Default Parameter Values in Python

Fredrik Lundh | July 17, 2008 | based on a comp.lang.python post

Python's handling of default parameter values is one of a few things that tends to trip up most new Python programmers (but usually only once).

What causes the confusion is the behaviour you get when you use a "mutable" object as a default value; that is, a value that can be modified in place, like a list or a dictionary.

An example:

```
  >>> def function(data=[]):
...     data.append(1)
...     return data
...
>>> function()
[1]
>>> function()
[1, 1]
>>> function()
[1, 1, 1]
```

As you can see, the list keeps getting longer and longer. If you look at the list identity, you'll see that the function keeps returning the same object:

```
>>> id(function())
12516768
>>> id(function())
12516768
>>> id(function())
12516768
```

The reason is simple: the function keeps using the same object, in each call. The modifications we make are "sticky".

Why does this happen?

Default parameter values are *always* evaluated when, and only when, the "def" statement they belong to is executed; see:

~~http://docs.python.org/ref/function.html~~ (dead link)

for the relevant section in the Language Reference.

Also note that "def" is an executable statement in Python, and that default arguments are evaluated in the "def" statement's environment. If you execute "def" multiple times, it'll create a new function object (with freshly calculated default values) each time. We'll see examples of this below.

What to do instead?

The workaround is, as others have mentioned, to use a placeholder value instead of modifying the default value. **None** is a common value:

```
def myfunc(value=None):
   if value is None:
       value = []
   # modify value here
```

If you need to handle arbitrary objects (including None), you can use a sentinel object:

```
sentinel = object()
```

```
def myfunc(value=sentinel):
    if value is sentinel:
        value = expression
    # use/modify value here
```

In older code, written before “object” was introduced, you sometimes see things like

```
sentinel = ['placeholder']
```

used to create a non-false object with a unique identity; [] creates a new list every time it is evaluated.

Valid uses for mutable defaults

Finally, it should be noted that more advanced Python code often uses this mechanism to its advantage; for example, if you create a bunch of UI buttons in a loop, you might try something like:

```
for i in range(10):
    def callback():
        print "clicked button", i
    UI.Button("button %s" % i, callback)
```

only to find that all callbacks print the same value (most likely 9, in this case). The reason for this is that Python’s nested scopes *bind to variables, not object values*, so all callback instances will see the current (=last) value of the “i” variable. To fix this, use explicit binding:

```
for i in range(10):
    def callback(i=i):
        print "clicked button", i
    UI.Button("button %s" % i, callback)
```

The “i=i” part binds the parameter “i” (a local variable) to the *current* value of the outer variable “i”.

Two other uses are local caches/memoization; e.g.

(It happened to me in one of the first Python programs I ever wrote, and it took several years before we spotted the (non-critical) bug, when someone looked a bit more carefully at the contents of a property file, and wondered what all those things were doing there...)

```
def calculate(a, b, c, memo={}):
    try:
        value = memo[a, b, c] # return already calculated value
    except KeyError:
        value = heavy_calculation(a, b, c)
        memo[a, b, c] = value # update the memo dictionary
    return value
```

(this is especially nice for certain kinds of recursive algorithms)

and, for highly optimized code, local rebinding of global names:

```
import math

def this_one_must_be_fast(x, sin=math.sin, cos=math.cos):
    ...
```

How does this work, in detail?

When Python executes a “def” statement, it takes some ready-made pieces (including the compiled code for the function body and the current namespace), and creates a new function object. When it does this, it also evaluates the default values.

The various components are available as attributes on the function object; using the function we used above:

```
>>> function.func_name
'function'
>>> function.func_code
<code object function at 00BEC770, file "<stdin>", line 1>
>>> function.func_defaults
([1, 1, 1],)
>>> function.func_globals
{'function': <function function at 0x00BF1C30>,
'__builtins__': <module '__builtin__' (built-in)>,
'__name__': '__main__', '__doc__': None}
```

Since you can access the defaults, you can also modify them:

```
>>> function.func_defaults[0][:] = []
>>> function()
[1]
>>> function.func_defaults
([1],)
```

However, this is not exactly something I'd recommend for regular use…

Another way to reset the defaults is to simply re-execute the same "def" statement. Python will then create a new binding to the code object, evaluate the defaults, and assign the function object to the same variable as before. But again, only do that if you know exactly what you're doing.

And yes, if you happen to have the pieces but not the function, you can use the **function** class in the **new** module to create your own function object.