@starfieldKt (Keita Hoshino)

Pythonで変数に変数代入したら思っていたのと違ったので値渡しと参照渡し等について整理した【ゼロからPython勉強してみる】

Python 初心者 備忘録

最終更新日 2024年05月15日 投稿日 2022年05月18日

はじめに

これまでの記事でとりあえずPythonを勉強する準備が整い、四則演算だとか変数に値を代入してみたりしてたのですが、その中で以下のような状況に出くわし、なんでか調べたので備忘録としてメモします。

int型のデータを変数に代入してた時

まずこの状況、`b=a`で`b`に`a`の値である`1`を与えて、その後`a`の値を変更しようが`b`の値が連動して変わらないんだなそういうもんなんだなと思ってました。

パターン1

```
>>>a = 1 #変数aに1という値を代入
>>>b = a #変数aに変数bの値を代入

#aもbも値は1
>>>print(a)
1
>>>print(b)
1

>>>a = 0 #変数aに0という値を代入

>>>print(a)
0
>>>print(b)
1           #bの値は1のまま
```

list型のデータを変数に代入してみた時

しかし、list型のデータで同じようなことをやってみたら、変数`b`の値も一緒に代わっていたのです。

パターン2

```
>>> a=[1,2]  #変数aにリスト型のデータ[1,2]を代入
>>> b=a      #変数bに変数aの値を代入
```

```
#aもbも値は同じ
>>> print(a)
[1, 2]
>>> print(b)
[1, 2]

>>> a.append(3) #変数aの値の末尾に3を加える

>>> print(a)    #変数aの値は[1, 2, 3]になっている
[1, 2, 3]

>>> print(b)    #変数bの値も[1, 2, 3]になってる
[1, 2, 3]
```

なぜこんなことが起きるのか

まず勘違いしていた

※コメントを受け内容を修正しました。

なぜパターン1とパターン2で先ほどのようなことが起きたのか、そもそもパターン1とパターン2では`b`に`a`を代入した後に`a`に対して行った内容が異なっていた。
パターン1では`a = 0`とし、パターン2では`a.append(3)`としていて当時の私は「どっちも変数に代入した値に変化を加えてるんだろう」と同列に考えていたのですが、パターン1では「`1`というオブジェクトを参照していたが、`0`というオブジェクトを参照させる」、パターン2では「aが参照している`[1,2]`というオブジェクトの末尾に`3`を加える」といった感じで全然違うことをしてたのです。

パターン2の際に`a.append(3)`ではなく`a = [1,2,3]`とした場合ならパターン1同様に参照する物を変えているのでaとbで値が異なります。

パターン3

```
>>> a = [1, 2]  # 変数aに [1, 2] という値を代入
>>> b = a       # 変数aに変数bの値を代入

#aもbも値は [1, 2]
>>> print(a)
[1, 2]
>>> print(b)
[1, 2]

>>> a = [1, 2, 3]  # 変数aに[1, 2, 3]という値を代入

>>> print(a)
[1, 2, 3]
>>> print(b)
[1, 2]          # 変数bの値は [1, 2] のまま
```

じゃあ逆に、パターン1の時にパターン2のように変数が参照している値を直接いじれないのかと言うとそれはできません。
パターン1で変数が参照しているint型のデータはその値を変更不可能とされているからです。このように編集不可能なもことを「イミュータブル」、逆にlist型のように編集できるのを「ミュータブル」といいます。

イミュータブルなデータ型

pythonでイミュータブルなデータ型は代表として`int型（整数）`、`bool型（真偽値）`、`str型（文字列）`、`tuple型（タプル）`等がある。

ミュータブルなデータ型

`ミュータブル`は`イミュータブル`の反対で`変更可能`ということである。
Pythonでの代表的なミュータブルなデータ型としては`list型（リスト）`、`dict型(辞書)`などがある。

変数と値の関係

この記事は再勉強後に編集で修正しているので、上記でなんでこんなことが起きたのかについては大体解決しているのですが、「参照するってなに！参照先が変わるってどういうことよ！」と最初に記事を書いていた時にはなっていたので、整理した記事は下記に残しておきます。

