Python: How do I pass a variable by reference?

The Python documentation seems unclear about whether parameters are passed by reference or value, and the following code produces the unchanged value 'Original'

```
class PassByReference:
    def __init__(self):
        self.variable = 'Original'
        self.Change(self.variable)
        print self.variable

    def Change(self, var):
        var = 'Changed'
```

up vote 721
down vote
favorite
401

Is there something I can do to pass the variable by actual reference?

python reference pass-by-reference

edited Jan 26 '13 at 9:19 — Honest Abe 2,29311326

asked Jun 12 '09 at 10:23 — David Sykes 9,67893767

share|improve this question

For a short explanation/clarification see the first answer to this stackoverflow question. As strings are immutable,

3 they won't be changed and a new variable will be created, thus the "outer" variable still has the same value. – PhilS Jun 12 '09 at 10:35

16 This is very very enlightening. – Animesh Jun 12 '09 at 11:06

7 The code in BlairConrad's answer is good, but the explanation provided by DavidCournapeau and DarenThomas is correct. – Ethan Furman Jan 7 '12 at 6:47

11 Before reading the selected answer, please consider reading this short text Other languages have "variables", Python has "names". Think about "names" and "objects" instead of "variables" and "references" and you should avoid a lot of similar problems. – lqc Nov 15 '12 at 0:39

add comment

13 Answers

active oldest votes

Arguments are passed by assignment. The rationale behind this is twofold:

1. the parameter passed in is actually a *reference* to an object (but the reference is passed by value)
2. some data types are mutable, but others aren't

So:

- If you pass a *mutable* object into a method, the method gets a reference to that same object and you can mutate it to your heart's delight, but if you rebind the reference in the method, the outer scope will know nothing about it, and after you're done, the outer reference will still point at the original object.
- If you pass an *immutable* object to a method, you still can't rebind the outer reference, and you can't even mutate the object.

To make it even more clear, let's have some examples.

List - a mutable type

Let's try to modify the list that was passed to a method:

```
def try_to_change_list_contents(the_list):
    print 'got', the_list
    the_list.append('four')
    print 'changed to', the_list

outer_list = ['one', 'two', 'three']

print 'before, outer_list =', outer_list
try_to_change_list_contents(outer_list)
print 'after, outer_list =', outer_list
```

Output:

```
before, outer_list = ['one', 'two', 'three']
got ['one', 'two', 'three']
changed to ['one', 'two', 'three', 'four']
after, outer_list = ['one', 'two', 'three', 'four']
```

Since the parameter passed in is a reference to `outer_list`, not a copy of it, we can use the mutating list methods to change it and have the changes reflected in the outer scope.

Now let's see what happens when we try to change the reference that was passed in as a parameter:

```
def try_to_change_list_reference(the_list):
    print 'got', the_list
    the_list = ['and', 'we', 'can', 'not', 'lie']
    print 'set to', the_list

outer_list = ['we', 'like', 'proper', 'English']

print 'before, outer_list =', outer_list
try_to_change_list_reference(outer_list)
```

```
print 'after, outer_list =', outer_list
```

Output:

```
before, outer_list = ['we', 'like', 'proper', 'English']
got ['we', 'like', 'proper', 'English']
set to ['and', 'we', 'can', 'not', 'lie']
after, outer_list = ['we', 'like', 'proper', 'English']
```

Since the `the_list` parameter was passed by value, assigning a new list to it had no effect that the code outside the method could see. The `the_list` was a copy of the `outer_list` reference, and we had `the_list` point to a new list, but there was no way to change where `outer_list` pointed.

up vote 850 down vote accepted

String - an immutable type

It's immutable, so there's nothing we can do to change the contents of the string

Now, let's try to change the reference

```
def try_to_change_string_reference(the_string):
    print 'got', the_string
    the_string = 'In a kingdom by the sea'
    print 'set to', the_string

outer_string = 'It was many and many a year ago'

print 'before, outer_string =', outer_string
try_to_change_string_reference(outer_string)
print 'after, outer_string =', outer_string
```

Output:

```
before, outer_string = It was many and many a year ago
got It was many and many a year ago
set to In a kingdom by the sea
after, outer_string = It was many and many a year ago
```

Again, since the `the_string` parameter was passed by value, assigning a new string to it had no effect that the code outside the method could see. The `the_string` was a copy of the `outer_string` reference, and we had `the_string` point to a new list, but there was no way to change where `outer_string` pointed.

