Python Local vs Global Variables

Introduction

When you write Python programs, variables do not all live in the same place.

Some variables are created **inside a function** and are only available there. These are called **local variables**.

Other variables are created **outside functions** and can be accessed from different parts of your program. These are called **global variables**.

Understanding the difference is important because it helps you:

- avoid `NameError`
- understand how functions access data
- prevent accidental changes to important values
- write cleaner and more predictable programs
- understand how Python handles variable scope

---

1. What Is a Local Variable?

A **local variable** is a variable created inside a function.

It belongs to that function and normally cannot be accessed directly from outside it.

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

    print(message)

greet()
```

Here:

```
message = "Hello"
```

is a local variable.

It exists inside `greet()`.

Important idea

Think of a function as a room.

A local variable is something kept **inside that room**.

Only code inside the function can directly use it.

---

2. Local Variable Example

```
def calculate_total():
    price = 500
    quantity = 2

    total = price * quantity

    print(total)

calculate_total()
```

The variables:

```
price
quantity
total
```

are all local variables.

They are available while Python is executing `calculate_total()`.

---

3. Local Variables Cannot Normally Be Used Outside the Function

```
def calculate():
    total = 500

calculate()

print(total)
```

This causes an error because `total` belongs to the function.

Python will raise:

```
NameError
```

The correct approach is often to return the value.

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

result = calculate()
```

```
print(result)
```

Now `result` exists outside the function and contains the returned value.

---

4. What Is a Global Variable?

A **global variable** is created outside functions.

```
app_name = "haas.dev"

def show_app():
    print(app_name)

show_app()
```

Here:

```
app_name = "haas.dev"
```

is a global variable.

The function can read it because Python looks for variables outside the function when it cannot find them locally.

---

5. Local vs Global

Consider this program:

```
website = "haas.dev"

def show_resource():
    resource = "Python Course"

    print(website)
    print(resource)

show_resource()
```

There are two different variables:

```
website
```

is global.

```
resource
```

is local.

Simple rule

Variable	Created where?	Available where?
Local	Inside a function	Inside that function
Global	Outside functions	Throughout the module, subject to scope rules

---

6. A Function Can Read a Global Variable

Python allows a function to read a global variable.

```
course = "Python"

def show_course():
    print(course)

show_course()
```

Output:

```
Python
```

Python checks:

1. Is `course` inside `show_course()`?
2. If not, is there a global `course`?
3. If found, use it.

---

7. Local Variables Take Priority

Suppose the same variable name exists locally and globally.

```
name = "Hafsa"

def show_name():
    name = "Asim"
    print(name)

show_name()

print(name)
```

Output:

```
Asim
Hafsa
```

Why?

Inside the function:

```
name = "Asim"
```

creates a local variable.

Outside the function:

```
name = "Hafsa"
```

is the global variable.

Python uses the local variable when the function is running.

---

8. Same Name Does Not Mean Same Variable

This is an important concept.

```
score = 90

def show_score():
    score = 50
    print(score)

show_score()

print(score)
```

Output:

```
50
90
```

The local `score` does not replace the global `score`.

There are two separate variable bindings.

Think of them as:

```
Global:
score → 90

Function:
score → 50
```

---

9. Changing a Local Variable

A local variable can be changed normally.

```
def counter():
    count = 1

    count = count + 1

    print(count)

counter()
```

Output:

```
2
```

The change only affects the local variable.

---

10. Changing a Global Variable

Reading a global variable is easy:

```
count = 10

def show_count():
    print(count)

show_count()
```

But assigning to that variable inside the function is different.

Consider:

```
count = 10

def increase():
    count = count + 1

increase()
```

This produces an error.

Why?

Because Python sees:

```
count = ...
```

inside the function and treats `count` as a local variable.

Then Python tries to read that local `count` before it has been given a value.

---

11. The `global` Keyword

If you intentionally want to modify a global variable from inside a function, Python provides the `global` keyword.

```
count = 10

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

increase()

print(count)
```

Output:

```
11
```

The line:

```
global count
```

tells Python:

> Use the global `count`, not a new local variable.

---

12. Example of `global`

```
visits = 0

def visit_page():
    global visits

    visits = visits + 1

visit_page()
visit_page()
visit_page()

print(visits)
```

Output:

```
3
```

Every function call changes the same global variable.

---

13. Should You Use `global` Frequently?

Usually, **no**.

Although `global` is valid Python, using many global variables can make programs harder to understand.

