Why do we use global keyin python

**ChatGPT said:**

ChatGPT

The global keyword in Python is used to modify or update a variable that is defined outside the current function (i.e., in the global scope). Normally, if you assign a value to a variable inside a function, Python treats it as a local variable, meaning it only exists within the function's scope. However, if you want to modify a global variable inside a function, you need to declare that variable as global to tell Python that you are referring to the global version of the variable rather than creating a new local one.

**Example:**

python

Copy code

x = 10 # Global variable

def modify\_global():

global x # Declare x as global

x = 20 # Modify the global variable

modify\_global()

print(x) # Output: 20

Without the global keyword, Python would treat x as a local variable within the function, and the global x wouldn't be affected.

4o

**You said:**

Ok , if x is defined outside the function it's global variable and if we want to change the value of x inside the function without keyword global will it treat as locally ??

**ChatGPT said:**

ChatGPT

Yes, if you try to assign a value to x inside the function **without** using the global keyword, Python will treat it as a **local variable**, meaning it will only exist within the function's scope. The global variable x defined outside the function will remain unchanged.

**Example:**

python

Copy code

x = 10 # Global variable

def modify\_without\_global():

x = 20 # This is treated as a new local variable inside the function

print("Inside function:", x) # Prints 20 (local x)

modify\_without\_global()

print("Outside function:", x) # Prints 10 (global x remains unchanged)

In this example:

* Inside the function, x = 20 creates a new local variable x.
* The global x outside the function is **not modified** and still holds the value 10.

4o

**You said:**

But why it is considered like that

**ChatGPT said:**

ChatGPT

In Python, variables are treated as **local by default** inside functions because of how **scope** and **namespaces** work. The main idea is to avoid accidental modifications to variables outside the function, allowing functions to operate independently on local data without unintentionally affecting global variables.

Here’s why it works this way:

**1. Local and Global Namespaces:**

* **Local Scope**: A function has its own namespace (or scope), where it can define and manipulate variables. These variables are considered **local** to the function and disappear once the function finishes executing.
* **Global Scope**: Variables defined outside any function or class exist in the **global namespace** and can be accessed from anywhere in the program unless shadowed by a local variable.

**2. Avoiding Side Effects:**

Without using the global keyword, Python assumes you want to work with a **new variable** local to the function. This prevents unintended side effects where changes made inside a function might accidentally modify a global variable, potentially causing bugs that are hard to track down.

**3. Explicit Intent:**

By requiring the global keyword, Python forces you to explicitly declare when you want to modify a global variable. This makes it clear to anyone reading the code that you're deliberately changing a global variable, rather than accidentally creating a local one.

**Example:**

python

Copy code

x = 10 # Global variable

def modify\_without\_global():

x = 20 # Local variable (independent of global x)

print("Inside function:", x) # 20

modify\_without\_global()

print("Outside function:", x) # 10

In this case:

* The function creates a **local** x, which doesn't affect the **global** x. This makes the function self-contained, avoiding unintentional modifications to the global variable.

**Why It's Useful:**

* If functions could modify global variables without explicit declaration, it could lead to unpredictable behavior in large programs. By keeping local and global variables separate, Python helps maintain **modular** and **reliable** code.

To modify the global variable intentionally, you need the global keyword to tell Python to use the global variable rather than creating a new local one. This behavior makes code more predictable and easier to debug.

4o

**You said:**

Ok explain about nonlocal keyword

**ChatGPT said:**

ChatGPT

The nonlocal keyword in Python is used to modify variables in a **nested (enclosing) function's scope**. It allows you to work with variables that are not local to the current function but also not global—instead, they are in an **outer, non-global function**.

This is especially useful when you're dealing with **nested functions** (functions defined within other functions) and you want the inner function to modify a variable from the outer (enclosing) function.

**Scope Levels in Python:**

* **Global Scope**: Variables defined at the top level of a script or module.
* **Enclosing Scope**: Variables defined in the outer function of a nested function.
* **Local Scope**: Variables defined inside the current function.
* **Built-in Scope**: Predefined Python names like print, len, etc.

The nonlocal keyword allows you to modify variables in the **enclosing scope** (the function that surrounds the current one), but it **does not affect global variables**.

**Example:**

python

Copy code

def outer\_function():

x = 10 # This is in the enclosing scope

def inner\_function():

nonlocal x # Refers to the x in the outer\_function

x = 20 # Modifies the x in the enclosing scope (outer\_function)

print("Inner function:", x) # 20

inner\_function()

print("Outer function:", x) # 20 (the x in outer\_function is modified)

outer\_function()

**Key Points:**

1. **Without nonlocal**: If you don’t use nonlocal, Python would treat the variable x in the inner\_function as **local** to the inner function and wouldn't modify the x in the outer\_function.
2. **With nonlocal**: By using nonlocal, you're telling Python to look for the variable x in the **enclosing (outer)** function and modify it, rather than creating a new local variable in the inner function.

