In Python, the concept of "pass by reference" is a bit nuanced because Python handles variables differently compared to languages like C++ or Java. In Python, all variables are references to objects in memory, and when you pass a variable to a function, you're passing a reference to the object, not the actual value. However, whether or not changes made inside a function reflect outside the function depends on whether the object is **mutable** or **immutable**.

**Mutable vs Immutable Objects**

* **Immutable objects**: These are objects whose state cannot be changed after they are created. Common examples include integers, strings, and tuples.
* **Mutable objects**: These are objects whose state can be changed after they are created. Common examples include lists, dictionaries, and sets.

**How Mutable and Immutable Objects Behave When Passed to a Function**

1. **Immutable objects** (like integers, strings, and tuples): When passed to a function, any changes made inside the function do not affect the original object because Python creates a new object for the modified value. The reference to the new object is local to the function.
2. **Mutable objects** (like lists or dictionaries): When passed to a function, changes made inside the function do affect the original object, because the function operates on the reference to the original object.

**Example 1: Immutable Object (Integer)**

Let's look at how an immutable object behaves when passed to a function:

python

Copy code

def modify\_immutable(x):

x = x + 1 # Creates a new integer object

print("Inside function:", x)

a = 10

modify\_immutable(a)

print("Outside function:", a)

**Output**:

bash

Copy code

Inside function: 11

Outside function: 10

**Explanation**:

* a is an integer (immutable), and when it's passed to modify\_immutable, a new object is created inside the function when x is modified. The original a remains unchanged.

**Example 2: Mutable Object (List)**

Now, let's see how a mutable object behaves when passed to a function:

python

Copy code

def modify\_mutable(lst):

lst.append(4) # Modifies the original list

print("Inside function:", lst)

my\_list = [1, 2, 3]

modify\_mutable(my\_list)

print("Outside function:", my\_list)

**Output**:

bash

Copy code

Inside function: [1, 2, 3, 4]

Outside function: [1, 2, 3, 4]

**Explanation**:

* my\_list is a list (mutable), and when it's passed to modify\_mutable, the original list is modified in place. The changes inside the function affect the original list outside the function because lst and my\_list both refer to the same list object in memory.

**Summary**

* **Immutable objects**: If you modify them inside a function, the changes do not affect the original object outside the function.
* **Mutable objects**: If you modify them inside a function, the changes affect the original object outside the function because both the original and the parameter inside the function reference the same object in memory.

This behavior is often referred to as **"pass-by-object-reference"** or **"pass-by-assignment"** in Python, because the variables themselves are references to objects in memory.