Department of Computer Science and Engineering (UG Studies)
PES University, Bangalore, India

Introduction to Computing using Python (UE19CS101)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Decorators in Python

First-Class Objects

In Python, functions are first-class objects. This means that **functions can be passed around and used as arguments**, just like any other object (string, int, float, list, and so on). 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")
```

Note:

- **To create an f-string, prefix the string with the letter " f ". The string itself can be formatted in much the same way that you would with `str.format()`.**
- **F-strings provide a concise and convenient way to embed python expressions inside string literals for formatting.**

Here, say_hello() and be_awesome() are regular functions that expect a name given as a string. The greet_bob() function however, expects a function as its argument. We can, for instance, pass it the say_hello() or the be_awesome() function:

```
>>> greet_bob(say_hello)
'Hello Bob'

>>> greet_bob(be_awesome)
'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_hello function is named without parentheses. This 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 usual.

What are decorators in Python?

Python has an interesting feature called decorators to add functionality to an existing code.
This is also called **metaprogramming** as a part of the program tries to modify another part of the program at compile time.

Basically, a decorator takes in a function, adds some functionality and returns it.

Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class.

Simple Decorators

Now that you've seen that functions are just like any other 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_hello():
    print("Hello!")

say_hello = my_decorator(say_hello)
```

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

```
print(say_hello())
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

To understand what's going on here, look back at the previous examples. We are literally just applying everything you have learned so far.
The so-called decoration happens at the following line:

```
say_hello = my_decorator(say_hello)
```

In effect, the name `say_hello` now points to the `wrapper()` inner function. Remember that you return wrapper as a function when you call `my_decorator(say_hello):`

```
print(say_hello)
<function my_decorator.<locals>.wrapper at 0x7f3c5dfd42f0>
```

However, wrapper() has a reference to the original `say_hello()` as `func`, and calls that function between the two calls to print().

Put simply: **decorators wrap a function, modifying its behavior.**

Before moving on, let's have a look at a second example. Because `wrapper()` is a regular Python function, the way a decorator modifies a function can change dynamically. So as 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_hello():
    print("Hello!")

say_hello = not_during_the_night(say_hello)
print(say_hello())
```

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

```
print(say_hello())
```

Syntactic Sugar!

Note:

- In computer science, syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express.
- It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.

The way you decorated `say_hello()` above is a little clunky. First of all, you end up typing the name say_hello three times. In addition, the decoration gets a bit hidden away below the definition of the function.
Instead, Python allows you to **use decorators in a simpler way with the @ symbol**, sometimes called the "pie" syntax. The following example does the exact 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_hello():
    print("Hello!")
```

So, **@my_decorator** is just an easier way of saying **say_hello = my_decorator(say_hello)**. It's how you apply a decorator to a function.

Reusing Decorators

Recall that a decorator is just a regular Python function. All the usual tools for easy reusability are available. Let's move the decorator to its own module that 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 inner function whatever you want, and a generic name like wrapper() is usually okay. You'll see a lot of decorators in this article. To keep them apart, we'll 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_hello():
    print("Hello!")
```

When you run this example, you should see that the original say_hello() is executed twice:

```
print(say_hello())
Hello!
Hello!
```

Decorating Functions With Arguments

Say that 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 raises 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 inner function wrapper_do_twice() does not take any arguments, but name="World" was passed to it. You could fix this by letting wrapper_do_twice() accept one argument, but then it would not work for the say_hello() function you created earlier.

The solution is to use *args and **kwargs in the inner wrapper function. Then it will accept an arbitrary number of positional and keyword arguments. 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 **wrapper_do_twice()** inner function now accepts any number of arguments and passes them on to the function it decorates. Now both your **say_hello()** and **greet()** examples works.

```
@do_twice
def say_hello():
    print("Hello!")

say_hello()
```

Output:

```
Hello!
Hello!
```

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

greet("World")
```

Output:

```
Hello World
Hello World
```

```
@do_twice
```

```
def greet(name1,name2,name3):
    print(f"Hello {name1}, {name2}, {name3}")

greet("Ram","Laxman","Hanuman")
```

Output:

```
Hello Ram, Laxman, Hanuman
Hello Ram, Laxman, Hanuman
```

Returning 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:

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

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

hi_adam = return_greeting("Adam")
print(hi_adam)
```

Output:

```
Creating greeting
Creating greeting
None
```

Oops, your decorator ate the return value from the function.

Because the `do_twice_wrapper()` doesn't explicitly return a value, the call return_greeting("Adam") ended up returning None.

To fix this, you need to **make sure the wrapper function returns the return value of the decorated function**. Change your decorators.py file:

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

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

```
hi_adam = return_greeting("Adam")
print(hi_adam)
```

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

```
Creating greeting
Creating greeting
'Hi Adam'
```

****