Python Scope

Table of contents

Global variables 1
Local variables and shadowing 2
The `global` keyword 2

The **scope** of a name is the region of code where that name is recognized (i.e. accessible/visible). Python resolves names using the **LEGB** rule, searching in order:

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

If a name isn't found at any level, Python raises a `NameError`.

Introduced in 2025-09-09-scope[1] (Lecture 7B), which also draws the LEGB cascade and the course's global/local scope boxes.

Global variables

Defined outside any function (module level); accessible everywhere in the module *provided* no local variable shadows the name. Avoid globals where possible — they make code harder to maintain.

Constants are conventionally named in `SNAKE_CASE_CAPS` (`PI`, `GRID_SIZE`, `NUMBER_OF_DAYS_IN_WEEK`), but unlike in other languages Python does **not** protect their value: a constant can simply be reassigned at run-time.

[1] courses/csse1001/2025-09-09-scope.pdf

Local variables and shadowing

Assigning to a name anywhere inside a function body makes that name **local to the entire function** — even before the assignment line executes. So does taking it as a **parameter**. This causes two common gotchas:

- **Shadowing**: a local variable (or parameter) with the same name as a global temporarily hides the global inside that function. The global itself is untouched — it still holds its old value after the call returns.
- `UnboundLocalError`: if a function reads a name before assigning to it, and also assigns to that same name later in its body, Python treats it as local throughout — so the read fails, because the local doesn't have a value yet.

The `global` keyword

Declares that a name inside a function refers to the module-level variable, rather than creating a local shadow:

```
def foo():
    global x
    x = x + 1
```

- It is good practice to declare a global explicitly wherever a function writes to one.
- A name cannot be declared as both a function parameter and `global` in the same function — that is a `SyntaxError` ("name 'x' is parameter and global").
- Declaring `global x` does not by itself create `x` at module level. The name only comes into existence when the assignment actually runs, i.e. when the function is **invoked** — until then, reading `x` at the prompt is still a `NameError`.
- Globals and locals mix freely in one expression: `def foo(y): return x*y` reads `x` globally and `y` locally.