Photo by Shamim Nakhaei

クラスと self と __init__

はじめに

まず __init__ が、かなり難しくて、つまずきどころかなと思います。 なぜ難しいかというと２つの理由があります。

１つ目は、難しい概念が色々と隠れているからです。 ２つ目は、自動的に色々とやってくれているからです。

そのため理解する必要はなく、 まず `__init__` の動作を　覚えてしまうこと　が大事かなと感じたりもします。

伝わるかは厳しいのですが、以下の順に見ていきたいと思います。

Step 1. オブジェクトってなに？

「値」と「処理」をまとめたものです。

ここに `tora` という猫がいたとします。

`tora` は２つの「値」を持っています。 `tora` は、名前 `name` があり、 `とら` と言います。 `tora` は、動物として分類 `family` されていて、 `猫科` に属します。

```
>>> tora.name
'とら'
```

```
>>> tora.family
'猫'
>>>
```

`tora` は２つの「処理」を行うことができます。 `tora` は、鳴く `say` することができます、 `にゃー` と。 `tora` は、唸る `growl` することができます、 `ウー` と。

```
>>> tora.say()
にゃー
>>> tora.growl()
ウー
>>>
```

Step 2. クラスってなに？

共通 の「値」と「処理」をまとめたものです。

猫は、動物として分類 `family` されています。 みな鳴くこと `say` ができますし、唸ること `growl` もできます。 もし Step 1　で書かれた猫を Python で書けば次のようになります。

```py
#
# 対話モード >>> に
# コピペで動きます。
#
class Cat:
    family = '猫'

    def say(self):
        print('にゃー')

    def growl(self):
        print('ウー')

tora = Cat()
tora.family
tora.say()
tora.growl()
```

```
>>> tora.family
'猫'
>>> tora.say()
にゃー
>>> tora.growl()
ウー
>>>
```

Step 3. インスタンス化ってなに？

クラスからオブジェクトを作ること

書式は以下の通りです。

```py
オブジェクト = クラス名()
```

```
>>> tora = Cat()  # <--- ここがインスタンス化
>>> tora.family
'猫'
>>> tora.say()
にゃー
>>> tora.growl()
ウー
>>>
```

インスタンス化したばかりのオブジェクトの名前 `name` を使おうとするとエラーになります。

```py
#
# 対話モード >>> に
# コピペで動きます。
#
class Cat:
    family = '猫'

    def say(self):
        print('にゃー')
```

```
    def growl(self):
        print('ウー')

tora = Cat()
tora.family
tora.say()
tora.growl()

tora.name  # <--- エラーになります。
```

```
>>> tora.name  # <--- エラーになります。
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Cat' object has no attribute 'name
>>>
```

なんでエラーになったのでしょうか？ それはまだ名前 `name` をつけてあげていないからです。

> 吾輩わがはいは猫である。名前はまだ無い。
>
> 夏目漱石 - 吾輩は猫である

名前 `name` は猫ごとに違います。 生まれたての猫には、まだ名前 `name` がないという訳です。

エラーにならないように名前をつけてあげたいと思います。

```py
#
# 対話モード >>> に
# コピペで動きます。
#
class Cat:
    family = '猫'
```

```
    def say(self):
        print('にゃー')

    def growl(self):
        print('ウー')

tora = Cat()
tora.family
tora.say()
tora.growl()
tora.name = 'とら'  # <---  1行追加しました。
tora.name
```

```
>>> tora.name
'とら'
>>>
```

エラーが消えました。

Step 4. __init__ ってなに？

インスタンス化する時の処理を書く関数

インスタンス化する時に実行する処理を追加する オブジェクトをインスタンス化する時に、ちょっと改造したいことがあります。

例えば、猫クラス `Cat` をインスタンス化してから名前をつけるのは面倒です。

```py
tora = Cat()          # インスタンス化してから
tora.name = 'とら'   # 名前をつける
```

