Python Variable Scope

Introduction

When we create a variable in Python, that variable is not automatically available everywhere in our program.

Where a variable can be **accessed or used** is called its **scope**.

For example:

```
name = "Hafsa"

def greet():
    print(name)

greet()
```

The variable `name` is created outside the function, so the function can access it.

But consider:

```
def greet():
    message = "Hello"
    print(message)

greet()

print(message)
```

This produces an error because `message` was created inside the function.

The key idea is:

> **Scope determines where a variable exists and where Python can access it.**

Understanding scope becomes especially important when working with functions, loops, nested functions, and larger Python projects.

---

1. What Is Scope?

Think of scope as the **area where a variable is visible**.

For example:

```
name = "Hafsa"

print(name)
```

`name` can be accessed because it was created in the current/global scope.

But:

```
def greet():
    message = "Hello"
    print(message)
```

`message` exists inside the function's scope.

We can think of it like this:

```
Program
│
├── Global Scope
│
└── Function Scope
    └── Local Variables
```

A variable's location determines where it can normally be used.

---

2. Local Scope

A variable created inside a function usually has **local scope**.

Example:

```
def greet():
    message = "Hello"
    print(message)

greet()
```

`message` is a local variable.

It can be used inside `greet()` :

```
def greet():
    message = "Hello"
    print(message)
```

But not outside:

```
def greet():
    message = "Hello"

greet()
```

```
print(message)
```

Python will raise:

```
NameError
```

because `message` does not exist in the scope where `print(message)` is executed.

---

3. Global Scope

A variable created outside functions is usually in the **global scope**.

Example:

```
name = "Hafsa"

def greet():
    print(name)

greet()
```

Here:

```
name
```

is a global variable.

It can be accessed from the function because Python can look outside the function when searching for a variable.

---

4. Local vs Global Scope

Consider:

```
name = "Hafsa"

def greet():
    message = "Hello"

    print(name)
    print(message)

greet()
```

There are two different scopes.

Global scope

```
name = "Hafsa"
```

Local scope

```
message = "Hello"
```

Inside `greet()`, Python can access both:

```
name → global
message → local
```

But outside the function:

```
print(name)
```

works.

While:

```
print(message)
```

does not.

---

5. Why Does Scope Exist?

Scope helps prevent variables from interfering with each other.

Imagine two functions:

```
def calculate_student():
    marks = 90
    print(marks)

def calculate_employee():
    marks = 50000
    print(marks)
```

Both functions use a variable called `marks`.

But they don't interfere with each other.

When:

```
calculate_student()
```

`marks` means:

```
90
```

Inside:

```
calculate_employee()
```

`marks` means:

```
50000
```

Local scope keeps these variables separate.

---

6. Same Variable Name in Different Functions

Two functions can have local variables with the same name.

```
def function_one():
    value = 10
    print(value)

def function_two():
    value = 20
    print(value)

function_one()
function_two()
```

Output:

```
10
20
```

The two `value` variables are separate local variables.

Changing one does not change the other.

---

7. Local Variables Are Created When the Function Runs

Consider:

```
def greet():
    message = "Hello"
    print(message)
```

Defining the function does not execute the function body.

The local variable is created when the function is called:

```
greet()
```

During the function execution:

```
message exists
```

After the function finishes, that local variable is no longer available through the function's local scope.

---

8. Scope and Function Parameters

Function parameters are also local to the function.

Example:

```
def greet(name):
    print(name)
```

Here:

```
name
```

is a local variable/parameter inside the function.

This works:

```
greet("Hafsa")
```

But this does not:

```
print(name)
```

because `name` is not defined in the global scope.

---

9. Scope With `return`

A function can send a value outside its local scope using `return`.

Example:

```
def calculate_total():
    total = 500
    return total

result = calculate_total()

print(result)
```

Here:

```
total
```

is local.

But its value is returned and stored in:

```
result
```

which exists in the outer scope.

This is one of the main ways functions communicate results without exposing their local variables.

---

10. A Local Variable Is Not Automatically Global

Consider:

```
def create_user():
    username = "Hafsa"

create_user()

print(username)
```

This does not work.

Calling the function does not make `username` a global variable.

`username` remains local to `create_user()`.

---

11. Global Variables Inside Functions

A function can read a global variable.

Example:

```
website = "haas.dev"

def show_website():
    print(website)

show_website()
```

Output:

```
haas.dev
```

Python searches for `website` inside the function first.

If it doesn't find it there, Python looks outside the function.

---

12. Local Variable Takes Priority

What happens if a local and global variable have the same name?

```
name = "Global Hafsa"

def show_name():
    name = "Local Hafsa"
    print(name)

show_name()

print(name)
```

Output:

```
Local Hafsa
Global Hafsa
```

Inside the function, Python uses the local variable.

