Python Decorator Demystify

What are Decorators

First and foremost, a decorator is a function that takes in a function and return another, letting you add extra behavior without changing the original function's source code.

```
def decorator(fn):
  def wrapper(*arg, **kwargs):
    print("Before")
    ret = fn(arg, kwargs)
    print("After")
    return ret
  return fn
```

This is a simple decorator that allows you to surround the execution of your function with a before and after print statement.

> It is important to note that, if you're not planning on modifying or extend any ability for a decorator, you can actually just return the function, without writing that wrapper function. You only need to write the wrapper function if you **need to extend or add onto what the original function is doing.** This will be important later.

You can then use the decorator by using it with a function like such:

```
@decorator
def add(a, b):
  return a + b
```

When you call `add(1, 2)` it will print "Before" and "After" and then returns you the value of 3 from the function.

What Python is essentially doing underneath is something like this:

```
add = decorator(add)
```

It calls the decorator function with your add function as it's argument and reassign it to add. Then the actual function that gets called is wrapper.

Decorator with Arguments

If you want to pass arguments to your decorator as to customize the decorator further, you can! However, you will need more layer of nesting:

```
def decorator_with_arg(arg):
  def decorate(fn):
    def wrapper(*arg, **kwargs):
      print("before is called with", arg)
      ret = fn(arg, kwargs)
      print("after is called with", arg)
      return ret
    return wrapper
  return decorate
```

Now you can see this is a little bit more complicated, if you want your decorator to have inputs from the user, then you will need to change the arguments for the most upper level function to take in what you want. In this case, it is just `arg` which can be anything since we are just printing it out.

But inside you can see that it is the decorator! So the most outer layer is returning decorate, which we have defined previously to be a function that takes in a function and returns another extending it's capability. You can see that it is another layer of nesting!

```
@decorator_with_arg("jessie")
def add(a, b):
  return a + b
```

If you decorate your function with it and call it it will print "before is called with jessie" and "after is called with jessie" and return you the value of a + b.

Functionality Python is doing something like this:

```
add = decorator_with_arg("jessie")(add)
```

The first function call results in a decorator being resulted, and it is then called with your function as it's parameter to decorate.

What if you need argument, but don't want to modify the function

If all you want to do is pass argument to the decorator, i.e. for logging purposes, you actually can just shorten it to 2 level of nesting, you do not need three.

```
def decorator_with_arg(arg):
  def decorator(fn):
    print("Inside the decorator", arg)
    return fn
  return decorator
```

So same thing if you call it with:

```
@decorator_with_arg("mark")
def add(a, b):
  return a + b


// underneath
add = decorator_with_arg("mark")(add)
```

The decorated function will function the same as it was before, BUT when it was initially loaded by python and decorated during runtime, it will print "Inside the decorator mark". But since the actual function itself is not changed, it is just returned as it is (no wrapper function within it, the same fn is returned as it is). This can help you achieve with meta logging, or registering of functions without changing the functionality of the original method.

Revision #1
Created 2026-09-06 23:07:54 UTC by Tamarine
Updated 2026-09-06 23:24:54 UTC by Tamarine