UnboundLocalError trying to use a variable (supposed to be global) that is (re)assigned (even after first use)

Asked 16 years ago Modified 18 days ago Viewed 116k times

313

When I try this code:

```
a, b, c = (1, 2, 3)

def test():
    print(a)
    print(b)
    print(c)
    c += 1
test()
```

I get an error from the `print(c)` line that says:

```
UnboundLocalError: local variable 'c' referenced before assignment
```

or in some older versions:

```
UnboundLocalError: 'c' not assigned
```

If I comment out `c += 1`, all the `print`s are successful.

I don't understand: why does printing `a` and `b` work, if `c` does not? How did `c += 1` cause `print(c)` to fail, even when it comes later in the code?

It seems like the assignment `c += 1` creates a **local** variable `c`, which takes precedence over the global `c`. But how can a variable "steal" scope before it exists? Why is `c` apparently local here?

See also How to use a global variable in a function? for questions that are simply about how to reassign a global variable from within a function, and Is it possible to modify a variable in python that is in an outer (enclosing), but not global, scope? for reassigning from an enclosing function (closure).

See Why isn't the 'global' keyword needed to access a global variable? for cases where OP *expected* an error but *didn't* get one, from simply accessing a global without the `global` keyword.

See How can a name be "unbound" in Python? What code can cause an `UnboundLocalError`? for cases where OP *expected* the variable to be local, but has a logical error that prevents assignment in every case.

See How can "NameError: free variable 'var' referenced before assignment in enclosing scope" occur in real code? for a related problem caused by the `del` keyword.

python **scope** **global-variables** **local-variables** **shadowing**

Share
Improve this question
Follow

edited Sep 21 at 0:54

Karl Knechtel
61.2k ● 14 ● 126 ● 183

asked Dec 16, 2008 at 3:06

tba
6,561 ● 8 ● 45 ● 63

15 Answers

Sorted by: Highest score (default)

324

Python treats variables in functions differently depending on whether you assign values to them from inside or outside the function. **If a variable is assigned within a function, it is treated by default as a local variable**. Therefore, when you uncomment the line, you are trying to reference the local variable `c` before any value has been assigned to it.

If you want the variable `c` to refer to the global `c = 3` assigned before the function, put

```
global c
```

as the first line of the function.

As for python 3, there is now

```
nonlocal c
```

that you can use to refer to the nearest enclosing function scope that has a `c` variable.

Share
Improve this answer
Follow

edited Dec 2 at 18:49

zyxue
8,760 ● 6 ● 57 ● 82

answered Dec 16, 2008 at 3:12

recursive
85.9k ● 36 ● 154 ● 245

8 Thanks. Quick question. Does this imply that Python decides the scope of each variable before running a program? Before running a function? – tba Dec 16, 2008 at 3:46

16 The variable scope decision is made by the compiler, which normally runs once when you first start the program. However it is worth keeping in mind that the compiler might also run later if you have "eval" or "exec" statements in your program. – Greg Hewgill Dec 16, 2008 at 3:48

3 Okay thank you. I guess "interpreted language" doesn't imply quite as much as I had thought. – tba Dec 16, 2008 at 3:53

1 Ah that 'nonlocal' keyword was exactly what I was looking for, it seemed Python was missing this. Presumably this 'cascades' through each enclosing scope that imports the variable using this keyword? – Brendan Nov 17, 2009 at 20:54

8 @brainfsck: it is easiest to understand if you make the distinction between "looking up" and "assigning" a variable. Lookup falls back to a higher scope if the name is not found in the current scope. Assignment is always done in the local scope (unless you use `global` or `nonlocal` to force global or nonlocal assignment) – Steven Sep 13, 2011 at 12:00

96

Python is a little weird in that it keeps everything in a dictionary for the various scopes. The original a,b,c are in the uppermost scope and so in that uppermost dictionary. The function has its own dictionary. When you reach the `print(a)` and `print(b)` statements, there's nothing by that name in the dictionary, so Python looks up the list and finds them in the global dictionary.