Outside the function, Python uses the global variable.

So:

```
Local variable → takes priority inside its scope
Global variable → used outside when no local variable shadows it
```

---

13. What Is Variable Shadowing?

When a local variable has the same name as a variable from an outer scope, the local variable **shadows** the outer one.

Example:

```
language = "Python"

def show_language():
    language = "JavaScript"
    print(language)

show_language()
```

Output:

```
JavaScript
```

The local `language` temporarily hides the global `language` inside the function.

The global variable is still there:

```
print(language)
```

Output:

```
Python
```

---

14. LEGB Rule

Python uses a useful rule called **LEGB** to find variables.

LEGB stands for:

```
L → Local
E → Enclosing
G → Global
B → Built-in
```

Python generally searches in this order:

```
Local
   ↓
Enclosing
   ↓
Global
   ↓
Built-in
```

This is one of the most important ideas about Python scope.

---

15. L: Local

Python first checks the current function.

```
name = "Global"

def greet():
    name = "Local"
    print(name)

greet()
```

Python finds `name` locally.

Output:

```
Local
```

---

16. E: Enclosing

Enclosing scope occurs when we have a function inside another function.

Example:

```
def outer():
    message = "Hello"

    def inner():
        print(message)

    inner()

outer()
```

`message` is not local to `inner()`.

But it exists in the surrounding `outer()` function.

Therefore it is in the **enclosing scope** of `inner()`.

---

17. Nested Functions and Scope

Consider:

```
def outer():
    name = "Hafsa"

    def inner():
        print(name)

    inner()

outer()
```

When `inner()` looks for `name`:

1. It checks its own local scope.
2. It doesn't find `name`.
3. It checks the enclosing `outer()` scope.
4. It finds `name`.

Output:

```
Hafsa
```

---

18. G: Global

If Python doesn't find a variable locally or in an enclosing function, it checks the global scope.

Example:

```
website = "haas.dev"

def show():
    print(website)

show()
```

Python finds `website` globally.

---

19. B: Built-in

If Python doesn't find a variable in local, enclosing, or global scope, it can check Python's built-in names.

For example:

```
def calculate():
    numbers = [10, 20, 30]
    print(len(numbers))
```

`len()` is not defined inside `calculate()`.

It is a Python built-in function.

So Python finds it in the built-in scope.

---

20. LEGB Example

Consider:

```
name = "Global"

def outer():
    name = "Enclosing"

    def inner():
        name = "Local"
```

```
        print(name)

    inner()

outer()
```

Output:

```
Local
```

Why?

Python checks:

```
Local → found "Local"
```

So it stops searching.

If we remove the local variable:

```
name = "Global"

def outer():
    name = "Enclosing"

    def inner():
        print(name)

    inner()

outer()
```

Now Python checks:

```
Local → not found
Enclosing → found
```

Output:

```
Enclosing
```

---

21. The `global` Keyword

Sometimes we intentionally want to modify a global variable from inside a function.

By default, assigning to a variable inside a function creates a local variable.

Example:

```
count = 0

def increase():
    count = count + 1
```

This causes an error because Python treats `count` inside the function as local.

To explicitly tell Python that we want the global variable:

```
count = 0

def increase():
    global count
    count = count + 1

increase()

print(count)
```

Output:

```
1
```

---

22. How `global` Works

The statement:

```
global count
```

means:

> Use the global variable named `count` instead of creating a new local variable.

Example:

```
score = 10

def update_score():
    global score
    score = 20

update_score()

print(score)
```

Output:

```
20
```

---

23. Should You Use `global` Often?

Usually, avoid relying heavily on global variables.

For example, instead of:

```
score = 0

def increase():
    global score
    score += 1
```

it is often cleaner to return the new value:

```
def increase(score):
    return score + 1

score = 0
score = increase(score)
```

This makes the function easier to understand and test.

The main lesson is:

> Use local variables and return values when possible. Use `global` only when shared global state is genuinely needed.

---

24. The `nonlocal` Keyword

`nonlocal` is used with nested functions.

It allows an inner function to modify a variable from its enclosing function.

Example:

```
def counter():
    count = 0

    def increase():
        nonlocal count
        count += 1
        return count

    return increase
```

Now:

```
counter_function = counter()

print(counter_function())
print(counter_function())
print(counter_function())
```

Output:

```
1
2
3
```

The inner function changes the `count` variable belonging to the enclosing `counter()` function.

---

25. `global` vs `nonlocal`

Keyword	Changes
`global`	Global scope variable
`nonlocal`	Enclosing function variable

Example:

```
global total
```

means use a global variable.

While:

```
nonlocal total
```

means use a variable from an enclosing function.

---

26. Scope Inside Loops

Python has an important difference from some other programming languages.

A normal `for` or `while` loop does **not** create a separate local scope.