I hope this clears things up a little.

EDIT: It's been noted that this doesn't answer the question that @David originally asked, "Is there something I can do to pass the variable by actual reference?". Let's work on that.

How do we get around this?

As @Andrea's answer shows, you could return the new value. This doesn't change the way things are passed in, but does let you get the information you want back out:

```
def return_a_whole_new_string(the_string):
    new_string = something_to_do_with_the_old_string(the_string)
    return new_string

# then you could call it like
my_string = return_a_whole_new_string(my_string)
```

If you really wanted to avoid using a return value, you could create a class to hold your value and pass it into the function or use an existing class, like a list:

```
def use_a_wrapper_to_simulate_pass_by_reference(stuff_to_change):
    new_string = something_to_do_with_the_old_string(stuff_to_change[0])
    stuff_to_change[0] = new_string

# then you could call it like
wrapper = [my_string]
use_a_wrapper_to_simulate_pass_by_reference(wrapper)

do_something_with(wrapper[0])
```

Although this seems a little cumbersome.

edited May 18 at 13:05 answered Jun 12 '09 at 11:18

share|improve this answer

mit
4,39032042

Blair Conrad
55.5k128388

47 Then the same is in C, when you pass "by reference" you're actually passing *by value* the reference... Define "by reference" :P – Andrea Ambu Jun 12 '09 at 11:52

27 I'm not sure I understand your terms. I've been out of the C game for a while, but back when I was in it, there was no "pass by reference" - you could pass things, and it was always pass by value, so whatever was in the parameter list was copied. But sometimes the thing was a pointer, which one could follow to the piece of memory (primitive, array, struct, whatever), but you couldn't change the pointer that was copied from the outer scope - when you were done with the function, the original pointer still pointed to the same address. C++ introduced references, which behaved differently. – Blair Conrad Jun 12 '09 at 12:09

31 your answer is most enlightening. big help for me learning python. big +1. – Doug T. Nov 29 '09 at 4:24

6 @Zac Bowling I don't really get how what you're saying is relevant, in a practical sense, to this answer. If a Python newcomer wanted to know about passing by ref/val, then the takeaway from this answer is: **1-** You *can* use the reference that a function receives as its arguments, to modify the 'outside' value of a variable, as long as you don't reassign the parameter to refer to a new object. **2-** Assigning to an immutable type will *always* create a new object, which breaks the reference that you had to the outside variable. – Cam Jackson Sep 8 '11 at 23:50

25 -1. The code shown is good, the explanation as to how is completely wrong. See the answers by DavidCournapeau or DarenThomas for correct explanations as to why. – Ethan Furman Jan 7 '12 at 6:41

show **29** more comments

The problem comes from a misunderstanding of what variables are in Python. If you're used to most traditional languages, you have a mental model of what happens in the following sequence:

```
a = 1
a = 2
```

You believe that `a` is a memory location that stores the value `1`, then is updated to store the value `2`. That's not how things work in Python. Rather, `a` starts as a reference to an object with the value `1`, then gets reassigned as a reference to an object with the value `2`. Those two objects may continue to coexist even though `a` doesn't refer to the first one anymore; in fact they may be shared by any number of other references within the program.

When you call a function with a parameter, a new reference is created that refers to the object passed in. This is separate from the reference that was used in the function call, so there's no way to update that reference and make it refer to a new object. In your example:

up vote 146 down vote

```
    self.variable = 'Original'
    self.Change(self.variable)

def Change(self, var):
    var = 'Changed'
```

`self.variable` is a reference to the string object `'Original'`. When you call `Change` you create a second reference `var` to the object. Inside the function you reassign the reference `var` to a different string object `'Changed'`, but the reference `self.variable` is separate and does not change.

