Open in app Get started

Published in Starbugs Weekly 星巴哥技術專欄

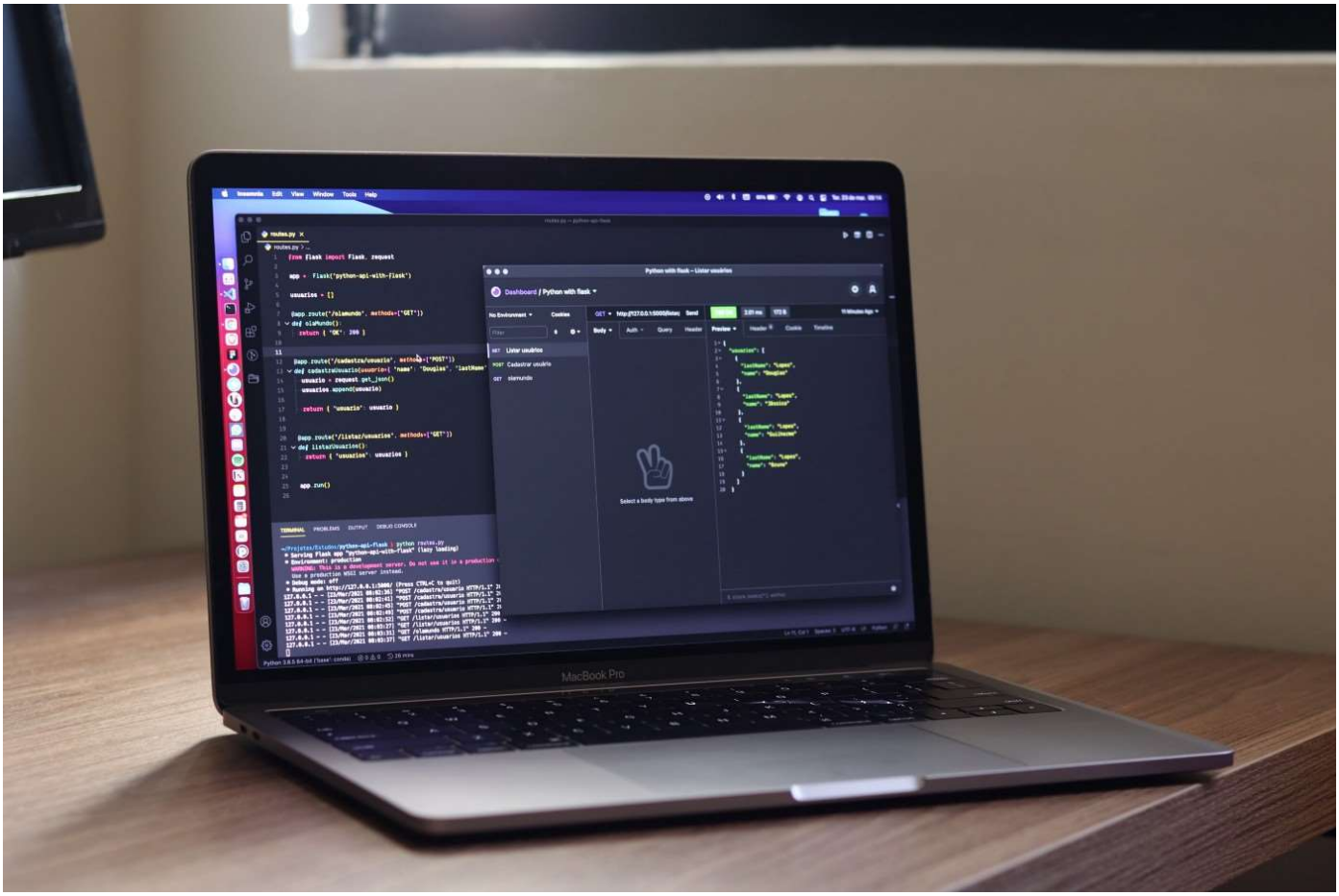
Luka Huang Follow

Sep 21, 2021 · 14 min read

Save

# Python 是 Pass By Value, Pass by Reference, 還是 Pass by Sharing？

Python 到底是 Call by Value, Call by Reference, 還是 Call by Sharing？



常見的語言的求值策略(Evaluation strategy)大概離不開這幾個，例如：JavaScript (Call by Value, Call by Sharing)、Java (Call by Value)、C (Call by Value)、C++ (Call by Value, Call by Reference)

這次使用使用淺顯易懂的語言，並搭配圖片來說明到底 Python 中是 Pass by Value, Pass by Reference, 還是 Pass Sharing？

```
本篇目錄：
1. 用 C++ 理解 Pass by Value
2. 用 C++ 理解 Pass by Reference
3. 來看看 Python 是不是 Pass by Value
4. 來看看 Python 是不是 Pass by Reference
5. 來看看 Python 是不是 Pass By Sharing
6. Python 到底是 Pass by Value, Pass Value 還是 Pass Sharing 呢？
```

## 1. 用 C++ 理解 Pass by Value

C++ 是探討 Pass by Value 與 Pass by Reference 的最佳選擇。我們先來理解 Pass by Value。

```
#include <iostream>

void passByValue(int num)
{
    printf("Before: the num is %d\n", num);

    num = 3;

    printf("After: the num is %d\n", num);
} // num is destroyed here

int main()
{
    int number{ 7 };
    printf("Before: number is %d\n", number);

    passByValue(number);

    printf("After: number is %d\n", number);
    return 0;
}
```

**執行結果:**

Before: number is 7

Before: the num is 7

可以看到函式內部的數並不會受到影響，遵守作用域( Scope)範圍。我們再執行另一個程式，查看記憶體位址的改變。

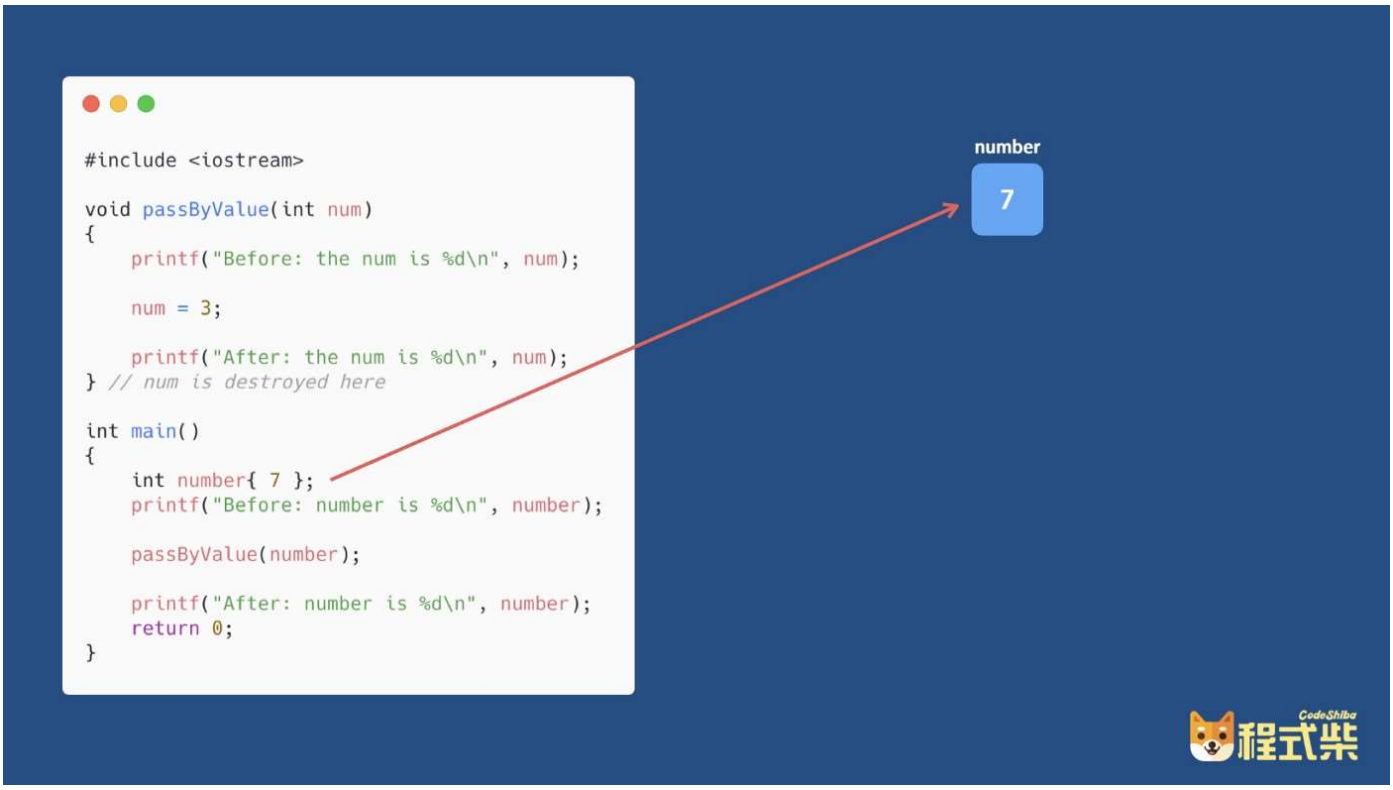
```
#include <iostream>

void passByValue(int num)
{
    printf("The Address of num is %p\n",(void*)&num);

    num = 3;

    printf("The Address of num is %p\n",(void*)&num);
}

int main()
{
    int number{ 7 };

    printf("The Address of number is %p\n",(void*)&number);

    passByValue(number);

    printf("The Address of number is %p\n",(void*)&number);
    return 0;
}
```

