Call by Value vs. Call by Reference in Python

1 Overview

Python does not use traditional *call by value* or *call by reference* semantics. Instead, it uses what is best described as **call by object reference** (or **call by sharing**).

This means:

- Function arguments are *references to objects*, not copies.
- The reference itself is passed by value.

In other words, the function receives a new reference to the same object, but reassigning the parameter inside the function does not affect the caller's variable.

2 Analogy

A variable name is like a sticky note pointing to an object. When you pass that variable to a function, Python gives the function a *copy of the sticky note*. Both notes point to the same object, but if one note is moved to a new object, the other is unaffected.

3 Examples

Example A – Immutable types (int, str, tuple)

```
1  def modify(x):
2      x = x + 1  # rebinds local x
3
4  a = 10
5  modify(a)
6  print(a)  # -> 10 (unchanged)
```

Explanation: `x` originally points to the same object as `a` (the integer 10). `x = x + 1` creates a new integer (11) and rebinds `x` to it. `a` still points to the old object.

Example B – Mutable types (list, dict, set)

```
def modify(lst):
lst.append(99)  # mutates the object

a = [1, 2, 3]
modify(a)
print(a)  # -> [1, 2, 3, 99]
```

Explanation: `lst` and `a` both reference the same list. Mutation is visible outside the function.

Example C – Rebinding a mutable

```
def reassign(lst):
lst = [42]  # new list assigned locally

a = [1, 2, 3]
reassign(a)
print(a)  # -> [1, 2, 3]
```

Rebinding inside the function does not change the caller's variable, since only the local reference is modified.

4 Summary

Type of Object	Operation in Function	Effect Outside
Immutable (int, str, tuple)	Rebind or modify	No effect
Mutable (list, dict, set)	Mutate in place	Visible outside
Any type	Rebind variable	No effect

Table 1: Behavior of argument passing in Python

5 Conclusion

Python = Call by Object Reference (Call by Sharing).

- The function receives a copy of the reference to the same object.
- If the object is mutable and you mutate it, the caller sees the change.

- If you rebind the local variable, the caller does not.

This model explains why Python functions sometimes seem to behave like call-by-value and sometimes like call-by-reference, depending on whether the argument object is mutable.