Decorators

Decorators can be thought of as functions which modify the *functionality* of another function. They help to make your code shorter and more "Pythonic".

To properly explain decorators we will slowly build up from functions. Make sure to restart the Python and the Notebooks for this lecture to look the same on your own computer. So lets break down the steps:

Functions Review

In [1]:
```
def func():
    return 1
```

In [2]:
```
func()
```

Out[2]: 1

Scope Review

Remember from the nested statements lecture that Python uses Scope to know what a label is referring to. For example:

In [6]:
```
s = 'Global Variable'

def func():
    print locals()
```

Remember that Python functions create a new scope, meaning the function has its own namespace to find variable names when they are mentioned within the function. We can check for local variables and global variables with the local() and globals() functions. For example:

In [7]:
```
print globals()
```

```
{'_dh': [u'/Users/marci/Udemy-Complete-Python-Bootcamp'], '__': '', '_i': u"s
= 'Global Variable'\n\ndef func():\n    print locals()", 'quit': <IPython.cor
e.autocall.ZMQExitAutocall object at 0x1037e0a10>, '__builtins__': <module '_
_builtin__' (built-in)>, 's': 'Global Variable', '_ih': ['', u'def func():\n
   return 1', u'func()', u"s = 'Global Variable'\n\ndef func():\n    print lo
cals()", u'print globals()', u'print globals().keys()', u"s = 'Global Variabl
e'\n\ndef func():\n    print locals()", u'print globals()'], '__builtin__': <
module '__builtin__' (built-in)>, '_2': 1, 'func': <function func at 0x10445a
a28>, '__name__': '__main__', '___': '', '_': 1, '_sh': <module 'IPython.cor
e.shadowns' from '//anaconda/lib/python2.7/site-packages/IPython/core/shadown
s.pyc'>, '_i7': u'print globals()', '_i6': u"s = 'Global Variable'\n\ndef fun
c():\n    print locals()", '_i5': u'print globals().keys()', '_i4': u'print g
lobals()', '_i3': u"s = 'Global Variable'\n\ndef func():\n    print locals
()", '_i2': u'func()', '_i1': u'def func():\n    return 1', '__doc__': 'Autom
atically created module for IPython interactive environment', '_iii': u'print
globals()', 'exit': <IPython.core.autocall.ZMQExitAutocall object at 0x1037e0
a10>, 'get_ipython': <bound method ZMQInteractiveShell.get_ipython of <IPytho
n.kernel.zmq.zmqshell.ZMQInteractiveShell object at 0x1037c3990>>, '_ii': u'p
rint globals().keys()', 'In': ['', u'def func():\n    return 1', u'func()',
u"s = 'Global Variable'\n\ndef func():\n    print locals()", u'print globals
()', u'print globals().keys()', u"s = 'Global Variable'\n\ndef func():\n    p
rint locals()", u'print globals()'], '_oh': {2: 1}, 'Out': {2: 1}}
```

Here we get back a dictionary of all the global variables, many of them are predefined in Python. So let's go ahead and look at the keys:

In [8]:
```
print globals().keys()
```

```
['_dh', '__', '_i', 'quit', '__builtins__', 's', '_ih', '__builtin__', '_2',
'func', '__name__', '___', '_', '_sh', '_i8', '_i7', '_i6', '_i5', '_i4', '_i
3', '_i2', '_i1', '__doc__', '_iii', 'exit', 'get_ipython', '_ii', 'In', '_o
h', 'Out']
```

Note how **s** is there, the Global Variable we defined as a string:

In [10]:
```
globals()['s']
```

Out[10]:
```
'Global Variable'
```

Now lets run our function to check for any local variables in the func() (there shouldn't be any)

In [11]:
```
func()
```

```
{}
```

Great! Now lets continue with building out the logic of what a decorator is. Remember that in Python **everything is an object**. That means functions are objects which can be assigned labels and passed into other functions. Lets start with some simple examples:

In [12]:
```
def hello(name='Jose'):
    return 'Hello '+name
```

In [13]:
```
hello()
```

Out[13]:
```
'Hello Jose'
```

Assign a label to the function. Note that e are not using parentheses here because we are not calling the function hello, instead we are just putting it into the greet variable.

