python; modifying list inside a function

Asked 6 years, 7 months ago Active 5 months ago Viewed 56k times

28

6

Suppose I have function with list parameter, and inside its body I want to modify passed list, by copying elements of an array to the list:

```
def function1 (list_arg):
   a = function2()    #function2 returns an array of numbers
   list_arg = list(a)

list1 = [0] * 5
function1(list1)
list1
[0,0,0,0,0]
```

When doing it like this, it doesn't work. After executing `function1(list1)`, `list1` remains unchanged. So, how to make `function1` return `list1` with the same elements (numbers) as array `a`?

python list Edit tags

edited Apr 25 at 10:44

pfnuesel
9,667 10 48 60

asked Feb 26 '14 at 22:20

roberto
319 1 4 11

what does `'a = some array'` mean? – msvalkon Feb 26 '14 at 22:21

So, what exactly do you expect the `list1` to contain when your function returns? What is the content of `a` and where is it defined? – msvalkon Feb 26 '14 at 22:24

1 @user155 FYI, the standard syntax for comments is `#Your comment here`. Using strings is usually reserved for when you want multiline comments or docstrings. – Asad Saeeduddin Feb 26 '14 at 22:26

2 Reading this may help your understanding, and not just of functions. – John Y Feb 26 '14 at 22:43

excellent explanation in the above site from "John Y" – Joonho Park Jun 13 '19 at 2:20

4 Answers

Active | Oldest | Votes

35

If you assign something to the variable `list_arg`, it will from then on point to the new value. The value it pointed to *before* that assignment (your original list) will stay unchanged.

If you, instead, assign something to *elements* of that list, this will change the original list:

```
list_arg[:] = list(a)
```

This will make your code work as you wanted it.

But keep in mind that in-place changes are hard to understand and probably can confuse the next developer who has to maintain your code.

answered Feb 26 '14 at 22:38

Alfe
44.4k 14 74 132

5

You can operate on the list to change its values (eg, append something to it, or set its values) but changes will be reflected outside of the function only if you operate on the reference to the passed in object:

```
def function1 (list_arg):
   list_arg.append(5)
```

If you have questions when doing this, print out the `id` s:

```
def function1 (list_arg):
   print 1, id(list_arg)
   list_arg[:] = ["a", "b", "c"]
   print 2, id(list_arg)
   list_arg = range(10)
   print 3, id(list_arg)

x = [1,2,3]
function1(x)
print x
```

prints:

```
1 4348413856
2 4348413856
3 4348411984
['a', 'b', 'c']
```

That is, `x` is changed in place, but assigning to the function's local variable `list_arg` has no impact on `x` , because is then just assigns a different object to `list_arg` .

edited Feb 26 '14 at 22:56

answered Feb 26 '14 at 22:24

tom10
56.6k 7 111 122

Nice one! I'm still trying to figure out the question. :/ – Drewness Feb 26 '14 at 22:27

6

What I think you are asking is why after calling `f(a)` , when `f` re-assigns the `a` you passed, `a` is still the "old" `a` you passed.

The reason for this is how Python treats variables and pass them to functions. They are passed *by reference*, but the reference is passed *by value* (meaning that a copy is created). This means that the reference you have inside `f` is actually a copy of the reference you passed. This again implies that if you

Now, if you rather than reassigning the local variable/reference inside `f` (which won't work, since it's a copy) perform mutable operations on it, such as `append()`, the list you pass will have changed after `f` is done.

See also the question [How do I pass a variable by reference?] which treats the problem and possible solutions in further detail.

TL;DR: Reassigning a variable inside a function won't change the variable you passed as an argument outside the function. Performing mutable operations on the variable, however, will change it.

edited May 23 '17 at 10:31
Community ♦
1 1

answered Feb 26 '14 at 22:38

Håvard S
20.4k 5 55 67

2

You're changing a reference to a local variable. When you pass in list_arg this way:

```
def function1 (list_arg):
```

list_arg is a reference to an underlying list object. When you do this:

```
    list_arg = list(a)
```

You're changing what list_arg means within the function. Since the function exits right after that, list_arg = list(a) has no effect.

If you want to actually change the reference to the list you have to do assign it to the result of the function.

```
def function1 ():
   'a = some array'
   return list(a)

list1 = [0] * 5
list1 = function1()
```

Or you could modify the contents of the list without changing the reference.

```
def function1(list_arg):
    del list_arg[:]  # Clears the array
    'a = some array'
    list_arg.extend(a)
```

answered Feb 26 '14 at 22:30

Geoff Genz
1,656 14 17
