

DECORATORS

CS 3080: Python Programming



University of Colorado
Colorado Springs

Functions

- By definition, a decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.
- Before you can understand decorators, you must first understand how functions work. For our purposes, a function returns a value based on the given arguments.

```
def add_one(number):
```

```
    return number + 1
```

```
add_one(2)          # 3
```

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). Only a reference to the function is passed.

```
def sayHello(name):  
    return "Hello {}".format(name)
```

```
def sayBye(name):  
    return "Bye {}".format(name)
```

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

```
tellBob(sayHello)    # 'Hello Bob'
```

```
tellBob(sayBye)      # 'Bye Bob!'
```

tellBob(sayHello) refers to two functions, but in different ways:

- **sayHello is named without parentheses: only a reference to the function is passed. The function is not executed.**
- **tellBob() is written with parentheses, so it will be called**

Inner functions

- Functions inside other functions are called **inner functions**.

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

secondChild()

firstChild()

parent()

What happens when you call the parent() function?

- the order in which the inner functions are defined does not matter: printing only happens when the inner functions are called/executed.
- the inner functions are not defined until the parent function is called: they are locally scoped to parent()

```
>>> parent()
```

Printing from the parent() function

Printing from the secondChild() function

Printing from the firstChild() function

Returning functions from functions

- Python also allows you to use functions as return values.

```
def parent(num):  
    def firstChild():  
        return "String from the firstChild() function"  
  
    def secondChild():  
        return "String from the secondChild() function"  
  
    if num == 1:  
        return firstChild  
    else:  
        return secondChild
```

```
print(parent(1)())  
print(parent(2)())
```

What happens when you call the parent() functions?

```
print(parent(2))
```

What happens when you call the parent() functions?

- **returning firstChild without parentheses means that you are returning a reference to the function firstChild**
- **firstChild() with parentheses refers to the result of calling the function**

Output:

Printing from the firstChild() function

Printing from the secondChild() function

Output:

<function parent.<locals>.firstChild at 0x153db0b26>

Returning functions from functions

- Python also allows you to use functions as return values.

```
def parent(num):  
    def firstChild():  
        return "String from the firstChild() function"  
  
    def secondChild():  
        return "String from the secondChild() function"  
  
    if num == 1:  
        return firstChild  
    else:  
        return secondChild  
first = parent(1)  
print(first)
```

What happens when you execute this function?

first()

Output: String from the firstChild function

Functions are just like any other object in Python!

Output:

<function parent.<locals>.firstChild at 0x153db0b26>

Simple decorator

```
def myDecorator(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 sayWhee():
```

```
    print("Whee!")
```

```
decoratedSayWhee = myDecorator(sayWhee)
```

```
decoratedSayWhee()
```

What happens when you execute
this function?

Decorators wrap a function, modifying its behavior.

<<< Decoration happens at this line!
decoratedSayWhee is a reference to
the wrapper function

```
decoratedSayWhee()
# Something is happening before the function is called.
# Whee!
# Something is happening after the function is called.
```

Second example

- Write a decorator that will only run the decorated function between 7 am and 10 pm.

Syntactic sugar

- Python allows you to use decorators in a simpler way with the @ symbol

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

@myDecorator

Same as: sayWhee = myDecorator(sayWhee)

```
def sayWhee():  
    print("Whee!")
```

Reusing decorators

- Create a new file
- Create decorator called doTwice(), which runs twice any function that it decorates.
- Try it with the sayWhee() function

How to import modules from a different directory (various ways):

<https://blog.finxter.com/python-how-to-import-modules-from-another-folder/>

How to write a module/package: <https://blog.finxter.com/how-to-write-a-python-module-package/>

Reusing decorators

Create a file called decorators.py with following content

```
def doTwice(func):  
    def wrapperDoTwice():  
        func()  
        func()  
    return wrapperDoTwice
```

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

```
from decorators import doTwice
```

```
@doTwice
```

```
def sayWhee():  
    print("Whee!")
```

```
sayWhee()
```

*Output:
Whee!
Whee!*

Decorating Functions With Arguments

```
from decorators import doTwice
```

```
@doTwice
```

```
def sayHello(name):
```

```
    print("Hello {}".format(name))
```

```
sayHello('Bob')    # This will break! wrapperDoTwice() takes 0
```

```
# positional arguments but 1 was given
```

Decorating Functions With Arguments

Where do we put the
function arguments?

```
def myDecorator(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 sayHello(name):  
    print("Hello {}".format(name))
```

```
decoratedSayHello = myDecorator(sayHello)  
decoratedSayHello("Bob")
```

This will break!

