NAPREDNE TEHNIKE U PYTHON-U: DEKORATORI
ADVANCED PYTHON TECHNIQUES: DECORATORS

Tomo Popović, *Electrical and Computer Engineering, Texas A&M University*

Sadržaj: *Programski jezik Python je postao veoma popularan zadnjih godina. Python je sve više prisutan i u edukaciji zbog veoma čitljive sintakse koja omogućava brzo učenje programiranja i principa računarskih nauka. Ovaj rad istražuje dekoratore u Python-u, koji se smatraju naprednom tehnikom čak i od strane iskusnih Python programera. U radu je dat opis mehanizma dekoratora kao i njihova implementaicija korišćenjem klasa i funkcija. Opcije za implementaciju su objašnjene korišćenjem ilustrativnih primera i njihove vizualizacije.*

Abstract: *In recent years, Python had become an extremely popular programming language. Python is becoming more and more present in education as its syntax gets out of the way and makes it easy to learn and use to introduce computer science concepts. This paper explores the inner workings of Python decorators, which is considered an advanced topic even for experienced programmers. The paper provides an insight into decorators mechanism, as well as their implementation using classes and functions. Implementation options are discussed using examples and visualization.*

1. INTRODUCTION

In recent years, the Python programming language has become extremely popular. Python is very flexible, versatile, fun, easy to learn, and it enables programmers to be more productive. In Python, one can write scripts that run in OS command line, cross-platform GUI applications, web applications, financial applications, as well as scientific code similar to Matlab [1,2]. Getting started with Python and its ecosystem is easy. The syntax gets out of the way and the code almost resembles a pseudo-code. In addition, Python is open source and it is supported by great and extremely active community.

Advanced Python techniques may require more exploration to be understood and used properly. Even competent and experienced programmers often do not utilize all of the idioms and advanced concepts available in the language. Python decorators would fit into that category. While decorators are fairly straightforward to use, their implementation may not be as easy to grasp. Even some definitions of Python decorators seem ambiguous since they mix the description of how to use them vs. how to implement them [3,4].

This paper is an attempt to introduce and explore internals of Python decorators, their implementation, and use in our own code or libraries. Section 2 provides background on Python objects, functions, introduction to Python decorators, as well as a visualization tool called Lumpy. Section 3 illustrates implementation of decorators using classes and functions, while Section 4 provides discussions and conclusions. References are given at the end.

2. BACKGROUND

2.1. Decorators in Python

Decorators in Python are applied to functions and have a purpose to augment their behavior. A decorated function is typically wrapped by another function, which is called instead the original one. The idea is that a decorator adds some new attributes to the original function, adds new functionality to it, and even replaces it with a completely different behavior.

The use of decorators assumes a line of code placed above a function or method starting with character `@`, followed by the decorator's name and arguments. Decorators are used in various frameworks, such as popular *Nose* testing framework [5], and they provide an easy syntax to use. For example, the Nose testing tools module provides the function `timed`, which passes if the decorated function, a function being tested, finishes within a specified time limit (Fig. 1).

```
@timed(.1)
def test_that_fails():
    time.sleep(.2)
```

Fig. 1. Using the `timed` test function from Nose

A decorator is typically a function that takes another function as an argument, and then returns a new function [6]. The returned function is used to replace the functionality of the original, also called decorated, function. The use of decorators sintactically looks similar to annotations in Java and C#. However, a better comparison would be to compare the decorators with macros in C/C++, as explained in [4]. Sometimes, decorators in Python get mixed with decorator design pattern [4,7]. Although the decorator pattern can be implemented using decorators, they can also be used to implement other design patterns [7].

Decorators can be used to implement loggers, timers, format conversions, frameworks, etc. They offer a powerful mechanism and easy decoration syntax to provide for modified functionality of decorated functions.

2.2. Everything is an Object

Everything is an object in Python [8]. All the values, classes, functions, object instances, and other language constructs are objects, which means they can have attributes, member functions, they can be passed as arguments, etc. This concept is particularly obvious when when considering how Python handles assignments. Even integers can be seen as objects in Python (please refer to example in Fig. 2).

```
a = 2
b = [1,2,3]
c = b
print a.__add__(2)
print (2).__add__(3)
```

Fig. 2. Everything in object in Python

2.3. Functions in Python

Functions are code constructs that are supposed to do something, typically generate an output, based on their arguments. In Python, functions can have a "procedural role" without returning a value (they return a `None`). There are also functions without arguments. In general, functions in Python are fairly straightforward and have a very clear syntax [1,8].

