

How do I pass a variable by reference?

Asked 15 years, 6 months ago Modified 6 months ago Viewed 2.2m times



I wrote this class for testing:

3337



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

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

When I tried creating an instance, the output was `Original`. So it seems like parameters in Python are passed by value. Is that correct? How can I modify the code to get the effect of pass-by-reference, so that the output is `Changed`?

Sometimes people are surprised that code like `x = 1`, where `x` is a parameter name, doesn't impact on the caller's argument, but code like `x[0] = 1` does. This happens because *item assignment* and *slice assignment* are ways to **mutate** an existing object, rather than reassign a variable, despite the `=` syntax. See [Why can a function modify some arguments as perceived by the caller, but not others?](#) for details.

See also [What's the difference between passing by reference vs. passing by value?](#) for important, language-agnostic terminology discussion.

python

reference

parameter-passing

pass-by-reference

Share

Improve this question

Follow

edited Jul 12, 2023 at 16:23



Karl Knechtel

61.2k ● 14 ● 126 ● 183

asked Jun 12, 2009 at 10:23



David Sykes


49.7k ● 18 ● 74 ● 81

33 For a short explanation/clarification see the first answer to [this stackoverflow question](#). As strings are immutable, they won't be changed and a new variable will be created, thus the "outer" variable still has the same value. – [PhilS](#) Jun 12, 2009 at 10:35

11 The code in BlairConrad's answer is good, but the explanation provided by DavidCournapeau and DarenThomas is correct. – [Ethan Furman](#) Jan 7, 2012 at 6:47

- 79 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, 2012 at 0:39
-
- 4 Working link: [Other languages have "variables", Python has "names"](#) – [Abraham Sangha](#) Apr 22, 2020 at 23:30
-
- 13 New official how of lqc's link: david.goodger.org/projects/pycon/2007/idiomatic/... – [Ray Hulha](#) Jun 9, 2020 at 19:56
-

42 Answers

Sorted by: Highest score (default) 

1

2

Next



3570



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.

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 string, 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.

Share

edited Apr 3, 2017 at 2:13

answered Jun 12, 2009 at 11:18

Improve this answer



random_user

848 ● 1 ● 7 ● 18



Blair Conrad

241k ● 25 ● 136 ● 112

Follow

74 -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, 2012 at 6:41

4 "...parameter passed in..." is incorrect use of terminology. A [parameter](#) is a named entity in a function (or method) definition that specifies an argument (or in some cases, arguments) that the function can accept. An [argument](#) is a value passed to a function (or method) when calling the function. – [Honest Abe](#) Feb 10, 2014 at 23:47

2 @EthanFurman You are wrong. Even if you read the link in DavidCournapeau's answer that is [Call By Object](#), you will see that all of the expressions "call by object", "call by sharing" or "call by object reference", have the same meaning that is using an address to access the object. And for sure the address has to be determined from namespace. I can't accept that "Call by object reference" is different from "Call by reference". – [Happy Ahmad](#) Jul 9, 2018 at 9:21

3 @ML_Pro, "pass by assignment" seems to be a term made up by the Python documenters to describe "pass by value". For the user, I see no functional difference between passing a value that is a reference (or handle) as happens in languages such as Java or C# and what Python does, and I'd never use the term "pass by assignment"; it was edited into the answer, I assume to align with the documentation. – [Blair Conrad](#) Aug 4, 2018 at 9:57

3 @HappyAhmad it absolutely *is* different to call by reference. If Python supported call by reference, you could do something like `def foo(&var): var = 2` then `x = 0; y = 1; foo(x); foo(y)` then `print(x, y)` would print `2 2` – [juanpa.arrivillaga](#) Mar 13, 2021 at 19:58



