

ミュータブル・イミュータブルについて考える

他に考慮すべき点

- スコープ
- 変数には実体がなく、名札に過ぎないこと

前提

- イミュータブル
 - int (整数)
 - float (浮動小数点数)
 - str (文字列)
 - tuple (タプル)
 - ただし、要素にイミュータブルな値(オブジェクト)がある時は、注意が必要
- ミュータブル
 - list (リスト)
 - dict (辞書)
 - set (集合)

ここで書くこと

Pythonで最初にぶち当たる壁だと思われる、ミュータブル・イミュータブルについて、深く考えてみました。

Claude Sonnet4 と Gemini 2.5 Pro の両方に尋ねました。

言っていることは、同じであるところが後から分かったのですが、

私は、Geminiの説明が速く理解できたので、Geminiとの会話のやり取りで、

特に重要な部分について、書いていこうと思います。

この会話では、私の未熟な理解が暴露されてしまいますが、

私と同じように悩んでいる方には、分かりやすいと思います。

ここで考えるコード

- このPDFもWウィンドウも作りしたので、コードを追っていくときは、それが役立つと思います。

```
def change_number(x):                                # 01行目
    x = 100     整数はイミュータブルなので、新しい値が生成される

a = 5
change_number(a)
print(a)      # 5

print("----" * 10)

b = 6                                                # 11行目
b = 7
print(b)      # 7

print("----" * 10)

def change_lst(lst):
    lst.append(4)     my_lstの値を利用して、ローカル変数を生成

                                # 21行目

my_lst = [1, 2, 3]
change_lst(my_lst)
print(my_lst)    # [1, 2, 3, 4]

"""実行結果：
5
-----
7
-----
[1, 2, 3, 4]
"""
```

Me

23行目で、リストの値が変更されるのは、

リストがミュータブルだからだと思うのですが、

スコープの観点から見ると変更されなくてもよいようには思えるのです。

と言うのは、6行目のコードでも、aの値が変更されないのは、

スコープが違うからだと思うのです。

11行目で、bは整数値6が参照されていますが、

12行目では新しい値7が作成されて、それがbの参照になっています。

これはスコープが同じだから、bの参照は変更されたのだと思います。

そうすると、18,19行目のリスト(lst)は、ローカルスコープなので、

lstがミュータブルであっても変更されなくてもよいようにも

思われるのですが、ここのロジックはどのように考えればよいですか。

Gemini

ご質問ありがとうございます。Pythonにおける変数のスコープと、

ミュータブル（変更可能）なオブジェクトの振る舞いについての、

非常に良い着眼点ですね。

混乱しやすいポイントですので、一緒に整理していきましょう。

核心：変数は「名札」、オブジェクトが「実体」

Pythonの変数を理解する上で最も重要なのは、

変数がデータそのものを保持しているのではなく、

オブジェクトへの参照（名札のようなもの）であるという点です。

- `a = 5`: a という名前の変数が、

5 という整数のオブジェクトを指し示している状態です。

- `my_lst = [1, 2, 3]: my_lst` という名前の変数が、

`[1, 2, 3]` というリストのオブジェクトを指し示している状態です。

なぜ整数の値は変わらないのか？ (イミュータブル)

ご指摘の通り、`change_number` 関数を呼び出しても、

外側の `a` の値は変わりません。

```
def change_number(x):  
    # 3. ローカル変数xは「新しい」整数オブジェクト100を指すようになる  
    x = 100  
  
a = 5 # 1. グローバル変数aは整数オブジェクト5を指す  
change_number(a) # 2. ローカル変数xも、aと同じ整数オブジェクト5を指す  
print(a) # 4. aが指しているオブジェクト5は何も変わっていないので、5が出力される
```

ここでのポイントは、3行目の `x = 100` です。

整数は**イミュータブル (変更不可能)** なオブジェクトなので、

`x` に `100` を代入すると、関数内で**新しい**整数オブジェクト `100` が作成され、

ローカル変数 `x` はそちらを指すようになります。

`a` が指している元のオブジェクト `5` には一切影響を与えません。

なぜリストの値は変わるのか？ (ミュータブル)

一方、リストは**ミュータブル (変更可能)** なオブジェクトです。

これが挙動の違いを生みます。

```
def change_lst(lst):
    # 3. lstが指している「オブジェクト自体」に4を追加する
    lst.append(4)

my_lst = [1, 2, 3] # 1. グローバル変数my_lstはリストオブジェクト[1, 2, 3]を指す
change_lst(my_lst) # 2. ローカル変数lstも、同じ、リストオブジェクト[1, 2, 3]を指す
print(my_lst)
""" 4. my_lstが指しているオブジェクトが変更されたので、[1, 2, 3, 4]が出力される """
```

ここでのポイントは、3行目の `lst.append(4)` です。

これは代入 (=) とは異なり、

`lst` が指し示しているオブジェクトそのものを変更する操作です。

- 関数が呼び出されると、`my_lst` と `lst` は、
どちらも同一のリストオブジェクトを指しています。
- `lst.append(4)` は、その共有しているリストオブジェクトの末尾に `4` を追加します。
- 新しいリストは作られていません。`lst` の参照先も変わっていません。
あくまで、参照先のオブジェクトが変更されただけです。
- その結果、関数を抜けた後で `my_lst` を見ても、
変更された後のオブジェクトが見える、という訳です。