In [14]:
```
greet = hello
```

In [15]:
```
greet
```

Out[15]:
```
<function __main__.hello>
```

In [16]:
```
greet()
```

Out[16]:
```
'Hello Jose'
```

This assignment is not attached to the original function:

In [17]:
```
del hello
```

In [18]:
```
hello()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-18-a803225a2f97> in <module>()
----> 1 hello()

NameError: name 'hello' is not defined
```

In [19]:
```
greet()
```

Out[19]:
```
'Hello Jose'
```

Functions within functions

Great! So we've seen how we can treat functions as objects, now lets see how we can define functions inside of other functions:

In [26]:
```
def hello(name='Jose'):
    print 'The hello() function has been executed'

    def greet():
        return '\t This is inside the greet() function'

    def welcome():
        return "\t This is inside the welcome() function"

    print greet()
    print welcome()
    print "Now we are back inside the hello() function"
```

In [27]:
```
hello()
```

```
The hello() function has been executed
         This is inside the greet() function
         This is inside the welcome() function
Now we are back inside the hello() function
```

In [29]:
```
welcome()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-29-efaf77b113fd> in <module>()
----> 1 welcome()

NameError: name 'welcome' is not defined
```

Note how due to scope, the welcome() function is not defined outside of the hello() function. Now lets learn about returning functions from within functions:

Returning Functions

In [34]:
```
def hello(name='Jose'):

    def greet():
        return '\t This is inside the greet() function'

    def welcome():
        return "\t This is inside the welcome() function"

    if name == 'Jose':
        return greet
    else:
        return welcome
```

In [39]:
```
x = hello()
```

Now lets see what function is returned if we set x = hello(), note how the closed parenthesis means that name ahs been defined as Jose.

In [40]:
```
x
```

Out[40]:
```
<function __main__.greet>
```

Great! Now we can see how x is pointing to the greet function inside of the hello function.

In [42]:
```
print x()
```

```
This is inside the greet() function
```

Lets take a quick look at the code again.

In the if/else clause we are returning greet and welcome, not greet() and welcome().

This is because when you put a pair of parentheses after it, the function gets executed; whereas if you don't put parenthesis after it, then it can be passed around and can be assigned to other variables without executing it.

When we write x = hello(), hello() gets executed and because the name is Jose by default, the function greet is returned. If we change the statement to x = hello(name = "Sam") then the welcome function will be returned. We can also do print hello()() which outputs now you are in the greet() function.

Functions as Arguments

Now lets see how we can pass functions as arguments into other functions:

In [43]:
```
def hello():
    return 'Hi Jose!'

def other(func):
    print 'Other code would go here'
    print func()
```

In [45]:
```
other(hello)
```

```
Other code would go here
Hi Jose!
```

Great! Note how we can pass the functions as objects and then use them within other functions. Now we can get started with writing our first decorator:

Creating a Decorator

In the previous example we actually manually created a Decorator. Here we will modify it to make its use case clear:

In [46]:
```
def new_decorator(func):

    def wrap_func():
        print "Code would be here, before executing the func"

        func()

        print "Code here will execute after the func()"

    return wrap_func

def func_needs_decorator():
    print "This function is in need of a Decorator"
```

In [47]:
```
func_needs_decorator()
```

```
This function is in need of a Decorator
```

In [50]:
```
# Reassign func_needs_decorator
func_needs_decorator = new_decorator(func_needs_decorator)
```

In [51]:
```
func_needs_decorator()
```

```
Code would be here, before executing the func
This function is in need of a Decorator
Code here will execute after the func()
```

So what just happened here? A decorator simple wrapped the function and modified its behavior. Now lets understand how we can rewrite this code using the @ symbol, which is what Python uses for Decorators:

In [52]:
```
@new_decorator
def func_needs_decorator():
    print "This function is in need of a Decorator"
```

In [53]:
```
func_needs_decorator()
```

```
Code would be here, before executing the func
This function is in need of a Decorator
Code here will execute after the func()
```

Great! You've now built a Decorator manually and then saw how we can use the @ symbol in Python to automate this and clean our code. You'll run into Decorators a lot if you begin using Python for Web Development, such as Flask or Django!