Python Closures

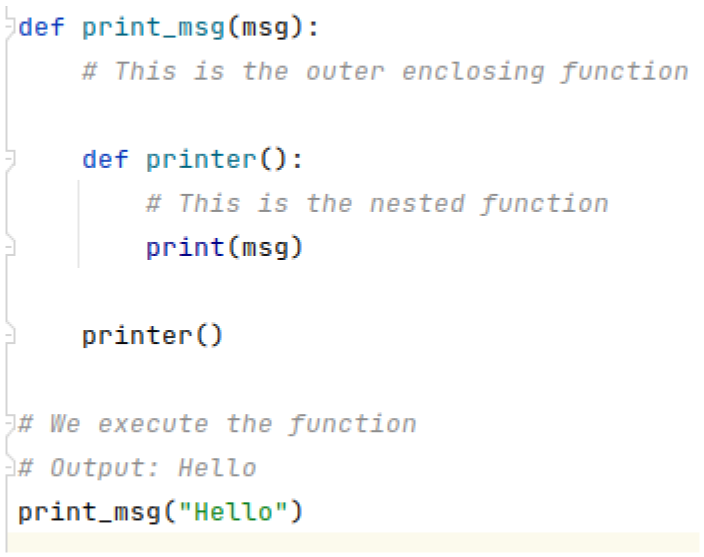
Nonlocal variable in a nested function:

Before getting into what a closure is, we have to first understand what a nested function and nonlocal variable is.

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

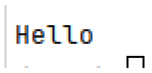
In Python, these non-local variables are read-only by default and we must declare them explicitly as non-local (using nonlocal keyword) in order to modify them.

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



```python
def print_msg(msg):
    # This is the outer enclosing function

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

    printer()

# We execute the function
# Output: Hello
print_msg("Hello")
```

Output:

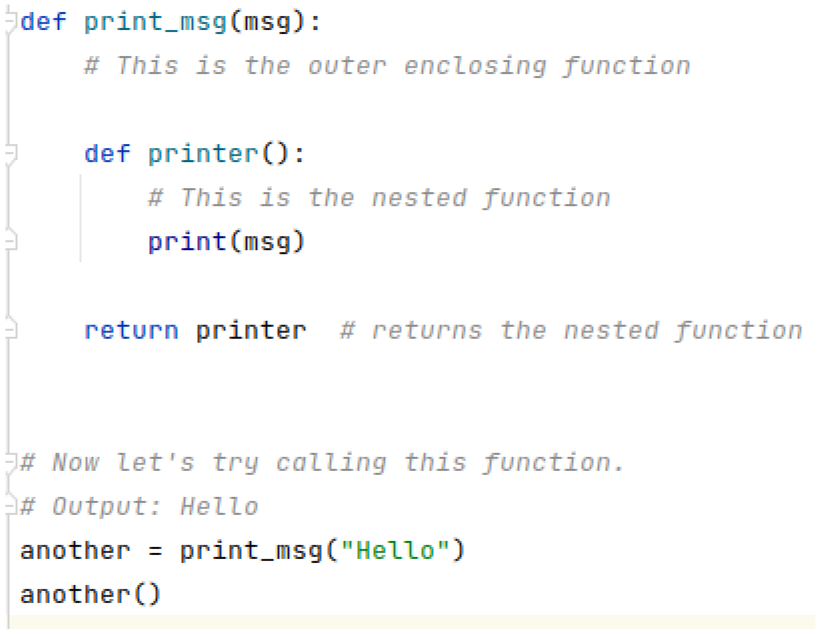


```
Hello
```

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

<u>Defining a Closure Function:</u>

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:



```python
def print_msg(msg):
    # This is the outer enclosing function

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

    return printer  # returns the nested function


# Now let's try calling this function.
# Output: Hello
another = print_msg("Hello")
another()
```

Output:

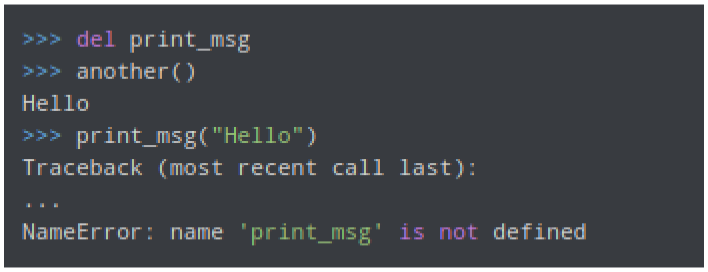
```
Hello
```

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 in this case) 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
```

Here, the returned function still works even when the original function was deleted.

When do we have closures?

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.

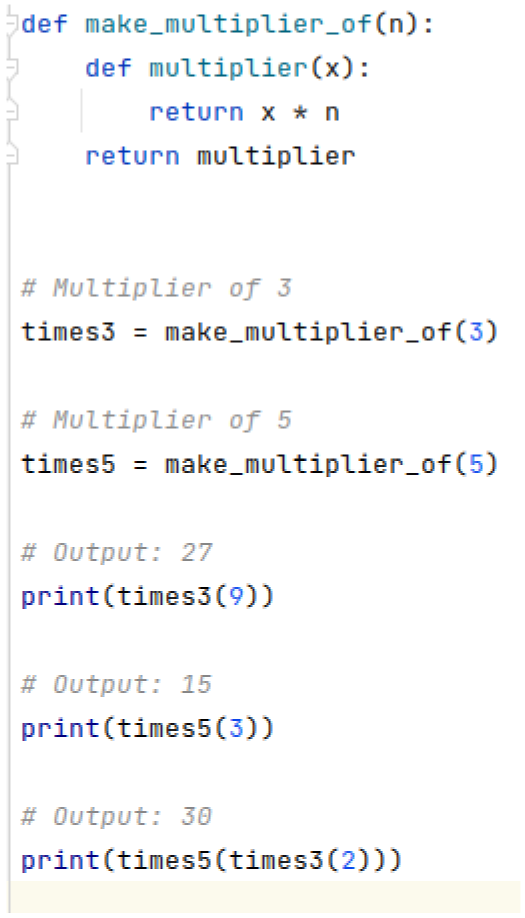
When to use closures?

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.

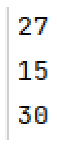
When there are few methods (one method in most cases) to be implemented in a class, closures can provide an alternate and more elegant solution. But when the number of attributes and methods get larger, it's better to implement a class.

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.



```python
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier


# Multiplier of 3
times3 = make_multiplier_of(3)

# Multiplier of 5
times5 = make_multiplier_of(5)

# Output: 27
print(times3(9))

# Output: 15
print(times5(3))

# Output: 30
print(times5(times3(2)))
```

Output:



```
27
15
30
```