Now we get to `c+=1`, which is, of course, equivalent to `c=c+1`. When Python scans that line, it says "aha, there's a variable named c, I'll put it into my local scope dictionary." Then when it goes looking for a value for c for the c on the right hand side of the assignment, it finds its *local variable named c*, which has no value yet, and so throws the error.

The statement `global c` mentioned above simply tells the parser that it uses the `c` from the global scope and so doesn't need a new one.

The reason it says there's an issue on the line it does is because it is effectively looking for the names before it tries to generate code, and so in some sense doesn't think it's really doing that line yet. I'd argue that is a usability bug, but it's generally a good practice to just learn not to take a compiler's messages *too* seriously.

If it's any comfort, I spent probably a day digging and experimenting with this same issue before I found something Guido had written about the dictionaries that Explained Everything.

Update, see comments:

It doesn't scan the code twice, but it does scan the code in two phases, lexing and parsing.

Consider how the parse of this line of code works. The lexer reads the source text and breaks it into lexemes, the "smallest components" of the grammar. So when it hits the line

```
c+=1
```

it breaks it up into something like

```
SYMBOL(c) OPERATOR(+=) DIGIT(1)
```

The parser eventually wants to make this into a parse tree and execute it, but since it's an assignment, before it does, it looks for the name c in the local dictionary, doesn't see it, and inserts it in the dictionary, marking it as uninitialized. In a fully compiled language, it would just go into the symbol table and wait for the parse, but since it WON'T have the luxury of a second pass, the lexer does a little extra work to make life easier later on. Only, then it sees the OPERATOR, sees that the rules say "if you have an operator += the left hand side must have been initialized" and says "whoops!"

The point here is that it *hasn't really started the parse of the line yet*. This is all happening sort of preparatory to the actual parse, so the line counter hasn't advanced to the next line. Thus when it signals the error, it still thinks its on the previous line.

As I say, you could argue it's a usability bug, but its actually a fairly common thing. Some compilers are more honest about it and say "error on or around line XXX", but this one doesn't.

Share

Improve this answer

Follow

edited Dec 10, 2016 at 7:09

Jonathan Leffler

752k ● 145 ● 946 ● 1.3k

answered Dec 16, 2008 at 3:25

Charlie Martin

112k ● 26 ● 196 ● 264

7 Note on implementation details: In CPython, the local scope isn't usually handled as a `dict`, it's internally just an array (`locals()` will populate a `dict` to return, but changes to it don't create new `locals`). The parse phase is finding each assignment to a local and converting from name to position in that array, and using that position whenever the name is referenced. On entry to the function, non-argument locals are initialized to a placeholder, and `UnboundLocalError`s happen when a variable is read and its associated index still has the placeholder value. – ShadowRanger Mar 25, 2016 at 19:49

Python 3.x does not keep local variables in a dictionary. The result of `locals()` is computed on the fly. This is why the error is called `UnboundLocalError` in the first place: the local variable exists in the sense that it was **reserved ahead of time, when the function was compiled**, but hasn't been bound (assigned) yet. This works fundamentally differently from adding something to the global namespace (which **is** effectively a dictionary), so it wouldn't make sense to report the problem as a generic `NameError`. – Karl Knechtel Feb 7, 2023 at 1:08

Which exactly what I said, modulo the implementation detail of how the locals() are presented. – Charlie Martin Feb 13, 2023 at 22:18

"Then when it goes looking for a value for c for the c on the right hand side of the assignment, it finds its local variable named c, which has no value yet, and so throws the error" But if I instead change the line `c += 1` as `c = 100`, why is the error still present? Python-3.10 – NameError Oct 26, 2023 at 12:05

@NameErrorit seems to work as expected for me – Charlie Martin Nov 5, 2023 at 21:58

53

Taking a look at the disassembly may clarify what is happening:

```
>>> def f():
...     print a
...     print b
...     a = 1

>>> import dis
>>> dis.dis(f)

  2           0 LOAD_FAST                0 (a)
              3 PRINT_ITEM
              4 PRINT_NEWLINE

  3           5 LOAD_GLOBAL              0 (b)
              8 PRINT_ITEM
              9 PRINT_NEWLINE

  4          10 LOAD_CONST               1 (1)
             13 STORE_FAST               0 (a)
             16 LOAD_CONST               0 (None)
             19 RETURN_VALUE
```

As you can see, the bytecode for accessing a is `LOAD_FAST`, and for b, `LOAD_GLOBAL`. This is because the compiler has identified that a is assigned to within the function, and classified it as a local variable. The access mechanism for locals is fundamentally different for globals - they are statically assigned an offset in the frame's variables table, meaning lookup is a quick index, rather than the more expensive dict lookup as for globals. Because of this, Python is reading the `print a` line as "get the value of local variable 'a' held in slot 0, and print it", and when it detects that this variable is still uninitialised, raises an exception.