Decorating Functions With Arguments

```
def myDecorator(func):  
    def wrapper(*args, **kwargs):  
        print("Something is happening before the function is called.")  
        func(*args, **kwargs)  
        print("Something is happening after the function is called.")  
    return wrapper
```

use `*args` and `kwargs` in the inner wrapper function: it will accept an arbitrary number of positional and keyword arguments**

```
def sayHello(name):  
    print("Hello {}".format(name))
```

```
decoratedSayHello = myDecorator(sayHello)
```

```
decoratedSayHello("Bob")
```

<https://realpython.com/python-kwargs->

<https://realpython.com/python-kwargs-and-args/>

Decorating Functions With Arguments

```
def doTwice(func):  
    def wrapperDoTwice(*args, **kwargs):  
        func(*args, **kwargs)  
        func(*args, **kwargs)  
    return wrapperDoTwice
```

The single asterisk operator `*` can be used on any iterable that Python provides, while the double asterisk operator `**` can only be used on dictionaries

Try This:

```
myList = [1, 2, 3]  
Print(*myList)
```

Returning Values From Decorated Functions

@doTwice

```
def sayHello(name):  
    return "Hello {}".format(name)
```

```
print(sayHello(Bob')) # None
```

**decorator ate the return value from the function
(the wrapper doesn't return a value explicitly)**

Returning Values From Decorated Functions

- To fix this, you need to make sure the wrapper function returns the return value of the decorated function.

```
def doTwice(func):  
    def wrapperDoTwice(*args, **kwargs):  
        func(*args, **kwargs)  
        return func(*args, **kwargs)  
    return wrapperDoTwice
```

Introspection

- **Introspection** is the ability of an object to know about its own attributes at runtime.

```
print.__name__ # print
```

```
print(sayHello.__name__) # wrapperDoTwice
```

- After being decorated, sayHello() has gotten very confused about its identity
- To fix this, decorators should use the **@functools.wraps** decorator, which will preserve information about the original function

Introspection

```
import functools
```

```
def doTwice(func):
```

```
    @functools.wraps(func)
```

```
        def wrapperDoTwice(*args, **kwargs):
```

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

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

```
        return wrapperDoTwice
```

```
print(sayHello.__name__) # sayHello
```

Decorator boilerplate template

```
import functools
```

```
def decorator(func):
```

```
    @functools.wraps(func)
```

```
    def wrapperDecorator(*args, **kwargs):
```

```
        # Do something before
```

```
        value = func(*args, **kwargs)
```

```
        # Do something after
```

```
        return value
```

```
    return wrapperDecorator
```

Real world examples

- A @timer decorator that will measure the time a function takes to execute and print the duration to the console.
- A @debug decorator that will print the arguments a function is called with as well as its return value every time the function is called.
- A @slowDown decorator that will sleep one second before it calls the decorated function

Nesting decorators

@debug

@doTwice

```
def sayHello(name):                                     # debug(doTwice(sayHello))
    print("Hello {}".format(name))
```

```
sayHello('Bob')           # Calling sayHello('Bob')
```

```
                            # Hello Bob
```

```
                            # Hello Bob
```

```
                            # 'sayHello' returned None
```

Nesting decorators

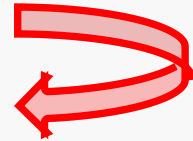
@doTwice

@debug

def **sayHello**(name):

print("Hello {}".format(name))

sayHello('Bob')



doTwice(debug(sayHello))

Calling sayHello('Bob')

Hello Bob

'sayHello' returned None

Calling sayHello('Bob')

Hello Bob

'sayHello' returned None

Decorators with arguments

```
@repeat(numTimes=4)
def sayHello(name):
    print("Hello {}".format(name))

sayHello('Bob')
```

Output
Hello Bob
Hello Bob
Hello Bob
Hello Bob

Passing arguments to your decorators,
Not just the decorated functions

How can we achieve this?

