Function Decorators and Closures:

Function decorators let us “mark” functions in the source code to enhance their behavior in some way. This is powerful stuff, but mastering it requires understanding closures.

One of the newest reserved keywords in Python is nonlocal, introduced in Python 3.0. You can have a profitable life as a Python programmer without ever using it if you adhere to a strict regimen of class-centered object orientation. However, if you want to implement your own function decorators, you must know closures inside out, and then the need for nonlocal becomes obvious.

Aside from their application in decorators, closures are also essential for effective asyn‐ chronous programming with callbacks, and for coding in a functional style whenever it makes sense.

Decorators 1.01:

A decorator is a callable that takes another function as argument (the decorated function).2 The decorator may perform some processing with the decorated function and returns it or replaces it with another function or callable object:

@decorate

def target():

print('running target()')

Has the same effect as writing this:

def target():

print('running target()')

target = decorate(target)

The end result is the same: at the end of either of these snippets, the target name does not necessarily refer to the original target function, but to whatever function is re‐ turned by decorate(target).

A decorator usually replaces a function with a different one:

>>> def deco(func):

... def inner():

... print('running inner()')

... return inner

...

>>> @deco

... def target():

... print('running target()')

...

>>> target()

running inner()

>>> target

<function deco.<locals>.inner at 0x10063b598>

Here what’s happening is that if a function receives another function as an argument, we have a decorator. The decorator is not born till you actually call it upon another function.

Strictly speaking, decorators are just syntactic sugar. As we just saw, you can always simply call a decorator like any regular callable, passing another function. Sometimes that is actually convenient, especially when doing metaprogramming—changing pro‐ gram behavior at runtime.

To summarize: the first crucial fact about decorators is that they have the power to replace the decorated function with a different one. The second crucial fact is that they are executed immediately when a module is loaded. This is explained next.

When Python Executes Decorators:

A key feature of decorators is that they run right after the decorated function is defined. That is usually at *import time*.

The registration.py module:

registry = []

def register(func):

    print('running register(%s)' % func)

    registry.append(func)

    return func

@register

def f1():

    print('running f1()')

@register

def f2():

    print('running f2()')

def f3():

    print('running f3()')

def main():

    print('running main()')

    print('registry ->', registry)

    f1()

    f2().

    f3()

if \_\_name\_\_=='\_\_main\_\_':

    main()

Here we have the list registry which holds the returned functions by the register decorator funcion. Then we have three functions being defined, but only two of them are being decorated, meaning that only two of them are being held in registry . Then we mess with the main function. Why? Because we demonstrate that the decorator functions run at import time. Here we are telling to main() to be executed only if its run as a script, not when is imported as a module.

The output of running registration.py as a script looks like this:

$ python3 registration.py

running register(<function f1 at 0x100631bf8>)

running register(<function f2 at 0x100631c80>)

running main()

registry -> [<function f1 at 0x100631bf8>, <function f2 at 0x100631c80>]

running f1()

running f2()

running f3()

$

Note that register runs (twice) before any other function in the module. When reg ister is called, it receives as an argument the function object being decorated—for example, <function f1 at 0x100631bf8.

After the module is loaded, the registry holds references to the two decorated functions: f1 and f2. These functions, as well as f3, are only executed when explicitly called by main.

If registration.py is imported (and not run as a script), the output is this::

>>> import registration

running register(<function f1 at 0x100631bf8>)

running register(<function f2 at 0x100631c80>)

>>>

At this time, if you look at the registry, here is what you get:

>>> registration.registry

[<function f1 at 0x10063b1e0>, <function f2 at 0x10063b268>]

>>>

The main point of this example is to emphasize that function decorators are executed as soon as the module is imported, but the decorated functions only run when they are explicitly invoked. This highlights the difference between what Pythonistas call *import* *time* and *runtime*.

Similar decorators are used in many Python web frameworks to add functions to some central registry—for example, a registry mapping URL patterns to functions that generate HTTP responses.

Variable Scope Rules:

Now let’s talk bout the variable’s scopes. This term refers to the place where the variables live, and from where are they called and returned to. Frist of all let’s look to an obvius example which is very easy for everybody to predict its behavior.

Function reading a local and a global variable:

>>> def f1(a):

... print(a)

... print(b)

...

>>> f1(5)

5

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 3, in f1

NameError: name 'b' is not defined

>>>

This was really easy; the variable b was not defined so it raises an error when calling the un-existing variable. And we can fix this by just defining another variable, let’s do a global one. The result will be that both lines will be printed with no errors. All is good. But what would happen is we define two variables, a local one, and a global one with the same name? You’ll think that the answer is very obvious, the local one will be printed right? But what would happen if the local variable were actually defined after the call?

Variable b is local, because it is assigned a value in the body of the function:

>>> b = 6

>>> def f2(a):

... print(a)

... print(b)

... b = 9

...

>>> f2(5)

5

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 3, in f2

UnboundLocalError: local variable 'b' referenced before assignment

>>>

Here you will be asking yourself why if there is a global variable, didn’t Python just print that one? The problem is that because of the way of python works and compiles the bytecode, the testing for the type of variables happens before everything else. We can see this very clear is we see the bytecode with dis. Summarizing, Python first notices that there are two local variables, a, and b. Then the function is called with a being the argument of the function, hence, a local variable. A is defined then and since b is also in the locals dictionary, the interpreter won’t check for global variable because it already has one which there is no need to be defined because he is expecting you to define it before call it. The variable is in the body, the interpreter knows it, now if you call it and then define it, you’ll get an error.=

If we want the interpreter to treat b as a global variable in spite of the assignment within the function, we use the global declaration:

>>> def f3(a):

... global b

... print(a)

... print(b)

... b = 9

...

>>> f3(3)

3

6

>>> b

9

>>> f3(3)

a = 3

b = 8

b = 30

>>> b

30

>>>

Closures:

A closure is a function with an extended scope that encompasses nonglobal variables. This variables are the ones that exist outside the closure function, but not in the global enviroment, but in the higher function’s.

Let’s jump right into a class that works like this so we can see this behavior in action and later the same but with a function.

class Averager():

    def \_\_init\_\_(self):

        self.series = []

    def \_\_call\_\_(self, new\_value):

        self.series.append(new\_value)

        total = sum(self.series)

        return total/len(self.series)

Here we have a class which has a variable called series, inside the instance of the class Averager. This holds all the values that we want to introduce in the future to make the average of all the previous values.

Now, let’s take a look to another code, this time written as a function.

def make\_averager():

    series = []

    def averager(new\_value):

        series.append(new\_value)

        total = sum(series)

        return total/len(series)

    return averager

return averager When invoked, make\_averager returns an averager function object. Each time an averager is called, it appends the passed argument to the series, and computes the current average