@ZenAku1072

Pythonは値渡しなのか？参照渡しなのか？[結論] 参照の値渡し

Python 参照渡し 値渡し 参照の値渡し

最終更新日 2023年10月11日 投稿日 2023年10月10日

Pythonは値渡しなのか？参照渡しなのか？

自分がプログラミングしていた時，Pythonが値渡しなのか，参照渡しなのか気になったので調べることにした．
すると，「値渡し」だったり，「参照渡し」だったり，ましてや「参照の値渡し」など，言っていることが人によってまちまちなので，実際に動かして検証してみることにした．

> 調べたサイト
> Pythonは「値渡し」です！「参照渡し」という誤解はなぜ生じるのか？
> Python に参照渡しは存在しない話
> 第16回.Pythonの引数は参照渡しだが・・・
> Pythonの関数の引数は参照渡しなのか？参照の値渡し
> 値渡しと参照渡しと参照の値渡しと
> 【Pythonプログラミング入門】ミュータブル・イミュータブルを解説！～VTuberと学習～ 【初心者向け】

値渡し・参照渡しとは

値渡し・参照渡しはプログラミングで引数をもつ関数への渡し方の種類のこと.
関数の引数で，実際に渡す値・変数を**実引数**，関数側で定義する変数を**仮引数**という.

- **値渡し**
  実引数の値を **コピーして** 渡す方法
  値をコピーして渡すため，元の値には**影響しない**
- **参照渡し**
  実引数の~~値が存在している~~ **変数の場所(メモリアドレス)** を渡す方法
  ~~値~~ 変数の場所を参照するため，元の ~~値~~ 変数に**影響する**

参照渡しにはさらに2種類にわかれる.

- **参照の値渡し**
  ~~参照先(~~ 値のメモリアドレス ~~)~~ をコピーして渡す
  元の値に**影響する時がある**
- **参照の参照渡し** (追記:10/11/2023 あまり呼ばないらしい)
  ~~参照先(~~ 変数のメモリアドレス ~~)~~ そのものを渡す
  元の値に**必ず影響する**

一般的には「参照の参照渡し」を**参照渡し**と呼び,
「参照の値渡し」はそのまま**参照の値渡し**と呼ぶ.

以下のサイトにわかりやすく説明が乗っているので気になる人はぜひ参考にしてください.

(追記: 10/11/2023)
かなりの誤解があったので改めて記述.
まず変数は2種類存在.

- **値型変数**: 値そのものを格納
- **参照型変数**: 値が格納されているアドレス(参照)を格納

それぞれに値渡し，参照渡しがある.

引数へ渡し方

	値型変数	参照型変数
値渡し	値型値渡し	参照型値渡し
参照渡し	値型参照渡し	参照型参照渡し

実際に動かしてみた

Pythonには「**メモリのどこに変数←値が格納されているか**」を表示する***id関数***があるので，以下のサンプルプログラムを使って実際に動かしてみた．
実行は以下の環境で行った．

- **OS**: Window11
- **Python**: 3.12.0

> 以下のプログラムはすべてスクリプトファイルとして記述した時の動作になります．
> Jupyter notebookではtest1の実行だけ全く異なる挙動をします．
> 参考にする場合は，必ずスクリプトファイルとして記述し実行して下さい．

test1

値渡し・参照渡しの検証をする前に，id関数とPython特有の挙動を確認する．
Pythonにはミュータブルとイミュータブルなオブジェクトがある．
詳しくは説明しないが，次のように覚えておけばいい．

Pythonオブジェクトの種類

- **イミュータブル**: 不可変なオブジェクト
- **ミュータブル**: 可変なオブジェクト

今回はイミュータブルに intオブジェクト , ミュータブルに listオブジェクトを使う.
イミュータブルだと値が変わると，アドレスが変わり，値そのものは常に同じ場所にあり続ける.
ミュータブルだと値が変わっても，アドレスは変わらないが，オブジェクトが変わると違う場所を参照する.

サンプルプログラム

```
# イミュータブル
print("int")
print(f"int 1 id: {id(1)}")
print(f"int 2 id: {id(2)}")
print(f"check: {id(1) == id(2)}")

t1 = 1
print(f"t1 id: {id(t1)}")
print(f"check: {id(t1) == id(1)}")
t1 = 2
print(f"t1 id: {id(t1)}")
print(f"check: {id(t1) == id(2)}")
t1 = 1
print(f"t1 id: {id(t1)}")
print(f"check: {id(t1) == id(1)}")

# ミュータブル
print("list")
print(f"list [1] id: {id([1])}")
print(f"list [2] id: {id([2])}")
print(f"check: {id([1]) == id([2])}")

l1 = [1]
print(f"l1 id: {id(l1)}")
print(f"check: {id(l1) == id([1])}")
l1 = [2]
print(f"l1 id: {id(l1)}")
print(f"check: {id(l1) == id([2])}")
l1 = [1]
print(f"l1 id: {id(l1)}")
print(f"check: {id(l1) == id([1])}")
```

