Pass by Value and Pass by Reference in Python

Pass by Value and Pass by Reference in Python

When we call a function in Python, we can pass some values to it, called **arguments**. Inside the function, these values are received as **parameters**.

Example:

```
def greet(name):
    print("Hello", name)

greet("Abhyas")
```

Output:
Hello Abhyas
Here:
- "Abhyas" is the **argument**.
- name is the **parameter**.
- Python passes the *object reference* of "Abhyas" to the function.

Pass by Value or Pass by Reference

Many programming languages work in one of these two ways:

Pass by Value

- The function gets a copy of the value.
- Changes inside the function don't affect the original variable.
- Example languages: C, Java (for primitive types).

Pass by Reference

- The function gets a direct reference to the original variable.
- Changes inside the function do affect the original variable.
- Example languages: C++ (with references).

How Python actually works?

Python does neither pure pass-by-value nor pure pass-by-reference. Its model is best described as **"pass-by-object-reference" or "call-by-assignment":**

Key terms in plain language

- **Mutable objects**: objects whose contents can be changed (lists, dicts, sets, most custom objects).
- **Immutable objects**: cannot be changed in place (ints, floats, strings, tuples).
- **Mutate**: change the object itself (e.g., list.append() modifies the list).
- **Rebind**: give a variable (name) a new object (e.g., x = x + 1 makes x refer to a new int).

Pass by Value and Pass by Reference in Python

When you call a function in Python and give it a variable:

```
def show(x):
    print(x)

a = [1, 2, 3]
show(a)
```

Python doesn't make a *copy* of "a". It **sends a reference (a kind of link)** to the same object in memory.
Inside the function, the parameter "**x**" becomes a **new name** for that same object. So both a and x point to the same thing.

So:
• If the object is mutable (list, dict, set, most custom objects), in-place changes inside the function are visible to the caller.
• If the object is immutable (int, float, str, tuple), you cannot change it in place — assigning the parameter to a new value only rebinds the local name and does not affect the caller.

Solved Example 1: Immutable example (int)

```
def add_one(x):
    print("inside before:", x, "id:", id(x))
    x = x + 1
    print("inside after :", x, "id:", id(x))

n = 10
print("outside before:", n, "id:", id(n))
add_one(n)
print("outside after :", n, "id:", id(n))
```

Stepwise explanation:
1. n label → box with value 10. id(n) shows the box address.
2. Call add_one(n): function parameter x gets its own label stuck on the *same* box (10).
3. Inside the function x = x + 1 creates a **new** integer object 11, and the label x is moved to that new box.
4. The original n label is still on box 10 — so outside nothing changed.

Takeaway: For immutable types, operations that look like changes actually create new objects and rebind the local name — the caller's name is unchanged.

Solved Example 2: Mutable example (list mutated)

```
def append_item(lst):
    print("inside before:", lst, "id:", id(lst))
```

Pass by Value and Pass by Reference in Python

```
    lst.append(99)
    print("inside after :", lst, "id:", id(lst))

my_list = [1, 2, 3]
print("outside before:", my_list, "id:", id(my_list))
append_item(my_list)
print("outside after :", my_list, "id:", id(my_list))
```

Stepwise explanation:
1. my_list label → box [1,2,3].
2. Call append_item(my_list): parameter lst labels the same box.
3. lst.append(99) **mutates** the box (adds 99). The box's contents change; its id stays the same.
4. Outside, my_list still labels the same box, so it sees the change.

Takeaway: Mutating methods change the object in place and are visible to the caller.

Solved Example 3: Mutable but rebinding inside function

```
def reassign(lst):
    print("inside before reassign:", lst, "id:", id(lst))
    lst = [9, 8, 7]   # new list created and local name lst is rebound
    print("inside after reassign :", lst, "id:", id(lst))

a = [1, 2, 3]
print("outside before:", a, "id:", id(a))
reassign(a)
print("outside after :", a, "id:", id(a))
```

Stepwise explanation:
1. a label → original box [1,2,3].
2. Call function; lst labels same original box.
3. lst = [9,8,7] creates a new box and moves the label lst to it (local rebind). This does NOT touch the original box.
4. Outside a still labels original box — unchanged.

Important point: Mutation vs rebind are very different.

Assignment

Ques 1: Immutable Type (Pass by Value Example)
Write a Python program where you define a function increment(num) that adds 1 to a number and prints it.
Then print the number again outside the function. Observe whether the value changes outside the function.

Pass by Value and Pass by Reference in Python

Ques 2. Mutable Type (Pass by Reference Example)
Create a list of numbers [1, 2, 3] and define a function add_item(lst) that appends 4 to it. Print the list inside and outside the function. Check whether both outputs are the same.

Ques 3. Reassignment Inside Function
Define a function modify_list(lst) that creates a new list inside the function and assigns it to lst. Print the list inside and outside the function. Does the change reflect outside?