Decorators with arguments

```
def repeat(numTimes): # This is another def that handles the arguments of the decorator
```

```
    def decoratorRepeat(func):
```

```
        @functools.wraps(func)
```

```
        def wrapperRepeat(*args, **kwargs):
```

```
            for _ in range(numTimes):
```

```
                value = func(*args, **kwargs)
```

```
            return value
```

```
        return wrapperRepeat
```

```
    return decoratorRepeat
```

```
@repeat(numTimes=4)
```

```
def sayHello(name):
```

```
    print("Hello {}".format(name))
```

```
sayHello('Bob')      # Hello Bob
```

```
# Hello Bob
```

```
# Hello Bob
```

```
# Hello Bob
```

*an inner function within an inner function
(decorator inception @_@)*

Decorators with arguments

```
def repeat(numTimes): # This is another def that handles the arguments of the decorator
```

```
    def decoratorRepeat(func):
```

```
        @functools.wraps(func)
```

```
        def wrapperRepeat(*args, **kwargs):
```

```
            for _ in range(numTimes):
```

```
                value = func(*args, **kwargs)
```

```
            return value
```

```
        return wrapperRepeat
```

```
    return decoratorRepeat
```

```
@repeat(numTimes=4)
```

```
def sayHello(name):
```

```
    print("Hello {}".format(name))
```

```
sayHello('Bob')      # Hello Bob
```

```
# Hello Bob
```

```
# Hello Bob
```

```
# Hello Bob
```

Innermost function: This is no different from the earlier wrapper functions you have seen, except that it is using **numTimes** supplied from the outside.

Decorators with arguments

```
def repeat(numTimes): # This is another def that handles the arguments of the decorator
```

```
    def decoratorRepeat(func):
```

```
        @functools.wraps(func)
```

```
        def wrapperRepeat(*args, **kwargs):
```

```
            for _ in range(numTimes):
```

```
                value = func(*args, **kwargs)
```

```
            return value
```

```
        return wrapperRepeat
```

```
    return decoratorRepeat
```

```
@repeat(numTimes=4)
```

```
def sayHello(name):
```

```
    print("Hello {}".format(name))
```

```
sayHello('Bob')      # Hello Bob
```

```
                    # Hello Bob
```

```
                    # Hello Bob
```

```
                    # Hello Bob
```

Decorator

function: decoratorRepeat() looks exactly like the decorator functions we have seen earlier

Decorators with arguments

```
def repeat(numTimes): # This is another def that handles the arguments of the decorator
```

```
    def decoratorRepeat(func):
```

```
        @functools.wraps(func)
```

```
        def wrapperRepeat(*args, **kwargs):
```

```
            for _ in range(numTimes):
```

```
                value = func(*args, **kwargs)
```

```
            return value
```

```
        return wrapperRepeat
```

```
    return decoratorRepeat
```

```
@repeat(numTimes=4)
```

```
def sayHello(name):
```

```
    print("Hello {}".format(name))
```

```
sayHello('Bob')      # Hello Bob
```

```
# Hello Bob
```

```
# Hello Bob
```

```
# Hello Bob
```

Outermost function:

repeat(numTimes=4) returns a reference to the decorator function, in this case decoratorRepeat.

Decorators with arguments

```
def repeat(numTimes): # This is another def that handles the arguments of the decorator
```

```
    def decoratorRepeat(func):
```

```
        @functools.wraps(func)
```

```
        def wrapperRepeat(*args, **kwargs):
```

```
            for _ in range(numTimes):
```

```
                value = func(*args, **kwargs)
```

```
            return value
```

```
        return wrapperRepeat
```

```
    return decoratorRepeat
```

```
@repeat(numTimes=4)           # Same as sayHello = repeat(numTimes=4)(sayHello)
```

```
def sayHello(name):
```

```
    print("Hello {}".format(name))
```

```
sayHello('Bob')           # Hello Bob
```

```
                           # Hello Bob
```

```
                           # Hello Bob
```

```
                           # Hello Bob
```

Decorators with arguments

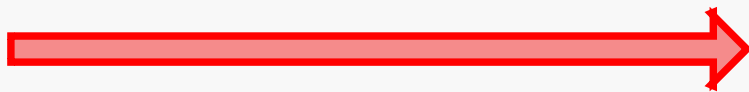
```
def repeat(numTimes): # This is another def that handles the arguments of the decorator
    def decoratorRepeat(func):
        @functools.wraps(func)
        def wrapperRepeat(*args, **kwargs):
            for _ in range(numTimes):
                value = func(*args, **kwargs)
            return value
        return wrapperRepeat
    return decoratorRepeat
```

@repeat

```
def sayHello(name):
    print("Hello {}".format(name))
```

```
sayHello('Bob')
```

wrapperRepeat is never executed



What if we now use the
decorator without arguments?