Share Improve this answer Follow

answered Dec 16, 2008 at 9:49

Brian

119k • 29 • 110 • 114

12

Python has rather interesting behavior when you try traditional global variable semantics. I don't remember the details, but you can read the value of a variable declared in 'global' scope just fine, but if you want to modify it, you have to use the `global` keyword. Try changing `test()` to this:

```
def test():
    global c
    print(a)
    print(b)
    print(c)    # (A)
    c+=1        # (B)
```

Also, the reason you are getting this error is because you can also declare a new variable inside that function with the same name as a 'global' one, and it would be completely separate. The interpreter thinks you are trying to make a new variable in this scope called `c` and modify it all in one operation, which isn't allowed in Python because this new `c` wasn't initialized.

Share

Improve this answer

Follow

edited Dec 10, 2016 at 6:53

Jonathan Leffler

752k • 145 • 946 • 1.3k

answered Dec 16, 2008 at 3:12

Mongoose

4,645 • 1 • 18 • 7

Thanks for your response, but I don't think it explains why the error is thrown at line (A), where I'm merely trying to print a variable. The program never gets to line (B) where it is trying to modify an un-initialized variable. – tba Dec 16, 2008 at 3:42

1 Python will read, parse and turn the whole function into internal bytecode before it starts running the program, so the fact that the "turn c to local variable" happens textually after the printing of the value doesn't, as it were, matter. – Vatine Dec 16, 2008 at 10:10

1 Python lets you access global variables in a local scope for reading, but not for writing. This answer has a nice work-around with explanation in comment below... +=1. – Mark Seagoe Jul 21, 2022 at 12:13

8

The best example that makes it clear is:

```
bar = 42
def foo():
    print bar
    if False:
        bar = 0
```

when calling `foo()` , this also **raises** `UnboundLocalError` although we will never reach to line `bar=0`, so logically local variable should never be created.

The mystery lies in "**Python is an Interpreted Language**" and the declaration of the function `foo` is interpreted as a single statement (i.e. a compound statement), it just interprets it dumbly and creates local and global scopes. So `bar` is recognized in local scope before execution.

For **more examples** like this Read this post: http://blog.amir.rachum.com/blog/2013/07/09/python-common-newbie-mistakes-part-2/

This post provides a Complete Description and Analyses of the Python Scoping of variables:

Share Improve this answer Follow

answered Jun 4, 2014 at 10:39

Sahil kalra

8,984 ● 4 ● 26 ● 29

1 Python is not any more "interpreted" than Java or C#, and in fact the decision to treat `bar` as a local variable in this code **requires** an up-front compilation step. – Karl Knechtel Sep 9, 2022 at 9:37

Summary

Python decides the scope of the variable **ahead of time**. **Unless explicitly overridden** using the `global` or `nonlocal` (in 3.x) keywords, variables will be recognized as **local** based on the **existence of any** operation that would *change the binding of* a name. That includes ordinary assignments, augmented assignments like `+=`, various less obvious forms of assignment (the `for` construct, nested functions and classes, `import` statements...) as well as *un*binding (using `del`). The actual execution of such code is irrelevant.

This is also explained in the documentation.

Discussion

Contrary to popular belief, **Python is not an "interpreted" language** in any meaningful sense. (Those are vanishingly rare now.) The reference implementation of Python compiles Python code in much the same way as Java or C#: it is translated into opcodes ("bytecode") for a *virtual machine*, which is then emulated. Other implementations must also compile the code - so that `SyntaxError`s can be detected

without actually running the code, and in order to implement the "compilation services" portion of the standard library.