**執行結果:**

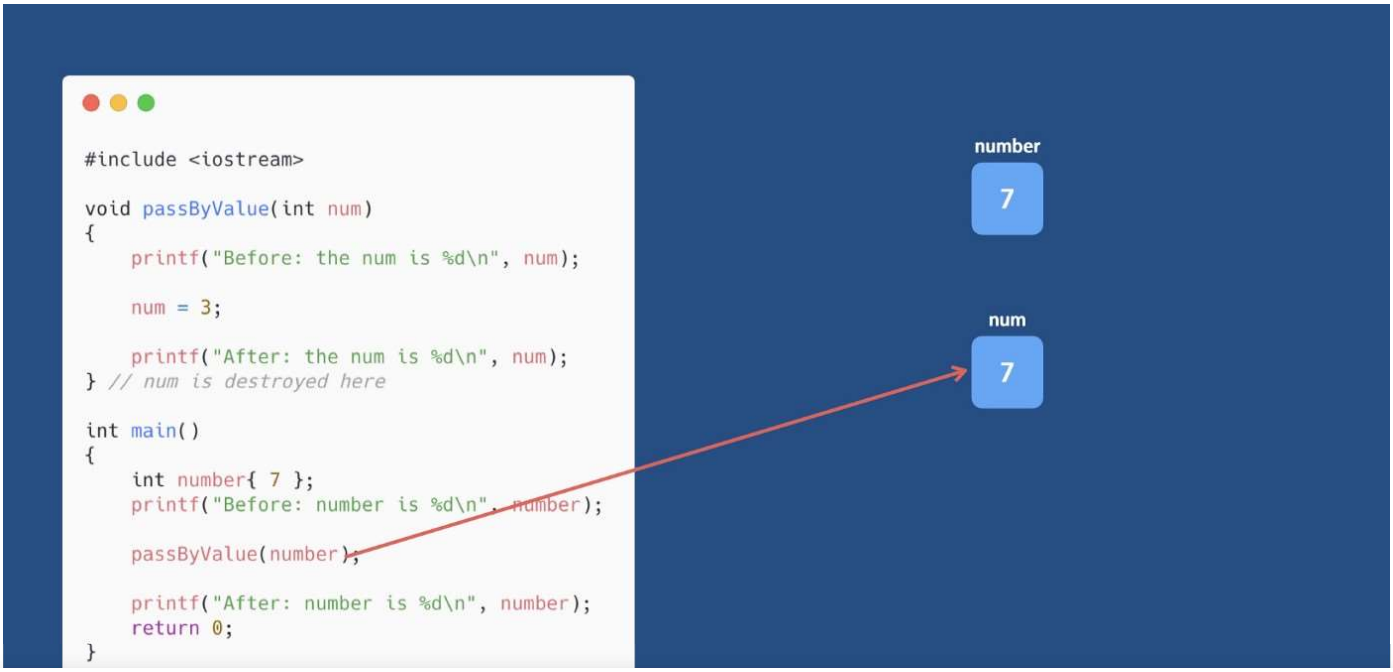
The Address of number is 0x7ffd0abe7b74
The Address of num is 0x7ffd0abe7b5c
The Address of num is 0x7ffd0abe7b5c

Open in app Get started

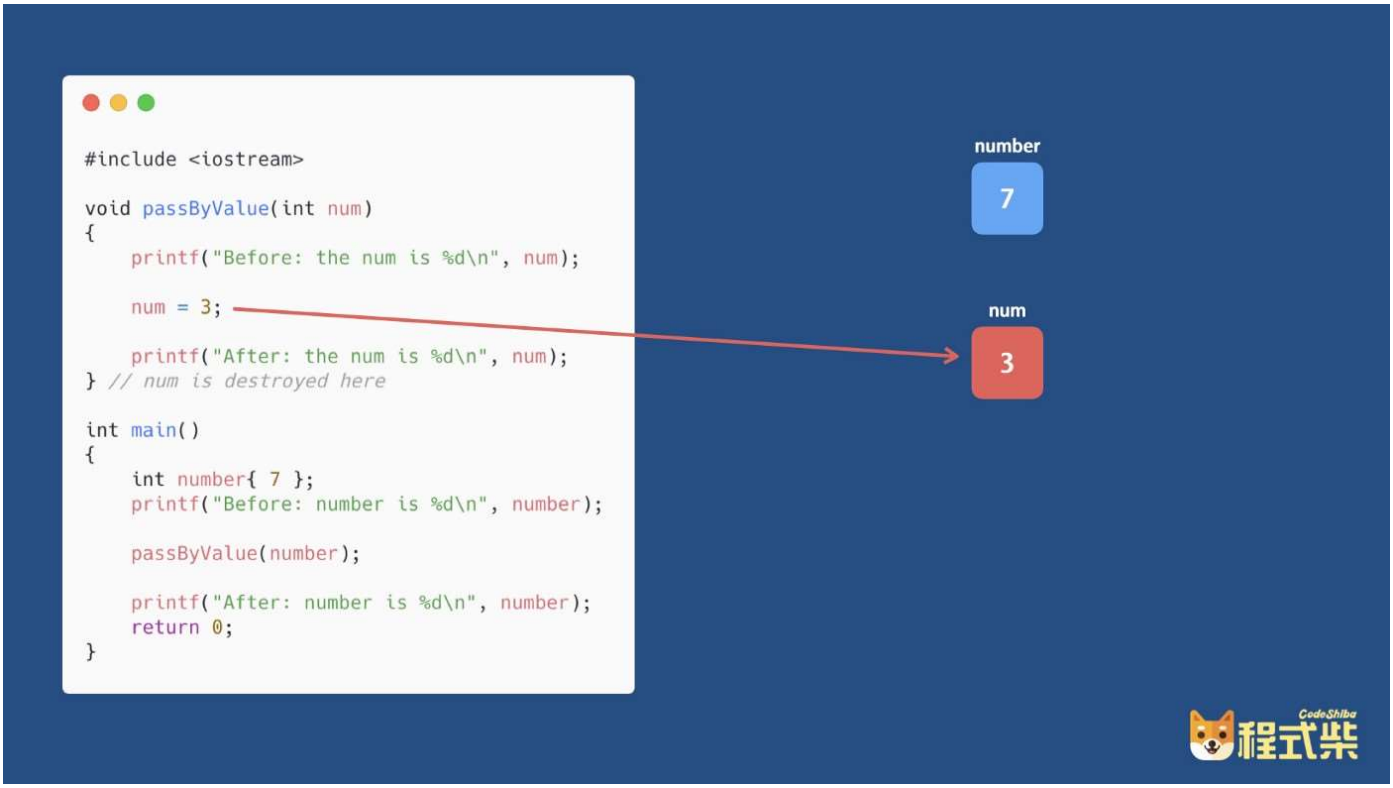
Step 1：宣告 number 為 7




Step 2：呼叫函式時，複製一份給 number 的值複製一份給 num。



step 3：修改 num 的值時，修改的是複製出來的那一份。因此原本的 number 不受影響。




## 2. 用 C++ 理解 Pass by Reference

```
#include <iostream>

void passByReference(int &num)
{
    printf("Before: the num is %d\n", num);

    num = 3;

    printf("After: the num is %d\n", num);
} // num is destroyed here

int main()
{
    int number{ 7 };
    printf("Before: number is %d\n", number);

    passByReference(number);

    printf("After: number is %d\n", number);
    return 0;
}
```

執行結果：

Before: number is 7

Before: the num is 7

跟剛剛不同的是，因為 Pass by Reference 傳的是參照，也就是地址，所以裡外等同於是一個變數。接著我們一樣把記憶體位址印出來。看看是不是真的是同樣的記憶體位址。