Pythonではどうなっているかはさておき、変数に値を渡すときにどのような渡し方があるのか調べてみたところ「値渡し」がどうとか「参照渡し」がどうとかPythonは「参照の値渡し」だとか書かれていました。
とりあえずそれらの単語が何なのかを整理してみます。

こんな感じのメモリのイメージ図を使用して整理しますが、あくまで自分で理解しやすくするための単純化したイメージで実際のメモリの読み書きの順序等とは異なることはご理解下さい。番地についても桁数も数字も仮にこうしているだけです。

メモリ

変数	番地	値
	０００１	
	０００２	
	０００３	
	０００４	
	０００５	
	０００６	
	０００７	
	０００８	
	０００９	
	００１０	
	００１１	
	００１２	
	００１３	
	００１４	
	００１５	

「値渡し」と「参照渡し」と「参照の値渡し」

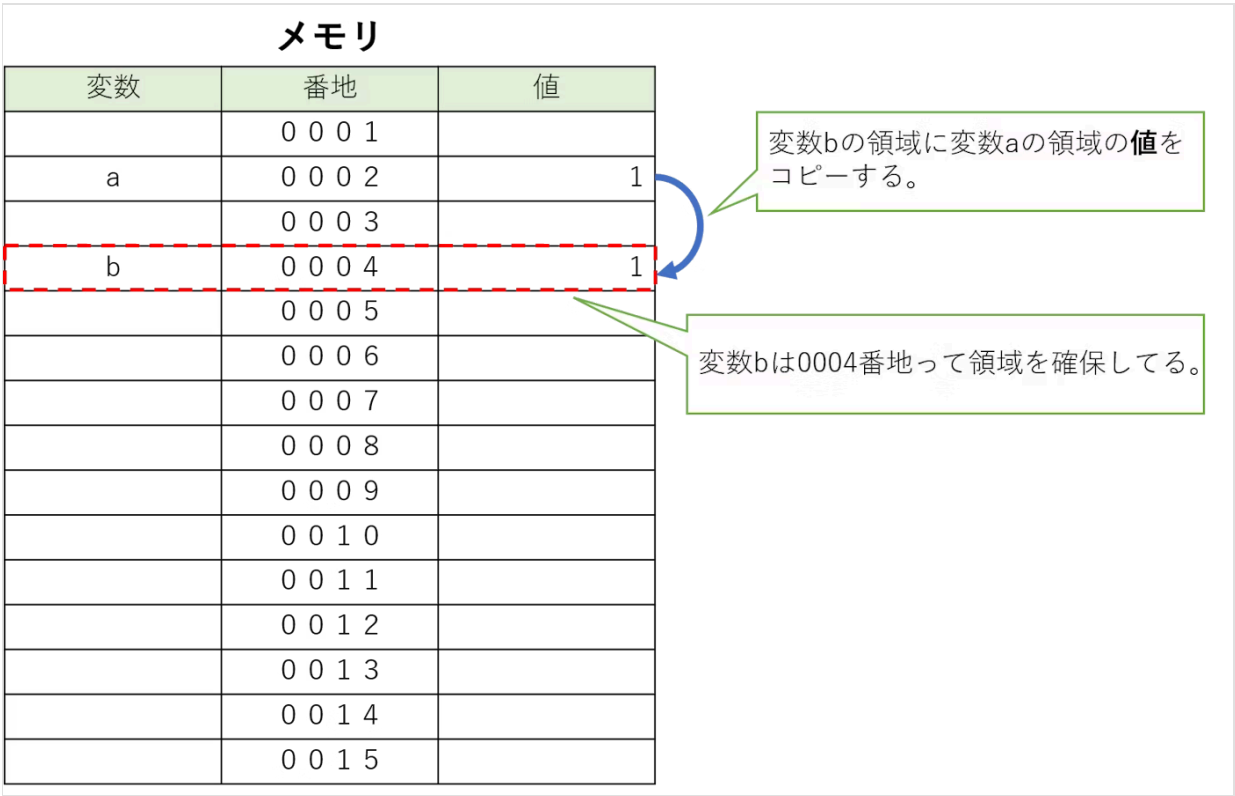
これらは関数や変数にデータを渡したり代入する時の方法の種類のようでした。
そもそも、コンピュータでプログラムを動かす時には「メモリ」に保存して、それらをつかって処理

