Department of Computer Science and Engineering (UG Studies)

PES University, Bangalore, India

Introduction to Computing using Python (UE19CS101)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Nested functions and Closures in Python

Inner functions (Nested functions)

Encapsulation: **You use inner functions to protect them from everything happening outside of the function, meaning that they are hidden from the global scope.**

A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope.

Example 1:

```
def outer(num1):
    def inner_inc(num1):  # inner function hidden from outer code
        return num1 + 1

    num2 = inner_inc(num1) # call internally inner function
    print(num1, num2)

inner_inc(10)  # Try calling inner function from here i.e., from outside
```

```
Traceback (most recent call last):
  File "inner.py", line 7, in <module>
    inner_inc()
NameError: name 'inner_inc' is not defined
```

```
outer(10) # call function outer
```

Output:

```
10 11
```

Note: Keep in mind that this is just an example. Although this code does achieve the desired result, it's probably better to make `inner_inc()` a top-level "private" function using a leading underscore: **`_inner_increment()`.**

`Example 2:` **The following recursive example is a slightly better use case for a nested function:**

```
def factorial(number):
    # Error handling
    if not isinstance(number, int):
        raise TypeError("Sorry. 'number' must be an integer.")
    if not number >= 0:
        raise ValueError("Sorry. 'number' must be zero or positive.")

    def inner_factorial(number): # inner function
        if number <= 1:
            return 1
        return number*inner_factorial(number-1)

    return inner_factorial(number)  # call inner function and return its result

print(factorial(4)) # Call the outer function.
```

Output:

```
24
```

Nonlocal variable in a nested function:

In Python, these non-local variables are read only by default and we must declare them explicitly as non-local (using nonlocal keyword) in order to modify them.

Following is an example of a nested function accessing a non-local variable.

```
def print_msg(msg):

      def printer():
            print(msg) # accessing a non-local variable msg
      printer() # call internally inner function

print_msg("Hello")
```

We can see that the nested function printer() was able to access the non-local variable msg of the enclosing function.

Closures and Factory Functions

What's a Closure?

A closure simply causes the inner function to remember the state of its environment when called. Beginners often think that a closure is the inner function, but it's really caused by the inner function. **The closure "closes" the local variable on the stack, and this stays around after the stack creation has finished executing.**

A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory.

- **It is a record that stores a function together with an environment**: a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.
- A closure—unlike a plain function—allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

Example 1: # Python program to illustrate closures

```
def outerFunction(text): # factory function
    text = text

    def innerFunction():# inner function
        print(text)

    # reference of inner function is returned by the factory function.
    return innerFunction  # Note we are returning function WITHOUT parenthesis

if __name__ == '__main__':
    myFunction = outerFunction('Hey!')
    myFunction()
```

Output:

```
Hey!
```

Important observations:

- **As observed from above code, closures help to invoke function outside their scope.**
- **The function innerFunction has its scope only inside the outerFunction. But with the use of closures we can easily extend its scope to invoke a function outside its scope.**

Example 2:

```
def generate_power(number):

    def nth_power(power):  # inner function
        return number ** power

    # reference of inner function is returned by the factory function.
    return nth_power

#Examples of use:
raise_two = generate_power(2)
raise_three = generate_power(3)

print(raise_two(7))
print(raise_three(5))
```

Output:

```
128
243
```

What's Happening in the Example

Let's take a look at what is going on in that example:

1. **generate_power() is a factory function, which simply means that it creates a new function each time it is called and then returns the newly created function**. Thus, raise_two() and raise_three() are the newly created functions.
2. What does this new, inner function do? It takes a single argument, power, and returns number**power.
3. Where does the inner function get the value of number from? This is where the closure comes into play: nth_power() gets the value of power from the outer function, the factory function. Let's step through this process:
    - Call the outer function: generate_power(2).
    - Build nth_power(), which takes a single argument power.
    - Take a snapshot of the state of nth_power(), which includes power=2.
    - Pass that snapshot into generate_power().
    - Return nth_power().

   To put it another way, the closure "initializes" the number bar in nth_power() and then returns it. Now, whenever you call that newly returned function, it will always see its own private snapshot that includes power=2.

When do we have a closure?

As seen from the above example, we have a closure in Python when a nested function references a value in its enclosing scope.

The criteria that must be met to create closure in Python are summarized in the following points.

- We must have a nested function (function inside a function).
- The nested function must refer to a value defined in the enclosing function.
- The enclosing function must return the nested function.

