

Stack Overflow is a community of 4.7 million programmers, just like you, helping each other.

Join them; it only takes a minute:

Sign up

Join the Stack Overflow community to:



Ask programming questions



Answer and help your peers



Get recognized for your expertise

## Using global variables in a function other than the one that created them



If I create a global variable in one function, how can I use that variable in another function?

Do I need to store the global variable in a local variable of the function which needs its access?

python global-variables scope

edited Sep 22 '14 at 12:58

igaurav  
1,143 ● 8 ● 21

asked Jan 8 '09 at 5:45

user46646  
22.4k ● 32 ● 58 ● 71

### 16 Answers

You can use a global variable in other functions by declaring it as `global` in each function that assigns to it:

```
globvar = 0

def set_globvar_to_one():
    global globvar    # Needed to modify global copy of globvar
    globvar = 1

def print_globvar():
    print globvar     # No need for global declaration to read value of globvar

set_globvar_to_one()
print_globvar()      # Prints 1
```

I imagine the reason for it is that, since global variables are so dangerous, Python wants to make sure that you really know that's what you're playing with by explicitly requiring the `global` keyword.

See other answers if you want to share a global variable across modules.

edited Oct 14 '14 at 20:01

l4mpi  
3,663 ● 3 ● 13 ● 37

answered Jan 8 '09 at 8:39

Paul Stephenson  
27.8k ● 7 ● 34 ● 43

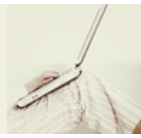
**319** It's extreme exaggeration to refer to globals as "so dangerous." Globals are perfectly fine in every language that has ever existed and ever will exist. They have their place. What you should have said is they can cause issues if you have no clue how to program. – [Anthony](#) Dec 22 '12 at 23:22

**73** I think they are fairly dangerous. However in python "global" variables are actually module-level, which solves a lot of issues. – [Fábio Santos](#) Jan 22 '13 at 15:41

**25** I was using hyperbole, but I think my underlying point is still valid. – [Anthony](#) May 7 '13 at 19:32

**28** @ArtOfWarfare Aren't `set_globvar_to_one` and `print_globvar` globals too? Everybody freaks out about global scalars (lists, objects, etc), but not global functions, which are first-class citizens, too. – [Hyperboreus](#) Nov 16 '13 at 8:56

**33** I disagree that the reason Python requires the `global` keyword is because globals are dangerous. Rather, it's because the language doesn't require you to explicitly declare variables and automatically assumes that a variable that you assign has function scope unless you tell it otherwise. The `global` keyword is the means that is provided to tell it otherwise. – [Nate C-K](#) Sep 3 '14 at 13:48



If I'm understanding your situation correctly, what you're seeing is the result of how Python handles local (function) and global (module) namespaces.

Say you've got a module like this:

```
# sample.py
myGlobal = 5

def func1():
    myGlobal = 42

def func2():
    print myGlobal

func1()
func2()
```

You might expecting this to print 42, but instead it prints 5. As has already been mentioned, if you add a 'global' declaration to `func1()`, then `func2()` will print 42.

```
def func1():
    global myGlobal
    myGlobal = 42
```

What's going on here is that Python assumes that any name that is *assigned to*, anywhere within a function, is local to that function unless explicitly told otherwise. If it is only *reading* from a name, and the name doesn't exist locally, it will try to look up the name in any containing scopes (e.g. the module's global scope).

When you assign 42 to the name `myGlobal`, therefore, Python creates a local variable that shadows the global variable of the same name. That local goes out of scope and is [garbage-collected](#) when `func1()` returns; meanwhile, `func2()` can never see anything other than the (unmodified) global name. Note that this namespace decision happens at compile time, not at runtime -- if you were to read the value of `myGlobal` inside `func1()` before you assign to it, you'd get an `UnboundLocalError`, because Python has already decided that it must be a local variable but it has not had any value associated with it yet. But by using the 'global' statement, you tell Python that it should look elsewhere for the name instead of assigning to it locally.

