オブジェクト.copy() と copy.deepcopy(オブジェクト)：

どちらを使うべき？

- はじめに：
    - ここは非常に分かりにくいところで、
      自分で分かっていたつもりでも、よくどうだったけと悩んでしまうところです。
        - ということで、今日PyCharmでコードを書いていた時、
          聞くきっけけは何だったのか忘れてしまいましたが、**Gemini**に徹底的に聞いてみました。
        - 今日聞いたことを忘れないうちにまとめておこうと思います。

copy.deepcopy()を使う場面

まず、リストの場合を考えてみます

- リスト：
    - ミュータブル(変更可能)
    - 形から言って箱を連想しやすいので、まず「箱」だと考えます。
      後にコンテナ(他のオブジェクトを格納するためのオブジェクト)と呼ぼうと思います。

リストを変更するとは

- まず簡単なコードで考えてみましょう

```
lst = [1, 2, 3]
lst_2 = lst.append(4)

print(f"lst: {lst}")
print(f"lst_2: {lst_2}")
"""実行結果
lst: [1, 2, 3, 4]
lst_2: None
"""
```

- リストはミュータブルであり、この時`lst`自体が変更されています
    - これはインプレースに行われるという言い方をします
    - lst_2 には、Noneが返っていますが、これはprint()関数が返したものです
    - つまり、lst_2 は、リストではありません。

- この場合、コピーしたい時、
  リストのcopy()メソッドを使うか、**copyモジュールのdeepcopy()関数**を使うべきでしょうか。

- これで、deepcopy()関数を使おうとは思いませんよね。
- では実際に使ってみて、片方の値を変更して問題が起きるか確かめます。

```
lst = [1, 2, 3]
lst.append(4)
print(lst)

"""実行結果
[1, 2, 3, 4]
"""
lst_copy = lst.copy()
lst_copy.append(5)

print(f"lst: {lst}")
print(f"lst_copy: {lst_copy}")
"""実行結果
lst: [1, 2, 3, 4]
lst_copy: [1, 2, 3, 4, 5]
"""
```

- コピー元は、元のままなので、期待通りになりました。

- 今度は、リストの中に、**リストをネストさせて**考えてみます。
- 問題が起きると思いますか。起きるとしたらなぜ起きるのでしょうか。

```
lst = [0, 1, [2, 3]]

lst_copy = lst.copy()
lst_copy[2][1] = 8

print(f"lst: {lst}")
print(f"lst_copy: {lst_copy}")

"""実行結果
lst: [0, 1, [2, 8]]
lst_copy: [0, 1, [2, 8]]
"""
```

- なんと、コピー元まで、コピーと同じ値になってしまいました
- これは、ほとんどの場合、意図しない結果になるでしょう。
- なぜなら、コピー元のリストは変更したくないから、コピーを作る場合が多いと思うからです

- 今度は、`import copy` → `copy.deepcopy(リスト)` という流れで実行してみようと思います

```
import copy

lst = [0, 1, [2, 3]]

# copy.deepcopy()関数を使う
lst_copy = copy.deepcopy(lst)
lst_copy[2][1] = 8

print(f"lst: {lst}")
print(f"lst_copy: {lst_copy}")

"""実行結果
lst: [0, 1, [2, 3]]
lst_copy: [0, 1, [2, 8]]
"""
```

- 今度は、コピー元は、元のままなので期待通りです。

ではなぜこの違いが生まれてくるのでしょう？

- `リスト.copy()メソッド`を使った場合こんなイメージになります
  - [ ]は、違う箱(コンテナ)と考えてください
    - [0, 1, [2, 3]] ➡ [0, 1, [2, 3]]
  - これが、copy()メソッドのイメージです
    - 外の箱は、コピーされて変更されているが、内側の箱は変更されていません。
      - つまり同じ参照なのです
  - リスト[2][1]を「8」に変更するということは、同じ箱の中の数字を入れ替えることなのです

copyモジュールのdeepcopy()関数 使った場合

- `copy.deepcopy(リスト)`を使った場合のイメージです
  - [ ]は、違う箱(コンテナ)と考えてください
    - [0, 1, [2, 3]] ➡ [0, 1, [2, 3]]
  - これが、copy.deepcopy()関数を使ったイメージです
    - つまり箱の総入れ替えです、
      箱が違えば、コピー元の値に影響を与えることは出来なくなりますね
    - そしてこの事はネストが深くなっても同様に働きます
      - [0, 1, [2, [3, 4]]] のような場合でも有効です

```
import copy

lst = [0, 1, [2, [3, 4]]]

# copy.deepcopy()関数を使う
```

```
lst_copy = copy.deepcopy(lst)
# コピーしたリストを変更する
lst_copy[2][1][0] = 8

print(f"lst: {lst}")
print(f"lst_copy: {lst_copy}")

"""実行結果(見やすいように整形しました)
lst:
    [0, 1, [2, [3, 4]]]
lst_copy:
    [0, 1, [2, [8, 4]]]
"""
# 期待通り、元のリストは変更されていません
```

ただ、copy.deepcopy()関数は、
浅いコピーに比べて処理時間が掛かり、メモリを多く使ってしまいます

- 小さなプログラムでは問題が無くても、大きなプログラムでは、なんと言うか「塵も積もれば山となる」と言えます

ここで重要なこと

- どんな時、浅いコピー（copy()メソッド）を避けるべきか？
  - オブジェクトが、中に違うオブジェクトを格納できるコンテナである場合、
    注意しなければなりません
- つまり内側の要素が、コンテナでなければ問題は起きません
- それは何故かというと、浅いコピーでも、
  外側のコンテナは、コピー(違うコンテナ)であるからです
- そして、内側の要素が、コンテナでない限りは、**違うコンテナ**に入っていると言えます
  - つまり「深いコピー」とは、ネストされたコンテナまで、コピーすると言えます
  - もっと言えば、全てが違う容器(コンテナ)になるのです
- タプルの場合：
  - タプルは不変であるが、そのコンテナには、リストも入るので要注意です！
  - 逆にタプルの中に、コンテナがなければ、コンテナとして扱わなくてもよい

