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 run every cell in this Notebook for this lecture to look the same on your own computer.

So let's 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 [ ]:

```
s = 'Global Variable'

def check_for_locals():
    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 `locals()` and `globals()` functions. For example:

In [ ]:

```
print(globals())
```

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 [ ]:

```
print(globals().keys())
```

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

In [ ]:

```
globals()['s']
```

Now let's run our function to check for local variables that might exist inside our function (there shouldn't be any)

In [ ]:

```
check_for_locals()
```

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 [3]:

```
def hello(name='Jose'):
    return 'Hello '+name
```

In [4]:

```
hello()
```

Out[4]:

```
'Hello Jose'
```

Assign another label to the function. Note that we are not using parentheses here because we are not calling the function **hello**, instead we are just passing a function object to the **greet** variable.

In [5]:

```
greet = hello
```

In [6]:

```
greet
```

Out[6]:

```
<function __main__.hello>
```

In [7]:

```
greet()
```

Out[7]:

```
'Hello Jose'
```

So what happens when we delete the name **hello**?

In [8]:

```
del hello
```

In [9]:

```
hello()
```

```
---------------------------------------------------------------
--------------
NameError                                 Traceback (most rec
ent call last)
<ipython-input-9-a75d7781aaeb> in <module>()
----> 1 hello()

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

In [10]:

```
greet()
```

Out[10]:

```
'Hello Jose'
```

Even though we deleted the name **hello**, the name **greet** *still points to* our original function object. It is important to know that functions are objects that can be passed to other objects!

Functions within functions

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

In [11]:

```
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 [12]:

```
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 [13]:

```
welcome()
```

```
---------------------------------------------------------------
--------------
NameError                                 Traceback (most rec
ent call last)
<ipython-input-13-a401d7101853> 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 [14]:

```
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
```

Now let's see what function is returned if we set x = hello(), note how the empty parentheses means that name has been defined as Jose.

In [15]:

```
x = hello()
```

In [16]:

```
x
```

Out[16]:

```
<function __main__.hello.<locals>.greet>
```

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

In [17]:

```
print(x())
```

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

Let's 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 parentheses 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 *This is inside the greet() function*.

Functions as Arguments

Now let's see how we can pass functions as arguments into other functions:

In [18]:

```
def hello():
    return 'Hi Jose!'

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

In [19]:

```
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 [20]:

```
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 [21]:

```
func_needs_decorator()
```

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

In [22]:

```
# Reassign func_needs_decorator
func_needs_decorator = new_decorator(func_needs_decorator)
```

In [23]:

```
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 simply wrapped the function and modified its behavior. Now let's understand how we can rewrite this code using the @ symbol, which is what Python uses for Decorators:

In [24]:

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

In [25]:

```
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!