How Python determines variable scope

During compilation (whether on the reference implementation or not), Python follows simple rules for decisions about variable scope in a function:

- If the function contains a `global` or `nonlocal` declaration for a name, that name is treated as referring to the global scope or the first enclosing scope that contains the name, respectively.
- Otherwise, if it contains any *syntax for changing the binding (either assignment or deletion) of the name, even if the code would not actually change the binding at runtime*, the name is **local**.
- Otherwise, it refers to either the first enclosing scope that contains the name, or the global scope otherwise.

Importantly, the scope is resolved **at compile time**. The generated bytecode will directly indicate where to look. In CPython 3.8 for example, there are separate opcodes `LOAD_CONST` (constants known at compile time), `LOAD_FAST` (locals), `LOAD_DEREF` (implement `nonlocal` lookup by looking in a closure, which is implemented as a tuple of "cell" objects), `LOAD_CLOSURE` (look for a local variable in the closure object that was created for a nested function), and `LOAD_GLOBAL` (look something up in either the global namespace or the builtin namespace).

There is no "default" value for these names. If they haven't been assigned before they're looked up, a `NameError` occurs. Specifically, for local lookups, `UnboundLocalError` occurs; this is a subtype of `NameError`.

Special (and not-special) cases

There are some important considerations here, keeping in mind that the syntax rule is implemented at compile time, with **no static analysis**:

- It **does not matter** if the global variable is a builtin function etc., rather than an explicitly created global:

  ```
  def x():
      int = int('1') # `int` is local!
  ```

  (Of course, it is a bad idea to shadow builtin names like this anyway, and `global` cannot help - just like using the same code outside of a function will still cause problems.)

- It **does not matter** if the code could never be reached:

```
y = 1
def x():
    return y # local!
    if False:
        y = 0
```

- It **does not matter** if the assignment would be optimized into an in-place modification (e.g. extending a list) - conceptually, the value is still assigned, and this is reflected in the bytecode in the reference implementation as a useless reassignment of the name to the same object:

```
y = []
def x():
    y += [1] # local, even though it would modify `y` in-place with `global`
```

- However, it **does** matter if we do an indexed/slice assignment instead. (This is transformed into a different opcode at compile time, which will in turn call `__setitem__`.)

```
y = [0]
def x():
    print(y) # global now! No error occurs.
    y[0] = 1
```

- There are other forms of assignment, e.g. `for` loops and `import`s:

```
import sys

y = 1
def x():
    return y # local!
    for y in []:
        pass

def z():
    print(sys.path) # `sys` is local!
    import sys
```

- Another common way to cause problems with `import` is trying to reuse the module name as a local variable, like so:

```
import random

def x():
    random = random.choice(['heads', 'tails'])
```

  Again, `import` is assignment, so there is a global variable `random`. But this global variable is **not special**; it can just as easily be shadowed by the local `random`.

- Deleting something also changes the name binding, e.g.:

```
y = 1
def x():
```

```
    return y # local!
    del y
```

The interested reader, using the reference implementation, is encouraged to inspect each of these examples using the `dis` standard library module.

Enclosing scopes and the `nonlocal` keyword (in 3.x)

The problem works the same way, *mutatis mutandis*, for both `global` and `nonlocal` keywords. (Python 2.x does not have `nonlocal`.) Either way, the keyword is necessary to assign to the variable from the outer scope, but is *not* necessary to *merely look it up*, nor to *mutate* the looked-up object. (Again: `+=` on a list mutates the list, but *then also reassigns* the name to the same list.)

Special note about globals and builtins

As seen above, Python does not treat any names as being "in builtin scope". Instead, the builtins are a fallback used by global-scope lookups. Assigning to these variables will only ever update the global scope, not the builtin scope. However, in the reference implementation, the builtin scope **can** be modified: it's represented by a variable in the global namespace named `__builtins__`, which holds a module object (the builtins are implemented in C, but made available as a standard library module called `builtins`, which is pre-imported and assigned to that global name). Curiously, unlike many other built-in objects, this module object can have its attributes modified and `del`d. (All of this is, to my understanding, supposed to be considered an unreliable implementation detail; but it has worked this way for quite some time now.)