**Use Case:**

nonlocal is useful when:

* You have a function inside another function, and you want the inner function to update a variable from the outer function.
* You want to preserve state between multiple calls to an inner function by modifying a variable in the outer function.

**Example Without nonlocal:**

python

Copy code

def outer\_function():

x = 10

def inner\_function():

x = 20 # This creates a new local variable x (does not modify the outer x)

print("Inner function:", x) # 20

inner\_function()

print("Outer function:", x) # 10 (the outer x remains unchanged)

outer\_function()

In this example, x in inner\_function is a **new local variable**, so it doesn't affect the x in outer\_function.

**When to Use nonlocal:**

* When you want to modify variables from an outer function inside a nested function.
* It provides a middle ground between **local** and **global** variable management in nested scopes.

nonlocal is most commonly used when you're working with **closures** or need to maintain state across function calls in a clean, controlled way.

4o

**You said:**

Ok now I have one more dobdt Variables declared globally outside of the function and tries to modify insideof the function without global keyword will be considered as local variable right??

**ChatGPT said:**

ChatGPT

Yes, you're right! When you declare a variable globally outside a function, and you try to modify it inside the function **without using the global keyword**, Python treats the variable as **local** to that function. It creates a new local variable with the same name, and the global variable remains unchanged.

This behavior occurs because Python assumes that any variable assigned a value inside a function is local to that function unless explicitly declared otherwise using the global keyword.

**Example:**

python

Copy code

x = 10 # Global variable

def modify\_variable():

x = 20 # Python treats this as a new local variable

print("Inside function:", x) # Prints 20 (local x)

modify\_variable()

print("Outside function:", x) # Prints 10 (global x remains unchanged)

**Explanation:**

* The assignment x = 20 creates a new **local** variable x inside the function. It **does not** modify the global x defined outside the function.
* When the function prints x, it refers to the local x, which is 20.
* After the function finishes, the global x remains unchanged as 10.

**Important Note:**

If you try to **access** a global variable inside a function without modifying it (i.e., without assigning a new value to it), Python will use the global version.

**Example of accessing without modifying:**

python

Copy code

x = 10 # Global variable

def access\_global():

print("Inside function:", x) # This will access the global x without modifying it

access\_global()

print("Outside function:", x) # Still 10

**Summary:**

* **Without the global keyword**, assigning a value to a variable inside a function creates a **new local variable**.
* **With the global keyword**, you can modify the global variable directly from within the function.

4o

**You said:**

Okay in the above mentioned case by me, will it takes the same memory allocation??

**ChatGPT said:**

ChatGPT

In the case you mentioned, where a global variable is declared outside of the function and then a local variable with the same name is created inside the function, **they do not share the same memory allocation**. The local and global variables are stored in different memory locations because they exist in different **scopes**.

**Key Points:**

1. **Global Variable**:
   * The global variable is stored in the **global namespace**.
   * It has its own memory allocation and exists throughout the program's execution.
2. **Local Variable**:
   * When you assign a new value to a variable inside a function (without using the global keyword), Python creates a **local variable** in the function's **local namespace**.
   * This local variable is stored in a separate memory location and only exists while the function is being executed.

**Example for Clarification:**

python

Copy code

x = 10 # Global variable (stored in global memory)

def modify\_variable():

x = 20 # Local variable (stored in local memory)

print("Inside function:", x) # 20 (refers to the local variable)

modify\_variable()

print("Outside function:", x) # 10 (refers to the global variable)

**Memory Allocation:**

* The global x = 10 is stored in the **global memory space** (outside the function).
* The local x = 20 inside the function is stored in a **separate memory space** in the function's local environment.

**Why Different Memory Allocations?**

Python treats variables in different scopes (global vs. local) separately. Even though both variables share the same name (x), they are in **different namespaces**:

* **Global namespace**: Where global variables reside.
* **Local namespace**: Where variables inside a function reside.

Since these namespaces are distinct, Python assigns different memory locations for variables with the same name but different scopes.

**What Happens with Memory?**

* When you create the local x inside the function, it gets its own memory allocation.
* Once the function finishes executing, the local x is destroyed, and its memory is freed.
* The global x remains unchanged and keeps its memory allocation until the program terminates.

**In Short:**

* **Global and local variables with the same name do not share memory**.
* They are stored in **separate memory spaces**, and each is handled independently, which avoids conflicts between the two.

4o