For example:

```
balance = 1000

def withdraw(amount):
    global balance
    balance = balance - amount

withdraw(200)
```

This works.

But a cleaner design is often:

```
def withdraw(balance, amount):
    return balance - amount

balance = 1000
```

```
balance = withdraw(balance, 200)

print(balance)
```

Now the function clearly shows:

```
Input → Function → Output
```

This makes the program easier to test and maintain.

---

14. Prefer Parameters and Return Values

Instead of depending heavily on global variables:

```
tax = 100

def calculate_total():
    return 500 + tax
```

you can make the dependency explicit:

```
def calculate_total(price, tax):
    return price + tax

total = calculate_total(500, 100)

print(total)
```

This is generally easier to understand.

The function tells us exactly what information it needs.

---

15. Global Constants

Global variables are not always bad.

A common use is storing **constants**.

Constants are values that should normally remain unchanged.

Python does not enforce constants, but developers commonly use uppercase names.

```
MAX_USERS = 100
APP_NAME = "haas.dev"
PI = 3.14159
```

These can be accessed from functions:

```
MAX_USERS = 100

def show_limit():
    print(MAX_USERS)

show_limit()
```

Using global constants can make sense because they represent shared configuration or fixed values.

---

16. Function Parameters Are Local Variables

Function parameters are local to the function.

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

greet("Hafsa")
```

Here:

```
name
```

is a local variable.

It only exists inside `greet()`.

Another example:

```
def calculate(price, quantity):
    total = price * quantity
    return total
```

These are local to the function:

```
price
quantity
total
```

---

17. Local Variable vs Global Variable Example

```
website = "haas.dev"

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

```
    print(website)
    print(title)
    print(category)

create_resource("Python Functions")
```

Here:

```
website
```

is global.

```
title
category
```

are local.

The function can read the global `website` while using its own local variables.

---

18. What Happens When Local and Global Variables Have the Same Name?

Example:

```
category = "Web Development"

def show_category():
    category = "Python"

    print(category)

show_category()

print(category)
```

Output:

```
Python
Web Development
```

The local variable hides the global variable **inside that function**.

This is called **shadowing**.

---

19. Variable Shadowing

Variable shadowing happens when a local variable uses the same name as a variable from an outer scope.

```
language = "JavaScript"

def learn():
    language = "Python"
    print(language)

learn()
```

Inside `learn()`:

```
language
```

refers to:

```
Python
```

The global value remains:

```
JavaScript
```

Best practice

Avoid using the same name for local and global variables unless there is a clear reason.

---

20. Local Variables Inside Multiple Functions

Each function has its own local scope.

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

def second():
    message = "Welcome"
    print(message)

first()
second()
```

Output:

```
Hello
Welcome
```

Both functions have a variable called:

```
message
```

but they are separate local variables.

---

21. Global Variables Can Be Shared

A global variable can be accessed by multiple functions.

```
app_name = "haas.dev"

def show_name():
    print(app_name)

def display_message():
    print("Welcome to", app_name)

show_name()
display_message()
```

Both functions can read:

```
app_name
```

because it is global.

---

22. A Practical Example: Student Result

Suppose we have a school program.

```
school_name = "ABC School"

def calculate_result(marks):
    total = sum(marks)

    print(school_name)
    print("Total:", total)

marks = [80, 75, 90]

calculate_result(marks)
```

Here:

```
school_name
```

is global.

```
marks
```

is a parameter/local variable inside the function.

```
total
```

is also local.

A cleaner design could be:

```
def calculate_result(marks):
    return sum(marks)

marks = [80, 75, 90]

total = calculate_result(marks)

print("ABC School")
print("Total:", total)
```

Now the function focuses only on calculating the result.

---

23. A Practical haas.dev Example

Imagine haas.dev has a resource system.

```
WEBSITE_NAME = "haas.dev"

def show_resource(title):
    category = "Python"

    print(WEBSITE_NAME)
    print("Resource:", title)
    print("Category:", category)

show_resource("Python Functions")
```

Here:

```
WEBSITE_NAME
```

is a global constant.

```
title
category
```

are local to the function.

This separation makes the code easier to understand.

---

24. Better Design With Parameters

Instead of depending on a global variable:

```
website = "haas.dev"

def show_resource(title):
    print(website)
    print(title)
```

we can write:

```
def show_resource(website, title):
    print(website)
    print(title)

show_resource("haas.dev", "Python Functions")
```

Now the function does not depend on hidden external data.

Its inputs are clear.

---

25. Local vs Global: A Mental Model

Imagine your program is a house.

Global variables

They are like things kept in the **common area**.

Different rooms can access them.

Local variables

They are like things kept **inside one room**.

Only that room can directly access them.

