Scope

Table of contents

Outline . . . 1
Definition: scope . . . 2
Global variables . . . 2
Constants . . . 2
Local variables shadow globals . . . 3
The `global` keyword . . . 3
A common gotcha: `UnboundLocalError` . . . 4
Shadowing . . . 5
Mixing globals and locals . . . 6
A `global` name doesn't exist until the function runs . . . 6
Python's LEGB rule for resolving names . . . 7
Worked: an invocation counter . . . 7
Summary . . . 8

Lecture 7B. See csse1001[1] for course logistics and python-scope[2] for the full reference on global/local scope and the LEGB rule.

Outline

- Definition of scope
- Global variables and constants
- Local variables and shadowing
- The `global` keyword
- The LEGB name-resolution rule

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-scope.pdf

Definition: scope

The **scope** of a variable is the region of the code where the variable's name is recognized (i.e. the variable is accessible/visible).

```
>>> x = 2
>>> y = 3
>>> def foo():
...     return x
>>> def bar():
...     return foo()*y
>>> foo()
2
>>> bar()
6
```

`x` and `y` are **global** variables available to all functions. The global scope is the whole snippet. Figure 1 below draws global scope as a purple box.

Global variables

A **global variable** (or simply "global") is defined outside of any function (at the module level) and can be accessed by all functions in the module. Anything declared outside a function is globally accessible *provided there is no local variable with the same name*. A variable declared globally is said to have **global scope**.

Avoid global variables if possible: they can make code difficult to maintain.

Constants

By convention, **constants** are defined in `SNAKE_CASE_CAPS`:

```
PI = 3.14159
NUMBER_OF_DAYS_IN_WEEK = 7
GRID_SIZE = 5
```

Warning: unlike in other languages, the value of a constant is *not protected* and *can be changed at run-time*:

```
>>> PI = 3.14159
>>> PI = 3
>>> PI
3
```

Local variables shadow globals

```
>>> x = 2
>>> def foo():
...     x = 7   # local variable
...     return
>>> foo()
>>> x
2
```

Despite having the same name, the `x` of `foo()` is assumed **local** — its *scope* is `foo()`. Assigning to `x` inside the function creates a brand-new local variable rather than touching the global one, which is why `x` is still `2` after the call.

Figure 1 draws this. A rounded box over a span of code lines *is* a scope, and the box's colour names the kind of scope.

The `global` keyword

We can specify that `foo` should be using `x` as a global. It is good practice to declare your globals when you use one:

```
>>> x = 2
>>> def foo():
...     global x
...     x = 7
...     return
>>> foo()
>>> x
7
```

If a function only ever *creates* a local variable (no `global` declaration), that name doesn't exist outside the function — outside of `foo` the variable `x` does not exist; `x` is a **local variable**:

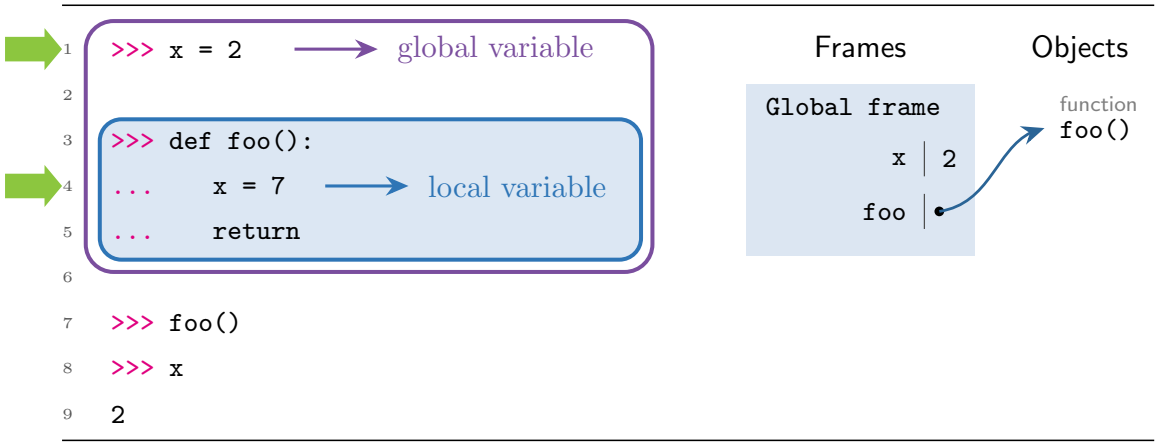

Figure 1: Scope boxes over a snippet. The purple outline is global scope and encloses the whole snippet; the blue outline with a pale blue fill is local scope and encloses only the body of the def, nested inside the purple box. Each box has a short arrow out to a colour-matched label. The green markers in the far-left margin flag lines 1 and 4 — the two assignments to x. Beside the code is the Python Tutor memory picture: the Global frame holds x bound to 2 and foo bound (blue arrow) to a function object, and there is no trace of foo's x at all, because the local x died when the call returned.

```
>>> def foo():
...     x = 2       # local variable
...     return x
>>> foo()
2
>>> x
NameError: name 'x' is not defined
```

A common gotcha: `UnboundLocalError`

```
>>> x = 2
>>> def foo():
...     x = x + 2
...     return
>>> foo()
UnboundLocalError: local variable 'x' referenced before assignment.
```

When `foo` creates `x`, it becomes local and thereby has no value at the time of assignment. As soon as `foo` assigns to `x` anywhere in its body, Python treats `x` as local *throughout the whole*

function — so the right-hand side `x + 2` tries to read a local `x` that doesn't have a value yet. Declaring `global x` first fixes this, since `x` then refers to the module-level variable throughout, and the effect accumulates across calls:

```
>>> x = 2
>>> def foo():
...     global x
...     x = x + 2
...     return
>>> foo()
>>> x
4
>>> foo()
>>> x
6
```

Shadowing

A *parameter* is the other way a local name comes into existence, and it shadows a global just as an assignment does. Figure 2 applies the same scope boxes to that case.

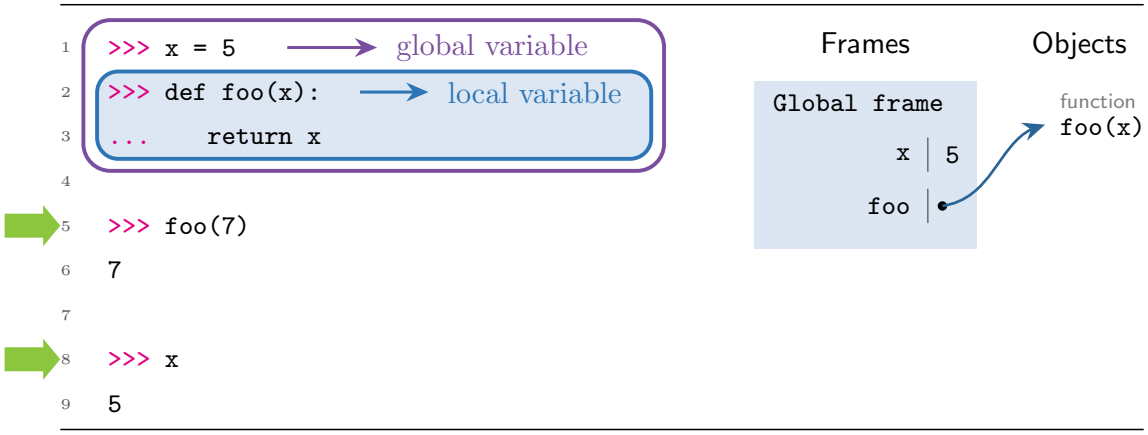

Figure 2: Shadowing by a parameter, in the same notation as Figure 1. The only structural change from that figure is where the blue local box starts: it opens on the def line itself, because here the local x is the parameter rather than an assignment in the body. foo(7) returns 7 while the global x is still 5 — the green margin markers flag exactly those two lines — and the Python Tutor panel shows why: the only x anywhere in the Global frame is the global one, bound to 5.

Despite having the same name, there are two `x`'s: one with a global scope and another with local. The parameter **shadows** the global variable, making it temporarily invisible inside `foo`.

A function can't declare a name as both a parameter and `global` at the same time:

```
>>> x = 5
>>> def foo(x):
...     global x
...     return
SyntaxError: name 'x' is parameter and global
```

Mixing globals and locals

```
>>> x = 5
>>> def foo(y):
...     return x*y
>>> foo(7)
35
>>> foo(x)
25
```

Globals and locals can be used in mixed computation.