してもらうのですが、メモリには「アドレス」だとか「番地」というものがありその番地毎に値などが保存されます。
その番地に保存された値を関数や変数に渡すときの方法がそれぞれ異なっています。

値渡し

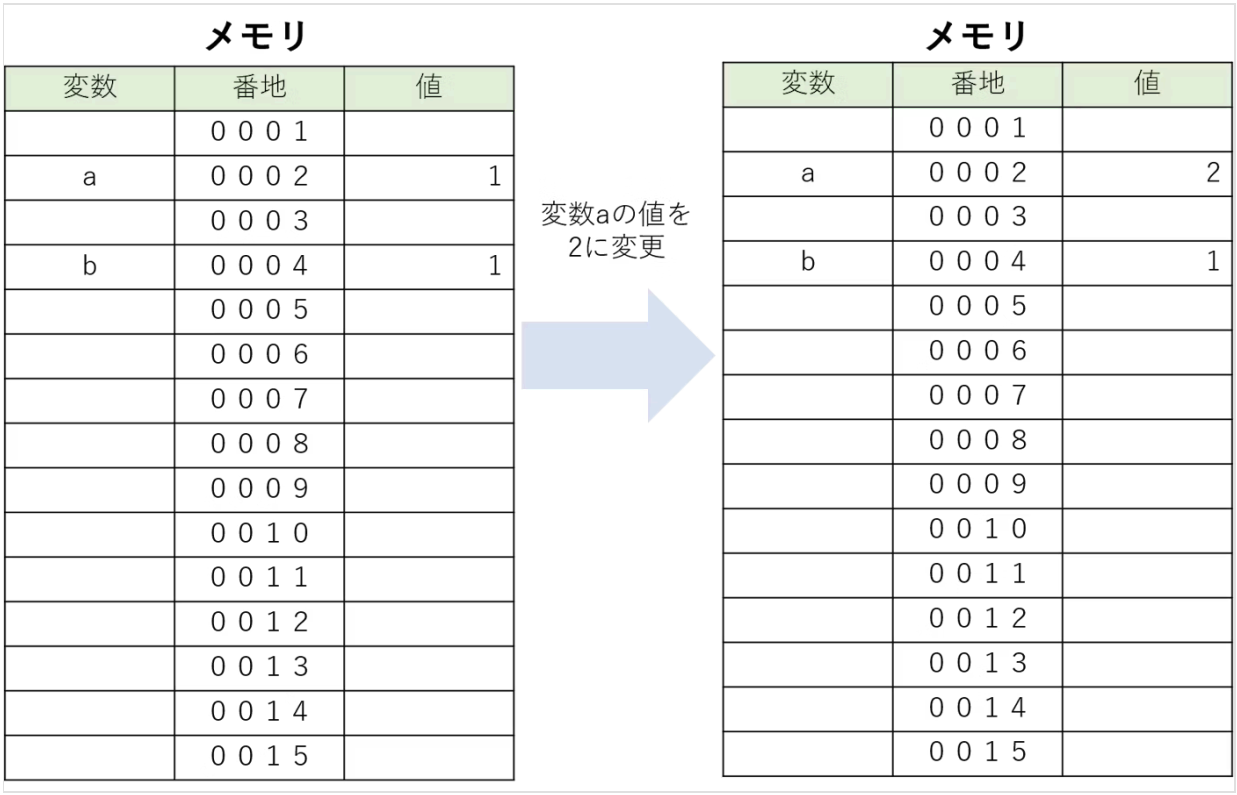
まず、値渡しをするような環境で`a`という変数に`1`という値を代入すると。
メモリ上ではこんな感じになります。

この時に変数`b`に変数`a`を代入すると、変数`b`が確保した領域(0004番地とする)には変数`a`が確保した領域0002番地に入っている値である`1`が入ります。