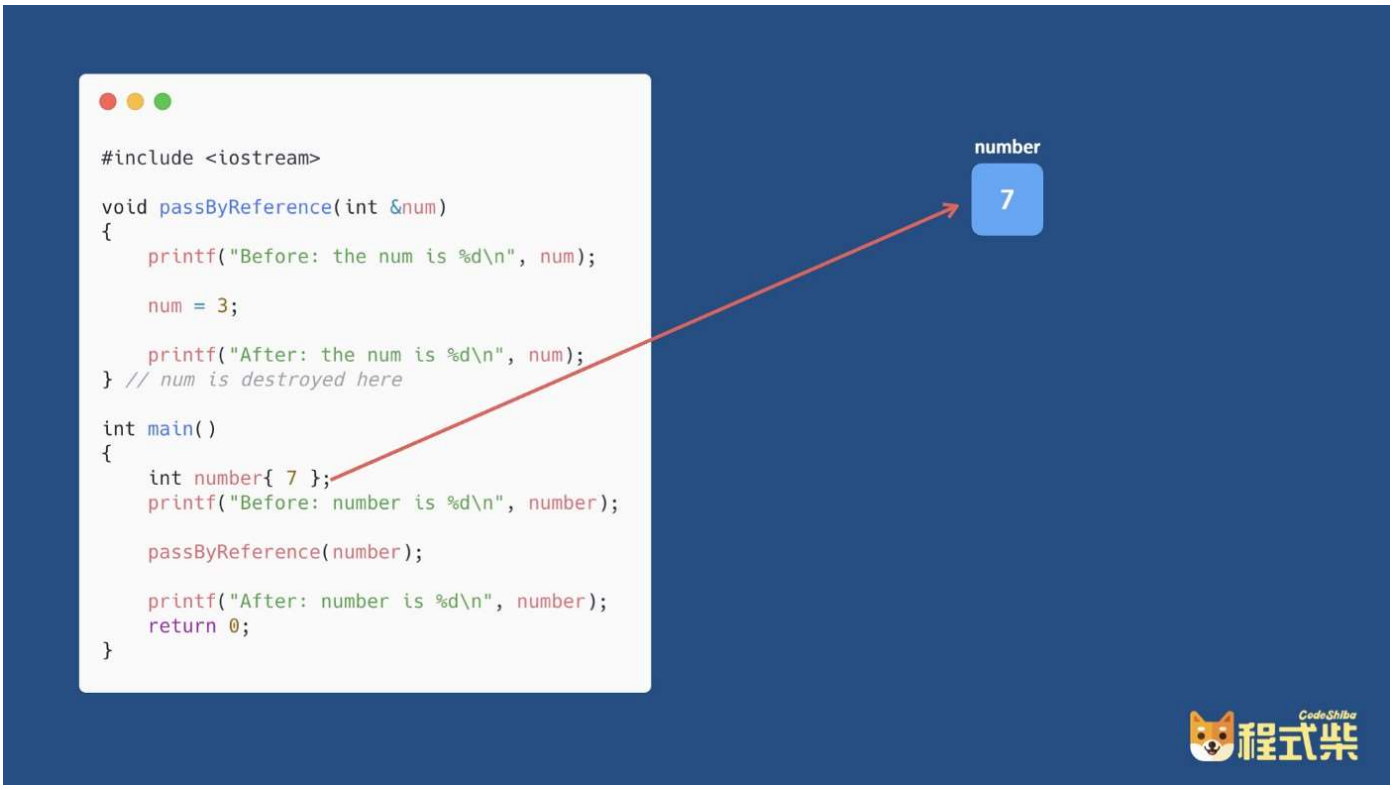
```
#include <iostream>

void passByReference(int &num)
{
    printf("The Address of num is %p\n",(void*)&num);

    num = 3;

    printf("The Address of num is %p\n",(void*)&num);
}

int main()
{
    int number{ 7 };

    printf("The Address of number is %p\n",(void*)&number);

    passByReference(number);

    printf("The Address of number is %p\n",(void*)&number);
    return 0;
}
```

**執行結果：**

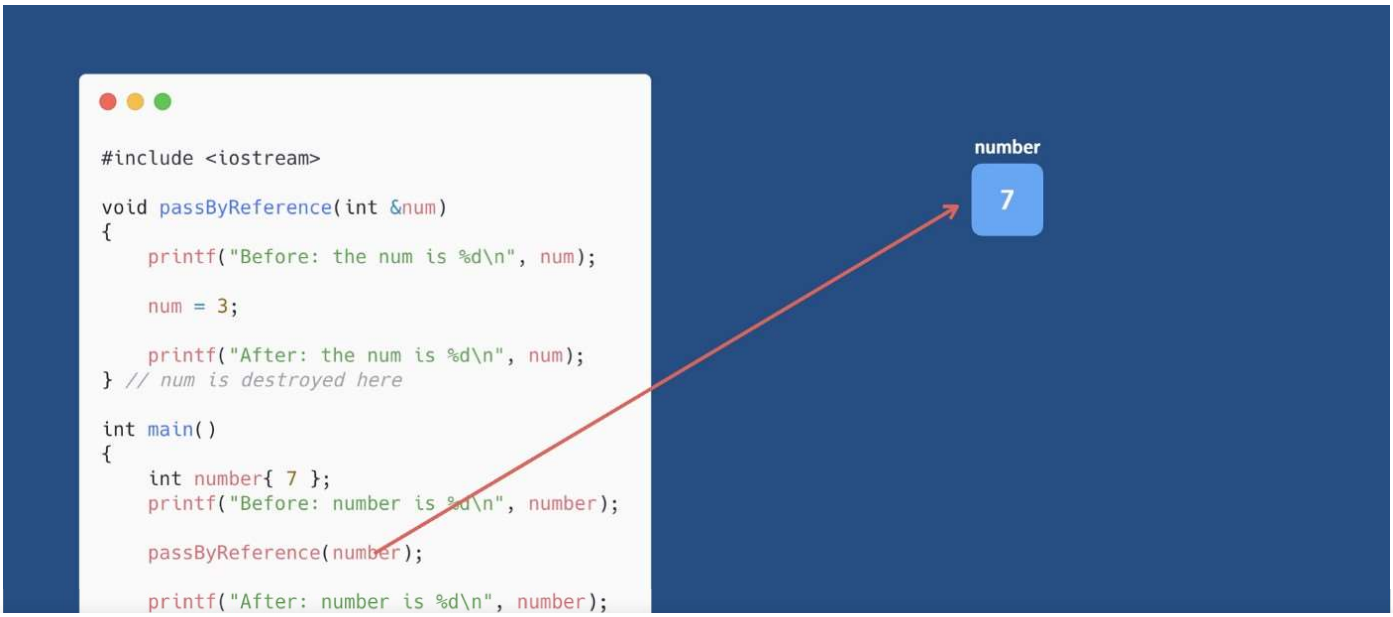
The Address of number is 0x7ffd45470b64
The Address of num is 0x7ffd45470b64
The Address of num is 0x7ffd45470b64

