Is Python pass-by-reference or pass-by-value?

robertheaton.com/2014/02/09/pythons-pass-by-object-reference-as-explained-by-philip-k-dick

09 Feb 2014

> "Suppose I say to Fat, or Kevin says to Fat, "You did not experience God. You merely experienced something with the qualities and aspects and nature and powers and wisdom and goodness of God." This is like the joke about the German proclivity toward double abstractions; a German authority on English literature declares, "Hamlet was not written by Shakespeare; it was merely written by a man named Shakespeare." In English the distinction is verbal and without meaning, although German as a language will express the difference (which accounts for some of the strange features of the German mind)."

> Valis, p71 (Book-of-the-Month-Club Edition)

Philip K. Dick is not known for his light or digestible prose. The vast majority of his characters are high. Like, really, really, really high. And yet, in the above quote from Valis (published in 1981), he gives a remarkably foresighted explanation of the notoriously misunderstood Python parameter passing paradigm. *Plus ça change, plus c'est omnomnomnom drugs.*

The two most widely known and easy to understand approaches to parameter passing amongst programming languages are pass-by-reference and pass-by-value. Unfortunately, Python is "pass-by-object-reference", of which it is often said:

> "Object references are passed by value."

When I first read this smug and overly-pithy definition, I wanted to punch something. After removing the shards of glass from my hands and being escorted out of the strip club, I realised that all 3 paradigms can be understood in terms of how they cause the following 2 functions to behave:

```
def reassign(list):
  list = [0, 1]

def append(list):
  list.append(1)

list = [0]
reassign(list)
append(list)
```

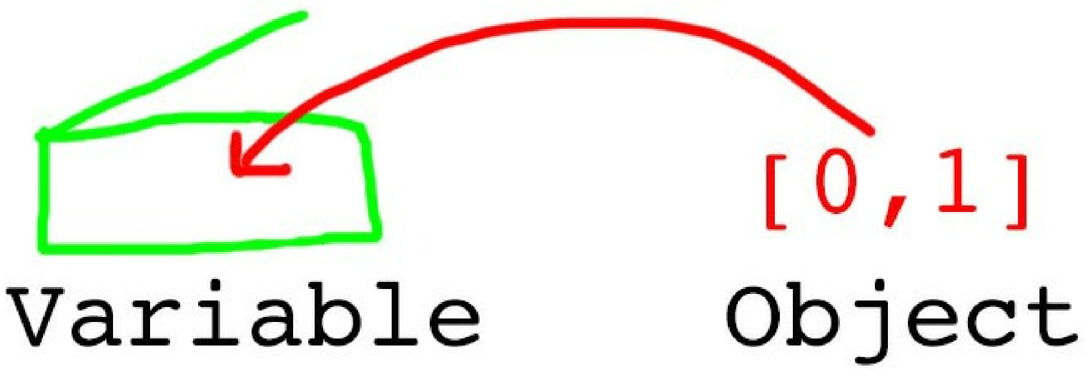
Let's explore.

The variable is not the object

“Hamlet was not written by Shakespeare; it was merely written by a man named Shakespeare.” Both Python and PKD make a crucial distinction between a thing, and the label we use to refer to that thing. “The man named Shakespeare” is a man. “Shakespeare” is just a name. If we do:

