Tasks – Scopes in Python

Practice **scope**: where a name is visible. In Python, variables defined **outside** a function are **global**; variables (and parameters) defined **inside** a function are **local**. Reading a global name inside a function works; **assigning** to a name inside a function creates a **local** variable unless you use `global`. Create each file, run it, and check the output.

Run scripts with: `python3 script_name.py`

Part 1 – Global and local

Task 1.1 – Read a global variable inside a function (`scope_read_global.py`)

- Create `scope_read_global.py`.
- At the **top level** (no indentation), assign `name = "Global"`.
- Define a function `show_name()` that has **no parameters** and **no assignment** to `name`. Inside the function, just **print** `name`. When Python looks up `name`, it finds the **global** variable.
- Call `show_name()`, then print `name` at the top level. You should see "Global" twice. **Reading** a global name inside a function is allowed.

Expected output:

```
Global
Global
```

Task 1.2 – Assign inside a function creates a local variable (`scope_local_shadows.py`)

- Create `scope_local_shadows.py`.

- At the top level, assign `x = 10` and print it: `print("before:", x)`.
- Define a function `set_x()` that **assigns** `x = 99` (and optionally prints `x` inside the function). In Python, this assignment makes `x` a **local** variable for that function; it does **not** change the global `x`.
- Call `set_x()`, then at the top level print `x` again: `print("after:", x)`. You should see 99 inside the function but **10** after the call — the global `x` was never changed. The local `x` **shadows** the global only inside the function.

Expected output (example):

```
before: 10
99
after: 10
```

Task 1.3 – Parameter is local (`scope_param_local.py`)

- Create `scope_param_local.py`.
- At the top level, assign `value = 100`.
- Define a function `double(value)` that takes a parameter named `value`, returns `value * 2`. The parameter **value** is **local** to the function; it has nothing to do with the global `value` except the name.
- Call and print `print(double(5))` (you should get 10). Then print the global: `print(value)`. It should still be 100. Parameters are local variables.

Expected output:

```
10
100
```

Part 2 – The global keyword

Task 2.1 – Use global to assign to the global variable (`scope_global_keyword.py`)

- Create `scope_global_keyword.py`.
- At the top level, assign `counter = 0` and print it.
- Define a function `increment()` that has **no parameters**. Inside the function, on the **first line**, write `global counter`. Then do `counter = counter + 1`. The `global` statement means "use the **global** name `counter`, don't create a local one." So the assignment changes the global variable.
- Call `increment()` three times, then at the top level print `counter`. You should see 3. Without `global`, the assignment would create a local `counter` and you'd get an error (or wrong behavior) when reading `counter` before assigning.

Expected output (example):

```
0
3
```

Task 2.2 – Read global, then use global to update (`scope_global_read_write.py`)

- Create `scope_global_read_write.py`.
- At the top level, assign `total = 10`.
- Define a function `add_to_total(amount)` that **adds** `amount` to `total` and prints the new total. Use `global total` so that the assignment `total = total + amount` (or `total += amount`) updates the global `total`.
- Call `add_to_total(5)`, then `add_to_total(3)`. Print `total` at the top level after the calls. You should see 15, then 18, and finally 18. The global is being updated from inside the function.

Expected output (example):

```
15
18
18
```

Part 3 – Each function has its own locals

Task 3.1 – Same variable name in two functions (`scope_two_functions.py`)

- Create `scope_two_functions.py`.
- Define a function `foo()` that assigns `x = 1` and prints `x`. Define another function `bar()` that assigns `x = 2` and prints `x`. Each function has its **own** local `x`; they are not the same variable.
- Call `foo()` then `bar()` then `foo()` again. You should see 1, 2, 1. The two `x` names live in different scopes.

Expected output:

```
1
2
1
```

Task 3.2 – Variable from one function not visible in another (`scope_not_visible.py`)

- Create `scope_not_visible.py`.
- Define a function `set_secret()` that assigns `secret = 42` and prints it. Define another function `get_secret()` that tries to **print** `secret` (and does nothing else). There is no global `secret` and no parameter; `secret` in `get_secret()` would be a name that was never assigned in that scope.
- Call `set_secret()` (you see 42). Then call `get_secret()` — you should get **NameError: name 'secret' is not defined**. Variables inside one function are **not** visible inside another. Each function's locals are private.

Expected output (then error):

```
42
NameError: name 'secret' is not defined
```

Part 4 – Loop and block scope

Task 4.1 – for-loop variable exists after the loop (`scope_loop_var.py`)

- Create `scope_loop_var.py` .
- At the top level, use a for loop: `for i in range(3):` and in the body print `i` . After the loop (same indentation as `for`), print `"after loop:", i` . In Python, the variable `i` is **not** limited to the loop body; it still exists after the loop and holds the last value (2). There is no “block scope” for loops like in some other languages.
- Run the script and observe that `i` is still 2 after the loop.

Expected output:

```
0
1
2
after loop: 2
```

Task 4.2 – Global and local with same name: print order (`scope_before_after.py`)

- Create `scope_before_after.py` .
- At the top level, assign `msg = "global"` and print it: `print(msg)` .
- Define a function `mix()` that first **prints** `msg` (so it reads the global), then **assigns** `msg = "local"` and prints `msg` again. The assignment creates a local `msg` ; from that point on inside the function, `msg` refers to the local one.
- Call `mix()` , then at the top level print `msg` again. You should see: global, then inside the function "global" then "local", then after the call "global". The global is only visible until you assign to the name; then the local shadows it for the rest of the function.

Expected output:

```
global
global
local
global
```

Done

You've seen: **global** (top-level) and **local** (inside a function) scope, **reading** a global inside a function, **assigning** inside a function creating a **local** (shadowing the global), **parameters** as locals, the **global** keyword to assign to a global from inside a function, **separate locals** per function, that one function's variables are **not** visible in another, and that **for-loop** variables in Python are not block-scoped (they exist after the loop).