こうやって１行で書けたら、良さそうです。

```py
tora = Cat()
```

インスタンス化する時に名前をつけたいです。 インスタンス化する時に `__init__` が、自動的に呼び出されるのでこれを使います。

```py
#
# コピペで動きます。
#
class Cat:
    family = '猫'

    def say(self):
        print('にゃー')

    def growl(self):
        print('ウー')

    def __init__(self):
        # 1. 第一引数 self には
        #    名前のない猫オブジェクトが
        #    自動的に代入されています。
        self.name = 'とら'

        # 2. self は return しない
        # return self

tora = Cat()
tora.name
```

```
>>> tora.name
とら
>>>
```

return はいらない。

関数とは違い return 文を使わなくても 自動的に self が返されます。

Step 5. self ってなに？

self という文字をたくさん見るのですが、 self には「生まれたての猫」が入っています。 もう少し補足します。

```py
#
# コピペで動きます。
#
class Cat:
```

```
    family = '猫'

    def say(self):
        print('にゃー')

    def growl(self):
        print('ウー')

    def __init__(生まれたての猫):
        生まれたての猫.name = 'とら'

tora = Cat()
tora.name
```

```
>>> tora = Cat()
>>> tora.name
'とら'
>>>
```

Step 6. __init__ を書き換える

すべての猫の名前が「たま」なのかな？と疑問に思われた方、それは非常に正しい判断です。 ちゃんと名前を変えてあげられるようにしたいと思います。

ここで大事なのは `__init__(self, name)` メソッドと、 `Cat("ドラえもん")` の 引数の数が違うこと です。

```py
#
# コピペで動きます。
#
class Cat:
    family = '猫'

    def say(self):
        print('にゃー')

    def growl(self):
        print('ウー')

    def __init__(self, name):  # <--- __init__ の引数
        self.name = name

tama = Cat('たま')              # <--- インスタンス化の引数
tama.name
dora = Cat('ドラえもん')
dora.name
```

```
>>> tama = Cat('たま')
>>> tama.name
'たま'
>>> dora = Cat('ドラえもん')
>>> dora.name
'ドラえもん'
>>>
```

はじめに

補足

◯ なんで self を書かないといけないの？

頭に self をつけていない変数は、あとから参照できません。 自分は Python を習いたての頃、 `self` をつけ忘れているのに気づけなくて普通に数日溶かしました笑

```py
#
# コピペで動きます（エラーで弾かれます）。
#
class Cat:
    def __init__(self):
        name = 'とら'

tora = Cat()
tora.name
```

```
>>> tora.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Cat' object has no attribute 'name
>>>
```

なぜ `self` をつけないと、名前がつけられないのでしょうか？ それは `生まれたての猫` の `name` という属性ではなく...

```py
def init(生まれたての猫, name):
    生まれたての猫['name'] = name

cat = {'name': '我輩は猫である名前はまだない'}
```

```
init(cat, 'とら')
cat['name']
```

```
>>> cat['name']
'とら'
>>>
```

たんなる `name` という変数に代入しているからです。

```py
def init(cat, name):
    name = name

cat = {'name': '吾輩は猫である名前はまだない'}
init(cat, 'とら')
cat['name']
```

```
>>> cat['name']
'吾輩は猫である名前はまだない'
>>>
```

知らなくても大丈夫ですが、もうちょっと細かくいうと、 `__init__` は、関数です。 そしてここで書いた `self` あるいは `生まれたての猫` は、関数 `__init__` の第一引数です。

- 関数とメソッドの違いってなに？

◯ PEP 8

`self` じゃなくてもいいんです。 `cat` でも動きます。

```py
#
# コピペで動きます。
#
class Cat:
    def __init__(cat):
        cat.name = 'とら'

    def say(self):
        print('にゃー')

    def growl(self):
        print('ウー')

tora = Cat()
tora.name
```

```
>>> tora.name
'とら'
>>>
```

ここで大事なことは `self` と書かなくても動くと言うことです。 じゃあなんでみんな `self` って書いているのでしょうか？

それは偉い人たちがみんなでそうやって書こうね、と決めたからです。 PEP 8 という文章で定められています。PEP 8 は、知らなくても大丈夫です。

- PEP 8 ってなに？

◯ クラス変数とインスンタンス変数

オブジェクトで共有する属性を「クラス変数」、 オブジェクトだけで使う属性を「インスタンス変数」と言います。

ここで大事なことはクラス変数には `self` が不要です。 そしてインスタンス変数には `self` が必要だということです。

```py
#
# コピペで動きます。
#
class Cat:
    #
    # クラス変数
    # self は不要
    #
    family = '猫'

    def say(self):
        print('にゃー')

    def growl(self):
        print('ウー')

    def __init__(self, name):
        #
        # インスタンス変数
        # self は必要
        #
        self.name = name

tama = Cat('たま')
tama.family
tama.name

dora = Cat('ドラえもん')
dora.family
dora.name
```

```
>>> tama = Cat('たま')
>>> tama.family
'猫'
>>> tama.name
'たま'
>>>
>>> dora = Cat('ドラえもん')
>>> dora.family
'猫'
```

```
>>> dora.name
'ドラえもん'
>>>
```

