Python Scope

Table of contents

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

The **scope** of a name is the region of code where that name is recognized. 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`.

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`, but Python does not actually protect their value from being reassigned.

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. 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.

- `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
```

A name cannot be declared as both a function parameter and `global` in the same function — this is a `SyntaxError`.