The only way around this is to pass a mutable object. Because both references refer to the same object, any changes to the object are reflected in both places.

```
    self.variable = ['Original']
    self.Change(self.variable)

def Change(self, var):
    var[0] = 'Changed'
```

answered Nov 15 '11 at 17:45

share|improve this answer

Mark Ransom
132k12114270

Good succinct explanation. Your paragraph "When you call a function..." is one of the best explanations I've heard of the

23 rather cryptic phrase that 'Python function parameters are references, passed by value.' I think if you understand that paragraph alone, everything else kind of just makes sense and flows as a logical conclusion from there. Then you just have to be aware of when you're creating a new object and when you're modifying an existing one. – Cam Jackson Nov 16 '11 at 0:03

1 But how can you reassign the reference? I thought you can't change the address of 'var' but that your string "Changed" was now going to be stored in the 'var' memory address. Your description makes it seem like "Changed" and "Original" belong to different places in memory instead and you just switch 'var' to a different address. Is that correct? – Glassjawed May 7 '12 at 1:10

4 @Glassjawed, I think you're getting it. "Changed" and "Original" are two different string objects at different memory addresses and 'var' changes from pointing to one to pointing to the other. – Mark Ransom May 7 '12 at 1:46

3 this is the best explanation and should be the accepted answer. Short, concise and clear. Others coin new, confusing terms like 'reference is passed by value', 'call be object', etc. This answer clears everything. Thanks :) – ajay Oct 23 '13 at 12:19

@MarkRansom Take my #respect, sir! just cleared off all my noobie confusions ... – kmonsoor Apr 1 at 11:08

add comment

It is neither pass-by-value or pass-by-reference - it is call-by-object. See this, by Fredrik Lundh:

http://effbot.org/zone/call-by-object.htm

Here is a significant quote:

> "...variables [names] are *not* objects; they cannot be denoted by other variables or referred to by objects."

up vote 146 down vote

In your example, when the `Change` method is called--a namespace is created for it; and `var` becomes a name, within that namespace, for the string object `'Original'`. That object then has a name in two namespaces. Next, `var = 'Changed'` binds `var` to a new string object, and thus the method's namespace forgets about `'Original'`. Finally, that namespace is forgotten, and the string `'Changed'` along with it.

share|improve this answer

edited Jan 26 '13 at 9:06 Honest Abe 2,29311326

answered Jun 12 '09 at 12:55 David Cournapeau 21.9k23359

10 I find it hard to buy. To me is just as Java, the parameters are pointers to objects in memory, and those pointers are passed via the stack, or registers. – Luciano Dec 13 '11 at 1:25

3 This is not like java. One of the case where it is not the same is immutable objects. Think about the trivial function lambda x: x. Apply this for x = [1, 2, 3] and x = (1, 2, 3). In the first case, the returned value will be a copy of the input, and identical in the second case. – David Cournapeau Dec 14 '11 at 1:53

6 No, it's *exactly* like Java's semantics for objects. I'm not sure what you mean by "In the first case, the returned value will be a copy of the input, and identical in the second case." but that statement seems to be plainly incorrect. – Mike Graham Nov 14 '12 at 20:58

8 It is exactly the same as in Java. Object references are passed by value. Anyone who thinks differently should attach the Python code for a `swap` function that can swap two references, like this: `a = [42] ; b = 'Hello'; swap(a, b) # Now a is 'Hello', b is [42]` – cayhorstmann Dec 20 '12 at 3:42

7 It is exactly the same as Java when you pass objects in Java. However, Java also have primitives, which are passed by copying the value of the primitive. Thus they differ in that case. – Claudiu Jul 17 '13 at 18:59

show **7** more comments

Think of stuff being passed **by assignment** instead of by reference/by value. That way, it is allways clear, what is happening as long as you understand what happens during normal assignment.

So, when passing a list to a function/method, the list is assigned to the parameter name. Appending to the list will result in the list being modified. Reassigning the list *inside* the function will not change the original list, since:

up vote 68

```
a = [1, 2, 3]
b = a
b.append(4)
b = ['a', 'b']
print a, b      # prints [1, 2, 3, 4] ['a', 'b']
```

down vote

Since immutable types cannot be modified, they *seem* like being passed by value - passing an int into a function means assigning the int to the functions parameter. You can only ever reassign that, but it won't change the originial variables value.

answered Jun 12 '09 at 12:17

share|improve this answer

Daren Thomas
26.7k2190145

3 Really great example code. – Aerovistae Sep 15 '13 at 5:09

add comment

Technically, **Python always uses pass by reference values**. I am going to repeat my other answer to support my statement.