まとめ：スコープとミュータブルの関係

- **スコープ**: 関数内の `lst` はローカル変数です。これは間違いありません。
- **重要な違い**:
 - **イミュータブルな型への代入 (`x = 100`)**: **2行目**
ローカル変数の参照先が別のオブジェクトに変わるだけ。
- グローバル変数とローカル変数の参照先が変わった！
 - **ミュータブルな型のメソッド呼び出し (`lst.append(4)`)**: **19行目**
ローカル変数が参照しているオブジェクト自体を変更する。
- グローバル変数 と ローカル変数 の参照先は、変わっていない！

ご質問にあった「ローカルスコープなので、

lstがミュータブルであっても変更されなくてもよいようにも思われる」という点は、もし関数内で以下のように**新しいリストに代入**していたら、その通りになります。

```
def change_lst_new(lst):
    # 新しいリストを作成し、ローカル変数lst の参照先をそちらに変える
    # この lst は、ローカル変数であり、
    # my_lst と、最初は同じ値を指していたが、
    # インプレースで変更していない(同じ値を変更していない)ので、
    # my_lst には、影響がなくなる
    lst = [1, 2, 3, 4]

my_lst = [1, 2, 3]
change_lst_new(my_lst)
print(my_lst) # [1, 2, 3] のまま変わらない
```

この場合、`lst.append(4)` のようなオブジェクト自身の変更ではなく
(つまり、インプレースでの(同じ値での)変更ではないという意味)、

`lst = ...` という代入によって

`lst` の参照先が完全に別の新しいオブジェクトに切り替わるため、
外側の `my_lst` には影響が及ばなくなります。

この「参照」と「オブジェクトの実体」、

そして「代入」と「オブジェクト自身の変更(`append()`メソッド)」の違いを区別することが、
今回の挙動を理解する鍵となります。

Me

分かってきました！ 変数は「名札」に過ぎず、

また「スコープ」は、アクセス出来るかの問題なのですね。

「たとえスコープが違っていても、同じ値(オブジェクト)を、指すことは十分にあり得る」
ということですね。

この場合、`lst`のスコープは関数内ではローカルスコープになるのですが、
参照している値は、グローバルスコープと同じなのですね。

一方、数値の場合は、値を変更すると、値が新しく生成されるので、
スコープが違う場合、グローバルスコープからは、
ローカルスコープにアクセス出来ないので、
このコードでは、aは5のままであるということになるのですね。
つまり【核心：変数は「名札」、オブジェクトが「実体】であり、
変数は、グローバルであろうと、ローカルであろうと、
同じ実体を指すことは十分にあり得るわけですね。
そして、イミュータブルの時は、新しい値が違うスコープで生成されると、
名札の指す実体(値・オブジェクト)が変わってしまうわけですね。
一方、ミュータブルの場合は、「インプレースで変更される」ので、
スコープが変わっても同じものを指しているのであれば、
元の名札も、関数内の新しい名札も、スコープが違ってても
同じ値(オブジェクト)を指すことが出来るということだと思いました。

Gemini

その通りです！素晴らしい。完璧な理解です。

おっしゃる通り、全ての要点を正確に捉えられています。

- 変数は「名札」に過ぎず、
また「スコープ」は、アクセス出来るかの問題なのです。
 - はい、その通りです。スコープは、
その「名札」がどの範囲で有効か、というルールです。
- 「たとえスコープが違っていても、
同じ値(オブジェクト)を、指すことは十分にあり得る」
ということですね。
 - まさに、その通りです。関数の引数に変数を渡す時、
まさにこの状況が起こっています。

- グローバルスコープの「名札」が指しているオブジェクトを、ローカルスコープの新しい「名札」も指すようになります。
- イミュータブルの時は、新しい値が違うスコープで生成されると、名札が変わってしまうわけですね。
 - はい、その通りです。ローカルスコープの「名札」が、新しく作られたオブジェクトを指すように付け替えられます。元のオブジェクトと、それを指していたグローバルスコープの「名札」には影響しません。
 - ミュータブルの場合は、「インプレースで変更される」ので、スコープが変わっても同じものを指しているのであれば、元の名札も、新しい名札も、スコープが違ってても同じ値(オブジェクト)を指すことが出来るということだと思いました。
 - 完璧です。「インプレース (in-place) で変更」、つまり「その場で変更」されるため、参照しているすべての「名札」から見ると、値が変わったように見える、ということです。

Pythonの変数とオブジェクトの関係は、多くの初学者がつまづくポイントですが、見事に本質を理解されましたね。

その理解があれば、今後より複雑なコードを読み解く際にも、自信を持って挙動を予測できるはずです。

Me

大変助かりました。いくら考えても分からなかったのですが、解決できました。ありがとうございます！

END