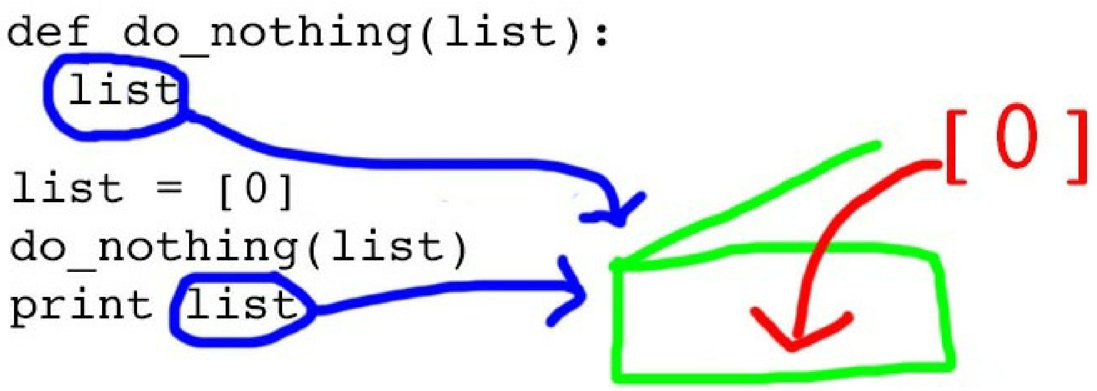
```
a = []
```

then `[]` is the empty list. `a` is a variable that points to the empty list, but `a` itself is not the empty list. I draw and frequently refer to variables as “boxes” that contain objects; but however you conceive of it, this difference is key.



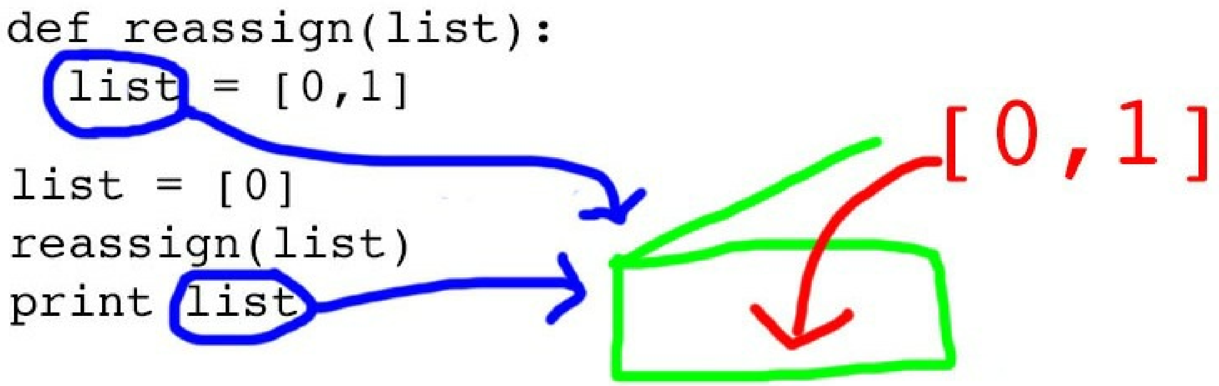
Pass-by-reference

In pass-by-reference, the box (the variable) is passed directly into the function, and its contents (the object represented by the variable) implicitly come with it. Inside the function context, the argument is essentially a complete alias for the variable passed in by the caller. They are both the exact same box, and therefore also refer to the exact same object in memory.



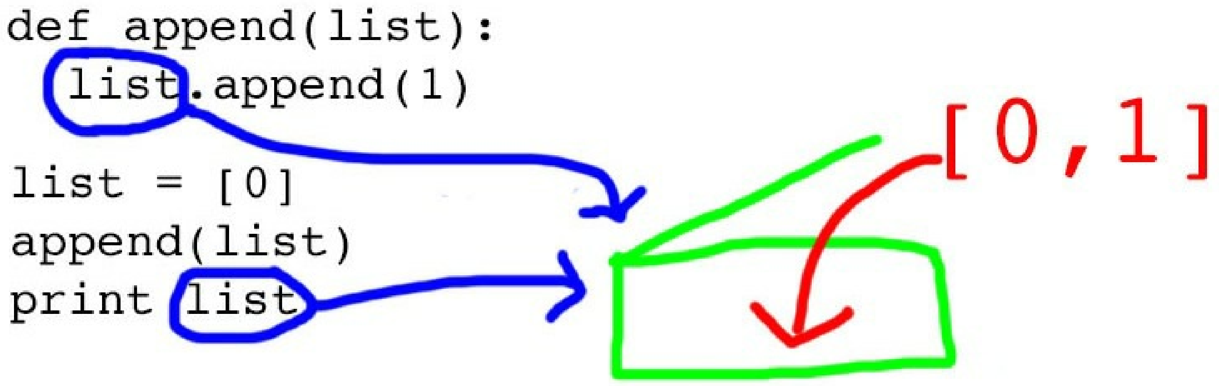
```
def do_nothing(list):
  list

list = [0]
do_nothing(list)
print list
```

