Scope

Table of contents

See csse1001[1] for course logistics — this note covers Lecture 7B's technical content. See python-scope[2] for the full reference on global/local scope and the LEGB rule.

Today's 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.)

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` inside `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.

The `global` keyword

We can specify that a function should use a name as a global (rather than create a local shadow). It's 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:

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

A common gotcha: `UnboundLocalError`

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

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

Shadowing

```
>>> x = 5
>>> def foo(x):
...     return x
>>> foo(7)
7
>>> x
5
```

Despite having the same name, there are two `x`'s: one with global scope, and another local to `foo` (its parameter). 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 freely used together in the same expression.

Python's LEGB rule for resolving names

When Python looks up a name, it searches (in order):

1. **L**ocal — the current function.
2. **E**nclosing — outer function(s), 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, Python raises an error.

Exercise: an invocation counter

Write a function `foo` that returns the number of times it has been called:

```
>>> foo()
1
>>> foo()
2
>>> foo()
3
```

```
count = 0
def foo():
    """ prints the number of times foo() has been called """
    global count
    count += 1
    print(count)
```

> The task asks for a function that *returns* the count, but the source's implementation only `print`s it (returning `None` implicitly) — this happens to produce identical output in the REPL shown above, since a `None` return isn't echoed, but the two aren't the same thing. Reproduced here exactly as given, with the discrepancy flagged rather than silently "fixed" into a `return count`.

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