Python Closures Explained

A CLOSURE is a function object that remembers values in enclosing scopes regardless of whether those scopes are still present in memory. If you have ever written a function that returned another function, you probably may have used closures even without knowing about them.

A quick look at closures

For example, consider the following function `generate_power_func` which returns another function.

```
def generate_power_func(n):
    print "id(n): %X" % id(n)
    def nth_power(x):
        return x**n
    print "id(nth_power): %X" % id(nth_power)
    return nth_power
```

The inner function `nth_power` is called a **closure** because, as you will see shortly, it will have access to `n` which is defined in `generate_power_func` (the enclosing scope) even after program flow is leaves it. If you want to get too technical, you can say that the function `nth_power` is closed over the variable `n`. Let's call `generate_power_func` and assign the result to another variable to examine this further.

```
>>> raised_to_4 = generate_power_func(4)
id(n): CCF7DC
id(nth_power): C46630
>>> repr(raised_to_4)
'<function nth_power at 0x00C46630>'
```

As expected, when `generate_power_func(4)` was executed, it created an `nth_power` function object (at `0x00C46630`), and returned it, which we just assigned to `raised_to_4` (you can see that `id(raised_to_4) == 0x00C46630 == id(nth_power)`). Now let's also delete the name of the original function `generate_power_func` from the global namespace.

```
>>> del generate_power_func
```

Now it's time for the closure magic ...

```
>>> raised_to_4(2)
16
```

Wait a minute! How did this work? We defined `n = 4` outside of the local scope of `nth_power`. How does `raised_to_4` (the `nth_power` function object) know that the value of `n` is 4? It makes sense that `generate_power_func` would know about `n` (and its value, 4) when the

program flow is *within* `generate_power_func`. But the program flow is currently not within `generate_power_func`. For that matter `generate_power_func` does not even exist in the namespace anymore.

The `nth_power` function object returned by `generate_power_func` is a closure because it knows about the details of the variable `n` from the *enclosing* scope.

Diving deeper - the __closure__ attribute and cell objects

Luckily for us, functions in Python are first class objects. So we can gain a little more understanding of the closures by inspecting the function objects. And all Python functions have a`__closure__`attribute that lets us examine the enclosing variables associated with a closure function.

The `__closure__` attribute returns a tuple of **cell objects** which contain details of the variables defined in the enclosing scope. Let's examine this.

```
>>> raised_to_4.__closure__
(<cell at 0x00FFFB70: int object at 0x00CCF7DC>,)
>>> type(raised_to_4.__closure__[0])
<type 'cell'>
>>> raised_to_4.__closure__[0].cell_contents
4
```

As you can see, the `__closure__` attribute of the function `raised_to_4` has a reference to `int` object at `0x00CCF7DC` which is none other than `n` (which was defined in `generate_power_func`).
In case you're wondering, every function object has `__closure__` attribute. If there is not data for the closure, the `__closure__` attribute will just be `None`. For example

```
>>> def f():
...     pass
...
>>> repr(f); repr(f.__closure__)
'<function f at 0x0153A330>'
'None'
```

Summary

- Closure is just a fancy name for a function that remembers the values from the enclosing lexical scope even when the program flow is no longer in the enclosing scope.
- If you've ever written a function that returned another function, you may have used closures.