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 asynchronous 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 program 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 obvious 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 past argument to the series, and computes the current average

Output from functional way:

>>> avg = make\_averager()

>>> avg(10)

10.0

>>> avg(11)

10.5

>>> avg(12)

11.0

What we are trying to do here is to create a callable object either from the function or the class. To do so using classes we need to define the \_\_call\_\_ method, but when we use the functional way, we don’t need to do so since the \_\_call\_\_ method is already defined. Either way, we just call avg(n) to include n in the series and get the updated mean.

In the Averager class’s case we can clearly see that the history is saved in the attribute self.series which is a list. Now let’s see where the history is stored inside of the function:

The history is obviously stored in the variable series, which is a local variable since the variable is initialized inside of the body of the function. But when avg(10) is called, make\_averager has already returned, and its local scope is long gone. Within averager, series is what we call a free variable. This is a technical term meaning a variable that is not bound in the local scope.

What this means is that the closure for averager extends the scope of that funcion to include the binding for the free variable series:

Inspecting the function:

>>> avg.\_\_code\_\_.co\_varnames

('new\_value', 'total')

>>> avg.\_\_code\_\_.co\_freevars

('series',)

Inspecting the returned averager object shows how Python keeps the names of local and free variables in the \_\_code\_\_ attribute that represents the compiled body of the function.

Visual representation of the actual closure, and the loss of the closure:

Closure

def make\_averager():

    series = []

Free

Variable

    def averager(new\_value):

        series.append(new\_value)

        total = sum(series)

        return total/len(series)

    return averager

Here we can see how the scope from the inner function averager has been extended till the variable series, this is why we say that the local scope is gone, because now this variable is inside another scope too.

Getting the closures within a function:

The binding for series is kept in the \_\_closure\_\_ attribute of the returned function avg. Each item in avg.\_\_closure\_\_ corresponds to a name in avg.\_\_code\_\_.co\_freevars. These items are cells, and they have an attribute called cell\_contents where the actual value can be found.

>>> avg.\_\_code\_\_.co\_freevars

('series',)

>>> avg.\_\_closure\_\_

(<cell at 0x107a44f78: list object at 0x107a91a48>,)

>>> avg.\_\_closure\_\_[0].cell\_contents

[10, 11, 12]

Closures summary:

To summarize: a closure is a function that retains the bindings of the free variables that exist when the function is defined, so that they can be used later when the function is invoked and the defining scope is no longer available. Note that the only situation in which a function may need to deal with external variables that are nonglobal is when it is nested in another function. We can access to its values by searching inside of the function using the \_\_closures\_\_ command, and then the command cell\_contents.

The nonlocal Declaration:

For this example, we’ll enhance our averager function from before by not storing a lot of values, but instead only two values, the total, and how many values we'd store till now. We’ll do so by adding the new value to a total variable and adding one to the count of how many values we have. This way we can get the math done faster, and we save ourselves a lot of memory as the averager increases its numbers.

def make\_averager():

count = 0

total = 0

def averager(new\_value):

count += 1

total += new\_value

return total / count

return averager

If we try this code, this is the result we’ll get:

>>> avg = make\_averager()

>>> avg(10)

Traceback (most recent call last):

...

UnboundLocalError: local variable 'count' referenced before assignment

>>>

The problem is that the statement count += 1 actually means the same as count = count + 1, when count is a number or any immutable type. So we are actually assigning to count in the body of averager, and that makes it a local variable. The same problem affects the total variable.

We did not have this problem in the other example because we never assigned to the series name; we only called series.append and invoked sum and len on it. So, we took advantage of the fact that lists are mutable.

But with immutable types like numbers, strings, tuples, etc., all you can do is read, but never update. If you try to re-bind them, as in count = count + 1, then you are implicitly creating a local variable count. It is no longer a free variable, and therefore it is not saved in the closure. To work around this, the nonlocal declaration was introduced in Python 3. It lets you flag a variable as a free variable even when it is assigned a new value within the function. If a new value is assigned to a nonlocal variable, the binding stored in the closure is changed.

Working code:

def make\_averager():

count = 0

total = 0

def averager(new\_value):

nonlocal count, total

count += 1

total += new\_value

return total / count

return averager

Here we are explicitly saying that the variables count, and total are not to be found in the local enviroment and therefore, we must go to a higher scope and retrieve them.

Implementing a Simple Decorator:

Now that we have Python closures covered, we can effectively implement decorators with nested functions. The decorator we’ll see next is a decorator that clocks every invocation of the decorated function and prints the elapsed time, the arguments passed, and the result of the call.