What may not be so obvious is that functions in Python are objects like everything else that is user-defined (Fig. 3). Functions have attributes, such as `__name__`, and they can be assigned to a variable. In this case the `func1` function object is assigned to variable `y` and then called using that new name. Instead of variables that contain values, Python language deals with names and bindings, which in this case means that the variable `y` is bound to the same object as name `func1`.

Functions are first class objects, which means that they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and returned from other functions as a result [6,8]. Another very important concept to introduce here are *closures*. A closure is a first-class function, a function that can be stored as a variable, and also has an ability to access other variables local to the scope in which it was created. The implementation of decorators rely on this very concept, which will be explained later.

```
def func1():
    print "Inside func1"

func1()
y = func1
y()
print y.__name__
```

Fig. 3. Functions are objects as well

2.4. Visualization Using Lumpy

Lympy is a Python module that generates UML object and class diagrams [8]. Lumpy is used from a running Python program and it can be a very useful debugging and educational tool since it generates a high-level visualization that are somewhat in line with UML standard. In this paper, Lumpy will be used to illustrate what happens when Python decorators are used. An example object diagram for previously shown code (Fig. 2 and 3) is created using Lumpy and depicted in Fig. 4. The object state consists of a module that contains main references, where variable `a` is bound to an integer value 2, `b` and `c` are bound to a list object, and `func1` and `y` are bound to the same function object named `func1`.

3. IMPLEMENTING DECORATORS

3.1. Implementation Using Classes

Decorators in Python are often introduced through functions. That can be confusing, especially if the audience is not used to the concept of functions being objects.

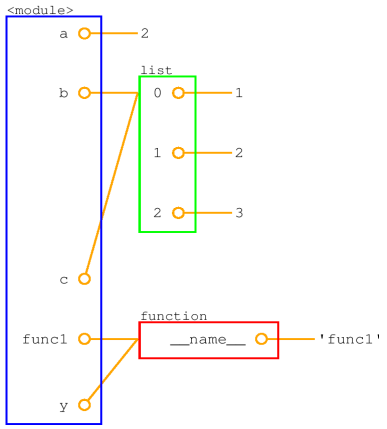

Fig. 4. Visualization of Python objects using Lumpy

We will here start with a decorator implementation example using classes, which makes it easier to relate decorators to object instances of that class. In Fig. 5, a class called `MyDecorator` is implemented. The constructor method `__init__()` defines an attribute and binds it to the object representing the decorated function f. The class also needs to be callable so the method `__call__()` is defined in which the decorated function is called and executed. The purpose of decorators in Python is to wrap the original function they decorate and to provide for some modified or additional behavior. In this simple example, we added printing before and after the call of the original function object that is being decorated.

```
class MyDecorator():
    def __init__(self, f):
        self.f = f
    def __call__(self, *args, **kwargs):
        print("Entering", self.f.__name__)
        self.f(*args, **kwargs)
        print("Exited", self.f.__name__)

# @MyDecorator
def func1():
    print("inside func1()")

@MyDecorator
def func2():
    print("inside func2()")
…
```

Fig. 5. Class implementation of a decorator

As shown in is shown in Fig. 6, the module object contains attribute `MyDecorator` referencing a class object with name `MyDecorator`. Since the decorator is commented for the first function, its name `func1` is bound directly to a function object named `func1`. Finally, the decorated function is bound to a `MyDecorator` that contains an attribute `f` bound to the original function `func2`. By calling `func2`, we actually call an instance of `MyDecorator` that exhibits modified behavior in its `__call__()` method.

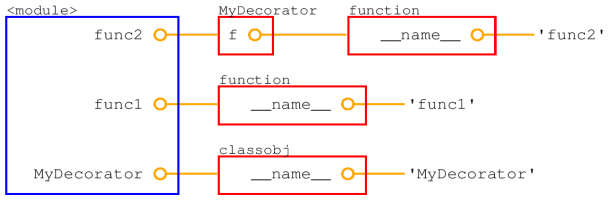

Fig. 6. Decorated function references a callable object

3.2. Implementation Using Functions

As mentioned before, decorators need to be implemented as callable objects so that that they can be used to replace the decorated functions. In the previous example, the decorated function was replaced by an instance of a class that has a `__call__()`. More commonly, decorators in Python are introduced by implementing functions as decorators. Since functions in Python are callable objects, they can be used to implement decorators as illustrated in Fig. 7.

```
def my_decorator(f):
    def wrapped_f(*args, **kwargs):
        print("Entering", f.__name__)
        f(*args, **kwargs)
        print("Exited", f.__name__)
    return wrapped_f

@my_decorator
def func1():
    print("inside func1()")

@my_decorator
def func2(x):
    print("inside func2() %d" % x)
…
print(func1.__name__)
print(func2.__name__)
```