Share

Improve this answer

Follow

edited Sep 21 at 0:52

answered Sep 9, 2022 at 10:52

Karl Knechtel

61.2k ● 14 ● 126 ● 183

1 Simply great. Don't bother with other answers. I still think the UnboundLocalError's message (e.g. `local variable 'c' referenced before assignment`) leaves much to be desired. I doubt anybody without intricate knowledge of Python's interpreter inner workings would find it clear – z33k Mar 11 at 22:57

@z33k from what I can tell, the usual problem in fixing the bug is in understanding *why* the variable is local. I don't really see how an error message could explain that, especially since the fact that it's local isn't in itself an error. – Karl Knechtel Mar 11 at 23:17

where is `c` referenced and *why* is it a problem? Are we talking about references here or is this some other meaning? Why a *generic NameError* is not sufficient? I know you mention this here but I still don't quite see it judging from what Python tells me and without interpreter knowledge. And I'm not a beginner Python user. – z33k Mar 12 at 0:10

1 It's "reference" in the ordinary English meaning of "try to do something with". It's a subtype because the local namespace works differently, and to try to give more information. Anyway, if

you have *concrete* suggestions for improving how Python does this (and I certainly am not claiming it's perfect), the right place for them is discuss.python.org. – Karl Knechtel Mar 12 at 1:03

4

Here are two links that may help

1: docs.python.org/3.1/faq/programming.html?highlight=nonlocal#why-am-i-getting-an-unboundlocalerror-when-the-variable-has-a-value

2: docs.python.org/3.1/faq/programming.html?highlight=nonlocal#how-do-i-write-a-function-with-output-parameters-call-by-reference

link one describes the error UnboundLocalError. Link two can help with with re-writing your test function. Based on link two, the original problem could be rewritten as:

```
>>> a, b, c = (1, 2, 3)
>>> print (a, b, c)
(1, 2, 3)
>>> def test (a, b, c):
...     print (a)
...     print (b)
...     print (c)
...     c += 1
...     return a, b, c
...
>>> a, b, c = test (a, b, c)
1
2
3
>>> print (a, b ,c)
(1, 2, 4)
```

Share

Improve this answer

Follow

edited Sep 13, 2011 at 4:00

Daniel X Moore

15k ● 17 ● 82 ● 94

answered Nov 16, 2009 at 22:12

mcdon

5,049 ● 3 ● 39 ● 36

3

This is not a direct answer to your question, but it is closely related, as it's another gotcha caused by the relationship between augmented assignment and function scopes.

In most cases, you tend to think of augmented assignment (`a += b`) as exactly equivalent to simple assignment (`a = a + b`). It is possible to get into some trouble with this though, in one corner case. Let me explain:

The way Python's simple assignment works means that if `a` is passed into a function (like `func(a)`; note that Python is always pass-by-reference), then `a = a + b` will not modify the `a` that is passed in. Instead, it will just modify the local pointer to `a`.

But if you use `a += b`, then it is sometimes implemented as:

```
a = a + b
```

or sometimes (if the method exists) as:

```
a.__iadd__(b)
```

In the first case (as long as `a` is not declared global), there are no side-effects outside local scope, as the assignment to `a` is just a pointer update.

In the second case, `a` will actually modify itself, so all references to `a` will point to the modified version. This is demonstrated by the following code:

```
def copy_on_write(a):
      a = a + a
def inplace_add(a):
      a += a
a = [1]
copy_on_write(a)
print a # [1]
inplace_add(a)
print a # [1, 1]
b = 1
copy_on_write(b)
print b # [1]
inplace_add(b)
print b # 1
```

So the trick is to avoid augmented assignment on function arguments (I try to only use it for local/loop variables). Use simple assignment, and you will be safe from ambiguous behaviour.

Share

Improve this answer

Follow

edited Dec 10, 2016 at 7:07

Jonathan Leffler

752k ● 145 ● 946 ● 1.3k

answered Jan 24, 2009 at 15:13

alsuren

178 ● 1 ● 2

2

The Python interpreter will read a function as a complete unit. I think of it as reading it in two passes, once to gather its closure (the local variables), then again to turn it into byte-code.