- クラス変数とインスンタンス変数ってなに？

__init__ が結構難しい

まず __init__ が、結構難しいです。 理由は大きく分けて 2 つあります。

理由 1 難しい概念がしれっと入っているから

まず第一に、「名前空間」、「スコープ」といった難しい概念がしれっと入っているからです。

- 名前空間ってなに？
- スコープってなに？

「スコープ」？そんなの知ってるよ！って感じですし、「名前空間」も「苗字と名前」くらいの簡単な概念なのですが。

ただ、言葉では理解できていても、使えるかどうかとなると、なかなか四苦八苦する感じになるかなと。 自分は四苦八苦しました笑

回数こなしていけば、動作を覚えてしまうと思います。 無理して、すぐに理解する必要はないかなとも思ったりもします。

なぜなら、掘り下げて理解しようとすると、やる気が続かないこともあるからです。 とりあえず、これはこういうものということで、先に進めたほうが、楽しいかなと思ったりもします。

とはいえ、掘り下げて１つずつ積み重ねたほうが理解が速い場合があるので、
それも人によりけりかなとも思いはするのですが...

理由２ 自動でいろんなことをしてくれるから

また第二に、いろんなことを自動的にしてくれているからです。

1. 関数とは違い自動的に、self にオブジェクトが代入される
2. 関数とは違い自動的に、インスタンス化した時に呼び出される
3. 関数とは違い自動的に、return していないのに self が返される

でも上の説明では、それを全て割愛しています... 自動的にやってくれるというのは、便利にはなるのですが、 見ていこうとすると説明が煩雑になり、逆に理解することが結構難しくなったりします。

下記の文章は、PEP 20 という Python のすごい人が書いた Python のコツみたいな文書からの引用です。 下手に自動化させるよりも面倒でもベタ書きしてもらった方がわかりやすいコードになるよ。 という意味だと個人的に思っています。

> 明示的であることは、暗黙的であるより良い。
> Explicit is better than implicit.
> PEP 20 - The Zen of Python

ポイント

__init__ は、難しい。動作を覚えてしまう。

クラスをいつ使えばいいの？

この節は Effective Python の 「項目22：辞書やタプルで記録管理するよりもヘルパークラスを使う」の劣化版です。

この手の文章を書くときに、機能を書くのはそんなに大変ではありません。 しかし、そのメリットがわかるようなサンプルコードを書くことはかなり難しいです。 Effective Python は、ちょっと難しい感じの文章なのでが、 サンプルコードは、とても良いもののように感じました。

すこし向きを変えて考えてみたいと思います。 class は、いつ使うべきでしょうか？ ごくごくデータが単純なときは tuple を使い、 すこし複雑になったら dict を使い、 厳しそうなら class を使えば、いいかなと思います。

以下、tuple, dict, class の 3 step について、ご説明いたします。 ぱっといい例が思いつきませんでしたので、 全て上の Cat クラスを説明いたします。

```py
#
# 対話モード >>> に
# コピペで実行できます。
#
class Cat:
    type = '猫科'

    def __init__(self, name, gender, age):
```

```
        self.name = name
        self.gender = gender
        self.age = age

    def say(self):
        print(self.name, 'にゃー')

    def grawl(self):
        print(self.name, 'ウー')

tama = Cat('たま', 'メス', 5)
tora = Cat('とら', 'オス', 3)

tama.type
tama.name
tama.say()
tama.grawl()

tora.type
tora.name
tora.say()
tora.grawl()
```

