karembe kadidia

Contents

- Decorators with state
- Classes of decorators

6. More real-world examples

- Slowdown code, revisited
- Creation of singletons
- Caching return values
- Added unit information
- JSON validation

7. Conclusion

8.Further reading

1.Definition: The Decorator is a structural design pattern that allows you to add dynamically new behaviors to objects by placing them at inside special objects called packers(wrappers).

2. **Functions:** Before you can understand decorators, you must first understand function operation. For our purposes, a function returns a value based on given arguments. Here is a very simple example:

```
def add_one(number):
    return number + 1
add_one(2)
# on aura 3 en sortie
```

In general, functions in Python can also have effects secondaries rather than just transforming an input into an output. The print() function is a basic example: it returns None all with the side effect of outputting something to the console. However, to understand decorators, it suffices to consider functions as something that transforms given arguments en valeur.

Note: In functional programming, you (almost) only work only with pure functions without side effects. Although it is not not a purely functional language, Python supports many functional programming concepts, including functions as only first class objects. - First Class Items:

: In Python, functions are first-class objects. This means that functions can be transmitted and used as arguments, like any other object (string, int, float, list, etc.) . Consider the following three functions:

```
def say_hello(name):
    return f"Hello {name}"

def be_awesome(name):
    return f"Yo {name}, together we are the awesomest!"

def greet_bob(greeter_func):
    return greeter_func("Bob")
```

Here say_hello() and be_awesome() are regular functions that expect a name given as a string. However, the greet_bob() function expects a function as an argument. One can, for example, pass it the say_hello() or the be_awesome() function:

```
greet_bob(say_hello)
# la sortie sera: 'Hello Bob'
greet_bob(be_awesome)
# la sortie sera: 'Yo Bob, together we are the awesomest!'
```

Note that greet_bob(say_hello) refers to two functions, but in different ways: greet_bob() and say_hello. The say_hellofunction is named without parentheses. That means that only a reference to the function is passed. The function is not executed. The greet_bob() function, on the other hand, is written with parentheses, so it will be called as per usual. - Internal functions : It is possible to define functions inside other functions. These functions are called internal functions. Here is an example function with two inner functions:

```
def parent():
    print("Printing from the parent() function")

    def first_child():
        print("Printing from the first_child() function")

    def second_child():
        print("Printing from the second_child() function")

    second_child()
    first_child()
```

What happens when you call the parent() function? Think about it for a minute. The output will be as follows:

```
parent()
Printing from the parent() function
Printing from the second_child() function
Printing from the first_child() function
```

Note that the order in which the internal functions are defined does not have of importance. As with all other functions, printing does not produced only when the internal functions are executed.

Also, inner functions are not defined until the function parent is not called. They have a parent() local scope: they only exist inside the function as local parent() variables. Try calling . You should get an error: first_child()

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'first_child' is not defined
```

Whenever you call parent(), the internal functions first_child() and second_child() are also called. But because of their scope locale, they are not available outside the parent() function. - Return functions from functions : Python also allows you to use functions as values back. The following example returns one of the internal functions to from the parent() outer function:

```
def parent(num):
    def first_child():
        return "Hi, I am Emma"

    def second_child():
        return "Call me Liam"

    if num == 1:
        return first_child
    else:
        return second_child
```

Note that you return first_child without the parentheses. remember that means you are returning a reference to the first_child function. Unlike first_child() in parentheses refers to the result of function evaluation. This can be seen in the following example:

```
first = parent(1)
second = parent(2)

first # appel de la fonction first
```

```
<function parent.<locals>.first_child at 0x7f599f1e2e18> # sortie de la fonction first

second  # appel de la fonction second
<function parent.<locals>.second_child at 0x7f599dad5268> # sortie de la fonction second
```

The somewhat cryptic output simply means that the firstvariable refers to the local first_child() function inside parent(), while second points to second_child().

You can now use first and second as if it were normal functions, even if the functions they point to are not directly accessible:

```
first() # appel de la fonction first
'Hi, I am Emma' # sortie de la fonction first
```

Finally, notice that in the previous example, you ran the functions internals in the parent function, for example first_child(). However, in this last example, you didn't add parentheses to the functions internals first_child——when returning. This way you have a reference to every function you might call in the future.Make sense ?

3. Simple Decorators: Now that you've seen that functions are like any another object in Python, you're ready to move on and see the magical beast that is the Python decorator. Let's start with an example:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def say_whee():
    print("Whee!")

say_whee = my_decorator(say_whee)
```

Can you guess what happens when you call say_whee()? Try it:

```
say_whee() # appel de la fonction say_whee
Something is happening before the function is called.
Whee!
Something is happening after the function is called.
```

To understand what is happening here, review the previous examples. We literally apply everything you have learned so far.