Python always uses pass-by-reference values. There isn't any exception. Any variable assignment means copying the reference value. No exception. Any variable is the name bound to the reference value. Always.

You can think about a reference value as the address of the target object. The address is automatically dereferenced when used. This way, working with the reference value, it seems you work directly with the target object. But there always is a reference in between, one step more to jump to the target.

Here is the example that proves that Python uses passing by reference:

up vote 21 down vote

If the argument was passed by value, the outer `lst` could not be modified. The green are the target objects (the black is the value stored inside, the red is the object type), the yellow is the memory with the reference value inside -- drawn as the arrow. The blue solid arrow is the reference value that was passed to the function (via the dashed blue arrow path). The ugly dark yellow is the internal dictionary. (It actually could be drawn also as a green ellipse. The colour and the shape only says it is internal.)

You can use the `id()` built-in function to learn what the reference value is (that is, the address of the target object).

In compiled languages, a variable is a memory space that is able to capture the value of the type. In Python, a variable is a name (captured internally as a string) bound to the reference variable that holds the reference value to the target object. The name of the variable is the key in the internal dictionary, the value part of that dictionary item stores the reference value to the target.

Reference values are hidden in Python. There isn't any explicit user type for storing the reference value. However, you can use a list element (or element in any other suitable container type) as the reference variable, because all containers do store the elements also as references to the target objects. In other words, elements are actually not contained inside the container -- only the references to elements are.

edited Jan 22 at 22:30

answered Sep 15 '12 at 18:53

share|improve this answer

Peter Mortensen
7,43585283

pepr
4,03411739

Actually this is confirmed its pass by reference value. +1 for this answer although the example wasnt good. – Geo Papas Oct 4 '12 at 17:03

5 Inventing new terminology (such as "pass by reference value" or "call by object" is not helpful). "Call by (value|reference|name)" are standard terms. "reference" is a standard term. Passing references by value accurately describes the behavior of Python, Java, and a host of other languages, using standard terminology. – cayhorstmann Dec 20 '12 at 3:54

1 @cayhorstmann: The problem is that *Python variable* has not the same terminology meaning as in other languages. This way, *call by reference* does not fit well here. Also, how do you *exactly* define the term *reference*? Informally, the Python way could be easily described as passing the address of the object. But it does not fit with a potentially distributed implementation of Python. – pepr Dec 20 '12 at 8:54

I like this answer, but you might consider if the example is really helping or hurting the flow. Also, if you replaced 'reference value' with 'object reference' you would be using terminology that we could consider 'official', as seen here: Defining Functions – Honest Abe Jan 13 at 22:10

1 There is a footnote indicated at the end of that quote, which reads: *"Actually, **call by object reference** would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it..."* I agree with you that confusion is caused by trying to fit terminology established with other languages. Semantics aside, the things that need to be understood are: dictionaries / namespaces, name binding operations and the relationship of name→pointer→object (as you already know). – Honest Abe Jan 16 at 6:28

show **3** more comments

I think it is important to note that the current post with the most votes (by Blair Conrad), while being correct with respect to its result, is misleading and is borderline incorrect based on its definitions. While there are many languages (like C) that allow the user to either pass by reference or pass by value, Python is not one of them.

David Cournapeau's answer points to the real answer and explains why the behavior in Blair Conrad's post seems to be correct while the definitions are not.

up vote 17 down vote

To the extent that Python is pass by value, all languages are pass by value since some piece of data (be it a "value" or a "reference") must be sent. However, that does not mean that Python is pass by value in the sense that a C programmer would think of it.