Fig. 7. Function implementation of a decorator

The Lympy object diagram for this implementation is shown in Fig. 8. It can be observed that both functions `func1` and `func2` are now replaced with instances of new function `wrapped_f`. However, it is very important to note that, although they have the same name, these two instances are two different objects in memory and they internally reference instances of the original functions `func1` and `func2`. Another important thing to note here is that `wrapped_f()` is a *closure*, because it captures, closes over, the actual value of `f`.

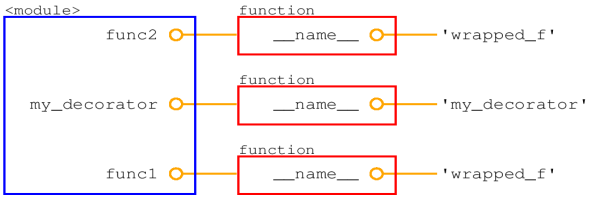

Fig. 8. Decorated functions are replaced with their closures

Printing names at the end of the program in Fig. 7 reveals that both names are the same even though internally the two reference two different function objects. If this behavior is not desirable, the names of decorated functions can be changed dynamically before it is being returned (Fig. 9 and 10). Python is very powerful when it come to types of information that can dynamically be obtained and modified about functions. Fig. 10 shows that decorated functions `func1` and `func2` now reference wrapper functions with different names.

```
def my_decorator(f):
    def wrapped_f(*args, **kwargs):
        print("Entering", f.__name__)
        f(*args, **kwargs)
        print("Exited", f.__name__)
    wrapped_f.__name__ = "wrapped_" + f.__name__
    return wrapped_f
```

Fig. 9. Function as a decorator with the name reference

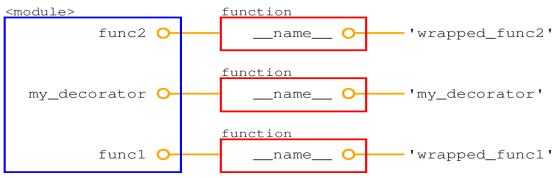

Fig. 10. Adjusting the names of decorated functions

3.3. Decorators with Arguments

When implementing decorators with arguments, there is different behavior that needs to be noted. If the decorator has arguments, the decorated function is not passed to the `__init__()` constructor, but to the `__call__()` method. The `__call__()` method is only invoked once, during the decoration process. It has one argument, which is the decorated function. In this situation, an internal wrapper function is needed (`wrapped_f` in Fig. 11). The arguments of the decorated function are to be passed to the wrapper, which can internally access both the arguments of the decorator and decorated function.

```
class MyDecorator():
    def __init__(self, arg1, arg2):
        print("Inside __init__()"
        self.arg1 = arg1
        self.arg2 = arg2
    def __call__(self, f):
        print("Inside __call__()")
        def wrapped_f(*args, **kwargs):
            print("Inside wrapped_f()")
            print("arg1", self.arg1)
            f(*args, **kwargs)
            print("After decorated f()")
        return wrapped_f

@MyDecorator("test", 1, 2)
def func1(a,b):
    print("inside func1()")
    return a+b
…
```

Fig. 11. Class implementation of a decorator with arguments

A template for the decorator function implementation with arguments is provided in Fig. 12. Although this implementation is more concise it needs some exploring to understand. At the time of decoration the decorator arguments are passed to `my_decorator` and the inside `wrapper` function is only called once. A reference to the decorated function is passed as a parameter to the `wrapper`, which defines an inner function `wrapped_f` that will replace the decorated function. Later in the code, whenever the decorated function is called, the corresponding instance of `wrapped_f` gets called instead. Please note that each instance of `wrapped_f` internally has access to the arguments originally passed to the decorator, as well as to an instance of the original decorated function. This inner function creates a closure for the decorated function `f`, but also for the arguments passed to the decorator function at the time of decoration. On the "outside" the object diagram looks exactly the same as in the previous example shown in Fig. 8. Each decorated function will be given a new instance of a `wrapped_f` that will close over the decorators arguments and the decorated function. If printed, the values of `func1` and `func2` reveal that they in fact reference two different instances of `wrapped_f`:

```
<function wrapped_f at 0x7f8d1f407230>
<function wrapped_f at 0x7f8d1f407320>
```

```
def my_decorator(arg1, arg2, arg3):
    def wrapper(f):
        print("Inside wrapper")
        def wrapped_f(*args, **kwargs):
            print("Inside wrapped_f()")
            print("arg1", arg1)
            f(*args, **kwargs)
            print("After decorated f()")
        return wrapped_f
    return wrapper

@my_decorator("test", 1, 2)
def func1(a,b):
    print("inside func1()")
    return a+b

@my_decorator("test", 1, 2)
def func2(x):
    print("inside func2() %d" % x)
…
```