より良い例は 書籍 Effective Python の 「項目22：辞書やタプルで記録管理するよりもヘルパークラスを使う」を、ご参照ください。

Step 1. タプル tuple

例えば、上で書いた Cat クラスも tuple を使えば、次のようになります。 クラス変数、共通の値である '猫科' は、直接、ベタに書き込んでしまいます。

```py
#
# 対話モード >>> に
# コピペで実行できます。
#
tama = ('猫科', 'たま', 'メス', 5)
```

```
tora = ('猫科', 'とら', 'オス', 3)

def say(cat):
    print(cat[1], 'にゃー')

def grawl(cat):
    print(cat[1], 'にゃー')


tama[0]
tama[1]
say(tama)
grawl(tama)

tora[0]
tora[1]
say(tora)
grawl(tora)
```

```
>>> tama[0]
'猫科'
>>> tama[1]
'たま'
>>> say(tama)
猫科 にゃー
>>> grawl(tama)
猫科 にゃー
>>>
>>> tora[0]
'猫科'
>>> tora[1]
'とら'
>>> say(tora)
猫科 にゃー
>>> grawl(tora)
猫科 にゃー
>>>
>>>
```

あまりたいしたものでないものに class を使うのは煩雑だったりします。 例えば上で書いた Cat クラスは __init__ メソッドが長いです。 そんな面倒なときは tuple を使います。

Python の tuple は list ととてもよく似ています。 もともとの由来は、このようなデータ型を表すために作られました。 ABC 言語の Compounds 型に起源をもっています。

- Python の tuple ってなに？ - Mastering Python

もし添字表記 `cat[0]` だと読みづらくて嫌だった場合は tuple をアンパックすることも 1 つの方法です。

```py
#
# 対話モード >>> に
# コピペで実行できます。
#
tama = ('猫科', 'たま', 'メス', 5)

def say(cat):
    _, name, _, _ = cat
    print(name, 'にゃー')

say(tama)
```

```
>>> say(tama)
たま にゃー
>>>
```

タプルの分割代入

Python では以下のように イテラブル を右辺において、 左辺の変数にまとめて代入できます。

```py
要素1, 要素2, 要素3, ... = イテラブル
```

```py
#
# 対話モード >>> に
# コピペで実行できます。
#
name, gender, age, family = ('たま', 'メス', 5, '猫科'
name
gender
age
family
```

```
>>> name
'たま'
>>> gender
'メス'
>>> age
5
>>> family
'猫科'
>>>
```

- 代入のいろいろな書き方

_ アンダーバー

決まりでは無いのですが Python では 使わない変数はアンダーバー `_` に代入することが多いです。

```py
#
# 対話モード >>> に
# コピペで実行できます。
#

# 10 回 'Hello, world!'
# 使わない変数は _ に
for _ in range(3):
    print('Hello, world!')
```

```
>>> for _ in range(3):
...     print('Hello, world!')
...
Hello, world!
Hello, world!
Hello, world!
>>>
```

Step 2. 辞書 dict

tuple で辛くなったら dict を使います。 うーん、この例でも、わかり辛い... すいません。 Effective Python にいい例があります。

```py
#
# 対話モード >>> に
# コピペで実行できます。
#

# 1. リテラルとして書いて、辞書を作る。
tama = {
    'type': '猫科',
    'name' :'たま',
    'gender': 'メス',
    'age': 5
}

# 2. インスタンス化する書き方で、辞書を作る。
tora = dict(
    type='猫科',
    name='とら',
    gender='オス',
    age=3
)

def say(cat):
    print(cat['name'], 'にゃー')

tama['type']
```

```
say(tama)
tora['type']
say(tora)
```

```
>>> tama['type']
'猫科'
>>> say(tama)
たま にゃー
>>> tora['type']
'猫科'
>>> say(tora)
とら にゃー
>>>
```

dict は 2 つの作り方があります。 上のコードでは tama はリテラルとして書く方法で辞書を作りました、 tora はインスタンス化する書き方で辞書を作りました。

リテラル なんていう小難しい言葉を使いましたが、 '0', 'abc' などの書いたままのものをリテラルと言ったりします。

> リテラル (literal) とは、いくつかの組み込み型の定数を表記したものです。
> 2.4. リテラル - Python 言語リファレンス

インスタンス化して作る方法はクォート `'` を書かなくて済み、 見た目も綺麗になるので、たまに使います。

Step 3. クラス class

tuple や dict で対応できなくなったら、class を使います。

```py
#
# 対話モード >>> に
# コピペで実行できます。
#
class Cat:
    type = '猫科'

    def __init__(self, name, gender, age):
        self.name = name
        self.gender = gender
        self.age = age

    def say(self):
        print(self.name, 'にゃー')

    def grawl(self):
        print(self.name, 'ウー')

tama = Cat('たま', 'メス', 5)
tora = Cat('とら', 'オス', 3)

tama.type
tama.name
tama.say()
tama.grawl()

tora.type
tora.name
tora.say()
tora.grawl()
```