If you want the behavior, Blair Conrad's answer is fine. But if you want to know the nuts and bolts of why Python is neither pass by value or pass by reference, read David Cournapeau's answer.

edited Aug 13 '12 at 20:21 answered Jun 27 '10 at 12:33

share|improve this answer

Matt Hulse
1,35011022

kobejohn
2,4701228

2 Yes, I think the link David Cournapeau gives is the more complete description – David Sykes Jun 28 '10 at 7:00

2 It is simply not true that all languages are call by value. In C++ or Pascal (and surely many others that I don't know), you have call by reference. For example, in C++, `void swap(int& x, int& y) { int temp = x; x = y; y = temp; }` will swap the variables passed to it. In Pascal, you use `var` instead of `&`. – cayhorstmann Dec 20 '12 at 3:49

I think actually we do not disagree for the most part. There is always some "value" passed regardless of whether it is a memory pointer or a copy of a data structure (the two that you are talking about I believe). Python passes neither of those. It passes names which seem like pointers at first glance but are not. So I think we only disagree on semantics here. No? – kobejohn Dec 20 '12 at 11:02

add comment

Effbot (aka Fredrik Lundh) has described Python's variable passing style as call-by-object: http://effbot.org/zone/call-by-object.htm

Objects are allocated on the heap and pointers to them can be passed around anywhere.

- When you make an assignment such as `x = 1000`, a dictionary entry is created that maps the string "x" in the current namespace to a pointer to the integer object containing one thousand.
- When you update "x" with `x = 2000`, a new integer object is created and the dictionary is updated to point at the new object. The old one thousand object is unchanged (and may or may not be alive depending on whether anything else refers to the object).
- When you do a new assignment such as `y = x`, a new dictionary entry "y" is created that points to the same object as the entry for "x".
- Objects like strings and integers are *immutable*. This simply means that there are no methods that can change the object after it has been created. For example, once the integer object one-thousand is created, it will never change. Math is done by creating new integer objects.
- Objects like lists are *mutable*. This means that the contents of the object can be changed by anything pointing to the object. For example, `x = []; y = x; x.append(10); print y` will print `[10]`. The empty list was created. Both "x" and "y" point to the same list. The *append* method mutates (updates) the list object (like adding a record to a database) and the result is visible to both "x" and "y" (just as a database update would be visible to every connection to that database).

up vote 11 down vote

Hope that clarifies the issue for you.

answered Mar 29 '13 at 4:41

share|improve this answer

Raymond Hettinger
49.3k1085167

I really appreciate learning about this from a developer. Is it true that the `id()` function returns the pointer's (object reference's) value, as pepr's answer suggests? – Honest Abe Jan 13 at 22:21

1 @HonestAbe Yes, in CPython the *id()* returns the address. But in other pythons such as PyPy and Jython, the *id()* is just a unique object identifier. – Raymond Hettinger Jan 14 at 9:03

add comment

You got some really good answers here.

```
x = [ 2, 4, 4, 5, 5 ]
print x  # 2, 4, 4, 5, 5

def go( li ) :
  li = [ 5, 6, 7, 8 ]  # re-assigning what li POINTS TO, does not
  # change the value of the ORIGINAL variable x

go( x )
print x  # 2, 4, 4, 5, 5  [ STILL! ]

raw_input( 'press any key to continue' )
```

up vote 7 down vote

edited Jan 27 '12 at 4:28 answered Jun 12 '09 at 12:16

share|improve this answer

Alex L
3,012835

bobobobo
17.2k16112180

yea, however if you do x = [2, 4, 4, 5, 5], y = x, X[0] = 1 , print x # [1, 4 ,4, 5, 5] print y # [1, 4, 4, 5, 5] – laycat Jun 29 at 3:37

add comment

A simple trick I normally use is to just wrap it in a list:

```
def Change(self, var):
    var[0] = 'Changed'

variable = ['Original']
self.Change(variable)
print variable[0]
```

up vote 6 down vote

(Yeah I know this can be inconvenient, but sometimes it is simple enough to do this.)

answered Aug 5 '11 at 22:52

share|improve this answer edited Aug 8 '11 at 10:39

AmanicA
1,9751029

add comment

In this case the variable titled `var` in the method `Change` is assigned a reference to `self.variable`, and you immediately assign a string to `var`. It's no longer pointing to `self.variable`. The following code snippet shows what would happen if you modify the data structure pointed to by `var` and `self.variable`, in this case a list:

```
>>> class PassByReference:
...     def __init__(self):
...         self.variable = ['Original']
...         self.change(self.variable)
```

```
...         print self.variable
...
...     def change(self, var):
...         var.append('Changed')
...
>>> q = PassByReference()
['Original', 'Changed']
>>>
```

up vote 3 down vote

I'm sure someone else could clarify this further.

answered Jun 12 '09 at 10:39

share|improve this answer

Mike Mazur
1,555816

add comment

The key to understanding parameter passing is to stop thinking about "variables". **There are no variables in Python.**

1. Python has names and objects.
2. Assignment binds a name to an object.
3. Passing an argument into a function also binds a name (the parameter name of the function) to an object.