The so-called decoration occurs at the following line:

```
say_whee = my_decorator(say_whee)
```

Indeed, the name say_whee now points to the inner wrapper() function. Remember that you return wrapper as a function when you call my_decorator(say_whee):

```
say_whee
<function my_decorator.<locals>.wrapper at 0x7f3c5dfd42f0>
```

However, wrapper() has a reference to the original say_whee() as a func, and calls this function between the two calls to print(). In simple terms:

decorators wrap a function, changing its behavior. Before continuing, let's look at a second example. Because wrapper() is a normal Python function, the way a decorator modifies a function can change dynamically. In order not to disturb your neighbors, the following example will only run the decorated code during the day:

```
from datetime import datetime

def not_during_the_night(func):
    def wrapper():
        if 7 <= datetime.now().hour < 22:
            func()
        else:
            pass  # Hush, the neighbors are asleep
    return wrapper

def say_whee():
    print("Whee!")

say_whee = not_during_the_night(say_whee)
```

If you try to call say_whee() after bedtime, nothing will happen:

```
say_whee()
```

Syntactic sugar!

The way you decorated say_whee() above is a bit clunky. First, you end up typing the name say_whee three times. In addition, the decoration hides a little under the definition of the function.

Instead, Python allows you to use decorators in a way easier with the @symbol** , sometimes called the "pie" syntax. The following example does exactly the same thing as the first decorator example:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_whee():
    print("Whee!")
```

So @my_decorator is just a simpler way of saying say_whee = my_decorator(say_whee). This is how you apply a decorator to a function. - Reuse decorators

Remember that a decorator is just a normal Python function. All the usual tools for easy reuse are available. Let's move the decorator to its own module which can be used in many other functions. Create a file called decorators.py with the following content:

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice
```

Note: You can name your internal function whatever you like, and a generic name like wrapper() is usually fine. You will see a lot of decorators in this article. To separate them, we will name the inner function with the same name as the decorator but with a wrapper_prefix.

You can now use this new decorator in other files by doing a regular import:

```
from decorators import do_twice

@do_twice
def say_whee():
    print("Whee!")
```

Lorsque vous exécutez cet exemple, vous devriez voir que l'original say_whee()est exécuté deux fois :

```
say_whee()
Whee!
Whee!
```

- **Decorating functions with arguments**

Let's say you have a function that accepts some arguments. Can you still decorate it? Let's try:

```
from decorators import do_twice

@do_twice
def greet(name):
    print(f"Hello {name}")
```

Unfortunately, running this code generates an error:

```
greet("World")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: wrapper_do_twice() takes 0 positional arguments but 1 was given
```

The problem is that the internal function wrapper_do_twice() takes no arguments, but name="World" was passed to it. You can solve this problem by letting wrapper_do_twice() accept an argument, but that wouldn't work for the say_whee() function you created earlier.

The solution is to use *argset**kwargs in the inner wrapper function. Then it will accept an arbitrary number of positional arguments and keywords. Rewrite decorators.py as follows:

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
```

```
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_do_twice
```

The internal wrapper_do_twice() function now accepts any how many arguments and passes them to the function it decorates. Now your say_whee() and greet() examples work:

```
say_whee()
Whee!
Whee!

greet("World")
Hello World
Hello World
```

- **Return values from decorated functions** What happens to the return value of decorated functions? Well, that's up to the decorator to decide. Let's say you decorate a simple function as follows:

```
from decorators import do_twice

@do_twice
def return_greeting(name):
    print("Creating greeting")
    return f"Hi {name}"
```

Try using it:

```
hi_adam = return_greeting("Adam")
Creating greeting
Creating greeting
>>> print(hi_adam)
None
```

Oops, your decorator ate the function's return value.

Since do_twice_wrapper() does not explicitly return a value, the return_greeting("Adam") call ended up returning None. To fix this you need to make sure the wrapper function returns the return value of the decorated function.** Edit your decorators.py file:

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

The return value of the last execution of the function is returned:

```
return_greeting("Adam")
Creating greeting
Creating greeting
'Hi Adam'
```