Python Closures

A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope. 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):
    """This is the outer enclosing function"""

    def printer():
        """This is the nested function"""
        print(msg)

    printer()
```

We execute the function as follows.

```
>>> print_msg("Hello")
Hello
```

We can see that the nested function `printer()` was able to access the non-local variable *msg* of the enclosing function. In the example above, what would happen if the last line of the function `print_msg()` returned the `printer()` function instead of calling it? This means the function was defined as follows.

```


def print_msg(msg):
    """This is the outer enclosing function"""

    def printer():
        """This is the nested function"""
        print(msg)

    return printer  # this got changed
```

Now let's try calling this function.

```
>>> another = print_msg("Hello")
>>> another()
Hello
```

That's unusual. The `print_msg()` function was called with the string `"Hello"` and the returned function was bound to the name *another*. On calling `another()`, the message was still remembered although we had already finished executing the `print_msg()` function. This technique by which some data (`"Hello"`) gets attached to the code is called closure in Python.

This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

```
>>> del print_msg
>>> another()
Hello
>>> print_msg("Hello")
Traceback (most recent call last):
...
NameError: name 'print_msg' is not defined
```

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):
        return x * n
    return multiplier
```

Here is how we can use it.

```
>>> times3 = make_multiplier_of(3)
>>> times5 = make_multiplier_of(5)

>>> times3(9)
27
>>> times5(3)
15
>>> times5(times3(2))
30
```

Decorators in Python make an extensive use of closures as well.

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 a closure function. Referring to the example above, we know `times3` and `times5` are closure functions.

```
>>> make_multiplier_of.__closure__
>>> times3.__closure__
(<cell at 0x0000000002D155B8: int object at 0x000000001E39B6E0>,)
```

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

```
>>> times3.__closure__[0].cell_contents
3
>>> times5.__closure__[0].cell_contents
5
```