```
PROGRAM
│
├── Global variables
│
├── Function A
│   └── Local variables
│
```

```
├── Function B
│   └── Local variables
│
└── Function C
    └── Local variables
```

Each function gets its own local space.

---

26. The Scope Lookup Idea

When Python sees a variable name inside a function, it looks for that name according to Python's scope rules.

A useful simplified order is:

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

This is known as **LEGB**.

For this topic, the most important part is:

```
Local → Global
```

Python first checks the function's local scope.

If it cannot find the variable there, it can look in the global scope.

---

27. Common Mistake: Assuming a Local Variable Is Global

Incorrect:

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

create_user()

print(username)
```

`username` is local.

Correct:

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

username = create_user()

print(username)
```

The function returns the value, and the caller stores it.

---

28. Common Mistake: Accidentally Creating a Local Variable

Consider:

```
score = 100

def update_score():
    score = 200
```

This does not change the global `score`.

It creates a new local variable.

```
print(score)
```

still gives:

```
100
```

If you intentionally want to change the global variable:

```
score = 100

def update_score():
    global score
    score = 200

update_score()

print(score)
```

Output:

```
200
```

---

29. Common Mistake: Using Too Many Global Variables

This can become difficult to maintain:

```
username = "Hafsa"
score = 90
course = "Python"
level = "Beginner"
status = "Active"
```

Many functions may read and modify these values.

As a program grows, it becomes difficult to know:

- where a value changed
- which function changed it
- what a function depends on
- why a value has a particular value

Prefer passing data explicitly when practical.

---

30. A Better Function Design

Instead of:

```
price = 1000
discount = 100

def calculate():
    return price - discount
```

prefer:

```
def calculate(price, discount):
    return price - discount

total = calculate(1000, 100)

print(total)
```

Now the function is more reusable.

You can use it with different values:

```
calculate(2000, 200)
calculate(5000, 500)
calculate(800, 50)
```

---

31. When Should You Use Local Variables?

Use local variables for values that belong to a specific operation.

Examples:

```
def calculate_area(width, height):
    area = width * height
    return area
```

```
def process_order(price, quantity):
    total = price * quantity
    return total
```

```
def create_username(first_name):
    username = first_name.lower()
    return username
```

These values only need to exist while the function performs its task.

---

32. When Can Global Variables Be Useful?

Global variables can be useful for:

- constants
- application configuration
- fixed settings
- values intentionally shared across a module

Example:

```
APP_NAME = "haas.dev"
MAX_RESOURCES = 100
DEFAULT_LANGUAGE = "Python"
```

But shared mutable state should be used carefully.

---

33. Local vs Global Comparison

Feature	**Local Variable**	**Global Variable**

Created	Inside function	Outside function
Scope	Function	Module/program level
Can function read it?	Yes	Yes
Can other functions directly access it?	No	Yes
Lifetime	During relevant execution/context	Generally available while module is loaded
Easier to control?	Usually	Can become harder
Best use	Function-specific data	Shared constants/configuration

---

34. Problem Solving Framework

When deciding whether a variable should be local or global, ask:

Question 1: Who needs this value?

If only one function needs it:

```
Local
```

If many parts of the program intentionally need it:

```
Possibly global
```

Question 2: Can I pass it as an argument?

If yes, prefer:

```
function(value)
```

instead of relying on hidden global state.

Question 3: Does the function produce a result?

If yes, consider:

```
return result
```

instead of modifying a global variable.

Question 4: Is it a fixed configuration value?

If yes, a global constant may make sense:

```
MAX_USERS = 100
```

---

35. Mini Project: Score Calculator

Create a program that calculates a student's average.

```
SCHOOL_NAME = "ABC School"

def calculate_average(marks):
    total = sum(marks)
    average = total / len(marks)

    return average

marks = [80, 75, 90, 85]

average = calculate_average(marks)

print(SCHOOL_NAME)
print("Average:", average)
```

Here:

```
SCHOOL_NAME
```

is global.

```
marks
```

is passed into the function.

```
total
average
```

are local variables.

This is a good separation of responsibilities.

---

36. Mini Project: Resource Counter

Create a small resource counter for haas.dev.