That is all there is to it. Mutability is irrelevant for this question.

Example:

```
a = 1
```

This binds the name `a` to an object of type integer that holds the value 1.

up vote 3 down vote

```
b = x
```

This binds the name `b` to the same object that the name `x` is currently bound to. Afterwards, the name `b` has nothing to do with the name `x` any more.

See sections 3.1 and 4.1 in the Python 3 language reference.

So in the code shown in the question, the statement `self.Change(self.variable)` binds the name `var` (in the scope of function `Change`) to the object that holds the value `'Original'` and the assignment `var = 'Changed'` (in the body of function `Change`) assigns that same name again: to some other object (that happens to hold a string as well but could have been something else entirely).

answered Feb 11 at 11:29

share|improve this answer edited Feb 11 at 11:56

Lutz Prechelt
42019

"Python has no variables" is a silly and confusing slogan, and I really wish people would stop saying it... :(The rest of this answer is good! – Ned Batchelder Jun 23 at 21:53

It may be shocking, but it is not silly. And I don't think it is confusing either: It hopefully opens up the recipient's mind for the explanation that is coming and puts her in a useful "I wonder what they have instead of variables" attitude. (Yes, your mileage may vary.) – Lutz Prechelt Jun 25 at 7:30

would you also say that Javascript has no variables? They work the same as Python's. Also, Java, Ruby, PHP, ... I think a better teaching technique is, "Python's variables work differently than C's." – Ned Batchelder Jun 25 at 11:09

add comment

as you can state you need to have a mutable object, but let me suggest you to check over the global variables as they can help you or even solve this kind of issues !

http://docs.python.org/3/faq/programming.html#what-are-the-rules-for-local-and-global-variables-in-python

example:

```
>>> def x(y):
...     global z
...     z = y
...

>>> x
<function x at 0x00000000020E1730>
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
>>> z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined

>>> x(2)
>>> x
<function x at 0x00000000020E1730>
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
>>> z
2
```

up vote 1 down vote

answered Feb 10 at 17:57

share|improve this answer

Nuno Aniceto
923

Why would you post this on a question that accepted an answer 5 years ago – Tim Castelijns Feb 10 at 18:06

I was tempted to post a similar response- the original questioner may not have known that what he wanted was in fact to use a global variable, shared among functions. Here's the link I would have shared: stackoverflow.com/questions/423379/… In answer to @Tim, Stack Overflow is not only a question and answer site, it's a vast repository of reference knowledge that only gets stronger and more nuanced- much like an active wiki- with more input. – Max P Magee Jun 30 at 18:39

add comment

Here is the simple (I hope) explanation of the concept `pass by object` used in Python.
Whenever you pass an object to the function, the object itself is passed (object in Python is actually what you'd call a value in other programming languages) not the reference to this object. In other words, when you call:

```
def change_me(list):
   list = [1, 2, 3]

my_list = [0, 1]
change_me(my_list)
```

The actual object - [0, 1] (which would be called a value in other programming languages) is being passed. So in fact the function `change_me` will try to do something like:

up vote -1 down vote

```
[0, 1] = [1, 2, 3]
```

which obviously will not change the object passed to the function. If the function looked like this:

```
def change_me(list):
   list.append(2)
```

Then the call would result in:

```
[0, 1].append(2)
```

which obviously will change the object. This answer explains it well.

answered Oct 2 '12 at 8:03

share|improve this answer

matino
7,30921834

The problem is that the assignment does something else than you expect. The `list = [1, 2, 3]` causes reusing the `list` name for something else and forgeting the originally passed object. However, you can try `list[:] = [1, 2, 3]` (by the way `list` is wrong name for a variable. Thinking about `[0, 1] = [1, 2, 3]` is a complete nonsense. Anyway, what do you think means *the object itself is passed*? What is copied to the function in your opinion? – pepr Oct 3 '12 at 20:46