Anything the function does to either the variable or the object it represents will therefore be visible to the caller. For example, the function could completely change the variable’s content, and point it at a completely different object:



```
def reassign(list):
  list = [0,1]

list = [0]
reassign(list)
print list
```

The function could also manipulate the object without reassigning it, with the same effect:

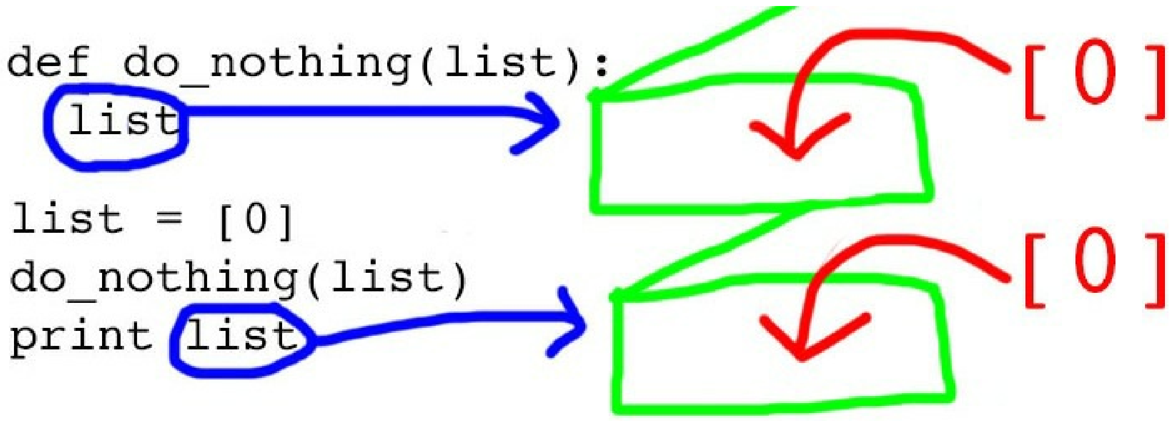


```
def append(list):
  list.append(1)

list = [0]
append(list)
print list
```

To reiterate, in pass-by-reference, the function and the caller both use the exact same variable and object.

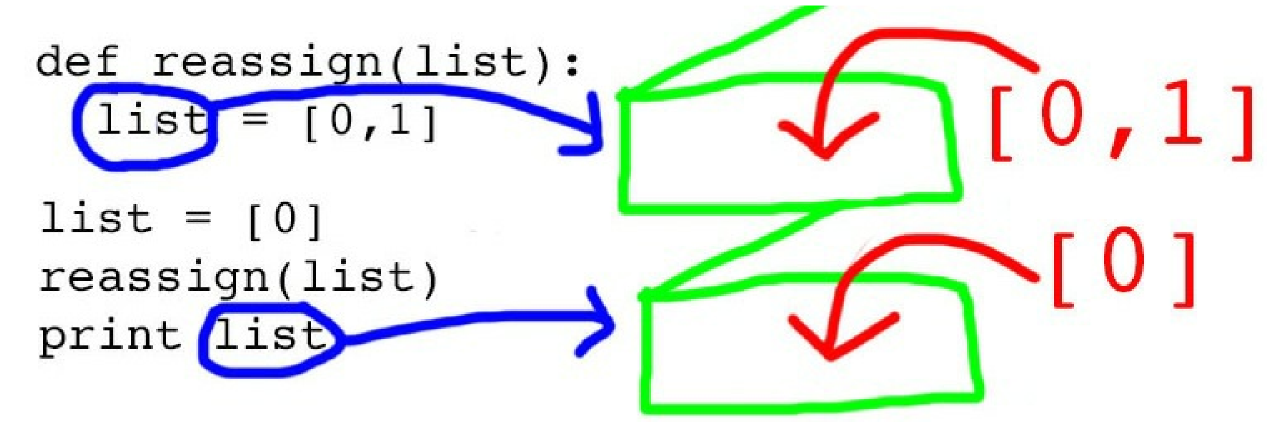
Pass-by-value

In pass-by-value, the function receives a copy of the argument objects passed to it by the caller, stored in a new location in memory.