As I'm sure you were already aware, any name used on the left of a '=' is implicitly a local variable. More than once I've been caught out by changing a variable access to a += and it's suddenly a different variable.

I also wanted to point out it's not really anything to do with global scope specifically. You get the same behaviour with nested functions.

Share Improve this answer Follow

answered Dec 16, 2008 at 8:58

James Hopkin
14k ● 1 ● 45 ● 72

2

`c+=1` assigns `c`, python assumes assigned variables are local, but in this case it hasn't been declared locally.

Either use the `global` or `nonlocal` keywords.

`nonlocal` works only in python 3, so if you're using python 2 and don't want to make your variable global, you can use a mutable object:

```
my_variables = { # a mutable object
    'c': 3
}

def test():
    my_variables['c'] +=1

test()
```

Share Improve this answer Follow

answered Nov 3, 2016 at 18:52

Colegram
116 ● 3

0

This issue can also occur when the `del` keyword is utilized on the variable down the line, after initialization, typically in a loop or a conditional block.

Share Improve this answer Follow

answered Jun 15, 2022 at 15:21

izilotti
4,927 ● 1 ● 50 ● 56

0

In this case of `n = num` below, `n` is a local variable and `num` is a global variable:

```
num = 10

def test():
  # ↓ Local variable
    n = num
       # ↑ Global variable
```

```
    print(n)

test()
```

So, there is no error:

```
10
```

But in this case of `num = num` below, `num` on the both side are local variables and `num` on the right side is not defined yet:

```
num = 10

def test():
   # ↓ Local variable
    num = num
         # ↑ Local variable not defined yet
    print(num)

test()
```

So, there is the error below:

> UnboundLocalError: local variable 'num' referenced before assignment

In addition, even if removing `num = 10` as shown below:

```
# num = 10 # Removed

def test():
   # ↓ Local variable
    num = num
         # ↑ Local variable not defined yet
    print(num)

test()
```

There is the same error below:

> UnboundLocalError: local variable 'num' referenced before assignment

So to solve the error above, put `global num` before `num = num` as shown below:

```
num = 10

def test():
    global num # Here
```

```
    num = num
    print(num)

test()
```

Then, the error above is solved as shown below:

```
10
```

Or, define the local variable `num = 5` before `num = num` as shown below:

```
num = 10

def test():
    num = 5 # Here
    num = num
    print(num)

test()
```

Then, the error above is solved as shown below:

```
5
```

Share

Improve this answer

Follow

edited Jan 8, 2023 at 3:26

answered Jan 8, 2023 at 2:56

Super Kai - Kazuya Ito

1

0

To fix this error, we can use the nonlocal keyword, which allows you to modify a variable in the nearest enclosing scope that is not global.

```
class S:
    def fun1(self):
        left_depth = right_depth = 0
        def inner():
            nonlocal left_depth
            nonlocal right_depth
            a = 5
            left_depth += 1
            print(a, left_depth)
        inner()

s = S()
s.fun1()
```

Share Improve this answer Follow

answered Oct 4 at 5:54

Rizwan Ahmed
11 ● 2

-1

The best way to reach class variable is directly accesing by class name

```
class Employee:
    counter=0

    def __init__(self):
        Employee.counter+=1
```

Share Improve this answer Follow

answered Dec 8, 2015 at 10:09

Harun ERGUL
5,942 ● 5 ● 57 ● 65

2 This has nothing to do with the question that was asked. – Karl Knechtel Feb 7, 2023 at 1:11

-1

You can also get this message if you define a variable with the same name as a method.

For example:

```
def teams():
    ...

def some_other_method():
    teams = teams()
```

The solution, is to rename method `teams()` to something else like `get_teams()`.

Since it is only used locally, the Python message is rather misleading!

You end up with something like this to get around it:

```
def get_teams():
    ...

def some_other_method():
    teams = get_teams()
```

Share

Improve this answer

Follow

edited Feb 11, 2023 at 11:30

answered Apr 18, 2022 at 15:47

JGFMK
8,866 ● 4 ● 62 ● 99

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.