@pepr objects aren't literals. They are objects. The only way to talk about them is giving them some names. That's why it's so simple once you grasp it, but enormously complicated to explain. :-) – Veky May 9 at 9:10

@Veky: I am aware of that. Anyway, the list literal is converted to the list object. Actually, any object in Python can exist without a name, and it can be used even when not given any name. And you can think about them as about anonymous objects. Think about objects being the elements of a lists. They need not a name. You can access them through indexing of or iterating through the list. Anyway, I insist on `[0, 1] = [1, 2, 3]` is simply a bad example. There is nothing like that in Python. – pepr May 12 at 11:05

@pepr: I don't necessarily mean Python-definition names, just ordinary names. Of course `alist[2]` counts as a name of a third element of alist. But I think I misunderstood what your problem was. :-) – Veky May 12 at 12:35

@Veky: All the names in Python are the same -- strings bound to references. There is no *definition name* vs. an *ordinary name*. The problem of matino is that binding the value inside the function to the name is just binding the value to the name. It cannot be explained as the interaction between the two list objects. The later list object knows nothing about the passed one and the passed one knows nothing about the assigned one. If name is the string bound to the object, then `alist[2]` cannot be called the name of the object. It is not a string. The `alist[2]` is reference to the object. – pepr May 13 at 11:58

show **1** more comment

protected by Robert Harvey♦ Apr 19 '12 at 22:49

Thank you for your interest in this question. Because it has attracted low-quality answers, posting an answer now requires 10 reputation on this site.

Would you like to answer one of these unanswered questions instead?

Not the answer you're looking for? Browse other questions tagged python reference pass-by-reference or ask your own question.

asked **5 years ago**

viewed **262925 times**

active **1 month ago**

Linked

6
Python : When is a variable passed by reference and when by value?
1
Python: Reassign value to variable (using a function)
1
Modify parameter as a side-effect in python
2
Send by ref/by ptr in python?
1
the parameter passing behavior in python function in Python
3
Why do list operations in python operate outside of the function scope?
2
List remains unaltered even after commiting a swap function in python
0
Without pointers, can I pass references as arguments in Python?

0
why doesn't an attribute value in a dictionary within a class refer to the same attribute of the class?
1
How to change the scope of a variable in a function? Python
see more linked questions…

Related

1388
Is Java “pass-by-reference” or “pass-by-value”?
1
Relink python object (i.e. pass by reference)
73
Javascript by reference vs. by value
3
Trouble understanding passing values and references in Python
6
Python : When is a variable passed by reference and when by value?
-1
passing by reference in Python
3
Python and reference passing. Limitation?
0
Bizarre reference behavior in Tree implementation
0
How python pass by reference?
3
Cython & C++: passing by reference

Hot Network Questions

- Is there a man section or other doc repository for data structure definitions?
- 12 Prisoners demonstrate first steam locomotive
- How do you decide when to go home for the day?
- What word means someone who’s satisfied with superficial knowledge?
- What recent programmes to alter highly-entrenched mathematical terminology have succeeded, and under what conditions do they tend to succeed or fail?
- Is it inappropriate to spend personal resources for the company's benefit?
- Software to view video files very fast
- Why can I use the same name for iterator and sequence in a Python for loop?
- How does one justify funding for mathematics research?
- How to join objects with Python?
- How can the compile-time be (exponentially) faster than run-time?
- Why does int i = 1024 * 1024 * 1024 * 1024 compile without error?
- Files Shown in Terminal not Show on Desktop
- 12.04 Update Manager message "New Hardware Support is Available"
- Understanding what contour package does
- Insert text below a figure with Tikz
- How should I handle my manager offering me an unwanted drink?
- Is the apparent lack of (Ricci) curvature in the Schwarzschild metric due to a choice of coordinates?
- Hogwarts: So why aren't the kids "doing it"?
- Optimize parsing more and more
- Free Wi-Fi to end users using hotspot paid ISP line
- Why are module-level variables in a Python exec inaccessible?
- Cannot cat or grep contents of file
- Is there any precautions I should take when using an ssd instead of an hdd?