.0

Reusing Decorator

We can reuse the decorator as well by recalling that decorator function. Let's make the decorator to its own module that can be used in many other functions. Creating a file called **mod_decorator.py** with the following code:

```
def do_twice(func):      # here, we are creating a function and passing the parameter
    def wrapper_do_twice():
     # here, we are creating a function and passing the parameter
        func()
        func()
    return wrapper_do_twice
We can import mod_decorator.py in another file.
from decorator import do_twice
@do_twice
def say_hello():
    print("Hello There")
say_hello()
```

We can import mod_decorator.py in other file.

```python
from decorator import do_twice
@do_twice
def say_hello():
    print("Hello There")
say_hello()
```

Output:

```
Hello There
Hello There
```

Python Decorator with Argument

We want to pass some arguments in function. Let's do it in following code:

```python
from decorator import do_twice
@do_twice
def display(name):
     print(f"Hello {name}")
display()
```

Output:

```
TypeError: display() missing 1 required positional argument: 'name'
```

As we can see that, the function didn't accept the argument. Running this code raises an error. We can fix this error by using ***args** and ****kwargs**in the inner wrapper function. Modifying the **decorator.py**as follows:

```python
def do_twice(func):
    def wrapper_function(*args,**kwargs):
        func(*args,**kwargs)
        func(*args,**kwargs)
   return wrapper_function
```

Now **wrapper_function()** can accept any number of argument and pass them on the function.

```python
from decorator import do_twice
@do_twice
def display(name):
      print(f"Hello {name}")
display("John")
```

Output:

```
Hello John
Hello John
```

Returning Values from Decorated Functions

We can control the return type of the decorated function. The example is given below:

```
from decorator import do_twice
@do_twice
def return_greeting(name):
    print("We are created greeting")
    return f"Hi {name}"
hi_adam = return_greeting("Adam")
```

Output:

```
We are created greeting
We are created greeting
```

Fancy Decorators

Let's understand the fancy decorators by the following topic:

Class Decorators

Python provides two ways to decorate a class. Firstly, we can decorate the method inside a class; there are built-in decorators like **@classmethod, @staticmethod** and **@property** in Python. The **@classmethod** and **@staticmethod** define methods inside class that is not connected to any other instance of a class. The @property is generally used to modify the getters and setters of a class attributes. Let's understand it by the following example:

Example: 1-

@property decorator - By using it, we can use the class function as an attribute. Consider the following code:

```
class Student:     # here, we are creating a class with the name Student
    def __init__(self,name,grade):
        self.name = name
        self.grade = grade
    @property
    def display(self):
        return self.name + " got grade " + self.grade

stu = Student("John","B")
print("Name of the student: ", stu.name)
print("Grade of the student: ", stu.grade)
print(stu.display)
```

Output:

```
Name of the student: John
Grade of the student: B
John got grade B
```

Example: 2-

@staticmethod decorator- The @staticmethod is used to define a static method in the class. It is called by using the class name as well as instance of the class. Consider the following code:

```
class Person:      # here, we are creating a class with the name Student
    @staticmethod
    def hello():        # here, we are defining a function hello
        print("Hello Peter")
per = Person()
per.hello()
Person.hello()
```

Output:

```
Hello Peter
Hello Peter
```

Singleton Class

A singleton class only has one instance. There are many singletons in Python including True, None, etc.

Nesting Decorators

We can use multiple decorators by using them on top of each other. Let's consider the following example:

```
@function1
@function2
def function(name):
      print(f "{name}")
```

In the above code, we have used the nested decorator by stacking them onto one another.

Decorator with Arguments

It is always useful to pass arguments in a decorator. The decorator can be executed several times according to the given value of the argument. Let us consider the following example:

Example:

```
Import functools      # here, we are importing the functools into our program
def repeat(num):     # here, we are defining a function repeat and passing parameter

# Here, we are creating and returning a wrapper function
    def decorator_repeat(func):
        @functools.wraps(func)
        def wrapper(*args,**kwargs):
            for _ in range(num):  # here, we are initializing a for loop and iterating till num

                value = func(*args,**kwargs)
            return value      # here, we are returning the value
        return wrapper    # here, we are returning the wrapper class
    return decorator_repeat
#Here we are passing num as an argument which repeats the print function
@repeat(num=5)
def function1(name):
    print(f"{name}")
```

Output:

```
JavatPoint
JavatPoint
JavatPoint
JavatPoint
JavatPoint
```

In the above example, **@repeat**refers to a function object that can be called in another function. The **@repeat(num = 5)**will return a function which acts as a decorator.

The above code may look complex but it is the most commonly used decorator pattern where we have used one additional **def** that handles the arguments to the decorator.

Note: Decorator with argument is not frequently used in programming, but it provides flexibility. We can use it with or without argument.

Stateful Decorators

Stateful decorators are used to keep track of the decorator state. Let us consider the example where we are creating a decorator that counts how many times the function has been called.

Example:

```
Import functools        # here, we are importing the functools into our program
def count_function(func):
# here, we are defining a function and passing the parameter func
@functools.wraps(func)
def wrapper_count_calls(*args, **kwargs):
wrapper_count_calls.num_calls += 1
print(f"Call{wrapper_count_calls.num_calls} of {func.__name__!r}")
return func(*args, **kwargs)
wrapper_count_calls.num_calls = 0
return wrapper_count_calls      # here, we are returning the wrapper call counts
@count_function
def say_hello():  # here, we are defining a function and passing the parameter
print("Say Hello")
say_hello()
say_hello()
```

Output:

```
Call 1 of 'say_hello'
Say Hello
Call 2 of 'say_hello'
Say Hello
```

In the above program, the state represented the number of calls of the function stored in **.num_calls**on the wrapper function. When we call **say_hello()**it will display the number of the call of the function.

Classes as Decorators

The classes are the best way to maintain state. In this section, we will learn how to use a class as a decorator. Here we will create a class that contains **__init__()** and take **func** as an argument. The class needs to be callable so that it can stand in for the decorated function.

To making a class callable, we implement the special **__call__()** method.

Code

```
import functools        # here, we are importing the functools into our program
class Count_Calls:      # here, we are creating a class for getting the call count
def __init__(self, func):
functools.update_wrapper(self, func)
self.func = func
self.num_calls = 0
```

```
def __call__(self, *args, **kwargs):
self.num_calls += 1
print(f"Call{self.num_calls} of {self.func.__name__!r}")
return self.func(*args, **kwargs)
@Count_Calls
def say_hello():  # here, we are defining a function and passing the parameter
print("Say Hello")
say_hello()
say_hello()
say_hello()
```

Output:

```
Call 1 of 'say_hello'
Say Hello
Call 2 of 'say_hello'
Say Hello
Call 3 of 'say_hello'
Say Hello
```

The **__init__()** method stores a reference to the function and can do any other required initialization.