```
WEBSITE = "haas.dev"

def show_resource_count(resources):
    count = len(resources)

    print(WEBSITE)
    print("Resources:", count)

resources = [
    "Python Functions",
    "Python Lists",
    "Python Dictionaries"
]

show_resource_count(resources)
```

Identify the variables

Global:

```
WEBSITE
```

Local:

```
resources
count
```

Note that the `resources` parameter is local to the function even though the caller also has a variable named `resources`.

---

37. 5 Minute Challenge

Write this program:

```
Global:
APP_NAME = "My App"

Function:
receive a list of products
calculate the total number of products
return the count

Outside:
call the function
print the app name
print the count
```

Expected structure:

```
APP_NAME = "My App"

def count_products(products):
    count = len(products)
    return count

products = ["Laptop", "Mouse", "Keyboard"]

result = count_products(products)

print(APP_NAME)
print(result)
```

Then modify the program so that:

1. The list contains five products.
2. The function returns the count.
3. The global variable remains unchanged.
4. The result is stored in a new local/outside variable.

---

38. Exercises

Exercise 1

Identify the local and global variables:

```
name = "Hafsa"

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

Exercise 2

What will this print?

```
x = 10

def test():
    x = 20
    print(x)

test()
print(x)
```

Exercise 3

Fix this code:

```
score = 50

def increase():
    score = score + 10

increase()

print(score)
```

Exercise 4

Rewrite this function so it does not depend on the global variable:

```
price = 1000

def show_price():
    print(price)
```

Exercise 5

Create a function:

```
calculate_discount(price, discount)
```

It should:

1. calculate the discounted price
2. store the result in a local variable
3. return the result
4. avoid using global variables

---

39. Mini Quiz

Question 1

Where is a local variable created?

A. Outside every function

B. Inside a function

C. Inside Python itself

D. Inside a module only

Answer: B

Question 2

What happens here?

```
name = "Hafsa"

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

show()
```

What is printed?

A. Hafsa

B. Asim

C. Error

D. Nothing

Answer: B

Question 3

Which keyword allows a function to modify a global variable?

A. `outer`

B. `global`

C. `public`

D. `shared`

Answer: B

Question 4

Which approach is usually preferable for passing information into a function?

A. Global variables

B. Parameters

C. Random variables

D. `print()`

Answer: B

Question 5

What is this?

```
MAX_USERS = 100
```

when created outside a function?

A. Local variable

B. Global constant-style variable

C. Function parameter

D. Loop variable

Answer: B

---

40. Interview Questions

Beginner

1. What is a local variable in Python?
2. What is a global variable?
3. What is the difference between local and global variables?
4. Can a function access a global variable?
5. Can a local variable be accessed outside its function?

6. What happens when a local and global variable have the same name?

Intermediate

7. What does the `global` keyword do?
8. Why can assigning to a global variable inside a function cause an error?
9. What is variable shadowing?
10. Why are function parameters considered local variables?
11. Why should global mutable state be used carefully?
12. Why are parameters and return values often preferable to global variables?

---

41. Cheat Sheet

```
LOCAL VARIABLE
Created inside a function.

def test():
    x = 10
```

```
GLOBAL VARIABLE
Created outside a function.

x = 10

def test():
    print(x)
```

```
LOCAL OVERRIDES GLOBAL INSIDE FUNCTION

x = 10

def test():
    x = 20
    print(x)

test()
print(x)
```

Output:

```
20
10
```

```
MODIFY GLOBAL VARIABLE

x = 10

def change():
    global x
    x = 20
```

```
PREFERRED PATTERN

def calculate(value):
    result = value * 2
    return result

answer = calculate(10)
```

Remember

```
Local → belongs to a function
Global → exists outside functions
Parameter → local to the function
return → sends a value back
global → explicitly allows modification of a global variable
```

---

42. Key Takeaways

- A **local variable** is created inside a function.
- A **global variable** is created outside functions.
- Local variables are normally available only inside their function.
- Functions can read global variables.
- A local variable with the same name as a global variable takes priority inside the function.
- This situation is called **shadowing**.
- Use `global` when you intentionally need to modify a global variable.
- Avoid unnecessary global state.
- Prefer **parameters and return values** for passing data between functions.
- Global constants can be useful for shared fixed values.
- Understanding local and global variables is essential for understanding Python scope.

Visit haas.dev for more resources and guides.

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

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