

なぜ、Pythonの「=」は、演算子ではないのか

この問題に関しては、ふとしたことから、望洋氏の本に書かれていたことを思いだして、Claudeに質問してみました。

直接関係ない所は、なるべく省きました。

ただ、関係ない部分も、在った方が良いと思ったところは残しました。

Me

...

聞きたいことはこれではなかったでした。

聞きたいことは、以前柴田望洋氏の本で読んだことで、「=」は、

正確には演算子ではないと言っていました、それはどういう意味なのでしょう。

...

そう言えば、`a = b = c`での働きについて説明している時だったと思います。

だんだん思い出してきました。

多くの書籍では、「cがbに代入され、bがaに代入される」ように、説明されているが、Pythonではそれが当てはまらないというような説明だったと思います。

Claude

柴田望洋氏の「=は正確には演算子ではない」という説明について、

非常に興味深い指摘ですね。これはPythonの代入の仕組みに関する重要な概念です。

「代入演算子」と呼ばれがちだが...

多くのプログラミング言語では=を「代入演算子」と呼びますが、

Pythonでは厳密には「代入文 (assignment statement)」の一部です。

重要な違い：`a = b = c`の動作

他の言語（C言語など）での一般的な解釈：

```
// 「右から左へ」の解釈  
a = b = c; // c→b→a の順で代入
```

Pythonでの実際の動作：

```
a = b = c = [1, 2, 3]  
# 実際には：すべての変数が同じオブジェクトを参照
```

Pythonの特殊性

```
# 実験してみましょう  
a = b = c = [1, 2, 3]  
print(id(a), id(b), id(c)) # 全て同じIDが表示される  
  
# リストを変更すると...  
a.append(4)  
print(a) # [1, 2, 3, 4]  
print(b) # [1, 2, 3, 4] ← bも変わる！ a と同じものを指しているから。  
print(c) # [1, 2, 3, 4] ← cも変わる！ a と同じものを指しているから。
```

なぜ「演算子ではない」のか

1. 演算子は値を返すが、Pythonの代入は値を返さない
2. オブジェクトの参照を結び付ける操作（バインディング）
3. 右辺を評価してから、左辺の全ての名前に同時に結び付ける

正確な理解

`a = b = c`は：

- 「cがbに代入され、bがaに代入される」のではなく
- 「右辺を評価した結果を、a, b, c全てに同時にバインドする」

Me

あなたが指摘されたコードを拡張して試してみました。

```
# 実験してみましょう
a = b = c = [1, 2, 3]
print(id(a), id(b), id(c)) # 全て同じIDが表示される

# リストを変更すると...
a.append(4)
print(a) # [1, 2, 3, 4]
print(b) # [1, 2, 3, 4] ← bも変わる！
print(c) # [1, 2, 3, 4] ← cも変わる！

lst_abc = [a, b, c]

print([id(eyedee) for eyedee in lst_abc])

"""実行結果（分かりやすくするために改行を入れました）

3036119804352 3036119804352 3036119804352

[1, 2, 3, 4]
[1, 2, 3, 4]
[1, 2, 3, 4]

[3036119804352, 3036119804352, 3036119804352]
"""
```

この意味するところはということなのか理解できないので教えてください。

Claude

素晴らしい実験ですね！この結果が示していることを詳しく説明します。

実行結果の意味

python

```
a = b = c = [1, 2, 3]
```

この一行で何が起こったかを図解すると：

```
メモリ上のオブジェクト: [1, 2, 3] (ID: 3036119804352)
                        ↑      ↑      ↑
                        a      b      c
```

全ての変数が同じオブジェクトを指しているのです。

なぜ全て同じIDなのか

python

```
lst_abc = [a, b, c]
print([id(eyedee) for eyedee in lst_abc])
# [3036119804352, 3036119804352, 3036119804352]
```

この結果は：

- `a`, `b`, `c` は別々の変数名だが
- 全て同じリストオブジェクトを参照している
- だから `id()` で調べると全て同じ値

重要なポイント：「コピー」ではない

もし「右から左へ代入」だったら：

python

```
# もしこうだったら（実際は違う）
c = [1, 2, 3]      # 新しいリスト作成
b = c.copy()       # cのコピーをbに
a = b.copy()       # bのコピーをaに
```