Open in app Get started

Step 1：宣告 number

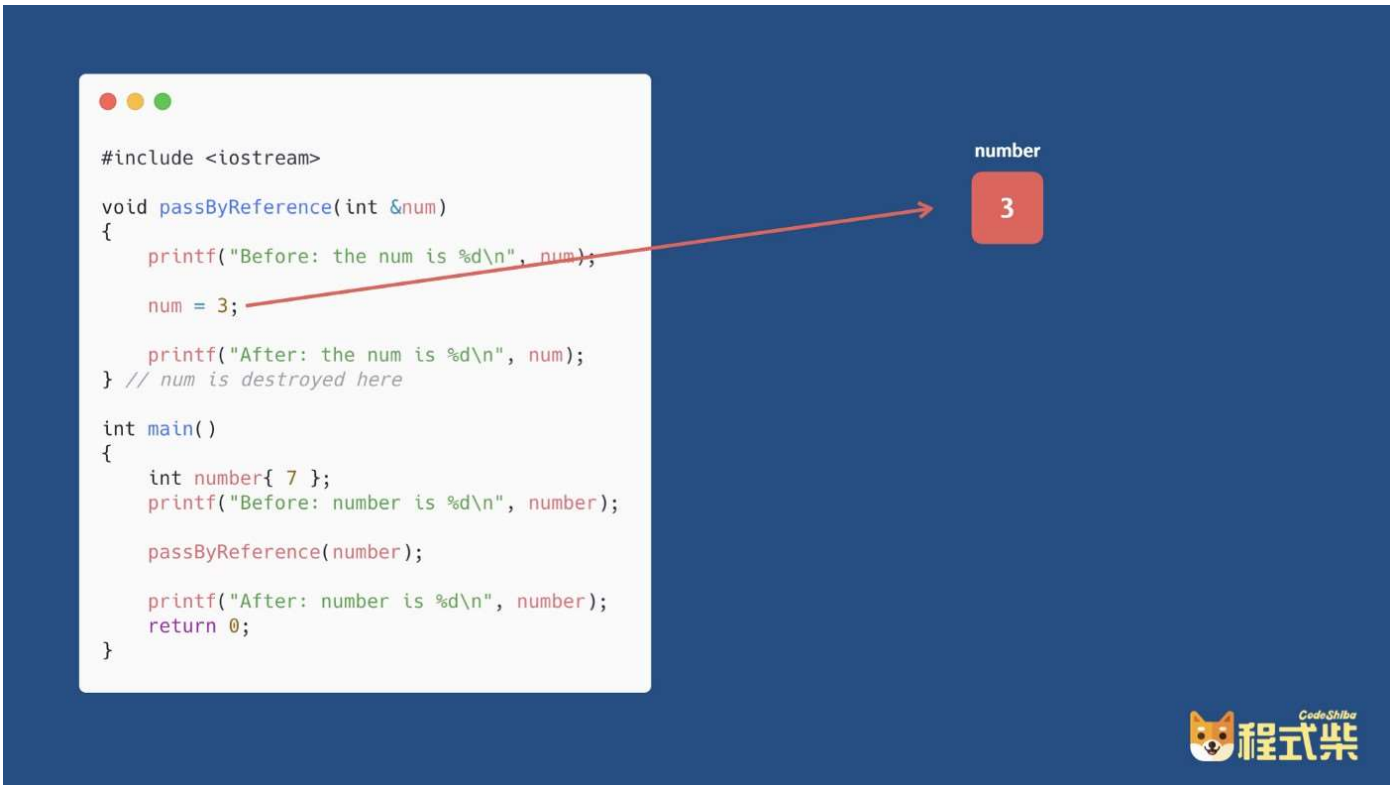



Step 2：呼叫 passByReference 時不會複製一份 number 的值，也就是「共用」同一份的值。

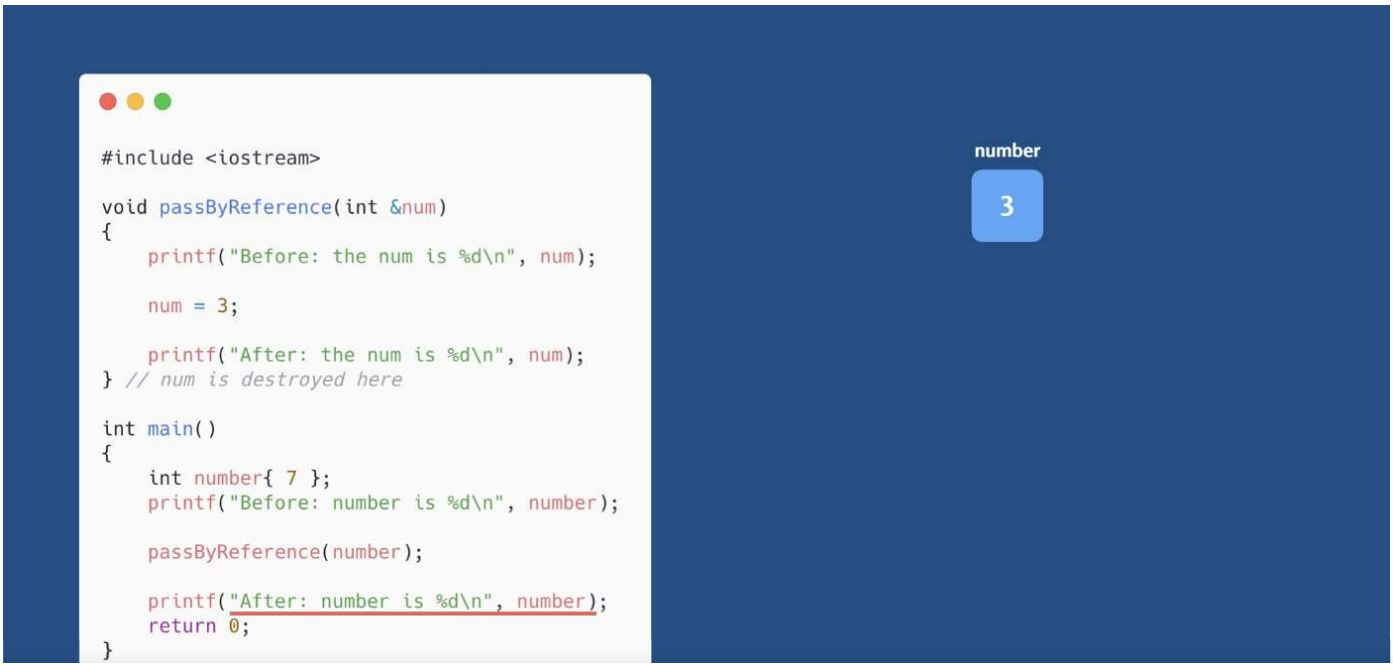


Open in app Get started

Step 3：因為共用一份資料，num 更新時，也更新到了 number。




Step 4: 看所以最後得到的結果是 `After: number is 3` ，因為 number 已經被改變



### 3. 來看看 Python 是不是 Pass by Value

接著我們來看看 Python 是否是 Pass by Value。

```
number = 7
def function_1(num):
  print("Before the function starts, the number is", num)
  num = 1
  print("After the function ends, the number is", num)

print("Before the function starts, the number is", number)
function_1(number)
print("After the function ends, the number is", number)
```

**執行結果:**

Before the function starts, the number is 7
Before the function starts, the number is 7
After the function ends, the number is 1
After the function ends, the number is 7

執行完 function_1 後，發現原本的 number，並**沒有**因為在函式中重新指派而改變成 1，還是維持原本的 7。

看起來跟 C++ 的 Pass Value 很像，接著我們再做一個實驗，把記憶體位置印出來。

```python
number = 7
def function_1(num):
  print("Before assigning, the num is %d", id(num))
  num = 1
  print("After assigning, the num is %d", id(num))

print("Before the function starts,the id is", id(number))
function_1(number)
print("After the function ends, the id is", id(number))
```

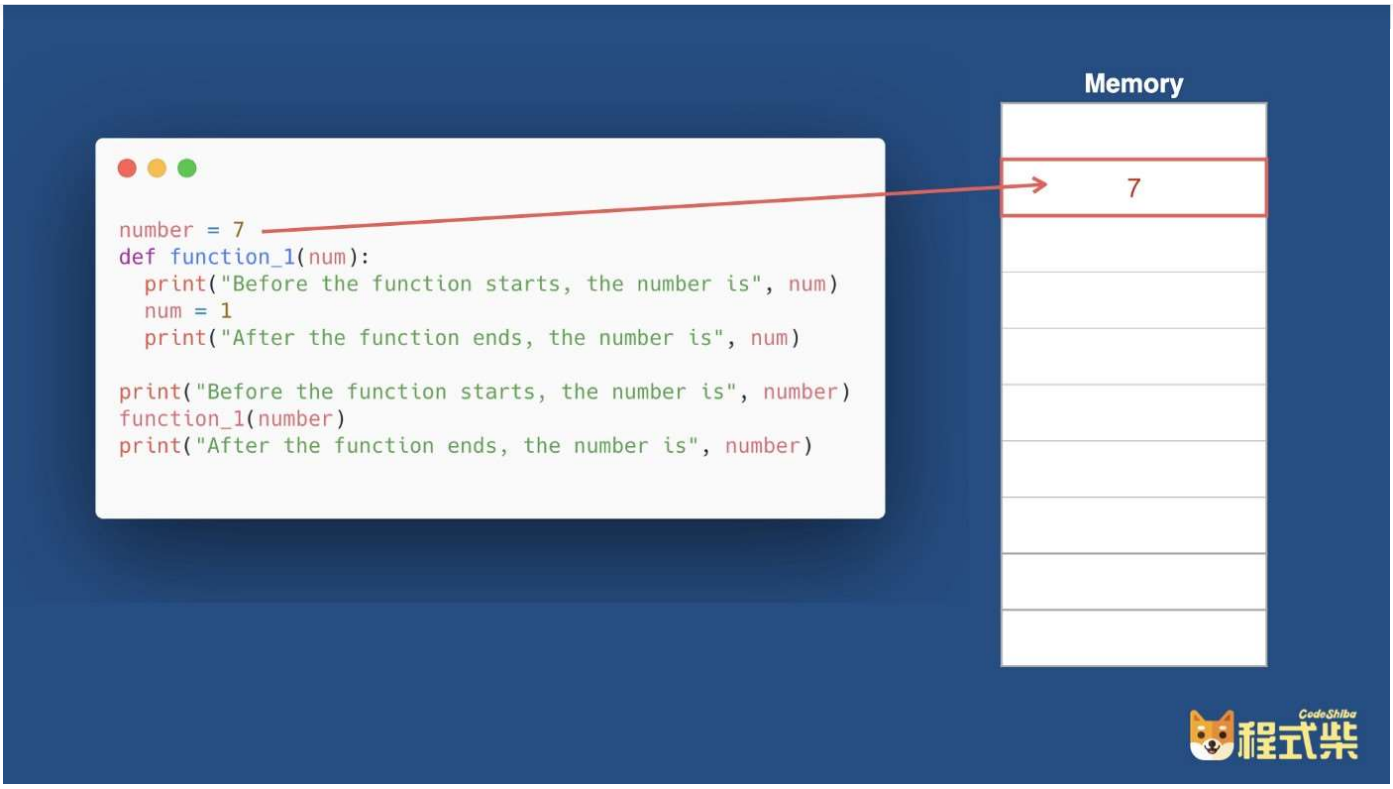
**執行結果：**

Before the function starts, the id is 4321770416
Before assigning, the num is %d 4321770416
After assigning, the num is %d 4321770224
After the function ends, the id is 4321770416