この時、変数`a`と`b`には同じ`1`という値が入っていますが、`a`と`b`が確保している領域の0002番地と0004番地にそれぞれ独立して`1`という値が入っているだけなので、`a`か`b`どちらかの値を変更しても

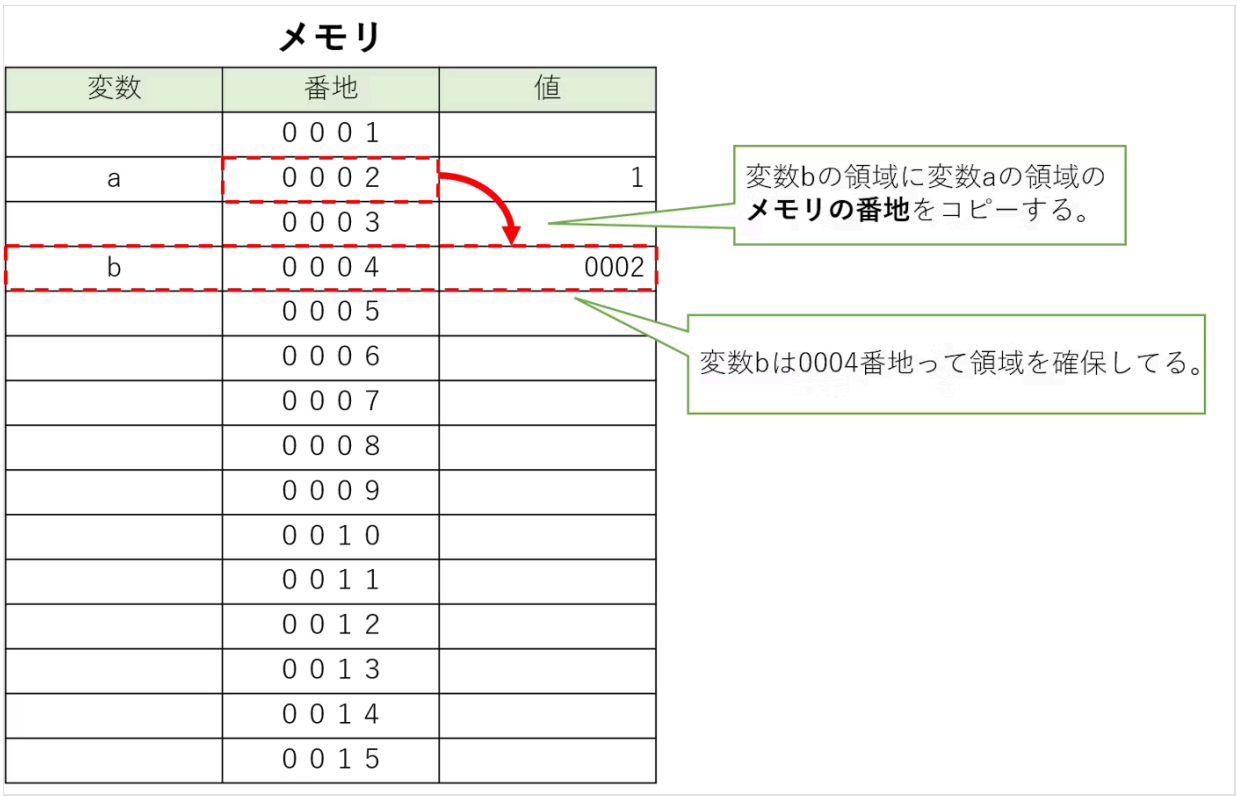
もう一方の変数の値に影響はありません。



このように値を直接渡して、渡した後はそれぞれ独立するようなものが値渡し。

参照渡し

今度は参照渡しですが、変数`a`の領域に値`1`が入っていて変数`b`に変数`a`を代入した時はこのようになります。
この時に変数`b`の値を求めると、`変数bの領域に入っているメモリの番地の領域に入っている値`つまり変数`a`の領域に入っている値が帰返ってきます。



