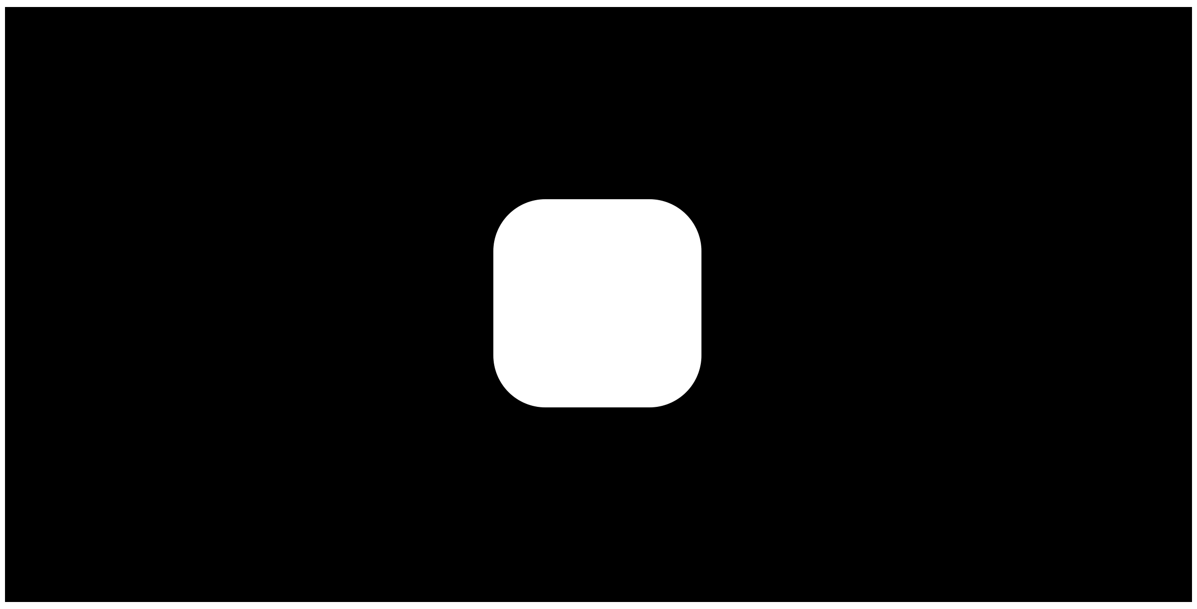
Part 1: Closures

Bookmark this page

Part 1: Closures

Closure



Start of transcript. Skip to the end.

[MUSIC PLAYING]

The venerable master, Qc Na was walking with his student Anton.

Hoping to prompt the master into a discussion, Anton said-- master,

I have heard that objects and classes are a very good thing.

Is this true?

Qc Na looked pityingly at his student and replied--

The venerable master Qc Na was walking with his student, Anton. Hoping to prompt the master into a discussion, Anton said “Master, I have heard that objects (and classes) are a very good thing. Is this true?” Qc Na looked pityingly at his student and replied, “Foolish pupil! Objects are merely a poor man’s closures.”

Chastised, Anton took his leave from his master and returned to his cell, intent on studying closures. He carefully read the entire “Lambda: The Ultimate…” series of papers and its cousins, and implemented a small Scheme interpreter with a closure-based object system. He learned much, and looked forward to informing his master of his progress.

On his next walk with Qc Na, Anton attempted to impress his master by saying “Master, I have diligently studied the matter, and now understand that objects are truly a poor man’s closures.” Qc Na responded by hitting Anton with his stick, saying “When will you learn? Closures are a poor man’s object.” At that moment, Anton became enlightened.

From: http://wiki.c2.com/?ClosuresAndObjectsAreEquivalent

What exactly are Closures, these mysterious things that offer programming enlightenment? Let’s continue to compare and contrast closures with objects.

- Objects have methods.
- Closures *are* methods — they are defined and behave like functions, but like object methods they carry and internal state and take it into account when returning results.
- Objects can, and generally do, carry a mutable state.
- Closures can, and often do, carry a mutable state.
- Objects control access to their attributes — their internal state — through Properties and Python’s lexical scoping rules. By default, however, object attributes are externally accessible.

- Closures by nature tend to close around their internal state and thereby prevent external access, thus in terms of access to internal state, internal attributes, this is the opposite of the default behavior of an object. In accordance with Python's Consenting Adults policy, a closure's internal state is still accessible via its __closure__() dunder, but this violates the spirit of a closure — so do so at your own risk.

Thus, objects (or classes) and closures are similar, but they are not the same.

The following represents the general form of a closure:

```
1 def closure(internal_state):
2     def return_function(arguments):
3         return internal_state combined with arguments
4     return return_function
```

Let's unpack that line by line.

1. The closure is defined like any other function with a name and arguments. In this case the name of the function is *closure* and its arguments are *internal_state*.

2. Inside the closure another function is defined. It also takes arguments. In this case, its name is *return_function*, because **this internally defined function itself will be returned by the closure**.

3. When calculating a return value, the internal function, *return_function*, uses both the internal state passed into the closure on line 1 when the closure was first defined, and also the arguments that will be passed into it later when it is used as a stand-alone function.

4. The closure uses the *internally defined* function, return_function, for its return value. Thus, just as a class is a template or factory for creating **stateful objects**, a closure is a template or factory for creating **stateful functions**, that is, stand-alone methods.

Closures act as factories for new functions, simplifying a process that would otherwise need additional code. Let's look at the following example.

```
def multiply(x):
    return x * 3
```

`multiply(x)` will take the value of x and multiply it by 3. That is the only thing this function will ever do. If we needed to have a multiplier by 5, we would be forced to code a new function, with very similar code, that will do so. Let us closures instead:

```
def make_multipler(n):
    def multiply(x):
        return x * n
    return multiply

times3 = make_multiplier(3) # times3(4) = 12
times5 = make_multiplier(5) # times5(4) = 20
```

In the example, *make_multiplier(n)* is the factory that will create as many functions as required, using *n* as the multiplier. Of course, you could argue that it would be easier to have the original *multiply(x)* function take two parameters instead, becoming *multiply(x, n)* and that would also work. Here is a more challenging example. This code will **partially** implement a counter that will increment itself every time the *increment()* function is called:

```
def increment():
    global count
    count += 1
    return count
```

I said **partially** because, in this case a global variable, *count* needs to be defined **outside** of the increment function, so that the current state of the counter is not lost on resetted (if you had something like *count = 0* inside of *increment()*).

Having *count* defined externally is in itself a risk, as it exposes it to other code that could accidentally modify it; it also becomes increasingly difficult to maintain if you needed more that one counter. Once again, closures come to the rescue:

```
def counter():
    count = 0
    def increment():
        nonlocal count
        count += 1
        return count
    return increment

mycounter = counter() # Creates one counter
mysecondcounter = counter() # Creates a second, INDEPENDENT counter
```

The beauty of this code is that each counter that is created is fully independent, that is, increment one counter will not affect any of the others. The *count* variable for each counter is generally only accessible to the counter itself, so the risk of other code modifying it by accident has been all but eliminated.

Note the use of the *nonlocal *keyword, which was introduced with Python 3, allowing to use the *counter* variable from the enclosing function without making it a global variable. Just for fun, let's make our closure counter factory even better

```
def counter(start=0):
    count = start
    def increment():
        nonlocal count
```