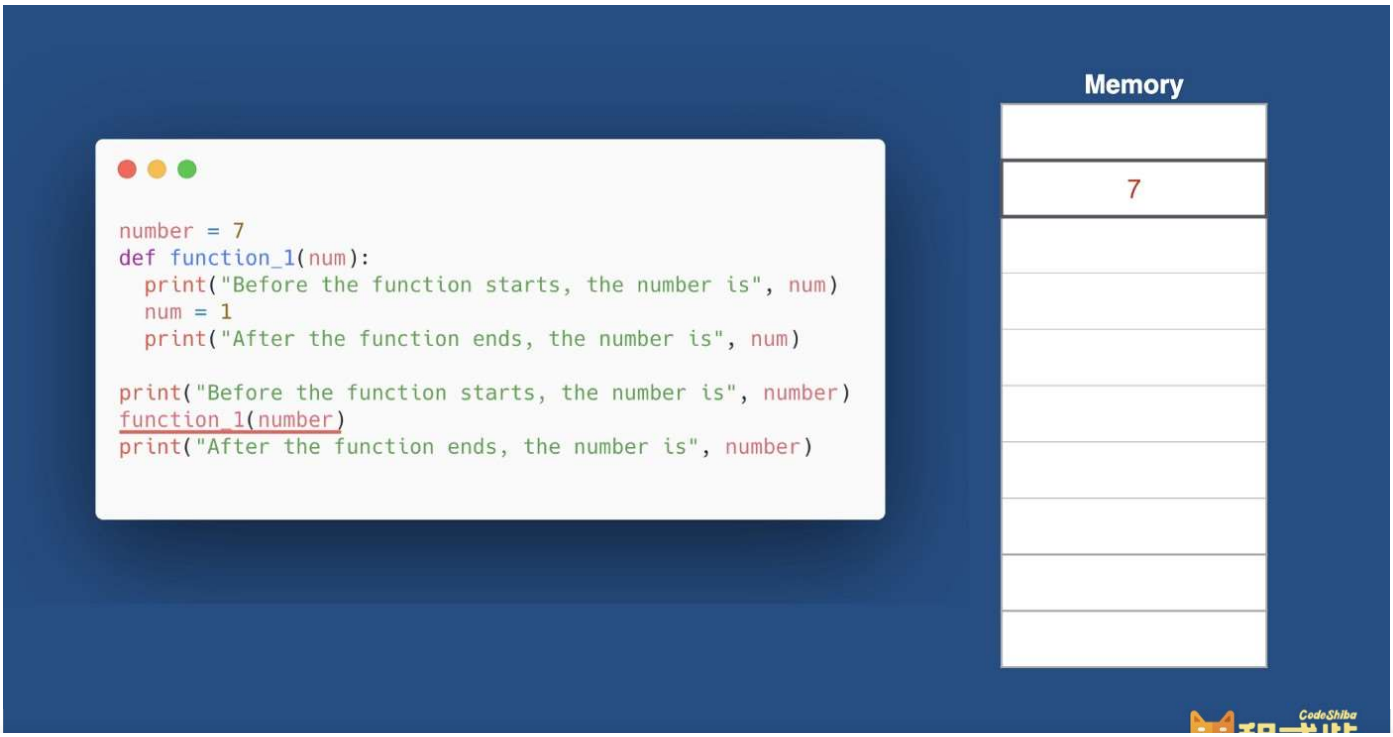
可以發現，行為雖然像是 Pass by Value，但是記憶體位址變化的時候，並不是執行 function_1 函式的當下，而是在指派 num = 1 之後。這是為什麼呢？

Step 1：將 7 指派給變數 number，此時 7 存到記憶體中。

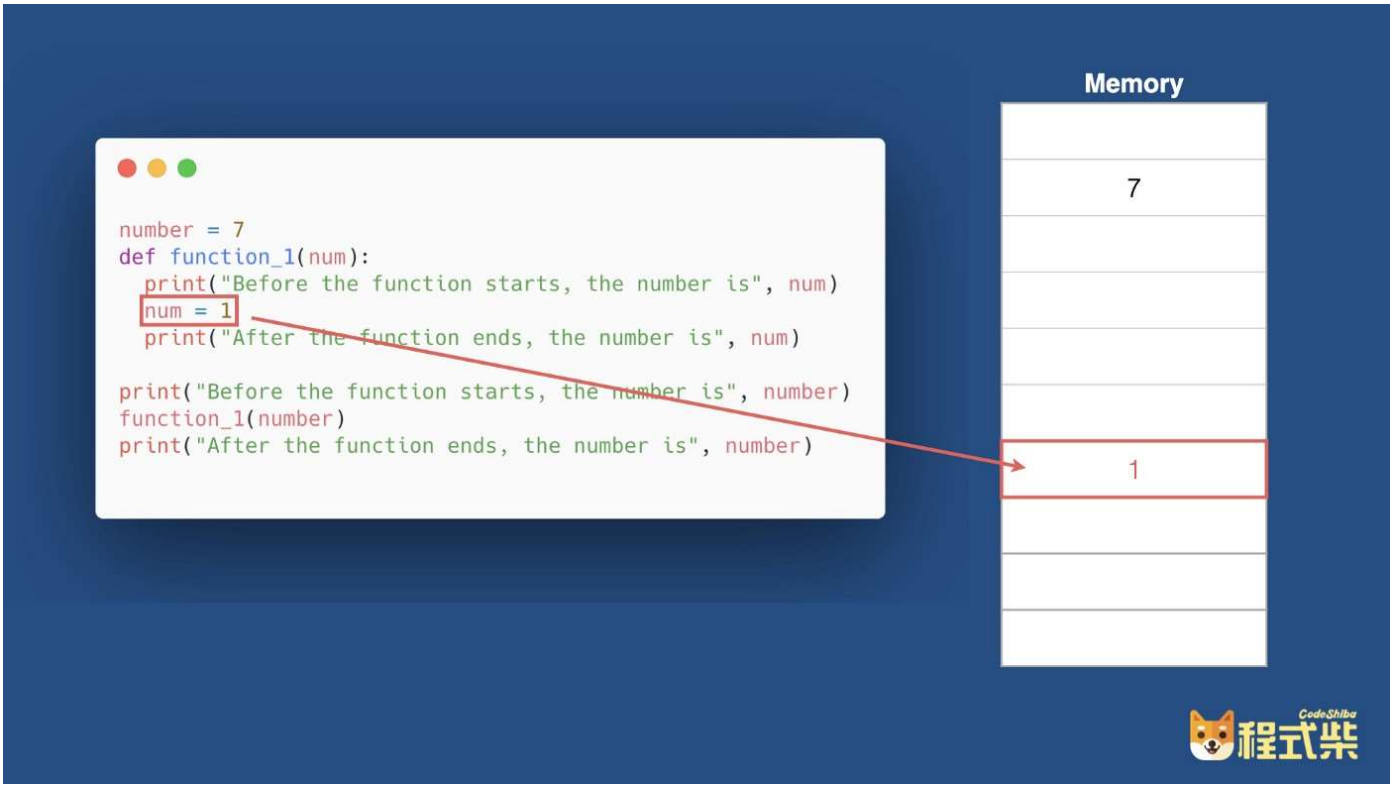
Open in app Get started



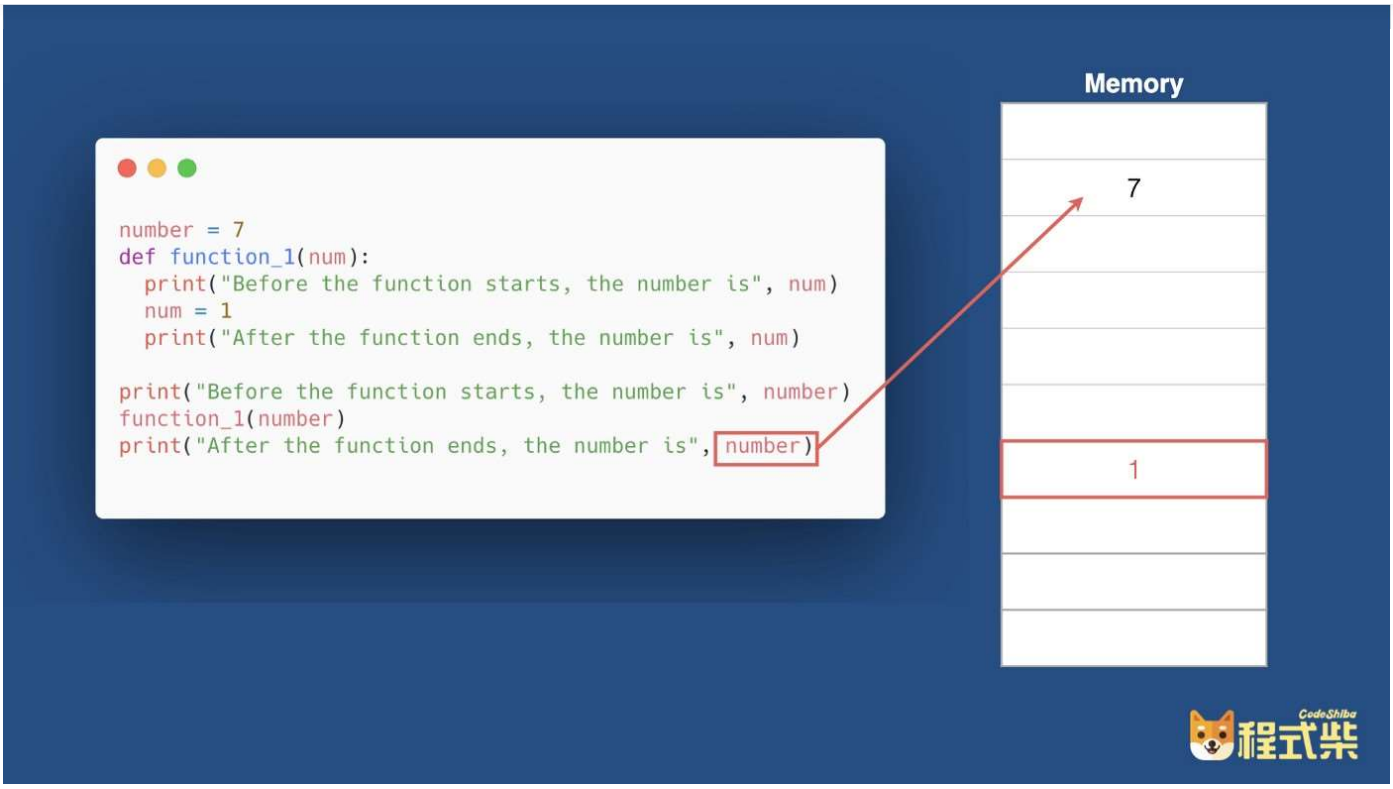

Step 2：呼叫函式 pass_by_value() 的時候，將 7 傳了進去。此時函式並不會複製一份值進去。



Step 3：當執行到 `num=1` 時， num 是整數型別，是 Imuttable Object 不可變物件，值無法改變，因此會新增一塊記憶體來儲存 num。