Decorators with arguments

```
def repeat(numTimes): # This is another def that handles the arguments of the decorator
    def decoratorRepeat(func):
        @functools.wraps(func)
        def wrapperRepeat(*args, **kwargs):
            for _ in range(numTimes):
                value = func(*args, **kwargs)
            return value
        return wrapperRepeat
    return decoratorRepeat
```

What if we now use the
decorator without arguments?

@repeat

```
def sayHello(name):
    print("Hello {}".format(name))
```

```
sayHello('Bob')
```

wrapperRepeat is never executed

Same as:

```
def sayHello(name):
    print("Hello {}".format(name))
```

```
sayHello = repeat(sayHello)
```

```
sayHello('Bob') # Now this is the
```

reference of wrapperRepeat

Both please – With and without arguments

```
def name(_func=None, *, kw1=val1, kw2=val2, ...):
```

```
    def decoratorName(func):
```

```
        ... # Create and return a wrapper function.
```

```
    if _func is None:
```

```
        return decoratorName
```

```
    else:
```

```
        return decoratorName(_func)
```

If @name has been called without arguments, the decorated function will be passed in as _func. If it has been called with arguments, then _func will be None

Both please – With and without arguments

```
def name(_func=None, *, kw1=val1, kw2=val2, ...):
```

```
    def decoratorName(func):
```

```
        ... # Create and return a wrapper function.
```

```
    if _func is None:
```

```
        return decoratorName
```

```
    else:
```

```
        return decoratorName(_func)
```

If: the decorator was called with arguments. Return a decorator function that can return a function wrapper

Else: the decorator was called without arguments. Apply the decorator to the function immediately

Both please

```
def repeat(_func=None, *, numTimes=2):  
    def decoratorRepeat(func):  
        @functools.wraps(func)  
        def wrapperRepeat(*args, **kwargs):  
            for _ in range(numTimes):  
                value = func(*args, **kwargs)  
            return value  
        return wrapperRepeat  
  
    if _func is None:  
        return decoratorRepeat  
    else:  
        return decoratorRepeat(_func)
```

Both please

```
def repeat(_func=None, *, numTimes=2):
    def decoratorRepeat(func):
        @functools.wraps(func)
        def wrapperRepeat(*args, **kwargs):
            for _ in range(numTimes):
                value = func(*args, **kwargs)
            return value
        return wrapperRepeat

    if _func is None:
        return decoratorRepeat
    else:
        return decoratorRepeat(_func)
```

Compare this with the original @repeat.
The only changes are the added _func parameter and the if-else at the end.

Both please

@repeat *# Same as: sayWhee = repeat(sayWhee)*

```
def sayWhee():  
    print("Whee!")
```

@repeat(numTimes=3) *# Same as: sayWhee2 = repeat(num_times=3)(sayWhee2)*

```
def sayWhee2():  
    print("Whee2!")
```

sayWhee() *# Whee!*

sayWhee2() *# Whee!*

Whee2!

Whee2!

Whee2!

Function attributes

- Everything in Python is an object, and almost everything has attributes and methods.
- In python, functions too are objects. So they have attributes like other objects.

```
def foo():
```

```
    pass
```

```
foo.gender = 'male'
```

```
foo.name = 'Bob'
```

```
print(foo.gender)          # male
```

```
print(foo.name)           # Bob
```

Stateful Decorators

- You can save the state of a function by using **function attributes**.

```
def countCalls(func):  
    @functools.wraps(func)  
    def wrapperCountCalls(*args, **kwargs):  
        wrapperCountCalls.numCalls += 1  
        print("Call {} of {}".format(wrapperCountCalls.numCalls, func.__name__))  
        return func(*args, **kwargs)  
    wrapperCountCalls.numCalls = 0  
    return wrapperCountCalls
```

```
@countCalls  
def passFunc():  
    pass
```

```
passFunc() # Call 1 of 'passFunc'  
passFunc() # Call 2 of 'passFunc'  
print(passFunc.numCalls) # 2
```