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>