906

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:

```
def __init__(self):
    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.

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

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

Share

Improve this answer

Follow

edited Apr 3, 2017 at 0:39



random_user

848 ● 1 ● 7 ● 18

answered Nov 15, 2011 at 17:45



Mark Ransom

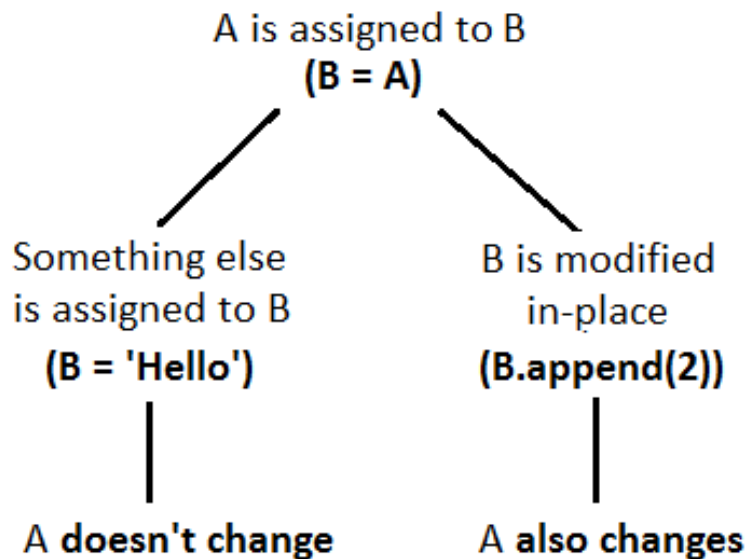
308k ● 44 ● 416 ● 647

-
- 4 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? – [Kashif](#) May 7, 2012 at 1:10
-
- 11 @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, 2012 at 1:46
-
- 2 Function calls do not create new references - use the id function inside and outside of the function to confirm that. The difference is what happens to the object when you attempt to change it inside the function. – [Tony Suffolk](#) 66 Mar 10, 2018 at 10:55
-

- 5 @TonySuffolk66 `id` gives the identity of the object referenced, not the reference itself.
– [Mark Ransom](#) Mar 10, 2018 at 14:24
- 2 @MinhTran in the simplest terms, a reference is something that "refers" to an object. The physical representation of that is most likely a pointer, but that's simply an implementation detail. It really is an abstract notion at heart. – [Mark Ransom](#) Oct 20, 2018 at 4:21

I found the other answers rather long and complicated, so I created this simple diagram to explain the way Python treats variables and parameters.

468




Share

edited May 25, 2016 at 15:30

answered Sep 4, 2014 at 16:05

Improve this answer

 [Zenadix](#)
16.2k ● 5 ● 28 ● 41

Follow

- 15 "A is assigned to B." Is that not ambiguous? I think in ordinary English that can mean either `A=B` or `B=A`. – [Hatshepsut](#) Jul 4, 2016 at 23:06

I do like the visual representation, but still misses the point of `mutable` vs `immutable` which makes the right leg moot since there will be no `append` available. (Still got an upvote for the visual rep though) :) – [Madivad](#) Jul 19, 2016 at 12:40

What do you mean by B is modified in-place? B is not an object – [Abhinav](#) Oct 20, 2019 at 3:57

@Abhinav in Python, EVERYTHING is an object - even simple integers. But some objects can be modified (mutable) and some can't. If the object has an `append` method then it must be mutable. – [Mark Ransom](#) Mar 24, 2021 at 16:00

- 1 "Something else is assigned to B" should be "B is assigned to something else". Names are assigned to values, not the other way around. Names refer to values, values don't know what names they have. – [timgeb](#) Feb 28, 2022 at 9:11



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

277

[Call By Object](#)



Here is a significant quote:



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

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

Follow

edited Feb 18, 2023 at 19:12



[Peter Mortensen](#)

31.6k ● 22 ● 109 ● 133

answered Jun 12, 2009 at 12:55



[David Cournapeau](#)

80.6k ● 9 ● 68 ● 71

-
- 25 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, 2011 at 1:25
-
- 10 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, 2011 at 1:53
-
- 34 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, 2012 at 20:58
-
- 28 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, 2012 at 3:42
-
- 28 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, 2013 at 18:59
-



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

229



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:

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

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 function's parameter. You can only ever reassign that, but it won't change the original variables value.

Share

Improve this answer

Follow

edited Nov 20, 2019 at 4:37



MurugananthamS

2,405 ● 4 ● 21 ● 52

answered Jun 12, 2009 at 12:17



Daren Thomas

70.2k ● 42 ● 155 ● 205

- 11 At first glance this answer seems to sidestep the original question. After a second read I've come to realize that this makes the matter quite clear. A good follow up to this "name assignment" concept may be found here: [Code Like a Pythonista: Idiomatic Python](#) – [Christian Groleau](#) Nov 22, 2017 at 21:45



There are no variables in Python

110

The key to understanding parameter passing is to stop thinking about "variables".

There are names and objects in Python and together they appear like variables, but it is useful to always distinguish the three.



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 to this question.

Example:

```
a = 1
```

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

```
b = x
```

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

See sections [3.1](#) and [4.2](#) in the Python 3 language reference.

How to read the example in the question

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).

How to pass by reference

So if the thing you want to change is a mutable object, there is no problem, as everything is effectively passed by reference.

If it is an [immutable](#) object (e.g. a bool, number, string), the way to go is to wrap it in a mutable object.

The quick-and-dirty solution for this is a one-element list (instead of `self.variable`, pass `[self.variable]` and in the function modify `var[0]`).

The more [pythonic](#) approach would be to introduce a trivial, one-attribute class. The function receives an instance of the class and manipulates the attribute.

Share

Improve this answer

Follow

edited Apr 27, 2020 at 10:17



faressalem

644 ● 8 ● 22

answered Feb 11, 2014 at 11:29



Lutz Prechelt

39.2k ● 11 ● 67 ● 89

57 "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, 2014 at 21:53

23 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, 2014 at 7:30

29 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, 2014 at 11:09

14 Yes, Java has variables. So does Python, and JavaScript, Ruby, PHP, etc. You wouldn't say in Java that `int` declares a variable, but `Integer` does not. They both declare variables. The `Integer` variable is an object, the `int` variable is a primitive. As an example, you demonstrated how your variables work by showing `a = 1; b = a; a++ # doesn't modify b`. That's exactly true in Python also (using `+= 1` since there is no `++` in Python)!
– Ned Batchelder Oct 29, 2014 at 16:29

7 The concept of "variable" is complex and often vague: **A variable is a container for a value, identified by a name.** In Python, the values are objects, the containers are objects (see the problem?) and the names are actually separate things. I believe it is much tougher to get an *accurate* understanding of variables in this manner. The names-and-objects explanation appears more difficult, but is actually simpler. – Lutz Prechelt Oct 9, 2015 at 10:51

Effbot (aka Fredrik Lundh) has described Python's variable passing style as call-by-object: <https://web.archive.org/web/20201111195827/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).

Hope that clarifies the issue for you.

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.

Share

Improve this answer

Follow

edited May 23, 2017 at 12:18



Community Bot
1 ● 1

answered Sep 15, 2012 at 18:53



pepr
20.7k ● 15 ● 81 ● 144

-
- 47 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, 2012 at 3:54
-
- 6 @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, 2012 at 8:54
-
- 1 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, 2014 at 22:10
-
- 1 Well, but the official says... "*arguments are passed using **call by value** (where the **value** is always an object **reference**, not the value of the object).*" This way, you may be tempted to substitute it textually as ... *arguments are passed using **call by reference***, which is a bit confusing because it is not true. The confusion is caused by a bit more complex situation where none of the classical terms fits perfectly. I did not find any simpler example that would illustrate the behaviour. – [pepr](#) Jan 14, 2014 at 23:31 ✎
-

- 2 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, 2014 at 6:28
-



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

62



```
def Change(self, var):  
    var[0] = 'Changed'  
  
variable = ['Original']  
self.Change(variable)  
print variable[0]
```



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

Share

[edited Aug 8, 2011 at 10:39](#)

answered Aug 5, 2011 at 22:52

Improve this answer



[AmanicaA](#)

5,477 ● 1 ● 39 ● 52

Follow

-
- 9 Nice. To pass by ref, wrap in []'s. – [Justas](#) Feb 1, 2015 at 5:49
-



You got some really good answers here.

29



```
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' )
```

Share

edited Jan 27, 2012 at 4:28

answered Jun 12, 2009 at 12:16

Improve this answer



Alex L

8,915 ● 6 ● 52 ● 76



bobobobo

67.1k ● 66 ● 271 ● 372

Follow

3 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, 2014 at 3:37



23



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']
>>>
```

I'm sure someone else could clarify this further.

Share Improve this answer Follow

answered Jun 12, 2009 at 10:39



Mike Mazur

2,519 ● 1 ● 16 ● 26

22

Python's pass-by-assignment scheme isn't quite the same as C++'s reference parameters option, but it turns out to be very similar to the argument-passing model of the C language (and others) in practice:

- Immutable arguments are effectively passed "**by value**." Objects such as integers and strings are passed by object reference instead of by copying, but because you can't change immutable objects in place anyhow, the effect is much like making a copy.
- Mutable arguments are effectively passed "**by pointer**." Objects such as lists and dictionaries are also passed by object reference, which is similar to the way C passes arrays as pointers—mutable objects can be changed in place in the function, much like C arrays.

