Python Conquers The Universe

Adventures across space and time with the Python programming language

Gotcha — Mutable default arguments

Posted on 2012/02/15

Goto start of series

Note: examples are coded in Python 2.x, but the basic point of the post applies to all versions of Python.

There's a Python gotcha that bites everybody as they learn Python. In fact, I think it was Tim Peters who suggested that every programmer gets caught by it exactly two times. It is call the *mutable defaults* trap. Programmers are usually bit by the mutable defaults trap when coding class methods, but I'd like to begin with explaining it in functions, and then move on to talk about class methods.

Mutable defaults for function arguments

The gotcha occurs when you are coding default values for the arguments to a function or a method. Here is an example for a function named `foobar`:

```
1 def foobar(arg_string = "abc", arg_list = []):
2     ...
```

Here's what most beginning Python programmers believe will happen when `foobar` is called without any arguments:

> *A new string object containing "abc" will be created and bound to the "arg_string" variable name. A new, empty list object will be created and bound to the "arg_list" variable name. In short, if the arguments are omitted by the caller, the `foobar` will always get "abc" and [] in its arguments.*

This, however, is *not* what will happen. Here's why.

The objects that provide the default values are not created at the time that `foobar` is called. They are created *at the time that the statement that defines the function is executed*. (See the discussion at Default arguments in Python: two easy blunders: "Expressions in default arguments are calculated when the function is defined, *not* when it's called.")

If `foobar`, for example, is contained in a module named `foo_module`, then the statement that defines `foobar` will probably be executed at the time when `foo_module` is imported.

When the `def` statement that creates `foobar` is executed:

- A new function object is created, bound to the name `foobar`, and stored in the namespace of `foo_module`.
- Within the `foobar` function object, for each argument with a default value, an object is created to hold the default object. In the case of `foobar`, a string object containing "abc" is created as the default for the `arg_string` argument, and an empty list object is created as the default for the `arg_list` argument.

After that, whenever `foobar` is called without arguments, `arg_string` will be bound to the default string object, and arg_list will be bound to the default list object. In such a case, `arg_string` will always be "abc", but `arg_list` may or may not be an empty list. Here's why.

There is a crucial difference between a string object and a list object. A string object is immutable, whereas a list object is mutable. That means that the default for `arg_string` can never be changed, but the default for `arg_list` can be changed.

Let's see how the default for arg_list can be changed. Here is a program. It invokes `foobar` four times. Each time that `foobar` is invoked it displays the values of the arguments that it receives, then adds something to each of the arguments.

```
def foobar(arg_string="abc", arg_list = []):
    print arg_string, arg_list
    arg_string = arg_string + "xyz"
    arg_list.append("F")

for i in range(4):
    foobar()
```

The output of this program is:

```
abc []
abc ['F']
abc ['F', 'F']
abc ['F', 'F', 'F']
```

As you can see, the first time through, the argument have exactly the default that we expect. On the second and all subsequent passes, the arg_string value remains unchanged — just what we would expect from an immutable object. The line

```
arg_string = arg_string + "xyz"
```

creates a new object — the string "abcxyz" — and binds the name "arg_string" to that new object, but it doesn't change the default object for the `arg_string` argument.

But the case is quite different with arg_list, whose value is a list — a mutable object. On each pass, we append a member to the list, and the list grows. On the fourth invocation of `foobar` — that is, after three earlier invocations — `arg_list` contains three members.

The Solution

This behavior is not a wart in the Python language. It really is a feature, not a bug. There are times when you really do want to use mutable default arguments. One thing they can do (for example) is retain a list of results from previous invocations, something that might be very handy.

But for most programmers — especially beginning Pythonistas — this behavior is a gotcha. So for most cases we adopt the following rules.

1. Never use a mutable object — that is: a list, a dictionary, or a class instance — as the default value of an argument.
2. Ignore rule 1 only if you really, *really*, *REALLY* know what you're doing.

So... we plan always to follow rule #1. Now, the question is *how* to do it... how to code `foobar` in order to get the behavior that we want.

Fortunately, the solution is straightforward. The mutable objects used as defaults are replaced by None, and then the arguments are tested for None.

```
def foobar(arg_string="abc", arg_list = None):
    if arg_list is None: arg_list = []
    ...
```

Another solution that you will sometimes see is this:

```
def foobar(arg_string="abc", arg_list=None):
    arg_list = arg_list or []
    ...
```

This solution, however, is *not* equivalent to the first, and should be avoided. See *Learning Python* p. 123 for a discussion of the differences. *Thanks to Lloyd Kvam for pointing this out to me.*

And of course, in some situations the best solution is simply not to supply a default for the argument.