ここで、変数`a`の値に変更を加えると、変数`b`の値は結局変数`a`の値を見ているので変数`b`の値を呼び出したときも同じ変更が加わった値が表示される。

メモリ

変数	番地	値
	0001	
a	0002	1
	0003	
b	0004	0002
	0005	
	0006	
	0007	
	0008	
	0009	
	0010	
	0011	
	0012	
	0013	
	0014	
	0015	

変数aの値を2に変更

メモリ

変数	番地	値
	0001	
a	0002	2
	0003	
b	0004	0002
	0005	
	0006	
	0007	
	0008	
	0009	
	0010	
	0011	
	0012	
	0013	
	0014	
	0015	

変数bの値を要求すると、①変数bの領域の値に参照先領域の番地が入ってるので、②書いてある参照領域を見に行って、③その領域に入っている値を返す。
このため変数aの値変更前は「1」が返って来て、変更後は変更後の値の「2」が返ってくる。

こんな感じのデータの渡し方が参照渡し。

参照の値渡し

最後に参照の値渡し、まず以下のような状態を考える。
変数`a`の領域には`1`という値の入っているメモリの番地が値として入っている。

メモリ

変数	番地	値
	0001	
a	0002	0010
	0003	
	0004	
	0005	
	0006	
	0007	
	0008	
	0009	
	0010	1
	0011	
	0012	
	0013	
	0014	
	0015	

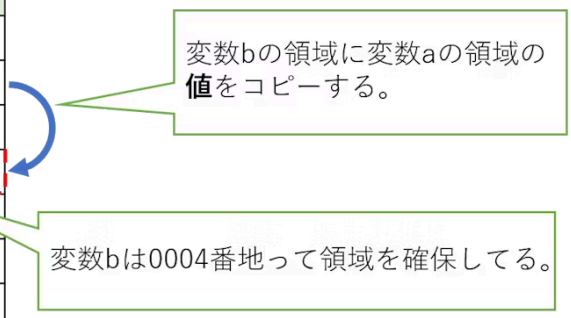
変数aの確保している領域の値には0010番地のアドレスが入っている

0010番地には１って値が入っている。

この時、変数`b`に変数`a`を代入すると以下のように変数`a`に値として入っていた0010番地のアドレスが変数`b`の領域に値としてコピーされる。

メモリ

変数	番地	値
	0001	
a	0002	0010
	0003	
b	0004	0010
	0005	
	0006	
	0007	
	0008	
	0009	
	0010	1
	0011	
	0012	
	0013	
	0014	
	0015	



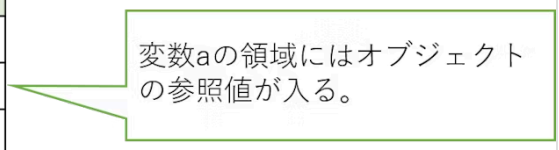
参照先の情報渡してるんだから参照渡しじゃんと思いましたが、参照の値渡しは値として持ってる参照先を値として他に渡している。
これが参照の値渡しのイメージ。

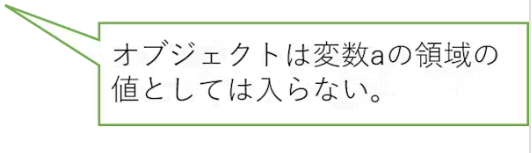
Pythonでの変数の扱い

話は少し変わりますが、今度はPythonでの変数の扱いについて。
Pythonでは変数の宣言が必要なく`a = 1`と変数と値を書くと変数を使えるようになりますが、この時メモリ上では以下のように`1`という値は変数`a`の確保した領域の値に入るのではなく、違う番地に入り、変数`a`の領域の値には代入したオブジェクト(値)の入ったメモリの番地（参照値）が入ります。
pythonでは他の変数でもこのように参照値を持ち、参照値の示す場所にオブジェクトが入ります。

メモリ

変数	番地	値
	0001	
a	0002	0010
	0003	
	0004	
	0005	
	0006	
	0007	
	0008	
	0009	
	0010	1
	0011	
	0012	
	0013	
	0014	
	0015	