Step 4 : 所以最後印出時，不會是 1，而是 7 ，因為 number 與 num 已經沒有共用記憶體位置




結論：Python 的 function_1 跟 C++的 Call by Value 最後的結果都有分開的記憶體位址。但是從記憶體位址變化的時機可以看得出來，讓 Python 有此現象的，是 Imuttable Object 不可變的特性，而非 Call by Value。

## 2. 來看看 Python 是不是Pass By Reference

接著我們來看看 Python 是否是 Pass by Reference。

```
object = {
  'name': 'Leo',
  'age': 25
}


def function_2(obj):
  print("before the variable assign, the obj is", obj)
  obj['name'] = '9m88'
  obj['age'] = 20
  print("after the variable assign, the obj is", obj)

print("before the function starts, the object is", object)
function_2(object)
print("after the function ends, the object is", object)
```

**執行結果:**

before the function starts, the object is `{'name': 'Leo', 'age': 25}`

before the variable assign, the obj is `{'name': 'Leo', 'age': 25}`

after the variable assign, the obj is `{'name': '9m88', 'age': 20}`

after the function ends, the object is `{'name': '9m88', 'age': 20}`

哇，外面 object 的內容居然被改變了，跟剛剛的 number 行為完全不一樣，這是怎麼回事？難道這是 Call by Reference？ 可是我們沒有傳地址進去呀。

一樣，我們先把記憶體位址印出來試試看。

```python
object = {
  'name': 'Leo',
  'age': 25
}


def function_2(obj):
  print("Before variable assign, the obj id is", id(obj))
  obj['name'] = '9m88'
  obj['age'] = 20
  print("After variable assign, the obj id is", id(obj))

print("Before variable assign, the object id is", id(object))
function_2(object)
print("After variable assign, the object id is", id(object))
```

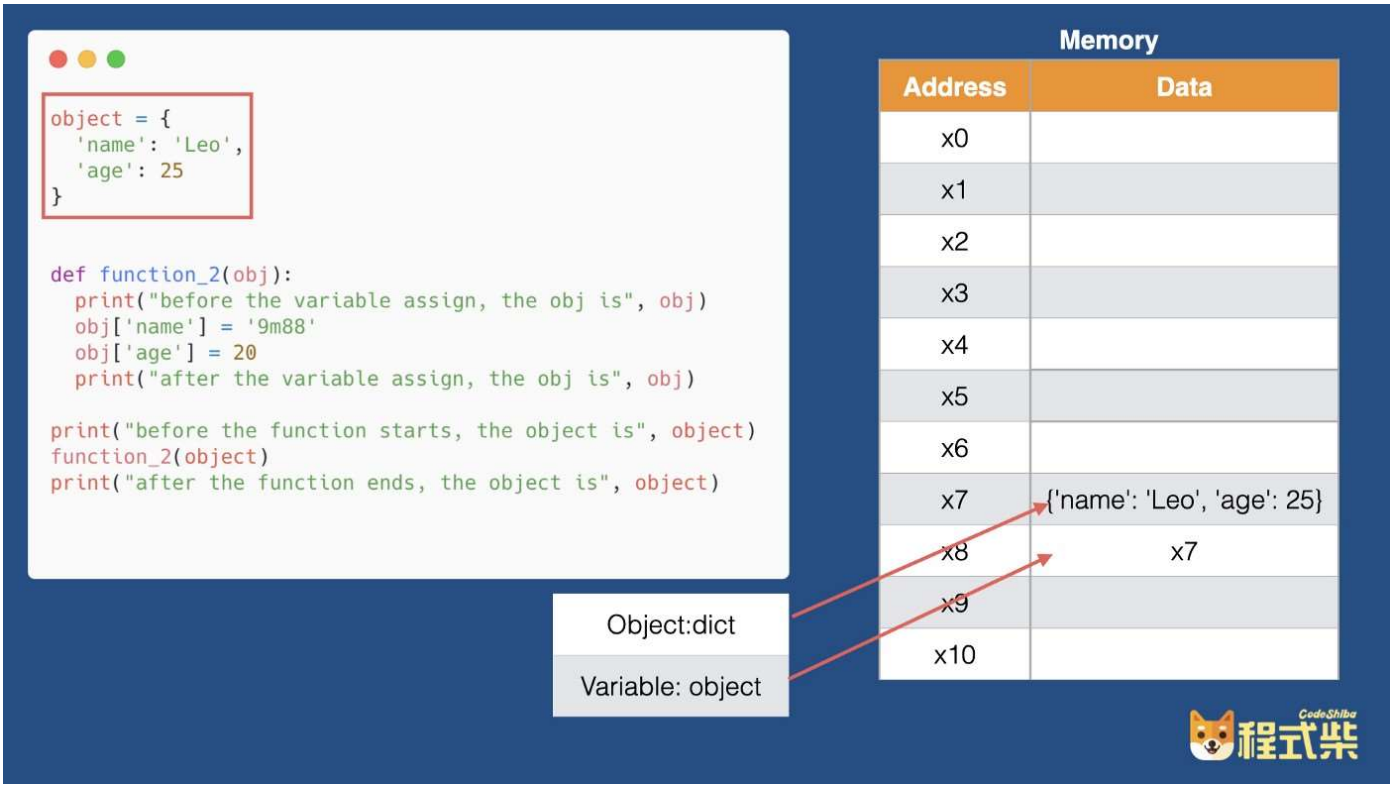
**執行結果：**

Before variable assign, the object id is 4508274560
Before variable assign, the obj id is 4508274560
After variable assign, the obj id is 4508274560
After variable assign, the object id is 4508274560

記憶體位址完全沒有改變，這是為什麼呢？因為剛剛我們使用字典 (dict) 這種資料型別，是屬於可變物件 (Mutable Object)，內容可以改變

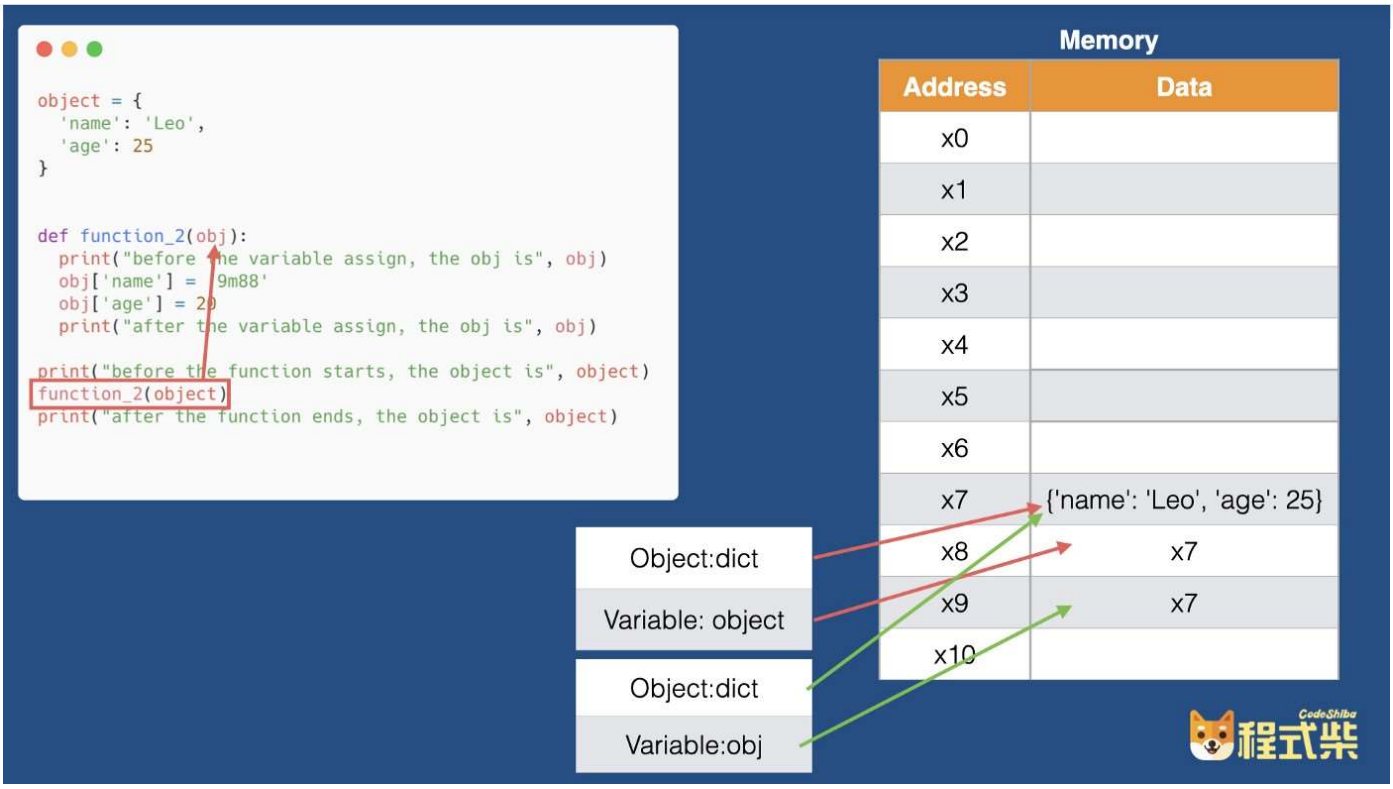
因為字典這種物件是 mutable 可變的，代表這種物件的值是可以改變的，所以想要來看看到底是記憶體怎麼儲存，才是可以改變的

