Jeff Knupp

PYTHON PROGRAMMER

Everything I know about Python...

Is Python call-by-value or call-by-reference? Neither. (/blog/2012/11/13/is-python-callbyvalue-or-callbyreference-neither)

One aspect of Python programming that trips up those coming from languages like C or Java is how arguments are passed to functions in Python. At a more fundamental level, the confusion arises from a misunderstanding about Python object-centric data model and its treatment of assignment. When asked whether Python function calling model is "call-by-value" or "call-by-reference", the correct answer is: **neither**. Indeed, to try to shoe-horn those terms into a conversation about Python's model is misguided. "call-by-object," or "call-by-object-reference" is a more accurate way of describing it. But what does "call-by-object" even mean?

In Python, (almost) everything is an object. What we commonly refer to as "variables" in Python are more properly called *names*. Likewise, "assignment" is really the *binding* of a name to an *object*. Each binding has a *scope* that defines its visibility, usually the *block* in which the name originates.

That's a lot of terminology all at once, but those basic terms form the cornerstone of Python's execution model (http://docs.python.org/3/reference/executionmodel.html). Compared to, say, C++, the differences are subtle yet important. A concrete example will highlight these differences. Think about what happens when the following C++ code is executed:

```
string some_guy = "Fred";
// ...
some_guy = "George";
```

In the above, the variable `some_guy` refers to a location in memory, and the value 'Fred' is inserted in that location (indeed, we can take the address of `some_guy` to determine the portion of memory to which it refers). Later, the contents of the memory location referred to by `some_guy` are changed to 'George'. The previous value no longer exists; it was overwritten. This likely matches your intuitive understanding (even if you don't program in C++).

Let's now look at a similar block of Python code:

```
some_guy = 'Fred'
# ...
some_guy = 'George'
```

Binding Names to Objects

On line 1, we create a *binding* between a *name*, `some_guy` , and a string *object* containing 'Fred'. In the context of program execution, the *environment* is altered; a binding of the name `some_guy` ' to a string object is created in the *scope* of the *block* where the statement occurred. When we later say `some_guy = 'George'` , the string object containing 'Fred' is unaffected. We've just changed the binding of the name `some_guy` . **We haven't, however, changed either the 'Fred' or 'George' string objects**. As far as we're concerned, they may live on indefinitely.

With only a single name binding, this may seem overly pedantic, but it becomes more important when bindings are shared and function calls are involved. Let's say we have the following bit of Python code:

```
some_guy = 'Fred'

first_names = []
first_names.append(some_guy)

another_list_of_names = first_names
another_list_of_names.append('George')
some_guy = 'Bill'

print (some_guy, first_names, another_list_of_names)
```

So what get's printed in the final line? Well, to start, the binding of `some_guy` to the string object containing 'Fred' is added to the block's *namespace*. The name `first_names` is bound to an empty list object. On line 4, a method is called on the list object

`first_names` is bound to, appending the object `some_guy` is bound to. At this point, there are still only two objects that exist: the string object and the list object. `some_guy` and `first_names[0]` *both refer to the same object* (Indeed, `print(some_guy is first_names[0])` shows this).

Let's continue to break things down. On line 6, a new name is bound: `another_list_of_names` . Assignment between names does not create a new object. Rather, both names are simply bound to the same object. As a result, the string object and list object are still the only objects that have been created by the interpreter. On line 7, a member function is called on the object `another_list_of_names` is bound to and it is *mutated* to contain a reference to a new object: 'George'. So to answer our original question, the output of the code is

```
Bill ['Fred', 'George'] ['Fred', 'George']
```

This brings us to an important point: there are actually two kinds of objects in Python. A *mutable* object exhibits time-varying behavior. Changes to a mutable object are visible through all names bound to it. Python's lists are an example of mutable objects. An *immutable* object does not exhibit time-varying behavior. The value of immutable objects can not be modified after they are created. They *can* be used to compute the values of **new** objects, which is how a function like string.join works. When you think about it, this dichotomy is necessary because, again, everything is an object in Python. If integers were not immutable I could change the meaning of the number '2' throughout my program.

It would be incorrect to say that "mutable objects can change and immutable ones can't", however. Consider the following:

```
first_names = ['Fred', 'George', 'Bill']
last_names = ['Smith', 'Jones', 'Williams']
name_tuple = (first_names, last_names)

first_names.append('Igor')
```

Tuples in Python are immutable. We can't change the tuple object `name_tuple` is bound to. But immutable containers may contain references to mutable objects like lists. Therefore, even though `name_tuple` is immutable, it "changes" when 'Igor' is appended to `first_names` on the last line. It's a subtlety that can sometimes (though very infrequently) prove useful.

By now, you should almost be able to intuit how function calls work in Python. If I call `foo(bar)` , I'm merely creating a binding within the scope of `foo` to the object the argument `bar` is bound to when the function is called. If `bar` refers to a mutable object and `foo` changes its value, then these changes will be visible outside of the scope of the function.

```
def foo(bar):
    bar.append(42)
    print(bar)
    # >> [42]

answer_list = []
foo(answer_list)
print(answer_list)
# >> [42]
```

On the other hand, if `bar` refers to an immutable object, the most that `foo` can do is create a name `bar` in its local namespace and bind it to some other object.

```
def foo(bar):
    bar = 'new value'
    print (bar)
    # >> 'new value'

answer_list = 'old value'
foo(answer_list)
print(answer_list)
# >> 'old value'
```

Hopefully by now it is clear why Python is neither "call-by-reference" nor "call-by-value". In Python a variable is not an alias for a location in memory. Rather, it is simply a binding to a Python object. While the notion of "everything is an object" is undoubtedly a cause of confusion for those new to the language, it allows for powerful and flexible language constructs, which I'll discuss in my next post.

Posted on Nov 13, 2012 by Jeff Knupp

> **Discuss Posts With Other Readers at discourse.jeffknupp.com (http://discourse.jeffknupp.com)!**

« Starting a Django 1.4 Project the Right Way (/blog/2012/10/24/starting-a-django-14-project-the-right-way)

Subscribe