A `global` name doesn't exist until the function runs

Declaring `global x` inside a function does not, by itself, create the module-level `x`. The binding only happens when the assignment actually executes — i.e. when the function is *invoked*:

```
>>> def foo():
...     global x
...     x = 2
...     return

>>> x
NameError:

>>> foo()
>>> x
2
```

Python's LEGB rule for resolving names

When Python looks up a name it searches four namespaces in a fixed order, dropping to the next only when the name is not found — and raising an error if it runs out. Figure 3 shows the cascade.

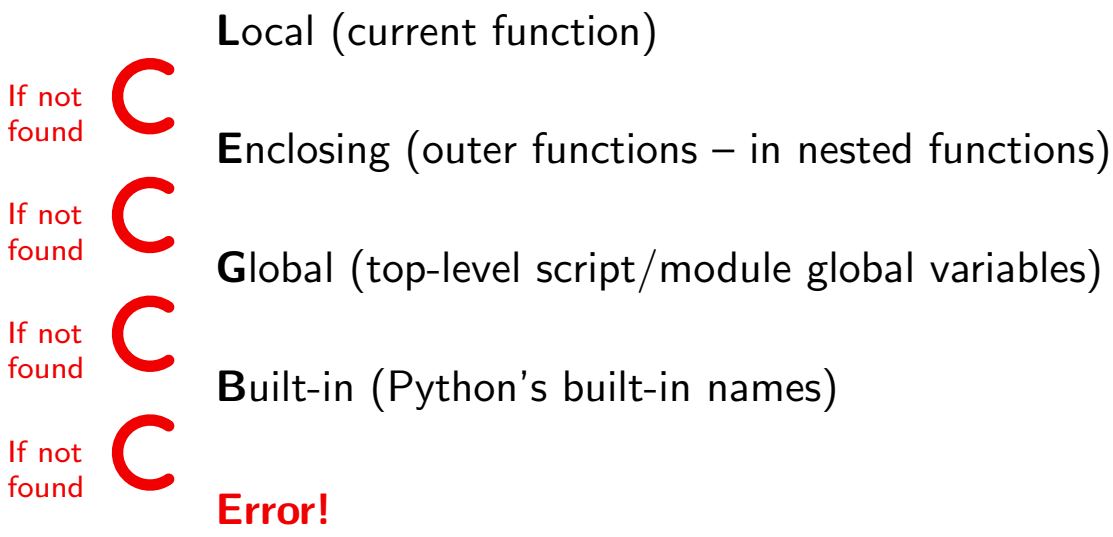

Figure 3: Python's LEGB rule. Read it top to bottom: each red hook is one 'if not found' step down to the next enclosing namespace, and the cascade bottoms out in a red Error! rather than in a default value. The bolded initial letters spell the mnemonic, and red marks the failure path.

1. **L**ocal — the current function.
2. **E**nclosing — outer functions, for nested functions.
3. **G**lobal — top-level script/module global variables.
4. **B**uilt-in — Python's built-in names.

If the name isn't found at any level: **Error!**

Worked: an invocation counter

Write a function `foo` that returns the number of times the function has been called/invoked. For example:

```
>>> foo()
1
```

```
>>> foo()
2
>>> foo()
3
```

A solution with the `global` keyword:

```
count = 0

def foo() -> int:
    """ Return the number of times foo() has been called. """
    global count
    count += 1
    return count
```

The state has to live *outside* the function: a `count = 0` inside `foo` would be re-created local and reset to `0` on every call, and `count += 1` without the `global` declaration is exactly the `UnboundLocalError` case above. `print(count)` in place of `return count` would produce the identical REPL display — the REPL doesn't echo a `None` return — while giving a function that returns nothing, so it would not satisfy the task as stated.

Summary

The places in your program that can access a **name** is called the **scope** of that name. The **global scope** is for things like **constants**, whereas names **declared in functions** get **locally scoped** to that function.

Next: 2025-09-09-object-oriented-programming[3] (Lecture 7C).

[3] courses/csse1001/2025-09-09-object-oriented-programming.pdf