Example:

```
for i in range(3):
    number = i

print(number)
```

Output:

```
2
```

The variable remains accessible because the loop itself does not create a new function-like scope.

However, function boundaries do create local scope.

---

27. Scope Inside `if` Statements

Similarly, an `if` block does not create a separate scope.

```
if True:
    message = "Hello"

print(message)
```

Output:

```
Hello
```

The variable is accessible because `if` does not create a new local scope.

This is different from a function.

---

28. Function Scope vs Block Scope

In Python:

```
Functions → create local scope
Classes → create their own namespace
if/else → do not create a new function-like scope
for/while → do not create a new function-like scope
```

For beginners, the most important distinction is:

```
def example():
    value = 10
```

`value` is local to the function.

But:

```
if True:
    value = 10
```

`value` remains accessible outside the `if` block in the same scope.

---

29. Scope and Lists

Global and local variables behave the same way with lists regarding name lookup.

Example:

```
skills = ["Python"]

def add_skill():
    skills.append("JavaScript")

add_skill()

print(skills)
```

Output:

```
['Python', 'JavaScript']
```

The function accesses the global list and modifies the list object.

However, assigning a new list inside the function creates a local variable:

```
skills = ["Python"]

def add_skill():
    skills = ["JavaScript"]

add_skill()

print(skills)
```

Output:

```
['Python']
```

The local assignment did not replace the global variable.

---

30. A Common Scope Mistake

Consider:

```
count = 10

def update():
    count = count + 1

update()
```

A beginner might expect:

```
11
```

But Python raises an error.

Why?

Because the assignment:

```
count = ...
```

makes `count` local to the function.

Python then tries to read that local `count` before it has a value.

If you intentionally want the global variable:

```
count = 10

def update():
    global count
    count = count + 1

update()

print(count)
```

Output:

```
11
```

---

31. A Common Scope Mistake: Assuming `return` Exposes the Variable

Consider:

```
def create_user():
    name = "Hafsa"

create_user()

print(name)
```

`name` is still local.

To make its value available outside, return it:

```
def create_user():
    name = "Hafsa"
    return name

name = create_user()

print(name)
```

Now the returned value is stored in a variable outside the function.

---

32. Scope in Real Projects

Suppose a haas.dev application has:

```
website_name = "haas.dev"
```

A function can read it:

```
def show_website():
    print(website_name)
```

But resource-specific information should generally remain local:

```
def create_resource():
    title = "Python Functions"
    category = "Python"

    return {
        "title": title,
        "category": category
    }
```

Here:

```
website_name → global
title → local
category → local
```

Keeping temporary data local helps prevent unrelated parts of a program from accidentally changing it.

---

33. Why Local Scope Is Usually Better

Local variables provide several benefits:

1. Less accidental modification

Other functions cannot directly access the local variable.

2. Easier debugging

You know where the variable is being used.

3. Reusable functions

The function does not depend heavily on external state.

4. Better organization

Each function can manage its own temporary data.

---

34. Scope and Function Design

Compare these approaches.

Global state

```
total = 0

def add_price(price):
    global total
    total += price
```

The function depends on external state.

Using parameters and return values

```
def add_price(total, price):
    return total + price

total = 0
total = add_price(total, 500)
```

The second approach makes the input and output explicit.

This often makes code easier to test and maintain.

---

35. Problem Solving Framework

When you encounter a scope problem, ask:

Step 1: Where was the variable created?

Was it:

```
inside a function?
outside a function?
inside another function?
```

Step 2: Where are you trying to access it?

```
same function?
another function?
global code?
nested function?
```

Step 3: Which LEGB level should Python find?

```
Local
Enclosing
Global
Built-in
```

Step 4: Are you trying to modify an outer variable?

If yes, consider whether:

```
return
parameter
global
nonlocal
```

is appropriate.

Prefer parameters and return values when they provide a clearer design.

---

36. Mini Project: Student Score Manager

Create a function that calculates a student's final score.

```
def calculate_score(marks, bonus):
    final_score = marks + bonus
```

```
    return final_score
```

Here:

```
marks → local parameter
bonus → local parameter
final_score → local variable
```

Use it:

```
score = calculate_score(80, 5)

print(score)
```

Output:

```
85
```

The function keeps its temporary calculation local and returns only the result the rest of the program needs.

---

37. Mini Project: haas.dev Resource Counter

Imagine we want to count how many resources have been created.

A simple version using global state:

```
resource_count = 0

def create_resource():
    global resource_count
    resource_count += 1
    return resource_count
```

Now:

```
print(create_resource())
print(create_resource())
print(create_resource())
```

Output:

```
1
2
3
```

This demonstrates `global`.

However, for larger applications, keeping state in an object or passing state explicitly can be easier to manage.