Share Improve this answer Follow

answered Mar 27, 2015 at 4:38



ajknzhol

6,450 ● 13 ● 49 ● 75

20

There are a lot of insights in answers here, but I think an additional point is not clearly mentioned here explicitly. Quoting from Python documentation [What are the rules for local and global variables in Python?](#)

In Python, variables that are only referenced inside a function are implicitly global. If a variable is assigned a new value anywhere within the function's body, it's assumed to be a local. If a variable is ever assigned a new value inside the function, the variable is implicitly local, and you need to explicitly declare it as 'global'. Though a bit surprising at first, a moment's consideration explains this. On one hand, requiring global for assigned variables provides a bar against unintended side-effects. On the other hand, if global was required for all global references, you'd be using global all the time. You'd have to declare as global every reference to a built-in function or to a component of an imported module. This clutter would defeat the usefulness of the global declaration for identifying side-effects.

Even when passing a mutable object to a function this still applies. And to me it clearly explains the reason for the difference in behavior between assigning to the object and operating on the object in the function.

```
def test(l):
    print "Received", l, id(l)
    l = [0, 0, 0]
    print "Changed to", l, id(l) # New local object created, breaking link to
                                global l
```



```
l = [1, 2, 3]
print "Original", l, id(l)
test(l)
print "After", l, id(l)
```

gives:

```
Original [1, 2, 3] 4454645632
Received [1, 2, 3] 4454645632
Changed to [0, 0, 0] 4474591928
After [1, 2, 3] 4454645632
```

The assignment to an global variable that is not declared global therefore creates a new local object and breaks the link to the original object.

Share

edited Feb 18, 2023 at 20:37

answered Sep 12, 2014 at 14:40

Improve this answer



Peter Mortensen

31.6k ● 22 ● 109 ● 133



Joop

8,108 ● 12 ● 45 ● 59

Follow



18



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 issue!

<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
```

Share

Improve this answer

Follow

edited Aug 9, 2016 at 2:23



John Smith

7,399 ● 7 ● 51 ● 63

answered Feb 10, 2014 at 17:57



Nuno Aniceto

1,982 ● 1 ● 17 ● 17

- 5 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/...](http://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, 2014 at 18:39



11



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:

```
[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.

Share

edited May 23, 2017 at 11:54

answered Oct 2, 2012 at 8:03

Improve this answer



Community Bot

1 • 1



matino

17.7k • 8 • 50 • 60

Follow

- 3 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 forgetting 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, 2012 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, 2014 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, 2014 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, 2014 at 12:35

Argh. My English is obviously much worse than my Python. :-) I'll try just once more. I just said you have to give object some names just to talk about them. By that "names" I didn't mean "names as defined by Python". I know Python mechanisms, don't worry. – [Veky](#) May 15, 2014 at 5:20



10



Aside from all the great explanations on how this stuff works in Python, I don't see a simple suggestion for the problem. As you seem to do create objects and instances, the Pythonic way of handling instance variables and changing them is the following:

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

    def Change(self):
        self.variable = 'Changed'
```

In instance methods, you normally refer to `self` to access instance attributes. It is normal to set instance attributes in `__init__` and read or change them in instance methods. That is also why you pass `self` as the first argument to `def Change`.

Another solution would be to create a static method like this:

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

    @staticmethod
    def Change(var):
        var = 'Changed'
        return var
```

Share

Improve this answer

Follow

edited Feb 18, 2023 at 20:41



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Feb 7, 2016 at 23:18



Dolf Andringa

2,170 ● 1 ● 22 ● 39



9

I used the following method to quickly convert some Fortran code to Python. True, it's not pass by reference as the original question was posed, but it is a simple workaround in some cases.



```
a = 0
b = 0
c = 0

def myfunc(a, b, c):
    a = 1
    b = 2
    c = 3
    return a, b, c

a, b, c = myfunc(a, b, c)
print a, b, c
```