変数の領域に入っている参照値は`id`という組み込み関数で確認できます。

例

```
>>> a=1
>>> print("値=",a,"参照値=",id(a))
値= 1 参照値= 1982607655152 #変数aは参照値が1982607655152の領域に入ってる値の1を参照している。
```

この時、変数`b`に変数`a`を代入してみます。

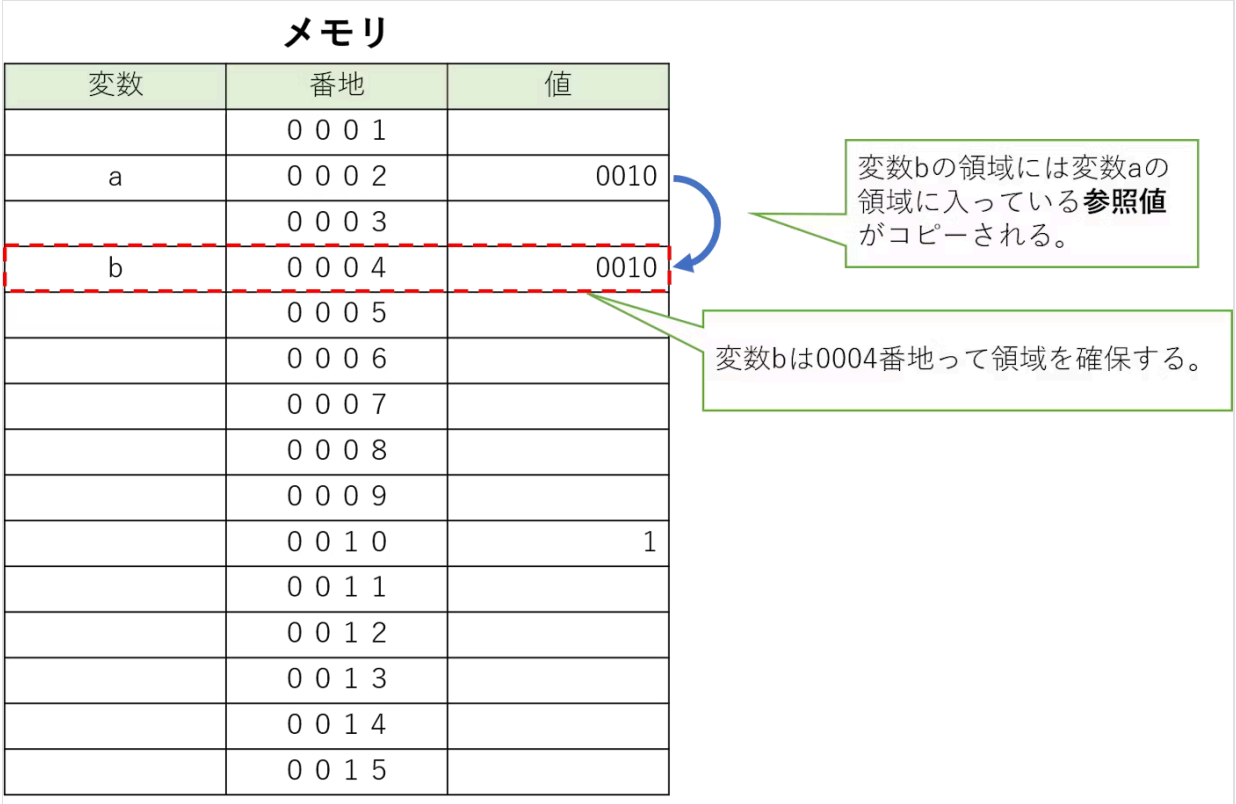
例

```
>>> a=1

>>> print("値=",a,"参照値=",id(a))
値= 1 参照値= 1982607655152 #変数aは参照値が1982607655152の領域に入ってる値の1を参照している。

>>> print("値=",b,"参照値=",id(b))
値= 1 参照値= 1982607655152 #変数bは変数aと同じ参照値が1982607655152の領域に入ってる値の1を参照している。
```

変数`b`の値と参照値を確認を確認すると、変数`a`と同じ領域を参照してそこの値を返してきてることがわかります。
イメージ図を下に示しますが、つまりPythonでのデータの渡し方は「参照の値渡し」ということです。