(I believe that this behavior originated largely through an optimization of local namespaces -- without this behavior, Python's VM would need to perform at least three name lookups each time a new name is assigned to inside a function (to ensure that the name didn't already exist at module/builtin level), which would significantly slow down a very common operation.)

edited Mar 16 '15 at 18:01



Michael

1,266 ● 15 ● 26

answered Jan 8 '09 at 9:19



Jeff Shannon

5,459 ● 1 ● 9 ● 7

65 You should write a Python text book :) – [alwbtc](#) May 7 '11 at 23:58

37 Good explanation. I find it convenient to remind myself from time to time that, in python, "assignment is not an operator." (Contrary to, say, C++, which is where I spend most of my programming time, hence the need for frequent reminders.) This leads to some apparently paradoxical results -- if `myGlobal` were an array -- `[5]` -- instead of a scalar, and `func1` did `myGlobal.append(42)`, then `func2` would print `[5, 42]` even without a "global" declaration. – [c-urchin](#) Jan 24 '12 at 22:58

7 learn the LEGB rule <http://stackoverflow.com/questions/291978/short-description-of-python-scoping-rules> – [laike9m](#) May 8 '13 at 2:23

1 Your answers is like meditation. I want to be still and meditate on this answer. This is so good. – [NullException](#) Apr 7 '14 at 16:23

1 Yep, good answer. If you understand how the language is interpreted (read books about language parsing) it explains a lot why the `global` keyword is needed. – [tiktak](#) Jul 11 '14 at 22:47

You may want to explore the notion of [namespaces](#). In Python, the [module](#) is the natural place for *global* data:

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, `modname.itemname`.

A specific use of global-in-a-module is described here - [how-do-i-share-global-variables-across-modules](#):

The canonical way to share information across modules within a single program is to create a

special configuration module (often called config or cfg). Just import the configuration module in all modules of your application; the module then becomes available as a global name. Because there is only one instance of each module, any changes made to the module object get reflected everywhere. For example:

File: config.py

```
x = 0 # Default value of the 'x' configuration setting
```

File: mod.py

```
import config
config.x = 1
```

File: main.py

```
import config
import mod
print config.x
```

edited Jul 11 '11 at 17:48



Dolph

19.2k ● 10 ● 42 ● 72

answered Jan 8 '09 at 5:59



gimel

38.8k ● 8 ● 48 ● 86

1 It seems to me like it would be cleaner to express these configuration variables using the ConfigParser class. In fact, that's what I'm trying to do at the moment, and I can't seem to figure it out. – g33kz0r Dec 22 '10 at 10:06

5 ConfigParser is not related to sharing global variables between functions; You will still need a "config" instance and some sharing strategy. – gimel Dec 22 '10 at 13:21

This seems to fail if, instead of doing 'import config' you do 'from config import x'; the changes to x in one module importing config are not visible to other modules importing config. Does anyone understand why? – BlueBomber May 21 '13 at 13:48

@BlueBomber: because `x` would be a *local* identifier to the module that imported it. Think this way: `a = SomeClass(); x = a.someattrib; x = 10`. Would you expect `a`'s values to have changed? No, right? Same with modules. – MestreLion Feb 2 '15 at 16:00

1 @BlueBomber: think of modules as objects (which they are). Importing is just assigning them to a local identifier. `from config import x` means, in layman's terms, the same as `x = config.x`. But be aware that if `x` is *mutable* (like a list), appending items to it *will* reflect in `config`! – MestreLion Feb 2 '15 at 16:07

Python uses a simple heuristic to decide which scope it should load a variable from, between local and global. If a variable name appears on the left hand side of an assignment, but is not declared global, it is assumed to be local. If it does not appear on the left hand side of an assignment, it is assumed to be global.

```
>>> import dis
>>> def foo():
...     global bar
...     baz = 5
...     print bar
...     print baz
...     print quux
...
>>> dis.disassemble(foo.func_code)
 3          0 LOAD_CONST          1 (5)
          3 STORE_FAST           0 (baz)

 4          6 LOAD_GLOBAL          0 (bar)
          9 PRINT_ITEM
         10 PRINT_NEWLINE

 5         11 LOAD_FAST           0 (baz)
         14 PRINT_ITEM
         15 PRINT_NEWLINE

 6         16 LOAD_GLOBAL          1 (quux)
         19 PRINT_ITEM
         20 PRINT_NEWLINE
         21 LOAD_CONST           0 (None)
         24 RETURN_VALUE

>>>
```

