Pythonのスコープ：LEGBルールと例外、そして管理キーワード

目次

1. スコープの基本：LEGBルール

- スコープとは、変数・関数が有効になる範囲のことである
- LEGBとは
    - スコープの狭い範囲 ➡ 広い範囲
    - local → enclosing → global → built-in

local(ローカル)・enclosing(エンクロージング)・global(グローバル)・ビルトイン(built-in)

- コードで確認
- 自分(変数・関数)が、今いる所から、より広い範囲に向けて探す
    - 最初に見つかったものを使う

```
third_global = "third"  # グローバルスコープ


def func():
    second_enclosing = "second"  # エンクロージングスコープ

    def inner():
        first_local = "first"  # ローカルスコープ

        # Pythonは、変数を、LEGB の順番で探し、
        # 最初に見つかったものを使います

        # ① ローカルスコープ
        print(first_local)
        """結果
        first
        """

        # ② エンクロージングスコープ
```

```
        print(second_enclosing)
        """結果
        second
        """

        # ③ グローバルスコープ
        print(third_global)
        """結果
        third
        """

        # ④ どこからでもアクセス出来るlen関数は、ビルトイン関数
        # ビルトイン関数自体を参照
        print(
            f"len関数はビルトインスコープに存在します: {len}"
        )
        """結果
        len関数はビルトインスコープに存在します: <built-in function len>
        """
        # ビルトイン関数の使用例
        print(f'文字列の長さ：len("fourth")：{len("fourth")}')
        """結果
        文字列の長さ：len("fourth")：6
        """

    inner()


func()
```

2. LEGBには収まらない特別なルール

if文・for文などは、特別なスコープを持たないことに注意

- 特に他の言語における `{ }` のことは、Pythonを書く時は忘れましょう

内包表記内の変数はその中で使われるのみである

- for文で実行すると、for文としてのスコープは持っていないので、
  ループの中で使われる変数は、残ったままです
- そして、その前などで同じ名前の変数を定義していたとしたら、上書きされることなどが考えられます
- 内包表記ではループ変数のスコープは、内包表記内に限定されます

```
fruit = '🍎'
print(f"私は{fruit}が好きです")

fruits = ["🍎", "🍍", "🍇"]

for fruit in fruits:
    print(fruit)
```

```
print(f"私の好きな果物は{fruit}です")  # 🍎から🍇に変わってしまっている
print("あなたはの好きな果物は何ですか？")

print("\n", "---" * 10, "\n")

food = "カレー"
print(f"わたしは{food}が好きです")

foods = ["カレー", "うどん", "そば"]

# 以前のコードで使った`food`を使ってしまっているが、
# リスト内包表記内ではスコープが閉ざされているので、問題は起きない
foods_lst = [food for food in foods]
print(foods)

# 内包表記内で使った変数foodに影響を受けていない
print(f"私の好きな食べ物は{food}です")
print("あなたはの好きな食べ物は何ですか？")


"""実行結果

私は🍎が好きです
🍎
🍍
🍇
私の好きな果物は🍇です
あなたはの好きな果物は何ですか？

 ------------------------------

わたしはカレーが好きです
['カレー', 'うどん', 'そば']
私の好きな食べ物はカレーです
あなたはの好きな食べ物は何ですか？
"""
```

クラススコープの立ち位置と特別な扱い方法

- スコープ構造

```
・LEGBルールに照らし合わせてみる
・関数スコープが、E・L(外側の関数がenclosing,内側の関数がlocalに当たる)
・クラススコープは、関数と並列関係

Built-ins
└── Global
    ├── 関数スコープ（ネスト可能）
    └── クラススコープ（特殊ルール）
```

- ここで、クラスが特別なルールになるのは、クラスと関数の構造が違うからです
    - 構造が違えばルールも違うのは当然と言えば当然でしょう

- コードで確認（説明はコード内に書いた方が説明に直結すると思うのでそうしました）

```python
class MyClass:
    class_var = "クラス変数"

    # クラス変数と同じレベルであればアクセス可能
    # ここでアクセス出来るのは、Pythonのスコープらしいそのままですね
    #   このprint()文は、Pythonインタプリタが、`class MyClass`を、
    #   読み込んでいる時に、実行されます
    #     クラスをインスタンス化した時に実行されるというわけではないです！
    print(class_var)
    """ 実行結果：クラス変数 """

    # コンストラクタ
    def __init__(self, instance_data):
        # インスタンス変数
        self.instance_data = instance_data
        # 以下は、print()文なので、インスタンス化してすぐに実行される
        print(f"コンストラクタ実行: {self.instance_data}")

    def method(self):
        # print(cls.class_var) # cls そもそも持っていない
        # print(class_var)  # ここはLEGBな感じで言うと関数なので見えない

        # self を使ってアクセス可能
        # これが有効なのは、class変数は、インスタンスにとっても共通だから。
        # この方法は実は推奨される方法である
        #   ただし、ここで変更すると、クラス変数ではなく、
        #   インスタンス変数に代わるので要注意である！
        print(self.class_var)

        # この書き方は有効
        # しかしこの形は、うまく行きそうではあるが、断定しずらいと思います
        #   実は推奨される方法ではない
        #   オブジェクト指向を深く理解すると分かるとAIに聞きました
        print(MyClass.class_var)

    @classmethod
    def class_method(cls):
        # print(self.class_var)  # selfは引数に無い
        # print(class_var)  # 見えないのは、直感的にも分かるかんじです

        # クラスメソッドは、cls を使ってアクセス可能
        # 要は、self からも、cls からも見えているということですね
        print(cls.class_var)

        # この中でもこの書き方は有効である
        # ただしオブジェクト指向の観点から推奨される方法ではない
        print(MyClass.class_var)

    # スタティックメソッドには、self(インスタンス) も cls(クラス) もない
    # アクセスするには、クラス名を使う
    @staticmethod
    def static_method(x, y):
        print(x, y)
```

```
# クラスの初期化
obj = MyClass("I'm obj data")
""" 実行結果：コンストラクタ実行：I'm obj data"""

# インスタンスメソッドにアクセス
obj.method()
"""「クラス変数」を2回出力する """

# クラスメソッドにアクセス
MyClass.class_method()
"""「クラス変数」を2回出力する """

# スタティックメソッドにアクセス
MyClass.static_method("static", "method")
""" static method """
```