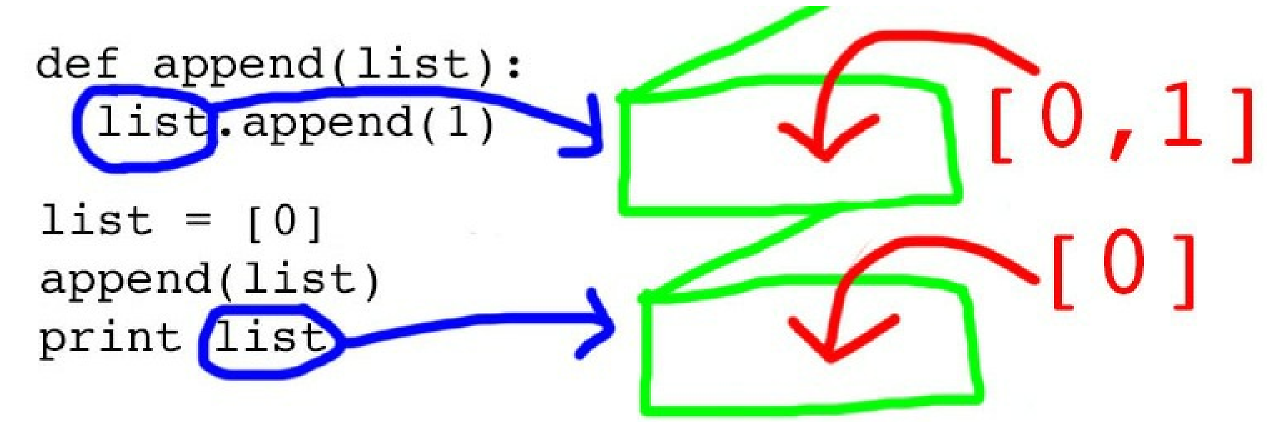
```
def do_nothing(list):
  list

list = [0]
do_nothing(list)
print list
```

The function then effectively supplies its own box to put the value in, and there is no longer any relationship between either the variables or the objects referred to by the function and the caller. The objects happen to have the same value, but they are totally separate, and nothing that happens to one will affect the other. If we again try to reassign:



```
def reassign(list):
  list = [0,1]

list = [0]
reassign(list)
print list
```

Outside the function, nothing happens. Similarly:



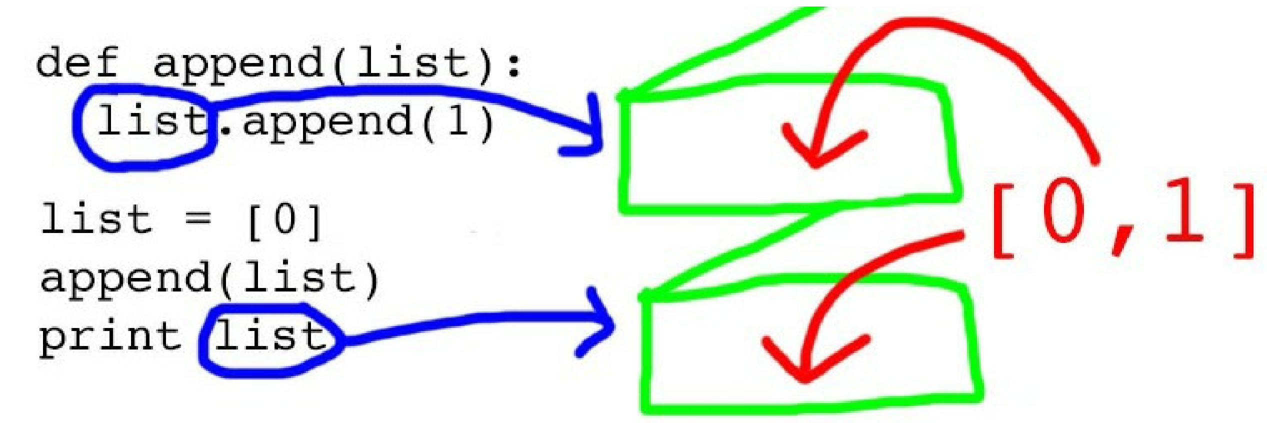
```
def append(list):
  list.append(1)

list = [0]
append(list)
print list
```

The copies of variables and objects in the context of the caller are completely isolated.