これらを踏まえてパターン1を見てみる

ではここまでを踏まえてパターン1の場合の流れを参照値も見ながら確認してみます。

例

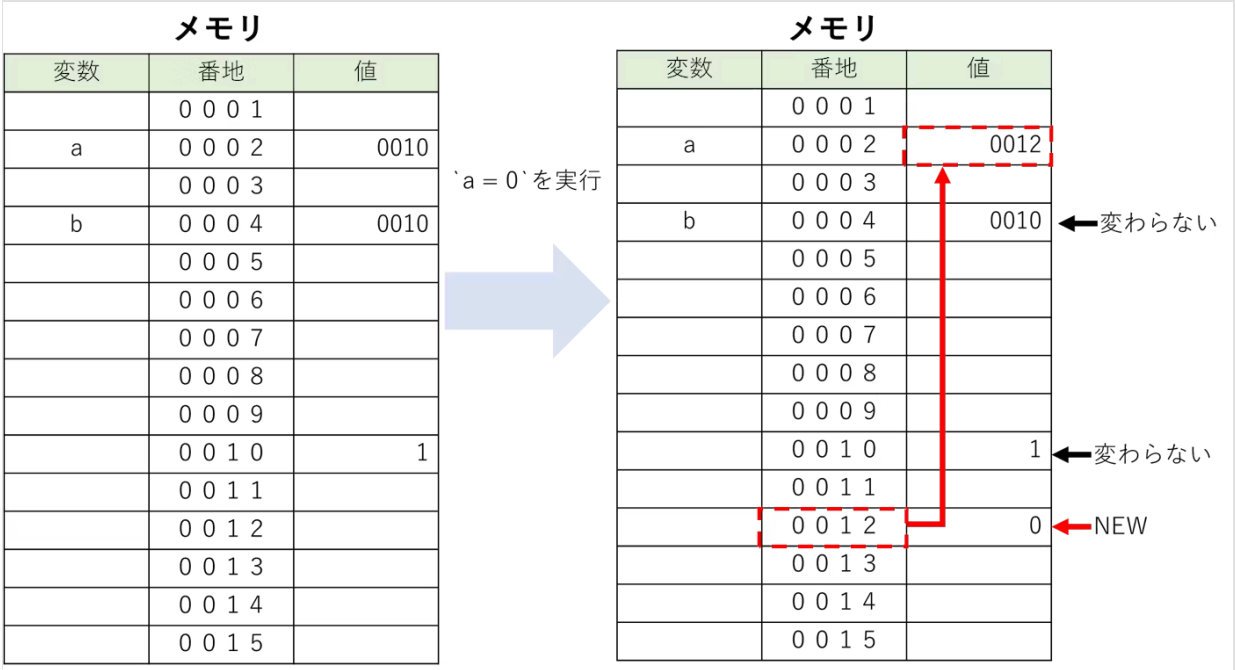
```
>>> a = 1
>>> b = a
```

```
>>> print("値=",a,"参照値=",id(a))
値= 1 参照値= 1982607655152 #変数aは参照値が1982607655152の領域に入ってる値の1を参照している。
>>> print("値=",b,"参照値=",id(b))
値= 1 参照値= 1982607655152 #変数bは変数aと同じ参照値が1982607655152の領域に入ってる値の1を参照している。

>>> a=0 #変数aの値を0とする

>>> print("値=",a,"参照値=",id(a))
値= 0 参照値= 1982607655120        #値は0になったが、参照値が先ほどと変わっている。
>>> print("値=",b,"参照値=",id(b))
値= 1 参照値= 1982607655152        #参照先も値も先ほどと変わっていない。
```

結果として変数`a = 0`とすることでaの持つ参照値が変わっています。一方で変数`b`は参照値が変わっていません。
当初は参照値`1982607655152`に入っている値の`1`を`0`に変えるつもりで`a = 0`としたのですが、そもそも`a = 0`では`a`に新たに`0`を代入しているだけでしたので、新しく参照値`982607655120`に値`0`が入り変数`a`の持つ参照値が`982607655120`に変わる形で値が変化しました。パターン3についても同様の結果になります。
なんならint型はイミュータブルなので参照値`1982607655152`に入っている値の`1`は変更不可です。
イメージ図で表すとこんな感じ。