まとめ

- クラス変数は、クラス内のトップレベルの変数であり、同じレベルであれば変数名だけでアクセスできる
- コンストラクタは、クラスがインスタンス化される時に、そのインスタンスのデータを生成する役割である
- インスタンスメソッドは、クラスのインスタンスが呼び出して使うものである
  - self から、クラス変数にアクセス出来る
    - この場合、値を変更すると、その値はインスタンス変数になる
    - それはスコープがインスタンスメソッド・エリアにある変数は、Pythonがそのように扱うからである
    - クラス変数にアクセス出来ると言っても、正確には、クラス変数の値にアクセスしているのである
- クラスメソッドは、クラス自身(`cls`)に紐づくメソッドであり、クラス変数へのアクセスなどを行います。
  - クラス変数へは `cls` を通してアクセスします。
    - メソッドの定義には `cls` が書かれますが、呼び出す際には記述しません。
  - その他に、`代替コンストラクタ` としての役割を果たすことができます。
    - これにより、通常のコンストラクタの引数をより柔軟に扱うことが可能になります。
    - 詳細なコード例は下の「コラム」で示します
- スタティックメソッドは、self も cls も、持っていない、便利関数のような役割である
  - そのクラスにあると便利な機能を受け持つことが多い
  - アクセスするには、`クラス名.スタティックメソッド` である

- クラスの初期化
  - オブジェクト = クラス名(コンストラクタのself以外の引数)
    - self しかない時は、空のカッコである
    - self は、書かないルールである
      - ただ、コンストラクタ定義にはあることを覚えておくべきである
- インスタンスメソッドにアクセス
  - オブジェクト(クラスのインスタンス).インスタンスメソッド(＊)
    - ＊：引数に self は書かない・その他の引数はメソッドの構文に合わせる
- クラスメソッドにアクセス
  - クラス名.クラスメソッド(＊)
    - ＊：引数に cls は書かない・その他の引き数はメソッドの構文に合わせる

- スタティックメソッドにアクセス
  - クラス名.スタティックメソッド(＊)
    - ＊：メソッドの構文に合わせて引数を決める

コラム：代替コンストラクタの使い方

- 以下は、クラスメソッドを代替コンストラクタとして使う方法です
  - 以下のコードでは、コンストラクタを呼ぶときの代替方法として使えます
  - 例えば、GUIだった場合、この方が使い勝手が良いと思われます
  - クラスメソッドやスタティックメソッドは、Pythonがコードを読み込んだ時点で使えます
    - 関数と同じ感覚ですね
      - インスタンスメソッドと違って初期化するコードは不要です

```
class AutoCookingMachine:  # 自動調理マシーン

    def __init__(self, menu, ingredients):
        self.menu = menu
        print(f"材料は{ingredients}ですね")
        print("少々お待ちください")
        print(f"{menu}の出来上がりです！")

    @classmethod
    def ingredients(cls):  # ingredient：料理の材料

        # input()関数をつかって、メニュー名、材料を入力する
        your_menu = input("何を作りますか：")

        your_list = []
        print("必要な材料を一つずつ入力してください")
        print("すべて入力したら「OK」を入力してください")

        while True:
            item = input("材料：")
            if item != "OK":
                your_list.append(item)
            else:
                materials_tuple = tuple(your_list)  # material: 材料
                # cls() を使って、__init__ に引数を渡すことができる
                return cls(your_menu, materials_tuple)


AutoCookingMachine.ingredients()

"""
何を作りますか：カレー
必要な材料を一つずつ入力してください
すべて入力したら「OK」を入力してください
材料：水
材料：カレー粉
材料：人参
材料：玉ねぎ
材料：チキン
```

```
材料：ジャガイモ
材料：OK
材料は('水', 'カレー粉', '人参', '玉ねぎ', 'チキン', 'ジャガイモ')ですね
少々お待ちください
カレーの出来上がりです！
"""
```