See how baz, which appears on the left side of an assignment in `foo()`, is the only `LOAD_FAST` variable.

answered Jul 12 '11 at 12:35



SingleNegationEliminatio

n  
77.8k ● 9 ● 139 ● 203

3 The heuristic looks for *binding operations*. Assignment is one such operation, importing another. But the target of a `for` loop and the name after `as` in `with` and `except` statements also are bound to. –

If you want to refer to global variable in a function, you can use **global** keyword to declare which variables are global. You don't have to use it in all cases (as someone here incorrectly claims) - if the name referenced in an expression cannot be found in local scope or scopes in the functions in which this function is defined, it is looked up among global variables. However, if you assign to a new variable not declared as global in the function, it is implicitly declared as local, and it can overshadow any existing global variable with the same name.

Also, global variables are useful, contrary to some OOP zealots who claim otherwise - especially for smaller scripts, where OOP is overkill.

answered Jan 8 '09 at 9:03



J S

502 ● 4 ● 6

- 1 Thank you, this perfectly answered why sometimes global variables worked fine and other times (particularly when changing the value) didn't work at all – [Nathan Garabedian](#) Jan 16 '13 at 22:21

In addition to already existing answers and to make this more confusing:

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.

Source: [What are the rules for local and global variables in Python?](#).

edited Jul 20 '14 at 10:36



Peter Mortensen

9,455 ● 11 ● 67 ● 98

answered Jul 4 '14 at 10:23



Rauni

986 ● 1 ● 8 ● 27

Global variables are much more fun when you deal with parallel execution. Here is an example of using a global variable within multiprocessing. We can clearly see that each process works with its own copy of the variable:

```
import multiprocessing
import time
import os
import sys
import random
def worker(a):
    oldValue = get()
    set(random.randint(0, 100))
    sys.stderr.write(' '.join([str(os.getpid()), str(a), 'old:', str(oldValue),
'new:', str(get()), '\n']))

def get():
    global globalVariable
    return globalVariable

globalVariable = -1
def set(v):
    global globalVariable
    globalVariable = v

print get()
set(-2)
print get()

processPool = multiprocessing.Pool(5)
results = processPool.map(worker, range(15))
```

Output:

```
27094 0 old: -2 new: 2
27094 1 old: 2 new: 95
27094 2 old: 95 new: 20
27094 3 old: 20 new: 54
27098 4 old: -2 new: 80
27098 6 old: 80 new: 62
27095 5 old: -2 new: 100
27094 7 old: 54 new: 23
27098 8 old: 62 new: 67
27098 10 old: 67 new: 22
```

27098 11 old: 22 new: 85  
27095 9 old: 100 new: 32  
27094 12 old: 23 new: 65  
27098 13 old: 85 new: 60  
27095 14 old: 32 new: 71

edited Jul 20 '14 at 10:37



Peter Mortensen

9,455 ● 11 ● 67 ● 98

answered Oct 3 '13 at 5:41



Bohdan

4,414 ● 4 ● 34 ● 43

You're not actually storing the global in a local variable, just creating a local reference to the same object that your original global reference refers to. Remember that pretty much everything in Python is a name referring to an object, and nothing gets copied in usual operation.

If you didn't have to explicitly specify when an identifier was to refer to a predefined global, then you'd presumably have to explicitly specify when an identifier is a new local variable instead (for example, with something like the 'var' command seen in JavaScript). Since local variables are more common than global variables in any serious and non-trivial system, Python's system makes more sense in most cases.