「対応できなくなったら」ってなんやねんって感じですが、 タプルの添字表記 `lst[0]` や辞書の添字表記 `dct['a']` が 面倒になったらクラスを使う くらいに自分は考えています。

データサイエンス系の人のつぶやきを見てたら、 あんまりクラス使わないけどね、って呟かれてたので。

ポイント

クラスは自分が使いたいと思うまで
無理して使わなくても大丈夫

クラスと辞書の使い分け

「クラスは自分が使いたいと思うまで使わない」なんて、 そんな手抜きでいいのしょうか？ どう考えても辞書 dict よりもクラス class の方が多機能だし、 多機能な方を使った方が良いのではないでしょうか？

使わない機能はなるべく、使わないようにする、というのが答えかなと、 個人的に思っています。 YAGNI と KISS という経験則があります。 一言でいえば、複雑なことはしないようにしましょうね。ということです。 あくまでも経験則で、論理的に正しいというわけではないのですが。

クラスと辞書の関係で言えばクラスが必要にならない限りは、 無理してクラスは使わないということかなと思っています。

◯ 例えば - タプルとリスト

Python にはタプルとリストがあります。 タプルとリストの違いは、変更できないか、できるかです。 一見、高機能なリストの方が良さそうに見えます。

以下の記事は `C#` の記事になってしまいますが、 変更できる `List` ではなく 変更できない `IReadOnlyList` を使おうと怒っています。

Python に言いかえれば、原則タプルを使い、 必要なときだけリストを使おうと言っています。 雰囲気だけ伝わればと思いました。

> `Items` の型が `List<T>` なので当然 `Add` したり `Remove` したり出来てしまう。 そんなことをすると、クエリーの結果が改ざんされてしまうことになり、 そのようなことが起こることを予期していない他の開発者のコードでバグ（というか予期しない動作）が多発するだろう。
>
> いくら、コーディングルールや、チーム内の暗黙知として、「 `Items` を変更するな」といった取り決めがあったとしても、実際に変更可能な実装になっている限り、不安は付きまとう。 人間はミスをする。
>
> 引数の型を何でも List にしちゃう奴にそろそろ一言いっておくか - Qiita

◯ 例えば - 辞書とクラス

例えば WSGI ではクラスではなく辞書が採用されています。 なに言ってるんだ？って感じだと思います。 WSGI っていう凄いものにも、クラスではなく辞書が使われているんだな、ということだけ押さえておいてください。

WSGI が何かは知らなくても全く問題ありません。 WSGI については、以下の動画がとてもわかりやすく、大変オススメです。

- 基礎から学ぶ Web アプリケーションフレームワークの作り方 - YouTube

動画はわかりやすいですが、自分はあまり理解していません。 WSGI は、ウェブアプリケーション、 例えば Django や Flask の中で使用されている仕組みです。

ウェブアプリケーションは極論、リクエストを受けたら、関数が起動しているだけです。 その時の関数が取る引数には、ユーザ定義クラス class ではなく、あえて dict が採用されています。

> environ はなぜ辞書でなければならないのか? サブクラスを使用すると何が問題なのか?
>
> 辞書を必要とする理由は、サーバ間の移植可能性を最大にすることである。
>
> PEP 3333: Python Web Server Gateway Interface

なぜユーザ定義クラス class はダメで辞書 dict なのか、わかっていないのですが、 メソッドを勝手に付け加えられたら困るからかなと思っています。

いずれにせよ、これだけ広く使われるものに対しても class ではなく dict が 採用されているので、ご自身が class 必要だなと感じるまでは、 dict を使った方が良いのではないかなと個人的に思ったり、思わなかったりしています。

◯ まとめ

入門系の書籍ではよく、「オブジェクト指向は再利用性が高まります。」って煽られるので、class 定義文を使わないといけないのかなと感じてしまうところがあったので書きました。

class 定義文を使わない方が良いという訳ではないのですが、YAGNI と KISS の精神を大切にして、そんなに無理して class 定義文を使わなくてもいいかなと思います。

おわりに

最後は、辞書とクラスの使い分けを元に YAGNI と KISS という経験則に触れました。どうやら、多機能であることが、必ずしも歓迎されなさそうです。

← クロージャと nonlocal

オブジェクト指向プログラミング →