Pass-by-object-reference

Python is different. As we know, in Python, “Object references are passed by value”.

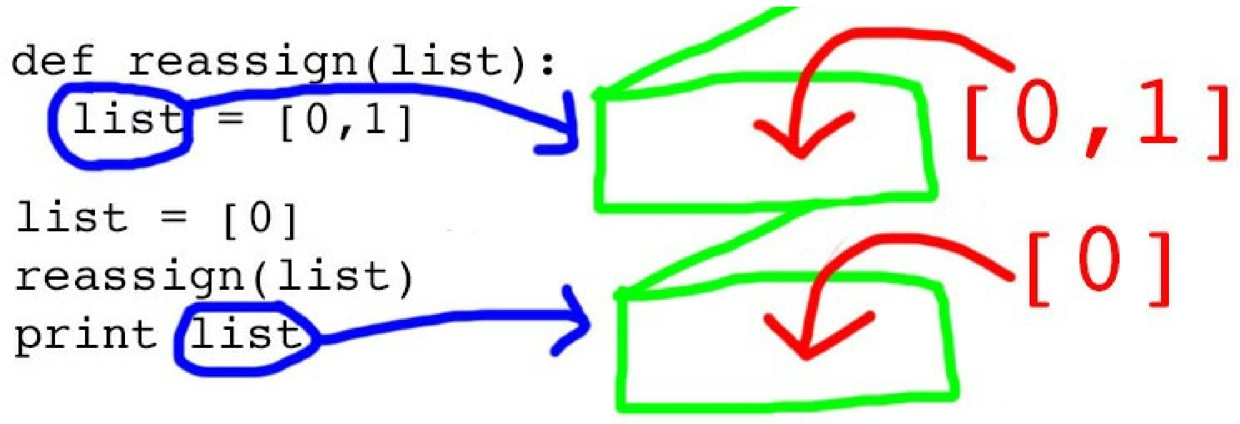
A function receives a reference to (and will access) the same object in memory as used by the caller. However, it does not receive the box that the caller is storing this object in; as in pass-by-value, the function provides its own box and creates a new variable for itself. Let’s try appending again:



```
def append(list):
  list.append(1)

list = [0]
append(list)
print list
```

Both the function and the caller refer to the same object in memory, so when the append function adds an extra item to the list, we see this in the caller too! They're different names for the same thing; different boxes containing the same object. This is what is meant by passing the object references by value - the function and caller use the same object in memory, but accessed through different variables. This means that the same object is being stored in multiple different boxes, and the metaphor kind of breaks down. Pretend it's quantum or something.

But the key is that they really are *different* names, and *different* boxes. In pass-by-reference, they were essentially the same box. When you tried to reassign a variable, and put something different into the function's box, you also put it into the caller's box, because they were the same box. But, in pass-by-object-reference:



```
def reassign(list):
  list = [0,1]

list = [0]
reassign(list)
print list
```

The caller doesn't care if you reassign the function's box. Different boxes, same content.

Now we see what Philip K. Dick was trying to tell us. A name and a person are different things. A variable and an object are different things. Armed with this knowledge, you can perhaps start to infer what happens when you do things like

```
listA = [0]
listB = listA
listB.append(1)
print listA
```

You may also want to read about the interesting interactions these concepts have with mutable and immutable types. But those are stories for another day. Now if you'll excuse me, I'm going to read *"Do Androids Dream Of Electric Sheep?"* - my meta-programming is a little rusty.

Useful links

http://foobarnbaz.com/2012/07/08/understanding-python-variables/
http://javadude.com/articles/passbyvalue.htm

Subscribe to my new work on programming, security, and a few other topics. Published a few times a month.

NEW: Also subscribe to my new series, Programming Feedback for Advanced Beginners

More on Programming

◀ Cookieless user tracking for douchebags

Lessons From A Silicon Valley Job Search ▶