Mutable defaults for method arguments

Now let's look at how the mutable arguments gotcha presents itself when a class method is given a mutable default for one of its arguments. Here is a complete program.

```
# (1) define a class for company employees
class Employee:
    def __init__ (self, arg_name, arg_dependents=[]):
        # an employee has two attributes: a name, and a list of his dependents
        self.name = arg_name
        self.dependents = arg_dependents

    def addDependent(self, arg_name):
        # an employee can add a dependent by getting married or having a baby
        self.dependents.append(arg_name)

    def show(self):
        print
        print "My name is.......: ", self.name
        print "My dependents are: ", str(self.dependents)
#-----------------------------------------------
#   main routine -- hire employees for the company
#-----------------------------------------------

# (2) hire a married employee, with dependents
joe = Employee("Joe Smith", ["Sarah Smith", "Suzy Smith"])

# (3) hire a couple of unmarried employess, without dependents
mike = Employee("Michael Nesmith")
barb = Employee("Barbara Bush")

# (4) mike gets married and acquires a dependent
mike.addDependent("Nancy Nesmith")

# (5) now have our employees tell us about themselves
joe.show()
mike.show()
barb.show()
```

Let's look at what happens when this program is run.

1. First, the code that defines the `Employee` class is run.
2. Then we hire Joe. Joe has two dependents, so that fact is recorded at the time that the `joe` object is created.
3. Next we hire Mike and Barb.
4. Then Mike acquires a dependent.
5. Finally, the last three statements of the program ask each employee to tell us about himself.

Here is the result.

```
My name is.......:  Joe Smith
My dependents are:  ['Sarah Smith', 'Suzy Smith']

```

```
My name is.......:  Michael Nesmith
My dependents are:  ['Nancy Nesmith']

My name is.......:  Barbara Bush
My dependents are:  ['Nancy Nesmith']
```

Joe is just fine. But somehow, when Mike acquired Nancy as his dependent, Barb *also* acquired Nancy as a dependent. This of course is wrong. And we're now in a position to understand what is causing the program to behave this way.

When the code that defines the `Employee` class is run, objects for the class definition, the method definitions, and the default values for each argument are created. The constructor has an argument `arg_dependents` whose default value is an empty list, so an empty list object is created and attached to the `__init__` method as the default value for `arg_dependents`.

When we hire Joe, he already has a list of dependents, which is passed in to the Employee constructor — so the `arg_dependents` attribute does not use the default empty list object.

Next we hire Mike and Barb. Since they have no dependents, the default value for `arg_dependents` is used. Remember — this is the empty list object that was created when the code that defined the `Employee` class was run. So in both cases, the empty list is bound to the `arg_dependents` argument, and then — again in both cases — it is bound to the `self.dependents` attribute. The result is that after Mike and Barb are hired, the `self.dependents` attribute of *both* Mike and Barb *point to the same object* — the default empty list object.

When Michael gets married, and Nancy Nesmith is added to his `self.dependents` list, Barb also acquires Nancy as a dependent, because Barb's `self.dependents` variable name is bound to the same list object as Mike's `self.dependents` variable name.

So this is what happens when mutuable objects are used as defaults for arguments in class methods. If the defaults are used when the method is called, *different class instances end up sharing references to the same object.*

And *that* is why you should never, *never*, ***NEVER*** use a list or a dictionary as a default value for an argument to a class method. Unless, of course, you really, *really*, ***REALLY*** know what you're doing.

Related

Python Decorators
In "Decorators"

Python Gotchas
In "Python gotchas"

Python & Java: A Side-by-Side Comparison
In "Java and Python"

This entry was posted in Python gotchas. Bookmark the permalink.

6 Responses to *Gotcha — Mutable default arguments*

TheBlackCat *says:*
2012/03/18 at 5:08 am

I've been bitten by this before. I know my opinion is not liked by many, but I think that anything that makes it easier to do uncommon, advanced tasks while making it harder to do common, simple tasks is a bug, not a feature. Same for something that makes life significantly harder for beginning users (and even many advanced users) while making it slightly easier for advanced users is a bug, not a feature.