📘 PYTHON PARAMETER PASSING & MUTABILITY (IN-DEPTH NOTES)

1️⃣ How Python passes arguments (CORE IDEA)

👉 Python uses “pass by object reference”

(Some books call it *pass by assignment*)

This means:

- A **reference (address) to the object** is passed to the function
- What happens next depends on **whether the object is mutable or immutable**

2️⃣ Mutable vs Immutable Objects

🔹 Immutable Objects (CANNOT change in place)

- int
- float
- bool
- str
- tuple
- None

👉 Any “change” creates a **new object**

🔹 Mutable Objects (CAN change in place)

- list
- dict
- set
- custom class objects

👉 Changes affect the **same object in memory**

3️⃣ What people mean by “Pass by Value” & “Pass by Reference”

❌ Python does NOT have true pass by value

❌ Python does NOT have true pass by reference

But to **understand**, we use these terms conceptually.

🔹 **Pass by Value (conceptual)**

- A **copy** of the value is passed
- Changes inside function **do NOT affect original**

👉 Python behaves like this for **immutable objects**

Example

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


a = 10
change(a)
print(a)
```

Output

10

✓ Original not changed

🔹 **Pass by Reference (conceptual)**

- A **reference to the same object** is passed
- Changes inside function **affect original**

👉 Python behaves like this for **mutable objects**

Example

```
def change(lst):
    lst.append(4)


a = [1, 2, 3]
change(a)
print(a)
```

Output

[1, 2, 3, 4]

✓ Original modified

🔽 What ACTUALLY happens (Important!)

🔍 Example with dictionary & boolean

```
def update_data(d, flag):
    d["x"] = 100
    flag = True

my_dict = {}
my_flag = None

update_data(my_dict, my_flag)

print(my_dict)
print(my_flag)
```

Output

{'x': 100}

None

🔍 Memory explanation

Dictionary (mutable)

my_dict ──▶ {}

↑

d (same object)

Boolean (immutable)

my_flag ──▶ None

flag = True (new object, local only)

5️⃣ Rebinding vs Mutation (VERY IMPORTANT)

🔹 Mutation → changes original

```
lst.append(10)

dict["a"] = 5
```

🔹 Rebinding → creates new object

```
lst = [1, 2]

x = 100

flag = True
```

⚠️ Tricky Example

```
def test(lst):
    lst = [9, 9, 9]   # rebinding


a = [1, 2, 3]

test(a)

print(a)
```

Output

```
[1, 2, 3]
```

❌ No change (rebinding, not mutation)

6️⃣ How to change immutable value outside a function

✅ Method 1: Return it (BEST WAY)

```
def change(x):
    return x + 10


a = 5

a = change(a)
```

✅ Method 2: Wrap inside mutable container

```
def change(flag):
    flag[0] = True

f = [None]
change(f)
print(f[0])
```

7️⃣ Tuple Question (IMPORTANT)

Given:

t = (10, 20, 30, 40)

❓ Is this possible?

t += (50)

❌ NO — ERROR

Because (50) is **NOT a tuple**

Python thinks it is:

50

So:

t += 50 ❌ TypeError

✅ Correct Way

t += (50,)

Why comma is required?

- Single-element tuple **must have a comma**

(50,) ✅ tuple

(50) ❌ int

⚠️ Important: tuple is immutable

t += (50,)

This does **NOT modify** the tuple.
It **creates a new tuple and rebinds t**.

🔍 Proof

t = (1, 2)

print(id(t))

t += (3,)

print(id(t))

IDs will be **different**.

🎱 One-Page Summary (WRITE THIS)

Python passes **object references**
Mutable objects → mutation affects outside
Immutable objects → reassignment does NOT
Tuple cannot be modified, only rebound
(50,) is tuple, (50) is int