Python Closures

Python closure is a nested function that allows us to access variables of the outer function even after the outer function is closed.

Before we learn about closure, let's first revise the concept of nested functions in Python.

Nested function in Python

In Python, we can create a function inside another function. This is known as a nested function. For example,

```
def greet(name):
    # inner function
    def display_name():
        print("Hi", name)

    # call inner function
    display_name()

# call outer function
greet("John")

# Output: Hi John
```

In the above example, we have defined the `display_name()` function inside the `greet()` function.

Here, `display_name()` is a nested function. The nested function works similar to the normal function. It executes when `display_name()` is called inside the function `greet()`.

Python Closures

As we have already discussed, closure is a nested function that helps us access the outer function's variables even after the outer function is closed. For example,

```
def greet():
    # variable defined outside the inner function
    name = "John"

    # return a nested anonymous function
    return lambda: "Hi " + name

# call the outer function
message = greet()

# call the inner function
print(message())

# Output: Hi John
```

In the above example, we have created a function named `greet()` that returns a nested anonymous function.

Here, when we call the outer function,

```
message = greet()
```

The returned function is now assigned to the *message* variable.

At this point, the execution of the outer function is completed, so the *name* variable should be destroyed. However, when we call the anonymous function using

```
print(message())
```

we are able to access the *name* variable of the outer function.

It's possible because the nested function now acts as a closure that closes the outer scope variable within its scope even after the outer function is executed.

Let's see one more example to make this concept clear.

Example: Print Odd Numbers using Python Closure

```
def calculate():
    num = 1
    def inner_func():
        nonlocal num
        num += 2
        return num
    return inner_func

# call the outer function
odd = calculate()

# call the inner function
print(odd())
print(odd())
print(odd())

# call the outer function again
odd2 = calculate()
print(odd2())
```

Output

```
3
5
7
3
```

In the above example,

```
odd = calculate()
```

This code executes the outer function `calculate()` and returns a closure to the odd number. T

That's why we can access the *num* variable of `calculate()` even after completing the outer function.

Again, when we call the outer function using

```
odd2 = calculate()
```

a new closure is returned. Hence, we get **3** again when we call `odd2()`.

When to use closures?

So what are closures good for?

Closures can be used to avoid global values and provide data hiding, and can be an elegant solution for simple cases with one or few methods.

However, for larger cases with multiple attributes and methods, a class implementation may be more appropriate.

```
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)))
```

Python Decorators make 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.

Table of Contents