Share

Improve this answer

Follow

edited Feb 18, 2023 at 20:23



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Aug 8, 2016 at 16:46



Brad Porter

449 ● 4 ● 6

Yes, this solves the 'pass by reference' in my use case as well. I have a function that basically cleans up values in a `dict` and then returns the `dict`. However, while cleaning up it may become apparent a rebuild of a part of the system is required. Therefore, the function must not only return the cleaned `dict` but also be able to signal the rebuild. I tried to pass a `bool` by reference, but ofc that doesn't work. Figuring out how to solve this, I found your solution (basically returning a tuple) to work best while also not being a hack/workaround at all (IMHO). – [kasimir](#) May 1, 2020 at 16:08



To simulate passing an object by reference, wrap it in a one-item list:

```
class PassByReference:
    def __init__(self, name):
        self.name = name

def changeRef(ref):
    ref[0] = PassByReference('Michael')

obj = PassByReference('Peter')
print(obj.name)

p = [obj]
changeRef(p)

print(p[0].name)
```

Assigning to an element of the list mutates the list rather than reassigning a name. Since the list itself has reference semantics, the change is reflected in the caller.

Share

Improve this answer

Follow

edited Jul 12, 2023 at 17:19



Karl Knechtel

61.2k ● 14 ● 126 ● 183

answered Apr 21, 2016 at 16:47



itmuckel

1,420 ● 16 ● 24

`p` is reference to a mutable list object which in turn stores the object `obj`. The reference 'p', gets passed into `changeRef`. Inside `changeRef`, a new reference is created (the new reference is called `ref`) that points to the same list object that `p` points to. But because lists are mutable, changes to the list are visible by *both* references. In this case, you used the `ref` reference to change the object at index 0 so that it subsequently stores the `PassByReference('Michael')` object. The change to the list object was done using `ref` but this change is visible to `p`. – Minh Tran Oct 20, 2018 at 3:19

So now, the references `p` and `ref` point to a list object that stores the single object, `PassByReference('Michael')`. So it follows that `p[0].name` returns `Michael`. Of course, `ref` has now gone out of scope and may be garbage collected but all the same. – Minh Tran Oct 20, 2018 at 3:22 ✎

- 1 You have *not* changed the private instance variable, `name`, of the original `PassByReference` object associated with the reference `obj`, though. In fact, `obj.name` will return `Peter`. The aforementioned comments assumes the definition Mark Ransom gave. – Minh Tran Oct 20, 2018 at 3:25 ✎
- 1 Point being, I don't agree that it's a *hack* (which I take to mean to refer to something that works but for reasons unknown, untested, or unintended by the implementer). You simply replaced one `PassByReference` object with another `PassByReference` object in your list and referred to the latter of the two objects. – Minh Tran Oct 20, 2018 at 3:32 ✎

Since it seems to be nowhere mentioned an approach to simulate references as known from e.g. C++ is to use an "update" function and pass that instead of the actual variable (or rather, "name"):



```
def need_to_modify(update):
    update(42) # set new value 42
    # other code

def call_it():
    value = 21
    def update_value(new_value):
        nonlocal value
        value = new_value
    need_to_modify(update_value)
    print(value) # prints 42
```

This is mostly useful for "out-only references" or in a situation with multiple threads / processes (by making the update function thread / multiprocessing safe).

Obviously the above does not allow *reading* the value, only updating it.

Share Improve this answer Follow

answered May 10, 2019 at 0:37



Daniel Jour

16.2k ● 2 ● 39 ● 65

I think it would be better to have a separate, language-agnostic Q&A for strategies for emulating pass-by-reference. – [Karl Knechtel](#) Jul 12, 2023 at 17:08



5



Given the way Python handles values and references to them, the only way you can reference an arbitrary instance attribute is by name:

```
class PassByReferenceIsh:
    def __init__(self):
        self.variable = 'Original'
        self.change('variable')
        print self.variable

    def change(self, var):
        self.__dict__[var] = 'Changed'
```

In real code you would, of course, add error checking on the dict lookup.

Share

edited Feb 18, 2023 at 20:31

answered Oct 31, 2016 at 15:33

Improve this answer



Peter Mortensen

31.6k ● 22 ● 109 ● 133



mARK bLOORE

176 ● 2 ● 7

Follow



4

Since your example happens to be object-oriented, you could make the following change to achieve a similar result:



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

    def change(self, var):
        setattr(self, var, 'Changed')

# o.variable will equal 'Changed'
o = PassByReference()
assert o.variable == 'Changed'
```

Share Improve this answer Follow

answered Sep 10, 2017 at 2:19



Jesse Hogan

3,223 ● 2 ● 17 ● 17

2 Although this works. It is not pass by reference. It is 'pass by object reference'.

– Bishwas Mishra Mar 14, 2018 at 11:48



4

You can merely use **an empty class** as an instance to store reference objects because internally object attributes are stored in an instance dictionary. See the example.