如果不了解 Imuttable 的人可以參考我寫的另外一篇文章。Python — Mutable vs Immutable。裡面講解了 Python 中可變與不可變的概念。

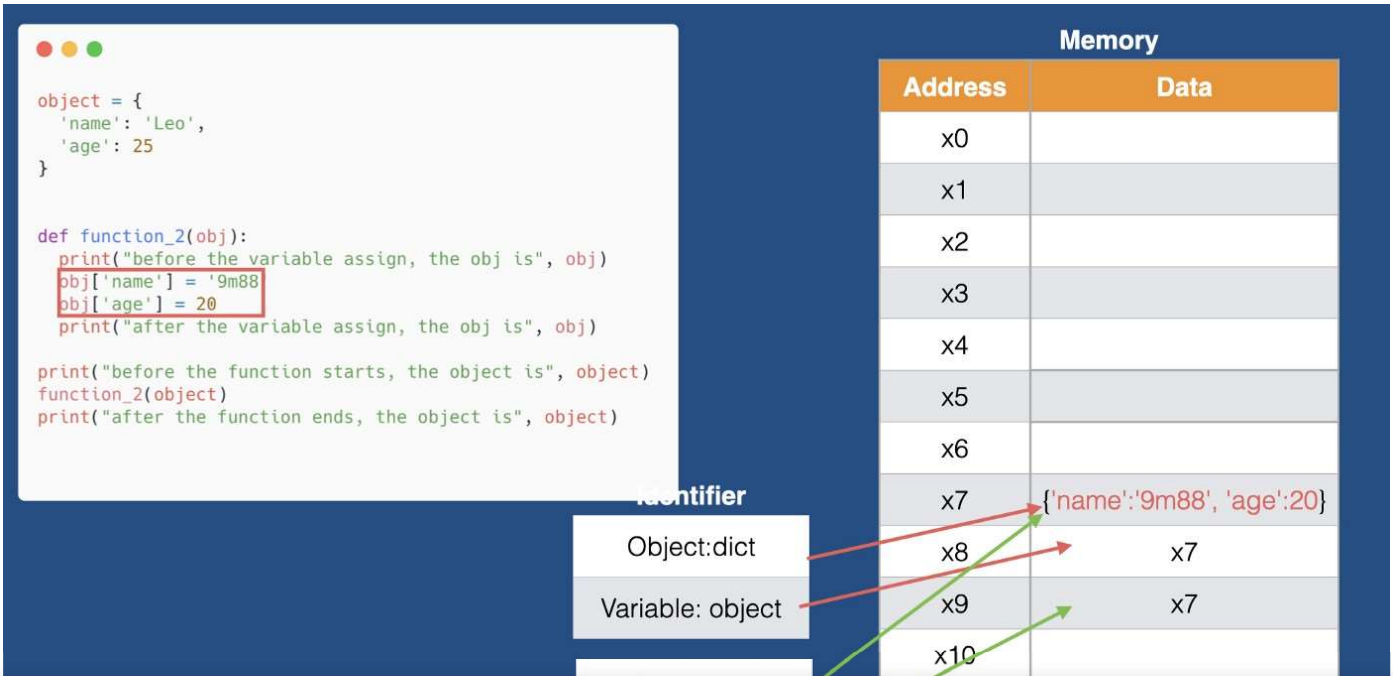



宣告 object 物件的時候，變數 Variable: object，指向一個記憶體位址為 `x8` 資料為 `x7` 。變數本身不直接儲存資料的，只會儲存資料的記憶體位址。而真正的資料儲存在地址為 `x7` 資料為 `{'name': 'Leo', 'age': 25}` 。

Open in app Get started




是呼叫function_2 函式的時候，object 的參照傳進去函式裡面，變數名稱變為 obj 。 變數 obj 將內容的地址，儲存在 x9 記憶體位址，資料為 x7。告訴變數 obj 可以去地址 x7 拿資料，這樣就會找到原本物件內容了。



接著實際執行 pass_by_reference() 函式，因為之前變數 obj 將資料指向跟變數 dict_object 相同的物件，**所以實際修改到的資料，是原本的資料**。最後印出來的時候發現，原始的物件內容已經被改變。

回顧一下 function_2 的執行結果，函式內部的改變會改變 object 外部的內容。

```
執行結果：
before the function starts, the object is {'name': 'Leo', 'age': 25}
before the variable assign, the obj is {'name': 'Leo', 'age': 25}
after the variable assign, the obj is {'name': '9m88', 'age': 20}
after the function ends, the object is {'name': '9m88', 'age': 20}
```

### 3. 來看看 Python 是不是 Pass by Sharing

接下來我們來看範例 3，將範例 2 稍作改變，來看看會發生什麼事情

```
dict_obj = {
  'name': 'Leo',
  'age': 25
}

def function_3(obj):
    print("before assign, the obj is", obj)
    obj = {
      'name': 'Amy',
      'age': 50
    }
    print("after assign, the obj is", obj)
print("before the function starts, the dict_obj is", dict_obj)
function_3(dict_obj)
print("after the function ends, the dict_object is", dict_object)
```

我們將 function_2() 方法改成 function_3()。並且把將方法的內容改成建立新物件的方式。

```
def function_3(obj):
    obj = {
      'name': 'Amy',
      'age': 50
    }
```