---

38. Mini Project: Nested Counter

Now use an enclosing variable with `nonlocal`.

```
def create_counter():
    count = 0

    def increase():
        nonlocal count
        count += 1
        return count

    return increase
```

Use it:

```
counter = create_counter()

print(counter())
print(counter())
print(counter())
```

Output:

```
1
2
3
```

The `count` variable belongs to `create_counter()`, while `increase()` modifies it.
This is a practical example of enclosing scope.

---

39. 5 Minute Challenge

Create this function:

```
def create_counter():
```

Requirements:

- Start `count` at `0`.
- Create an inner function called `increase`.
- Every time `increase()` is called, increase `count` by `1`.

- Return the new count.
- Use `nonlocal` to modify the enclosing variable.

Expected usage:

```
counter = create_counter()

print(counter())
print(counter())
print(counter())
```

Expected output:

```
1
2
3
```

---

40. Exercises

Exercise 1

Identify the scope of each variable:

```
website = "haas.dev"

def show():
    message = "Welcome"
    print(website)
    print(message)
```

Determine:

- `website`
- `message`

Exercise 2

What happens here?

```
def greet():
    name = "Hafsa"

greet()

print(name)
```

Explain why.

Exercise 3

What does this print?

```
name = "Python"

def show():
    name = "JavaScript"
    print(name)

show()
print(name)
```

Exercise 4

Fix this code:

```
count = 0

def increase():
    count += 1

increase()
```

Use the appropriate keyword.

Exercise 5

Create a nested function where an inner function modifies a variable from the outer function using `nonlocal`.

---

41. Mini Quiz

Question 1

What is scope?

A. The data type of a variable

B. The area where a variable can be accessed

C. The name of a function

D. The value of a variable

Answer: B

Question 2

Which scope is created by a function?

A. Local scope

B. Loop scope

C. `if` scope

D. Import scope

Answer: A

Question 3

What does LEGB stand for?

A. Local, External, Global, Basic

B. Local, Enclosing, Global, Built-in

C. List, Expression, Global, Boolean

D. Local, External, General, Built-in

Answer: B

Question 4

Which keyword allows a function to modify a global variable?

A. `outer`

B. `global`

C. `public`

D. `globalize`

Answer: B

Question 5

Which keyword allows an inner function to modify a variable from its enclosing function?

A. `global`

B. `outer`

C. `nonlocal`

D. `parent`

Answer: C

---

42. Interview Questions

1. What is variable scope?

Variable scope defines where a variable can be accessed in a Python program.

---

2. What is a local variable?

A variable created inside a function that is normally accessible only within that function.

---

3. What is a global variable?

A variable defined outside functions that belongs to the global scope.

---

4. What is LEGB?

LEGB is Python's variable lookup order:

```
Local
Enclosing
Global
Built-in
```

---

5. What does the `global` keyword do?

It tells Python that a variable inside a function refers to a variable in the global scope.

---

6. What does `nonlocal` do?

It allows a nested function to modify a variable from its enclosing function.

---

7. Do `if` statements create a new scope in Python?

No. An `if` block does not create a separate function-like local scope.

---

8. Do loops create a new scope in Python?

No. `for` and `while` loops do not create a separate function-like scope.

---

43. Cheat Sheet

Local

```
def greet():
    name = "Hafsa"
```

`name` is local to `greet()`.

Global

```
name = "Hafsa"
```

`name` is global.

Enclosing

```
def outer():
    name = "Hafsa"

    def inner():
        print(name)
```

`name` is in the enclosing scope of `inner()`.

Built-in

```
len()
print()
sum()
```

These are examples of built-in names.

`global`

```
count = 0

def increase():
    global count
    count += 1
```

`nonlocal`

```
def outer():
    count = 0

    def inner():
        nonlocal count
        count += 1
```

LEGB Quick View

```
┌─────────────────────────┐
│ Local                   │ ← Current function
├─────────────────────────┤
│ Enclosing               │ ← Outer/nested function
├─────────────────────────┤
│ Global                  │ ← Program/module level
├─────────────────────────┤
```

```
  | Built-in             | ← Python built-ins
  └──────────────────────┘
```

Python searches from **L → E → G → B**.

The first matching name is normally the one Python uses.

Key Takeaways

- Scope determines where a variable can be accessed.
- Variables created inside functions normally have local scope.
- Variables created outside functions are generally global.
- Nested functions introduce an enclosing scope.
- Python uses the **LEGB** rule to find variable names.
- `global` allows a function to modify a global variable.
- `nonlocal` allows a nested function to modify a variable from its enclosing function.
- `if`, `for`, and `while` blocks do not create function-like local scopes in Python.
- Function parameters are local to the function.
- `return` is often a cleaner way to move a result outside a function than relying on global variables.
- Local variables help keep functions independent and easier to maintain.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app

Resources:

https://dev-roast-app.vercel.app/resources