```
class RefsObj(object):
    "A class which helps to create references to variables."
    pass

...

# an example of usage
def change_ref_var(ref_obj):
    ref_obj.val = 24

ref_obj = RefsObj()
ref_obj.val = 1
print(ref_obj.val) # or print ref_obj.val for python2
change_ref_var(ref_obj)
print(ref_obj.val)
```

Share Improve this answer Follow

answered May 3, 2018 at 14:14



sergzach

6,754 ● 7 ● 51 ● 86



4

Since dictionaries are passed by reference, you can use a dict variable to store any referenced values inside it.



```
# returns the result of adding numbers `a` and `b`
def AddNumbers(a, b, ref): # using a dict for reference
    result = a + b
    ref['multi'] = a * b # reference the multi. ref['multi'] is number
    ref['msg'] = "The result: " + str(result) + " was nice!"
    return result

number1 = 5
number2 = 10
ref = {} # init a dict like that so it can save all the referenced values. this
is because all dictionaries are passed by reference, while strings and numbers
do not.

sum = AddNumbers(number1, number2, ref)
print("sum: ", sum) # the returned value
print("multi: ", ref['multi']) # a referenced value
print("msg: ", ref['msg']) # a referenced value
```

Share

edited Jun 1, 2021 at 8:29

answered May 5, 2019 at 14:14

Improve this answer

 **Liakos**
572 ● 7 ● 12

Follow



4



While pass by reference is nothing that fits well into Python and should be rarely used, there are some workarounds that actually can work to get the object currently assigned to a local variable or even reassign a local variable from inside of a called function.

The basic idea is to have a function that can do that access and can be passed as object into other functions or stored in a class.

One way is to use `global` (for global variables) or `nonlocal` (for local variables in a function) in a wrapper function.

```
def change(wrapper):
    wrapper(7)

x = 5

def setter(val):
    global x
    x = val

print(x)
```

The same idea works for reading and `del` eting a variable.

For just reading, there is even a shorter way of just using `lambda: x` which returns a callable that when called returns the current value of `x`. This is somewhat like "call by name" used in languages in the distant past.

Passing 3 wrappers to access a variable is a bit unwieldy so those can be wrapped into a class that has a proxy attribute:

```
class ByRef:
    def __init__(self, r, w, d):
        self._read = r
        self._write = w
        self._delete = d
    def set(self, val):
        self._write(val)
    def get(self):
        return self._read()
    def remove(self):
        self._delete()
    wrapped = property(get, set, remove)

# Left as an exercise for the reader: define set, get, remove as local
# functions using global / nonlocal
r = ByRef(get, set, remove)
r.wrapped = 15
```

Python's "reflection" support makes it possible to get an object that is capable of reassigning a name/variable in a given scope without defining functions explicitly in that scope:

```
class ByRef:
    def __init__(self, locs, name):
        self._locs = locs
        self._name = name
    def set(self, val):
        self._locs[self._name] = val
    def get(self):
        return self._locs[self._name]
    def remove(self):
        del self._locs[self._name]
    wrapped = property(get, set, remove)