この場合、3つの別々のリストができて、IDも異なるはずです。

Pythonでの実際の動作

python

```
a = b = c = [1, 2, 3]
```

これは内部的には：

1. `[1, 2, 3]` という1つのリストオブジェクトを作成

2. そのオブジェクトに `a`, `b`, `c` という**3つの名前**を付ける

なぜ一つを変更すると全部変わるのか

python

```
a.append(4)
# a, b, cは全て同じオブジェクトを指しているので
# どの名前でアクセスしても同じリストが変更される
```

比較：個別に作成した場合

python

```
# 個別に作成
a = [1, 2, 3]
b = [1, 2, 3]
c = [1, 2, 3]

print(id(a), id(b), id(c)) # 異なるIDが表示される
a.append(4)
print(a) # [1, 2, 3, 4]
print(b) # [1, 2, 3] ← 変わらない
print(c) # [1, 2, 3] ← 変わらない
```

これがPythonの「**代入は参照の結び付け**」という概念の核心です！

Me

今度はこんなコードに見ました。

```
# 実験してみましょう
a = b = c = [1, 2, 3]
print(id(a), id(b), id(c)) # 全て同じIDが表示される

# リストを変更すると...
a.append(4)
print(a) # [1, 2, 3, 4]
print(b) # [1, 2, 3, 4] ← bも変わる！
print(c) # [1, 2, 3, 4] ← cも変わる！

1st_abc = [a, b, c]
```

```

print([id(eyedee) for eyedee in lst_abc])

"""実行結果（改行を付けました）：
3036119804352 3036119804352 3036119804352

[1, 2, 3, 4]
[1, 2, 3, 4]
[1, 2, 3, 4]

[3036119804352, 3036119804352, 3036119804352]
"""

# a のみに、append(5) をするが、
# b, c も、同じ参照を指すはずである
a.append(5)
print([a_c for a_c in [a, b, c]])

print([id(eyedee) for eyedee in lst_abc])

# = ではなく、copy()メソッドを使う
x = [10, 20, 30]
y = x.copy()
z = y.copy()

lst_xyz = [x, y, z]

print([id(x_z) for x_z in lst_xyz])

"""実行結果（改行を付けました）：
[[1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5]]
[2051918375360, 2051918375360, 2051918375360]

[2051922065600, 2051956641920, 2051922084672]
"""

```

つまり、望洋氏が言いたかったことは、C言語における変数は、すべて値そのものを指すのだから、「代入=コピー」であるが、Pythonでは、`=` は「代入文」であり、特にC言語との違いは大きく、「参照を指定するだけ」であるということではないでしょうか。

(Java・C#などは、値を指すもの・参照をさすもの の2通りある)

Claude

素晴らしい実験と考察です！まさにその通りです。

実験結果の分析

パート1：同じ参照の確認

```
a.append(5)
# a, b, c全てが [1, 2, 3, 4, 5] になり、IDも同じ
```

パート2：copy()での独立したオブジェクト

```
x = [10, 20, 30]
y = x.copy() # 新しいオブジェクト作成
z = y.copy() # さらに新しいオブジェクト作成
# 結果：異なるID [2051922065600, 2051956641920, 2051922084672]
```

望洋氏の指摘の核心

おっしゃる通りです！

C言語：「代入 = コピー」

```
int a = 5;
int b = a; // aの値(5)をbにコピー
a = 10;    // bは5のまま（独立）
```

Python：「代入 = 参照の結び付け」

```
a = [1, 2, 3]
b = a # aが参照するオブジェクトをbも参照
a.append(4) # 同じオブジェクトなのでbも変わる
```

正確な表現

- **C言語**：代入 = 値のコピー
- **Python**：代入 = 参照の結び付け（バインディング）
- **Java/C#**：参照型は参照のコピー、値型は値のコピー

なぜ「演算子ではない」のか

```
# 演算子なら値を返すはず
result = (a = 5) # これはエラー！

# 実際の代入文
a = 5 # 文 (statement) であり、値を返さない
```

望洋氏の指摘は、

プログラミング言語の根本的な設計思想の違いを教えてくれる重要な概念ですね。

特にPythonの「すべてがオブジェクト」という哲学を理解する上で欠かせない知識です。

Me

ついでに、C#のコードで実験したら、本当に値を返しました

```
int i;
Console.WriteLine(i = 2); // 2
```

END