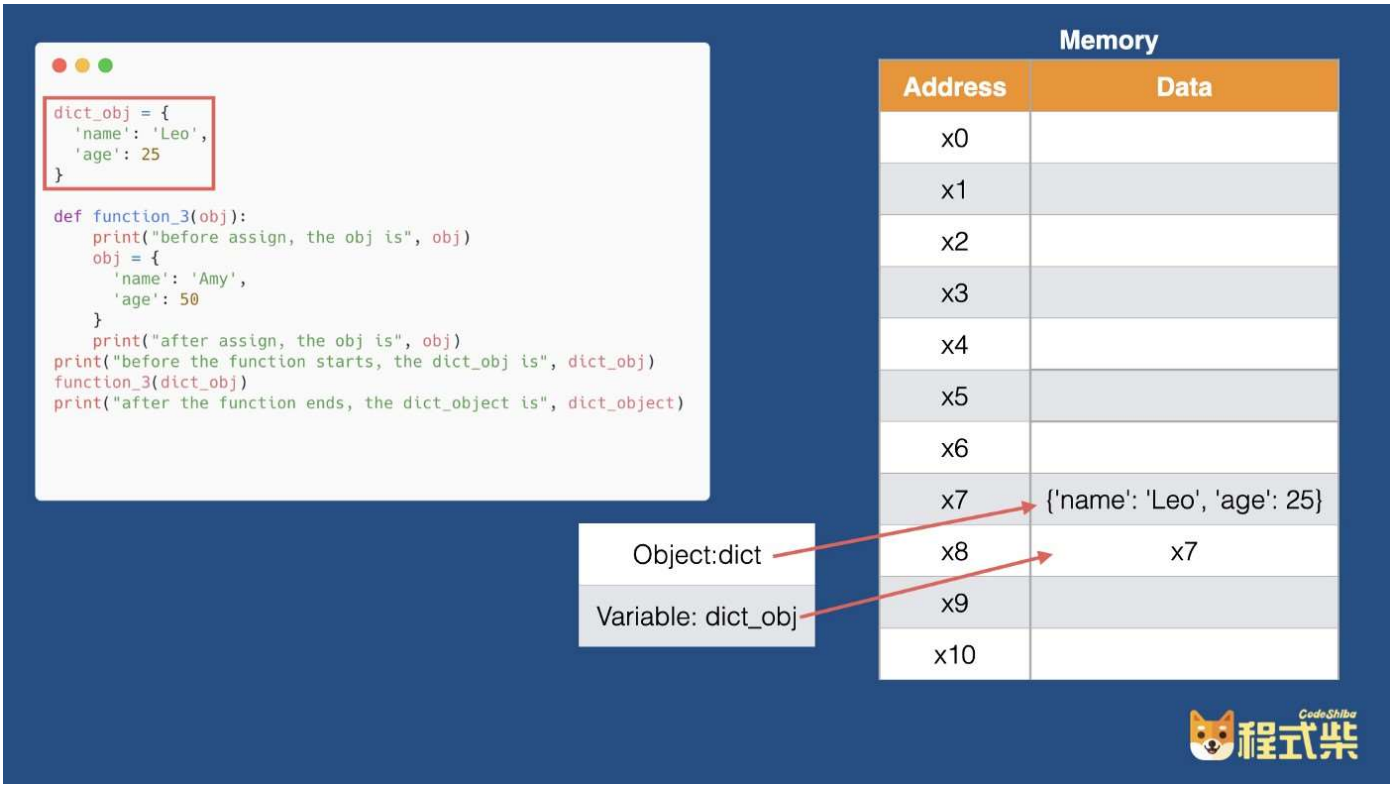
**執行結果:**

before the function starts, the dict_object is `{'name': 'Leo', 'age': 25}`

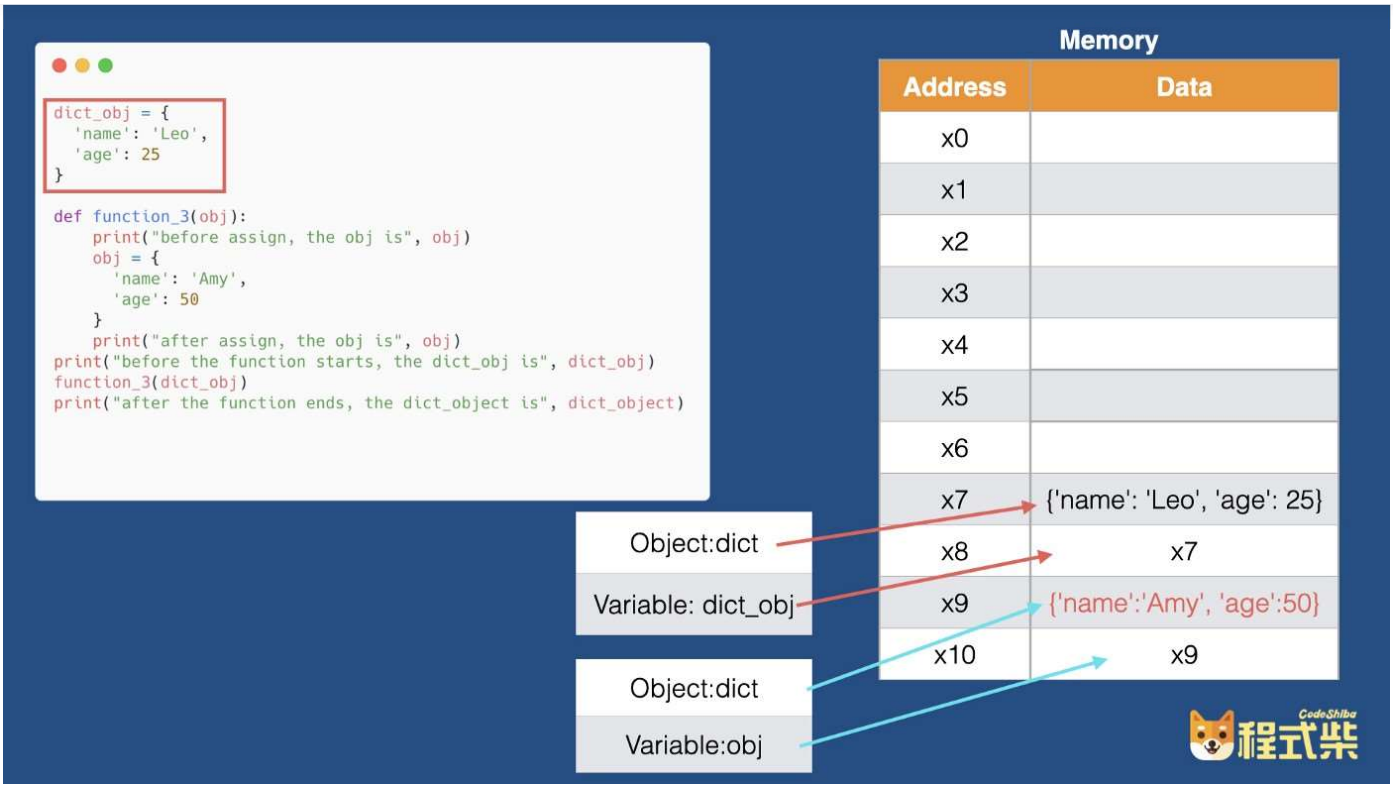
before assign, the obj is `{'name': 'Leo', 'age': 25}`

輸出結果再度變回跟 function_1 的結果一樣，無法改變外部物件，這是為什麼呢？

Step 1: 跟 function_2 完全相同。初始化字典物件內容。




Step 2: 這時就可以看得出來到底發生了什麼事情了，如果是直接使用建立一個新物件的來賦予變數 obj，那麼將不會改變原本變數 dict_obj 的值。




我們可以得出一個結論，在使用可變物件(Muttable Object)的時候，如果重新指派新的物件給它，它就不會保持共用記憶體，而是建立一個心的物件。

### Python 到底是 Pass by Value, Pass Value 還是 Pass Sharing 呢？

> 都不是，Python 是 Pass by Assignment！

我覺得最清楚的是這篇 Stack Overflow 的回答，簡單解釋，引數傳的是物件的參照(Call by Object Reference)，但此 Reference 是由 Pass by Value 的方式實作。

可以歸納出兩個結論：

- 如果你傳遞的可變物件 (Mutable Object)，傳遞的是物件的參照(記憶體位址)，所以可變物件的值是可以改變的，你可以因此修改到原始的物件的內容。
- 可變物件 (Mutable Object) 有一個例外是你重新指派一個新的物件給他，外部的作用域(Scope)將無法得知你做的事情，所以外部作用域會認舊的物件，而不是新指派的

3179

Arguments are passed by assignment. The rationale behind this is twofold:

1. the parameter passed in is actually a *reference* to an object (but the reference is passed by value)
2. some data types are mutable, but others aren't

So:

- If you pass a *mutable* object into a method, the method gets a reference to that same object and you can mutate it to your heart's delight, but if you rebind the reference in the method, the outer scope will know nothing about it, and after you're done, the outer reference will still point at the original object.
- If you pass an *immutable* object to a method, you still can't rebind the outer reference, and you can't even mutate the object.

To make it even more clear, let's have some examples.

https://stackoverflow.com/questions/986006/how-do-i-pass-a-variable-by-reference

最後再回到官方文件的定義，確認一下 Python 真的是 Pass by Assignment，打完收工。

## How do I write a function with output parameters (call by reference)?

Remember that arguments are passed by assignment in Python. Since assignment just creates references to objects, there's no alias between an argument name in the caller and callee, and so no call-by-reference per se. You can achieve the desired effect in a number of ways.

1. By returning a tuple of the results:

```
>>> def func1(a, b):
...     a = 'new-value'        # a and b are local names
...     b = b + 1              # assigned to new objects
...     return a, b            # return new values
...
>>> x, y = 'old-value', 99
>>> func1(x, y)
```

https://docs.python.org/3/faq/programming.html#how-do-i-write-a-function-with-output-parameters-call-by-reference

## 小結

這篇文章撰寫了第二次，包括圖片和觀念骨幹全部重新寫過一遍。原因是第一遍寫得觀念錯誤太多，熱愛分享，但是不能把錯誤的觀念帶給新手。所以又花了一個週末彙整資料繪製圖表，希望大家會喜歡。

## 系列文章：

- Python 版 一次搞懂 Pass by Value, Pass by Reference 與 Pass by Sharing << 你目前閱讀到這篇，
- 透過 C / C++ 搞懂 Pass by Value 與 Pass by Reference
- 為什麼 Java 的世界裡只有 Pass by Value？
- JS 版 一次搞懂 Pass by Value, Pass by Reference 與 Pass by Sharing

## 參考資料：


- https://simplernerd.com/python-list-tuple/
- https://www.youtube.com/watch?v=1YFss_4B_o4
- https://www.learncpp.com/cpp-tutorial/passing-arguments-by-value
- https://www.learncpp.com/cpp-tutorial/passing-arguments-by-reference
- Pass by value vs. pass by reference

Open in app Get started

歡迎訂閱，只會發送我覺得寫得不錯的文章。

Your email

Subscribe

By signing up, you will create a Medium account if you don't already have one. Review our Privacy Policy for more information about our privacy practices.

About Help Terms Privacy

Get the Medium app

Download on the App Store

GET IT ON Google Play