実行結果

```
int
int 1 id: 140733021866424
int 2 id: 140733021866456
check: False

t1 id: 140733021866424
check: True
t1 id: 140733021866456
check: True
t1 id: 140733021866424
check: True

list
list [1] id: 2813160118464
list [2] id: 2813160118464
check: True

l1 id: 2813160118464
check: False
l1 id: 2813160116480
check: False
l1 id: 2813160118464
check: False
```

test2

引数として渡された変数は関数内の初めまで同じ．(参照渡しのような動作)
代入が行われると違う場所に変わった．
関数外の変数には影響がなかった．(値渡しのような動作)
動作はイミュータブル，ミュータブルどちらでも同じだった．

サンプルプログラム

```
def func2_1(arg):
    """
    変数の値と格納されているアドレスを表示する関数(イミュータブル用)
    """
    print("===func2_1===")
    print(f"before id: {id(arg)}, arg = {arg}")
    arg = 2
    print(f"after id: {id(arg)}, arg = {arg}")
```

```
    print("===return===")


def func2_2(arg):
    """
    変数の値と格納されているアドレスを表示する関数(ミュータブル用)
    """
    print("===func2_2===")
    print(f"before id: {id(arg)}, arg = {arg}")
    arg = [2]
    print(f"after id: {id(arg)}, arg = {arg}")
    print("===return===")

# イミュータブル
val1 = 1
print(f"val1 id: {id(val1)}, val1 = {val1}")
func2_1(val1)
print(f"val1 id: {id(val1)}, val1 = {val1}")

# ミュータブル
val2 = [1]
print(f"val2 id: {id(val2)}, val2 = {val2}")
func2_2(val2)
print(f"val2 id: {id(val2)}, val2 = {val2}")
```

実行結果

```
イミュータブル
val1 id: 140733021866424, val1 = 1
===func2_1===
before id: 140733021866424, arg = 1
 after id: 140733021866456, arg = 2
===return===
val1 id: 140733021866424, val1 = 1

ミュータブル
val2 id: 2523838666944, val2 = [1]
===func2_2===
before id: 2523838666944, arg = [1]
 after id: 2523840213696, arg = [2]
===return===
val2 id: 2523838666944, val2 = [1]
```

test3

一番勘違いが起こるミュータブルなオブジェクトでの動作を確認する．
関数内で引数に代入や演算を行ったときは，外に影響は出なかった．(test2と同じ)
しかし，引数のメソッドを使うと，外にも影響がでた．

サンプルプログラム

```
def func3_1(arg1, arg2):
    """
    変数を足し算(結合)する関数
    """
    print("===func3_1===")
    print(f"before id: {id(arg1)}, arg1 = {arg1}")
    arg1 = arg1 + arg2
    print(f"after id: {id(arg1)}, arg1 = {arg1}")
    print("===return===")


def func3_2(arg1, arg2):
    print("===func3_2===")
    print(f"before id: {id(arg1)}, arg1 = {arg1}")
    arg1.append(arg2)
    print(f"after id: {id(arg1)}, arg1 = {arg1}")
    print("===return===")


val1 = [1]
val2 = [2]
print(f"val1 id: {id(val1)}, val1 = {val1}")
func3_1(val1, val2)

print(f"val1 id: {id(val1)}, val1 = {val1}")
print("add")
val1 = val1 + val2

print(f"val1 id: {id(val1)}, val1 = {val1}")
func3_2(val1, val2)

print(f"val1 id: {id(val1)}, val1 = {val1}")
print("append")
val1.append(3)
print(f"val1 id: {id(val1)}, val1 = {val1}")
```

実行結果

```
val1 id: 2255341541568, val1 = [1]
===func3_1===
before id: 2255341541568, arg1 = [1]
 after id: 2255343047232, arg1 = [1, 2]
===return===
val1 id: 2255341541568, val1 = [1]
add
val1 id: 2255343047232, val1 = [1, 2]
===func3_2===
before id: 2255343047232, arg1 = [1, 2]
 after id: 2255343047232, arg1 = [1, 2, [2]]
===return===
val1 id: 2255343047232, val1 = [1, 2, [2]]
append
val1 id: 2255343047232, val1 = [1, 2, [2], 3]
```

考察

-値渡しだと仮定する-

test3で関数を呼び出してメソッドを使ったときに影響が出ている説明ができない.

-参照渡しだと仮定する-

test2で関数内の足し算で影響を受けていない説明ができない.

-参照の値渡しだと仮定する-

test2では，代入でイミュータブルでは値，ミュータブルではオブジェクト自体が変ったため，test1の挙動から影響が出ないことが説明できる.
test3では，func3_1では代入がしているため，test2同様に影響が出なかったが，func3_2ではオブジェクト自身の関数(メソッド)を使ったため，元のオブジェクトに影響がでた.

結論

以上3つのtestプログラムから自分が出した結論は，Pythonは**参照の値渡し**であるといえる.