パターン2も見てみる

冒頭の`パターン2`のように値がミュータブルな場合、変更可能なのだからイミュータブルな値の時とは違って、新しいメモリの領域に新しい値を入れたりせずに指定した参照先の値を変更できる。

例

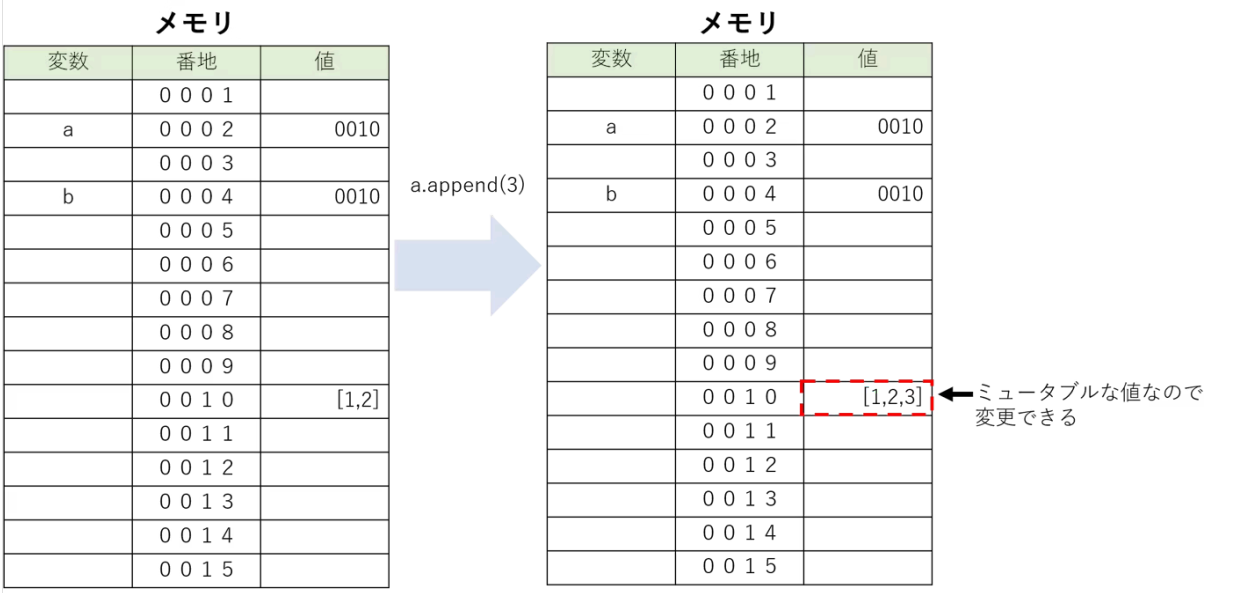
```
>>> a=[1,2]
>>> b=a

>>> print("値=",a,"参照値=",id(a))
値= [1, 2] 参照値= 2179606327936
>>> print("値=",b,"参照値=",id(b))
値= [1, 2] 参照値= 2179606327936

>>> a.append(3)                    #aの参照先の値[1,2]に3を追加する
```

```
# 参照先の値が変更され、aもbも同じ参照先を参照している
>>> print("値=",a,"参照値=",id(a))
値= [1, 2, 3] 参照値= 2179606327936
>>> print("値=",b,"参照値=",id(b))
値= [1, 2, 3] 参照値= 2179606327936
```

イメージ図で整理すると、こんな感じ。
0010番地の値が変わるだけで、`a`と`b`の持つ参照値が変わったりはしないのでどちらも同じ場所を参照するためどちらにも変更が反映される。`a.append(3)`の部分が`b.append(3)`や`a += [3]`でも同じようになる。
ただしパターン3のように`a = [1,2,3]`とした場合は新しくオブジェクトを代入しているので`a`の参照値は変わる。



参考にした記事