A simple decorator to output the running time of functions:

def clock(func):

def clocked(\*args):

t0 = time.perf\_counter()

result = func(\*args)

elapsed = time.perf\_counter() - t0

name = func.\_\_name\_\_

arg\_str = ', '.join(repr(arg) for arg in args)

print('[%0.8fs] %s(%s) -> %r' % (elapsed, name, arg\_str, result))

return result

return clocked

Importing the module:

import time

from clockdeco import clock

@clock

def snooze(seconds):

    time.sleep(seconds)

@clock

def factorial(n):

    return 1 if n < 2 else n\*factorial(n-1)

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

    print('\*' \* 40, 'Calling snooze(.123)')

    snooze(.123)

    print('\*' \* 40, 'Calling factorial(6)')

    print('6! =', factorial(6))

Output:

$ python3 clockdeco\_demo.py

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Calling snooze(123)

[0.12405610s] snooze(.123) -> None

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Calling factorial(6)

[0.00000191s] factorial(1) -> 1

[0.00004911s] factorial(2) -> 2

[0.00008488s] factorial(3) -> 6

[0.00013208s] factorial(4) -> 24

[0.00019193s] factorial(5) -> 120

[0.00026107s] factorial(6) -> 720

6! = 720

How it works:

Remember that this code:

@clock

def factorial(n):

    return 1 if n < 2 else n\*factorial(n-1)

Actually does this:

def factorial(n):

    return 1 if n < 2 else n\*factorial(n-1)

factorial = clock(factorial)

So factorial now actually holds a reference to the clocked function. From now on, each time factorial(n) is called, clocked(n) gets executed. In essence, clocked does the following:

1. Records the initial time t0.

2. Calls the original factorial, saving the result.

3. Computes the elapsed time.

4. Formats and prints the collected data.

5. Returns the result saved in step 2.

Decorators in the Standard Library:

Python has three built-in functions that are designed to decorate methods: property, classmethod, and staticmethod. We’ll discuss this decorators later. For now, let’s center our attention in this two decorators from the functools module:

Memorization with functools.lru\_cache:

The letters LRU stand for Least Recently Used, meaning that the growth of the cache is limited by discarding the entries that have not been read for a while. This decorator allows us to save time and resources when it comes to repeat several times the same expensive process by saving its result for an specific argument in a dictionary, and retrieving it if is called again, instead of calling the whole expensive process. the best example to explain this behavior is a Fibonacci function:

@functools.lru\_cache()

@clock

def factorial(n):

    return 1 if n < 2 else n\*factorial(n-1)

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

    print('\*' \* 40, 'Calling snooze(.123)')

    snooze(.123)

    print('\*' \* 40, 'Calling factorial(6)')

    print('6! =', factorial(6))

Execution time is halved, and the function is called only once for each value of n:

$ python3 fibo\_demo\_lru.py

[0.00000119s] fibonacci(0) -> 0

[0.00000119s] fibonacci(1) -> 1

[0.00010800s] fibonacci(2) -> 1

[0.00000787s] fibonacci(3) -> 2

[0.00016093s] fibonacci(4) -> 3

[0.00001216s] fibonacci(5) -> 5

[0.00025296s] fibonacci(6) -> 8

Memorization with functools.lru\_cache:

The new functools.singledispatch decorator in Python 3.4 allows each module to contribute to the overall solution, and lets you easily provide a specialized function even for classes that you can’t edit. If you decorate a plain function with @singledispatch, it becomes a generic function: a group of functions to perform the same operation in different ways, depending on the type of the first argument. This means that depending on the type of the argument that you give to the function, you can decide how the function will behave, and alter its result.

from functools import singledispatch

from collections import abc

import numbers

import html

@singledispatch

def htmlize(obj):

    content = html.escape(repr(obj))

    return '<pre>{}</pre>'.format(content)

@htmlize.register(str)

def \_(text):

    content = html.escape(text).replace('\n', '<br>\n')

    return '<p>{0}</p>'.format(content)

@htmlize.register(numbers.Integral)

def \_(n):

    return '<pre>{0} (0x{0:x})</pre>'.format(n)

@htmlize.register(tuple)

@htmlize.register(abc.MutableSequence)

def \_(seq):

    inner = '</li>\n<li>'.join(htmlize(item) for item in seq)

    return '<ul>\n<li>' + inner + '</li>\n</ul>'

This means that for each type of data we pass into the argument to the same function, we’ll get different results. This method is better than adding a sequence of if, elif, elif, elif command lines.

Stacked Decorators:

When two decorators @d1 and @d2 are applied to a function f in that order, the result is the same as f = d1(d2(f)).

In other words, this:

@d1

@d2

def f():

print('f')

Is the same as:

def f():

print('f')

f = d1(d2(f))

Parametrized Decorators:

When parsing a decorator in source code, Python takes the decorated function and passes it as the first argument to the decorator function. So how do you make a decorator accept other arguments? The answer is: make a decorator factory that takes those ar‐ guments and returns a decorator, which is then applied to the function to be decorated. Confusing? Sure. Let’s start with an example based on the simplest decorator we’ve seen, the register.

registry = []

def register(func):

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

    registry.append(func)

    return func

@register

def f1():

    print('running f1()')

print('running main()')

print('registry ->', registry)

f1()

A Parameterized Registration Decorator:

registry = set()

def register(active=True):

    def decorate(func):

        print('running register(active=%s)->decorate(%s)'% (active, func))

        if active:

            registry.add(func)

        else:

            registry.discard(func)

        return func

    return decorate

@register(active=False)

def f1():

    print('running f1()')

@register()

def f2():

    print('running f2()')

def f3():

    print('running f3()')

Explained:

In order to make it easy to enable or disable the function registration performed by register, we’ll make it accept an optional active parameter which, if False, skips registering the decorated function. Conceptually, the new register function is not a decorator but a decorator factory. When called, it returns the actual decorator that will be applied to the target function.