では、コンテナは何で、何はコンテナではないのか？

- **Gemini**に上げてもらったいくつかの例
  - リスト
  - 辞書
  - set

    - タプルは、コンテナだが、その中身が基本データ型(↓)であれば
      深いコピーを気にする必要はない
    - ここでは、理解に集中していますので、
      もっと詳しい例は、私も含めて経験を積んで学べば良いと思います

- コンテナには含まれないもの

    - int, float, str, bool
        - Java的には、プリミティブ型と言える（基本データ型）
    - コンテナであっても、使い方によっては、浅いコピーを適用できるもの
        - タプル：内容が基本データだけなら適用できる
            - リストなどが含まれていると危険である
        - frozeset：いつも浅いコピーが適用できる
            - frozensetには、add()メソッドが使えない
            - frozensetには、基本データ型や、
              基本データ型しか入っていないようなタプルしか入れることは出来ないのである

まとめ

- 浅いコピー（例：**リスト.copy()**）は、外側のコンテナ以外は、複製しない(コピーしない)。
- 深いコピー（例：**copy.deepcopy(リスト)**）は、全てのコンテナを複製する
    - ただし、時間が掛かり、メモリも多く使う
- 本格的なプログラムでは、浅いコピーと深いコピーの使い分けをしないと、
  プログラム全体の品質に関わってくる
- 使い分けの判断は、コピー元が
  変更される危険がある(同じ参照を見ている))ことを考慮に入れて考えることである
    - Pythonの場合、全ての変数は、参照です
        - つまり変数には型はなく、型を持っているのは値なのです

END

追記：

- ここまで書いてきて、自分でも理解も深まった感じがしました。
- ただ読み返してみたら、一点だけ、イメージに対して追加をした方が良いと思いました。
    - 確かにイメージと現実は違うのですが、考え方はつながっているし、
      それを受け入れるのが高級言語のよい所だと思います
- そこで、Claude氏に相談したところ良いアイデアを授かりました。
- **どこが今一つだったのか？**
    - 浅いコピーのイメージ：
        - 元の家:　　[テレビ, ベッド, **タンス[Tシャツ, ズボン, 本]**]
        - 引っ越した家:　[テレビ, ベッド, **タンス[Tシャツ, ズボン, 本]**]
    - この時、引っ越した家　が、コピーされたものをイメージしています

- しかし、このイメージでは、タンス(コンテナ)は、引っ越した新しい家に入っています
- これだと、例えば「本」を全部出して「おやつ」に入れ替えたとします
- その場合、新しい家なのだから、元の家の「本」まで「おやつ」に代わるわけが無いと思いますよね。
- そこで「元の家のタンス」と「引っ越した(コピーした)家のタンス」をどう関連付けるか
    - 実際のコンピュータでは、実際の箱ではなく、メモリアドレス(ポインタ)を使っています
    - そして、プログラミング言語のルールに従って、どのポインタに格納するかを決めているということです
    - それを分かりやすいイメージにしているのが、高級言語の役割と言えます
    - ただそれにも限界がありルールとして覚えるしかない部分もあるのです
    - もっともC言語に通じている方なら、難なく受け入れられるかもしれませんが。
        - でもそれはそれで学習コストが大変だと思います
        - 私はできることなら楽をしたいです
- そこで、Claude氏のアイデアをもとに考えたのが
  タンスの中身がドラえもんのどこでもドアのようにつながっていると、
  無理矢理考えれば、何とかつじつまが合わせられると思った次第です
    - これで、少し無理矢理ですが、つじつまがあったと思います
        - 私もずっと若ければC言語やコンピュータ自体の学習をしたかもしれませんが、
          取り敢えず正しくPythonを扱えればよいと思います。
          Pythonの作者も
          「体に叩き込んで覚える人の場合Pythonのヘンなとこも障害にならないかもしれない」と言っています。
        - 最も親切にも「Pythonのヘンなとこ」は「Pythonチュートリアル 第4版」にも作者が書いてくれています

END, again

実は、もう1ページ追加しました。
Claudeにも相談して、今までの説明より、より明快だと思います！
↓

deep copy VS shallow copy：

問題は ミュータブル イミュータブル なのである！

[1, 2, [3, 4], 5]

上記で述べた考え方も覚えるためには良い方法なのではと思いますが、今日ひらめきました。

何がひらめいたのかと言うと、例えば リストの [ ] は 箱などではない！

そんなことです。

これは分かりやすくするための目安であって、箱などで区切られていないのです！

リストの [ ] を便宜的に箱だと言ってしまうと、内側の箱は確かにミュータブルの場合注意が必要です。

実際の値の配置がどうなのかは知りませんが。

とにかく箱ではなくミュータブル・イミュータブル の視点で考えさえすればよいのです。

これで積年の悩みが晴れた気がしています。

Claudeにも相談しましたが、その視点でよいと言ってくれました。

取り敢えず、C言語やコンピュータの仕組みを覚えないといけないのかなという不安からは

今のところ解放されました。

もちろんゆとりがある人はやった方が良いのに越したことはないでしょう。

これで、このパートは、終われたと思っています！

The END