def change(x):
    x.wrapped = 7

def test_me():
    x = 6
    print(x)
    change(ByRef(locals(), "x"))
    print(x)
```

Here the `ByRef` class wraps a dictionary access. So attribute access to `wrapped` is translated to a item access in the passed dictionary. By passing the result of the builtin `locals` and the name of a local variable, this ends up accessing a local variable. The Python documentation as of 3.5 advises that changing the dictionary might not work, but it seems to work for me.

Improve this answer
Follow



Peter Mortensen
31.6k ● 22 ● 109 ● 133



textshell
2,056 ● 16 ● 21

Re "The Python documentation as of 3.5": Can you add the reference? (But **** **without** **** "Edit:", "Update:", or similar - the answer should appear as if it was written today.)
– Peter Mortensen Feb 18, 2023 at 20:29 ✎



3



Pass-by-reference in Python is quite different from the concept of pass by reference in C++/Java.

- **Java and C#:** primitive types (including string) pass by value (copy). A reference type is passed by reference (address copy), so all changes made in the parameter in the called function are visible to the caller.
- **C++:** Both pass-by-reference or pass-by-value are allowed. If a parameter is passed by reference, you can either modify it or not depending upon whether the parameter was passed as const or not. However, const or not, the parameter maintains the reference to the object and reference cannot be assigned to point to a different object within the called function.
- **Python:** Python is "pass-by-object-reference", of which it is often said: "Object references are passed by value." ([read here](#)). Both the caller and the function refer to the same object, but the parameter in the function is a new variable which is just holding a copy of the object in the caller. Like C++, a parameter can be either modified or not in function. This depends upon the type of object passed. For example, an immutable object type cannot be modified in the called function whereas a mutable object can be either updated or re-initialized.

A crucial difference between updating or reassigning/re-initializing the mutable variable is that updated value gets reflected back in the called function whereas the reinitialized value does not. The scope of any assignment of new object to a mutable variable is local to the function in the python. Examples provided by @blair-conrad are great to understand this.

Share

Improve this answer
Follow

edited Feb 18, 2023 at 20:35



Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered Jan 22, 2019 at 19:59



Alok Garg
174 ● 9



3



I found other answers a little bit confusing and I had to struggle a while to grasp the concepts. So, I am trying to put the answer in my language. It may help you if other answers are confusing to you too. So, the answer is like this-

When you create a list-



```
my_list = []
```

you are actually creating an object of the class list:

```
my_list = list()
```

Here, my_list is just a name given to the memory address (e.g., 140707924412080) of the object created by the constructor of the 'list' class.

When you pass this list to a method defined as

```
def my_method1(local_list):  
    local_list.append(1)
```

another reference to the same memory address 140707924412080 is created. So, when you make any changes/mutate to the object by using append method, it is also reflected outside the my_method1. Because, both the outer list my_list and local_list are referencing the same memory address.

On the other hand, when you pass the same list to the following method,

```
def my_method2(local_list2):  
    local_list2 = [1,2,3,4]
```

the first half of the process remains the same. i.e., a new reference/name local_list2 is created which points to the same memory address 140707924412080. But when you create a new list [1,2,3,4], the constructor of the 'list' class is called again and a new object is created. This new object has a completely different memory address, e.g., 140707924412112. When you assign local_list2 to [1,2,3,4], now the local_list2 name refers to a new memory address which is 140707924412112. Since in this entire process you have not made any changes to the object placed at memory address 140707924412080, it remains unaffected.

In other words, it is in the spirit that 'other languages have variables, Python have names'. That means in other languages, variables are referenced to a fixed address in memory. That means, in C++, if you reassign a variable by

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

the memory address where the value '1' was stored is now holding the value '2' And hence, the value '1' is completely lost. Whereas in Python, since everything is an object, earlier 'a' referred to the memory address that stores the object of class 'int'

which in turn stores the value '1'. But, after reassignment, it refers to a completely different memory address that stores the newly created object of class 'int' holding the value '2'.

Hope it helps.

Share Improve this answer Follow

answered Mar 18, 2023 at 14:32



Bhanuday Sharma

433 ● 5 ● 10



2



I am new to Python, started yesterday (though I have been programming for 45 years).

I came here because I was writing a function where I wanted to have two so-called out-parameters. If it would have been only one out-parameter, I wouldn't get hung up right now on checking how reference/value works in Python. I would just have used the return value of the function instead. But since I needed *two* such out-parameters I felt I needed to sort it out.

In this post I am going to show how I solved my situation. Perhaps others coming here can find it valuable, even though it is not exactly an answer to the topic question. Experienced Python programmers of course already know about the solution I used, but it was new to me.

From the answers here I could quickly see that Python works a bit like JavaScript in this regard, and that you need to use workarounds if you want the reference functionality.

But then I found something neat in Python that I don't think I have seen in other languages before, namely that you can return more than one value from a function, in a simple comma-separated way, like this:

```
def somefunction(p):  
    a = p + 1  
    b = p + 2  
    c = -p  
    return a, b, c
```

and that you can handle that on the calling side similarly, like this

```
x, y, z = somefunction(w)
```

That was good enough for me and I was satisfied. There isn't any need to use some workaround.

In other languages you can of course also return many values, but then usually in the form of an object, and you need to adjust the calling side accordingly.

The Python way of doing it was nice and simple.

If you want to mimic *by reference* even more, you could do as follows:

```
def somefunction(a, b, c):
    a = a * 2
    b = b + a
    c = a * b * c
    return a, b, c

x = 3
y = 5
z = 10
print(F"Before : {x}, {y}, {z}")

x, y, z = somefunction(x, y, z)

print(F"After  : {x}, {y}, {z}")
```

which gives this result

```
Before : 3, 5, 10
After  : 6, 11, 660
```

Share

Improve this answer

Follow

edited Feb 18, 2023 at 20:49



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jul 18, 2020 at 16:08



Magnus

1,754 ● 20 ● 15

- 4 "But then I found something neat in Python that I don't think I have seen in other languages before, namely that you can return more than one value from a function" No, you can't. What you are doing is returning a single value, a `tuple`, which is what the expression `a, b, c` creates. You then use *iterable unpacking* to unpack that tuple into separate variables. Of course, in effect, you can think of this as "returning multiple values", but you aren't actually doing that, you are returning a container. — [juanpa.arrivillaga](#) Aug 12, 2020 at 19:45
- 1 This rambles a lot and is mostly off the topic of OP's question. It refers instead to this concept: stackoverflow.com/questions/354883 - which is well addressed by other existing Q&A. — [Karl Knechtel](#) Jul 12, 2023 at 17:22

Alternatively, you could use [ctypes](#) which would look something like this:

```
import ctypes

def f(a):
    a.value = 2398 ## Resign the value in a function
```



```
a = ctypes.c_int(0)
print("pre f", a)
f(a)
print("post f", a)
```

As `a` is a `c int` and not a Python integer and apparently passed by reference. However, you have to be careful as strange things could happen, and it is therefore not advised.

Share

Improve this answer

Follow

edited Feb 18, 2023 at 20:53



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jan 28, 2021 at 11:06



Julian wandhoven

288 ● 2 ● 11

Use [dataclasses](#). Also, it allows you to apply type restrictions (aka "type hints").

```
from dataclasses import dataclass

@dataclass
class Holder:
    obj: your_type # Need any type? Use "obj: object" then.

def foo(ref: Holder):
    ref.obj = do_something()
```

I agree with folks that in most cases you'd better consider not to use it.

And yet, when we're talking about [contexts](#), it's worth to know that way.

You can design an explicit context class though. When prototyping, I prefer dataclasses, just because it's easy to serialize them back and forth.

Share

Improve this answer

Follow

edited Feb 18, 2023 at 21:05



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jun 28, 2021 at 17:29



Stepan Dyatkovskiy

970 ● 10 ● 17

1.9 mill views and 13 years later a decent solution comes up. I have implemented it calling the Holder "State" and adding a boolean value that can now be modified in a different function ... excellent and clean! – [Wolfgang Fahl](#) Nov 27, 2022 at 17:12

@WolfgangFahl several prior answers do functionally the same thing, just with a list element, ordinary object attribute etc. instead of a dataclass instance attribute. They all fundamentally work in the same way: since passing an object with reference semantics by value allows for mutation but not reassignment, we convert the desired reassignment into mutation by creating a wrapper object. – [Karl Knechtel](#) Jul 12, 2023 at 17:11

Agreed with @Karl Knechtel totally. Only reason I highlighted this case is that dataclasses support are quite friendly for serialization and provide minimum confusing API. So if somebody will extend your code there will be no temptation to perform list operations on what you supposed to use as a ref holder. – [Stepan Dyatkovskiy](#) Jul 16, 2023 at 8:35

There are already many great answers (or let's say opinions) about this and I've read them, but I want to mention a missing one. The one from [Python's documentation](#) in the FAQ section. I don't know the date of publishing this page, but this should be our true reference:

Remember that arguments are **passed by assignment** in Python. Since assignment just creates references to objects, there's no alias between an argument name in the caller and callee, and so **no call-by-reference** per se.

If you have:

```
a = SOMETHING

def fn(arg):
    pass
```

and you call it like `fn(a)`, you're doing exactly what you do in assignment. So this happens:

```
arg = a
```

An additional reference to `SOMETHING` is created. Variables are just symbols/names/references. They don't "hold" anything.

Share

Improve this answer

Follow

edited Feb 18, 2023 at 21:19



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Oct 4, 2022 at 8:39



S.B

16.3k ● 11 ● 33 ● 64

1

2

Next



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.