You *could* have a language which attempted to guess, using a global variable if it existed or creating a local variable if it didn't. However, that would be very error-prone. For example, importing another module could inadvertently introduce a global variable by that name, changing the behaviour of your program.

edited May 30 '11 at 21:09



Peter Mortensen

9,455 ● 11 ● 67 ● 98

answered Jan 9 '09 at 11:56



Kylotan

15.2k ● 3 ● 33 ● 62

As it turns out the answer is always simple.

Here is a small sample module. It is a way to show it in a main definition:

```
def five(enterAnumber,sumation):  
    global helper  
    helper = enterAnumber + sumation  
  
def isTheNumber():  
    return helper
```

Here is a way to show it in a main definition:

```
import TestPy  
  
def main():  
    atest = TestPy  
    atest.five(5,8)  
    print(atest.isTheNumber())  
  
if __name__ == '__main__':  
    main()
```

This simple code works just like that, and it will execute. I hope it helps.

edited Jul 20 '14 at 10:35



Peter Mortensen

9,455 ● 11 ● 67 ● 98

answered Oct 13 '13 at 16:07



user2876408

71 ● 1 ● 1

thanks, i'm new to python, but know a bit of java. what you said worked for me. and writing global a<ENTER> within the class.. seems to make more sense to me than within a function writing 'global a'.. I notice you can't say global a=4 – [barlop](#) Oct 19 '13 at 18:55

1 This is probably the simplest yet very useful python trick for me. I name this module `global_vars`, and initialize the data in `init_global_vars`, that being called in the startup script. Then, I simply create accessor method for each defined global var. I hope I can upvote this multiple times! Thanks Peter! – [swdev](#) Sep 18 '14 at 2:32

What if there are many many global variables and I don't want to have to list them one-by-one after a global statement? – [user1021819](#) Apr 10 '15 at 10:36

What you are saying is to use the method like this:

```
globvar = 5  
  
def f():  
    var = globvar  
    print(var)  
  
f()** #prints 5
```

but the better way is to use the global variable like this:

```

globavar = 5
def f():
    global globvar
    print(globvar)
f() #prints 5

```

both give the same output.

edited Oct 2 '15 at 19:08



Evan Carslake

1,405 ● 6 ● 15 ● 38

answered Dec 4 '14 at 6:27



gxyd

81 ● 1 ● 7

You need to reference the global variable in every function you want to use.

As follows:

```

var = "test"

def printGlobalText():
    global var #We are telling to explicitly use the global version
    var = "global from printGlobalText fun."
    print "var from printGlobalText: " + var

def printLocalText():
    #We are NOT telling to explicitly use the global version, so we are creating a
    local variable
    var = "local version from printLocalText fun"
    print "var from printLocalText: " + var

printGlobalText()
printLocalText()
"""
Output Result:
var from printGlobalText: global from printGlobalText fun.
var from printLocalText: local version from printLocalText
[Finished in 0.1s]
"""

```

edited Feb 4 '15 at 18:45



Peter Mortensen

9,455 ● 11 ● 67 ● 98

answered Dec 20 '14 at 12:45



Mohamed El-Saka

71 ● 1 ● 3

- 2 'in every function you want to use' is subtly incorrect, should be closer to: 'in every function where you want to update' – [spazm](#) Mar 19 '15 at 23:43

**If I create a global variable in one function, how can I use that variable in another function?**

## Globals without modification

You can just use it, so long as you don't expect to modify it. For example, we can create a global with the following function:

```

def create_global_variable():
    global global_variable # must declare it to be a global first
    # modifications are thus reflected on the module's global scope
    global_variable = 'Foo'

def use_global_variable():
    return global_variable + '!!!'