Fig. 12. Decorator with arguments using functions

There are three levels of functions in the implementation and the most inner one is the actual replacement function. Even though the functional implementation appears more concise and clear, it may not be as easy to understand as the implementation version using classes (Fig. 11).

3.4. An Example: Execution Timer

An example of the use of decorators is given in Fig. 13. The decorator function is designed to measure function execution time. The assumption is that the decorated functions (`pi_func1`, `pi_func2`, `pi_func3`) calculate number Pi. Adding the decorator to each of these functions means that an instance of the `ExecTimer` will be called and the original functions will be wrapped with code to measure execution time. The example output is given in Fig. 14.

```
from math import pi, fabs, sqrt
from datetime import datetime

class ExecTimer():
    def __init__(self, f):
        self.f = f
    def __call__(self, *args, **kwargs):
        print("Entering", self.f.__name__)
        t1 = datetime.now()
        result = self.f(*args, **kwargs)
        t2 = datetime.now()
        print("Exited", self.f.__name__)
        print("Execution time: %s" % (t2-t1))
        return result

@ExecTimer
def pi_func1(eps):
    """Gregory-Leibniz series"""
    …

@ExecTimer
def pi_func2(eps):
    """ Gauss-Legandre formula"""
    …

@ExecTimer
def pi_func3():
    return pi

eps = 0.00000001
print pi_func1(eps)
print pi_func2(eps)
print pi_func3()
```

Fig. 13. Checking execution time using decorators

```
('Entering', 'pi_func1')
Exited pi_func1, execution time: 0:00:29.527301
3.14159266359
('Entering', 'pi_func2')
Exited pi_func2, execution time: 0:00:00.000012
3.14159264621
('Entering', 'pi_func3')
Exited pi_func3, execution time: 0:00:00.000003
3.14159265359
```

Fig. 14. The output of the execution timer example

4. DISCUSSION AND CONCLUSIONS

A decorator is a callable object (typically a function) that accepts one argument, which is a reference bound to the function being decorated. This object is often called a wrapper object. A decorator can be implemented as a function or class and it has to be called with a single argument. For the class implementation, the `__call__()` method needs to be defined to implement the new behavior. During the decoration process, a callable object instance of that class is created. When calling the decorated function, the wrapper object is called, a function or a callable object depending on the implementation.

Due to the mechanism of binding names to functions as first-class objects that can be stored as variables, decorators can also be nested. This means that multiple decorators can be applied to a single function. Nesting will be resolved from the decorated function up. Nesting of decorators can be very powerful allowing the simultaneous use of decorators from different libraries. More good examples of decorators are given in [6], [9], and [10].

This paper provides a systematic and illustrative introduction to Python decorators. The implementation of decorators in Python is given using conceptual examples that are related to Python decorator mechanism. The examples are accompanied with object diagrams using Lumpy, which can be a useful learning tool. The paper covers templates for decorator implementation using both classes and functions. The paper addresses the peculiarities of the implementation of decorators with arguments. Hopefully, the text will motivate readers to use Python for learning computer science concepts, as well as to explore decorators and other advanced Python techniques.

REFERENCES

[1] Python Programming Language, http://www.python.org

[2] IPython Interactive Computing, http://ipython.org

[3] P. Eby, "Python 2.4 Decorators," *Dr. Dobb's*, No. 372, pp. 54-57, May 2005.

[4] B. Eckel, *Python 3 Patterns, Recipes and Idioms*, http://www.mindviewinc.com/Books/Python3Patterns

[5] *Nose* documentation, http://nose.readthedocs.org

[6] D. Beazley, B. K. Jones, *Python Cookbook*, 3rd ed, O'Reilly Media, May 2013.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addision-Wesley, Nov 1994.

[8] A. B. Downey, *Think Python*, O'Reilly Media, Aug 2012.

[9] M. Harrison, *Guide To: Learning Python Decorators*, CreateSpace, Sep 2013.

[10] *The ASPN online Python Cookbook*, http://aspn.activestate.com/ASPN/Cokbook/Python