3.スコープを管理する global文・nonlocal文

- ただし、global文・nonlocal文は避けるべきである
  - なぜなら、結局 `同じ名前の変数を違うスコープで使う` ことになり、管理が難しくなるから。
  - 変数の値がどこで、いつ変更されたのかを追いかけるのが困難になり、
    予期せぬバグの原因となります

global文の書き方

- 多くの場合、関数の中で、グローバル変数を使いたい場合に使います
- コード例で見てみましょう

```python
from random import randint

"""
randint(a, b)：a～bのランダムな整数を返す(aもbも含まれる)
"""

global_scope = 0


def point_generator():  # ポイント生成
    point = 10 * randint(-10, 10)
    # グローバル変数を代入する前(変更する前)に書くこと
    global global_scope  # global文
    global_scope = point
    return point


point_generator()

print(f"あなたのポイントは{global_scope}です")
"""実行結果
あなたのポイントは80です
"""
```

- 上記のコードに見られるように、`global_scope` を変更する前、
  つまり `グローバル変数を変更する前に` 、`global文` を書きます

- 今度は、グローバル変数を、global文を使わずに、グローバル変数を参照してみましょう

```
from random import randint

global_scope = 0


def point_generator():  # ポイント生成
    # グローバル変数を変更しない場合は、アクセス出来る
    print(f"global_scope: {global_scope}")


point_generator()  # global_scope: 0
```

- なんと、global文を書いていないのに、アクセス出来ました。
- これは何故なのでしょうか？
- つまり、global文が存在する意味は、ある種の警告なのです
  - `スコープの違うところで、global変数を変更します`と言っています
  - また`変更をしなければ問題は起きません`
- なぜ警告しなければならないか
  - 後々、グローバル変数を使う時、global文を使っていることに気付かないと、バグの原因になる
  - global文を見つけない限り、そのことには気づかず、ローカル変数だと思うでしょう
    - 特に、以前に書いたコードを見る時・大規模なコード・共同して書いている時
      - こんな時は、もしコードが上手く動いていたら、そのまま書き続けるかもしれません
        - 後から直す時その部分だけ検索して直せられればラッキーかもしれません

global文を使わない代替案（グローバル変数の値だけを使う方法）

- 関数の引数は、関数のスコープです
  - つまり、エンクロージング か ローカル の、スコープです
  - エンクロージングのエリアが無いので、ローカルとして考えましょう

```
from random import randint

"""
randint(a, b)：a～bのランダムな整数を返す(aもbも含まれる)
"""

global_scope = 0


def point_generator(value):  # ポイント生成
    point = 10 * randint(-10, 10)
    # ここでは、グローバル変数を使っていないので、global文は不要です
    # global global_scope  # global文
    value = point
    return value


# この文は、グローバル変数を引数に割り当てているように見えますが、
```

```
# この引数は、関数のスコープであり、ローカル変数です
# 単に、global_scope の値を使っているだけなのです
# ここで、グローバル変数が変更されたと思っては~~行きません~~（落とし穴）
result = point_generator(global_scope)
```
いけません

```
print(f"あなたのポイントは{result}です")
"""実行結果
あなたのポイントは100です
"""

print(f"グローバル変数としての`global_scope`の値：{global_scope}")
"""実行結果
グローバル変数としての`global_scope`の値：0
"""
```

nonlocal文の書き方

- `nonlocal`文は、**ネストされた関数**（関数の中に定義された関数）の中で、
  一つ外側の関数（エンクロージングスコープ）のローカル変数を変更したい場合に使います。

- `global`文が「グローバルスコープの変数を変更する」のに対し、
  `nonlocal`文は「**一つ外側の、グローバルではないスコープの変数を変更する**」
  と考えると分かりやすいでしょう。

- コード例で見てみましょう。

```
def outer_function():
    # outer_functionのローカル変数（エンクロージングスコープの変数）
    message = "Hello"
    print(f"outer_function 実行開始時: message = {message}")

    def inner_function():
        # inner_functionのローカルスコープ
        # ここで `message` を代入しようとすると...
        # もし `nonlocal` がないと、inner_functionの新しいローカル変数になる
        nonlocal message # nonlocal文：外側の`message`を変更することを宣言
        message = "World" # 外側の`message`が変更される
        print(f"inner_function 実行時: message = {message}")

    inner_function()
    print(f"outer_function 実行終了時: message = {message}")

outer_function()

"""実行結果
outer_function 実行開始時: message = Hello
inner_function 実行時: message = World
outer_function 実行終了時: message = World
"""
```

- 上記のコードのように、`inner_function`内で`nonlocal message`と宣言することで、
  `message = "World"`という代入が、`inner_function`のローカル変数を作るのではなく、
  外側の`outer_function`の`message`変数を直接変更します。

- **`nonlocal`文も、`global`文と同様に、使うべきではないとされることが多いです。**
  - 変数の変更箇所が分かりにくくなり、コードの可読性や保守性を低下させる原因となるためです。
  - 通常は、クロージャやクラス、あるいは引数や戻り値を使うことで代替できます。

END