```

Now writing the functions above does not actually run their code. We have to actually create the global variable first, so call the `create_global_variable` function:

```
>>> create_global_variable()
```

and now we can use the global variable:

```
>>> use_global_variable()
'Foo!!!'
```

## Modification of the global variable from inside a function

To point the global variable at a different object, you are required to use the `global` keyword again:

```

def change_global_variable():
    global global_variable
    global_variable = 'Bar'

```

Note that after writing this function, the code actually changing it has still not run:

```
>>> use_global_variable()
'Foo!!!'
```

So after calling the function:

```
>>> change_global_variable()
```

we can see that the global variable has been changed. The `global_variable` name now points to `'Bar'` :

```
>>> use_global_variable()
'Bar!!!'
```

Note that "global" in Python is not truly global - it's only global to the module level. So it is only available to functions written in the modules in which it is global. Functions remember the module in which they are written, so when they are exported into other modules, they still look in the module in which they were created to find global variables.

## Local variables with the same name

If you create a local variable with the same name, it will overshadow a global variable:

```
def use_local_with_same_name_as_global():
    # bad name for a local variable, though.
    global_variable = 'Baz'
    return global_variable + '!!!'

>>> use_local_with_same_name_as_global()
'Baz!!!'
```

But using that misnamed local variable does not change the global variable:

```
>>> use_global_variable()
'Bar!!!'
```

Note that you should avoid using the local variables with the same names as globals unless you know precisely what you are doing and have a very good reason to do so. I have not yet encountered such a reason.

edited Feb 3 at 13:36

answered Jan 1 at 19:55



Aaron Hall

38.1k ● 16 ● 115 ● 115

Try This :

```
def x1():
    global x
    x = 6

def x2():
    global x
    x = x+1
    print x

x = 5
x1()
x2()
```

edited Feb 4 '15 at 22:36

answered Feb 4 '15 at 19:19



Sagar Mehta

59 ● 3

following on and as an add on use a file to contain all global variables all declared locally and then 'import as'

```
file initval.py

Stocksin = 300
Prices = []

File getstocks.py

import initval as iv

Def getmystocks ():
    iv.StocksIn = getstockcount ()

Def getmycharts ():
    For ic in range (0,iv.StocksIn):
```

.....

answered Oct 24 '15 at 15:46



M Newton

30 ● 6

Writing to explicit elements of a global array does not apparently need the global declaration, though writing to it "wholesale" does have that requirement:

```
import numpy as np

hostValue = 3.14159
hostArray = np.array([2., 3.])
hostMatrix = np.array([[1.0, 0.0],[ 0.0, 1.0]])

def func1():
    global hostValue      # mandatory, else Local.
    hostValue = 2.0

def func2():
    global hostValue      # mandatory, else UnboundLocalError.
    hostValue += 1.0

def func3():
    global hostArray      # mandatory, else Local.
    hostArray = np.array([14., 15.])

def func4():
    # no need for globals
    hostArray[0] = 123.4

def func5():
    # no need for globals
    hostArray[1] += 1.0

def func6():
    # no need for globals
    hostMatrix[1][1] = 12.

def func7():
    # no need for globals
    hostMatrix[0][0] += 0.33

func1()
print "After func1(), hostValue = ", hostValue
func2()
print "After func2(), hostValue = ", hostValue
func3()
print "After func3(), hostArray = ", hostArray
func4()
print "After func4(), hostArray = ", hostArray
func5()
print "After func5(), hostArray = ", hostArray
func6()
print "After func6(), hostMatrix = \n", hostMatrix
func7()
print "After func7(), hostMatrix = \n", hostMatrix
```

edited Jan 8 at 22:35

answered Jan 7 at 20:41



Mike Lampton

11 • 4

I've found that when you try to assign a global variable to something else inside a function, it doesn't usually work. As in:

```
globalVar = 1

def functionname():
    global globalVar
    globalVar = 18
    print globalVar
```

What works for me to change it is when I assign it *outside* of the function, as in:

```
globalVar = 1
globalVar += 1

def functionname():
    global globalVar
    print globalVar
```

It works for me, but whatever.

answered Mar 6 at 19:04



AwesomeSauce

1 • 1

protected by Antti Haapala Mar 18 at 9:00

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 [reputation](#) on this site.

Would you like to answer one of these [unanswered questions](#) instead?