When to use closures?

So what are closures good for?

Closures can avoid the use of global values and provides some form of data hiding. It can also provide an object oriented solution to the problem.

When there are few methods (one method in most cases) to be implemented in a class, closures can provide an alternate and more elegant solutions. But when the number of attributes and methods get larger, better implement a class.

Here is a simple example where a closure might be more preferable than defining a class and making objects. But the preference is all yours.

```
def make_multiplier_of(n):

      def multiplier(x):      # inner function
            return x * n

      return multiplier

times3 = make_multiplier_of(3) # Multiplier of 3
times5 = make_multiplier_of(5) # Multiplier of 5

print(times3(9)) # Output: 27
print(times5(3)) # Output: 15
print(times5(times3(2))) # Output: 30
```

On a concluding note, it is good to point out that the values that get enclosed in the closure function can be found out.

All function objects have a __closure__ attribute that returns a tuple of cell objects if it is an inner function with closure. Referring to the example above, we know times3 and times5 are inner functions with closure.

```
print(make_multiplier_of.__closure__)  # output is None, because it's a outer function
print(times3.__closure__)  # (<cell at 0x0000000002D155B8: int object at 0x000000001E39B6E0>,)
```

The cell object has the attribute cell_contents which stores the closed value.

```
print(times3.__closure__[0].cell_contents) # output is 3
print(times5.__closure__[0].cell_contents) # output is 5
```

Some more examples on closures:

```
import logging
logging.basicConfig(filename='example.log', level=logging.INFO)

def logger(func):
    def log_func(*args):
        logging.info(
            'Running "{}" with arguments {}'.format(func.__name__, args))
        print(func(*args))
    # Necessary for closure to work (returning WITHOUT parenthesis)
    return log_func

def add(x, y):
    return x+y

def sub(x, y):
    return x-y

add_logger = logger(add)
sub_logger = logger(sub)

add_logger(3, 3)
add_logger(4, 5)

sub_logger(10, 5)
sub_logger(20, 10)
```

OUTPUT:

```
6
9
5
10
```

When and why to use Closures:

1. As closures are used as callback functions, they provide some sort of data hiding. This helps us to reduce the use of global variables.
2. When we have few functions in our code, closures prove to be efficient way. But if we need to have many functions, then go for class (OOP).

A Real World Example

```
def has_permission(page):
    def inner(username):
        if username == 'Admin':
            return "'{0}' does have access to {1}.".format(username, page)
        else:
            return "'{0}' does NOT have access to {1}.".format(username, page)
    return inner


current_user = has_permission('Admin Area')
print(current_user('Admin'))

random_user = has_permission('Admin Area')
print(random_user('Not Admin'))
```

This is a simplified function to check if a certain user has the correct permissions to access a certain page. You could easily modify this to grab the user in session to check if they have the correct credentials to access a certain route. Instead of checking if the user is just equal to 'Admin', you could query the database to check the permission and then return the correct view depending on whether the credentials are correct or not.

Conclusion

The use of closures and factory functions is the most common and powerful use for inner functions. In most cases, when you see a decorated function, the decorator is a factory function that takes a function as argument and returns a new function that includes the old function inside the closure. Stop. Take a deep breath. Grab a coffee. Read that again.

To put it another way, a decorator is just syntactic sugar for implementing the process outlined in the generate_power() example.

I'll leave you with a final example:

```
def generate_power(exponent):
    def decorator(f):
        def inner(*args):
            result = f(*args)
            return exponent**result
        return inner
    return decorator


@generate_power(2)
def raise_two(n):
    return n

print(raise_two(7))


@generate_power(3)
def raise_three(n):
    return n

print(raise_two(5))
```

If your code editor allows it, view generate_power(exponent) and generate_power(number) side-by-side to illustrate the concepts discussed. (Sublime Text has Column View, for example.)

If you have not coded the two functions, open the code editor and start coding. For new programmers, coding is a hands-on activity: like riding a bike, you just have to do it and do it solo. So back to the task at hand!

After you have typed the code, you can now clearly see that it is similar in that it produces the same results, but there are differences. For those who have never used decorators, noting these differences will be the first step in understanding them if you venture down that path.

If you'd like to know more about this syntax and decorators in general, check out our Primer on Python Decorators. Comment below with questions.

References:

1. **https://www.geeksforgeeks.org/python-closures/**

2. https://www.programiz.com/python-programming/closure
3. https://realpython.com/inner-functions-what-are-they-good-for/