# クラスとオブジェクトの操作

- 重要ポイント
  - クラス定義
  - self
  - インスタンス
  - インスタンス変数
  - クラス変数
  - 継承
  - クラスの判定

## クラスとは

- Pythonで扱うオブジェクトは`データとメソッド`を持っている。
  そんなオブジェクトの設計図であるclassはどんな構造をしているのかを探る
- クラスの例：(初めて学習する人はまだ分からなくても大丈夫であります)

```
class Dog:
    def __init__(self, name, age):
        self.name = name  # データ（属性）
        self.age = age    # データ（属性）

    def bark(self):       # メソッド（bark：吠える）
        return f"{self.name}が吠えています：ワンワン！"

    def get_info(self):   # メソッド（info：information,情報）
        return f"{self.name}は{self.age}歳です"
```

## クラス定義

- クラスは`データ（属性）とメソッド（機能）`を持つ
  - 例：犬クラス
    - データ：名前、年齢
    - メソッド：吠える、情報を取得する
- 文字列やコレクション（リスト・辞書など）も同様の構造を持つ組み込みクラス

## クラスの仕組みを探る

**最小のクラス定義　➡　動的にクラスにデータを追加していく方法**

- pass文のみの構成

```
# dog.py
class Dog:
    pass
```

```
# dog_test.py
from dog import Dog

bull = Dog()  # 初期化
bull.name = "Bull"  # Dogクラスにデータを追加している
bull.age = 3  # Dogクラスにデータを追加している

print(bull.name)
print(bull.age)
"""
Bull
3
"""
```

**コンストラクタ(__init__メソッド)を使って、**
**より簡単に、クラスオブジェクト(インスタンス)を作成する**

- コンストラクタ：__init__メソッド
  - 特別な関数と考える
  - 第1引き数は、`self`である（self：classのオブジェクト自身と言う意味）
    - コンストラクタ関数を呼び出す時は、`self は書かない`ことに注意する！

```
# dog2.py
class Dog2:
    # コンストラクタ（第1引き数は、selfである）
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
# dog2_test.py
from dog2 import Dog2

# この場合、1行で、Dog2オブジェクトを生成できる
alex = Dog2("Alex", 4)
```

```
print(f"Alex: {alex.name}, {alex.age}歳")  # Alex: Alex, 4歳
```

### クラスオブジェクトが持つ特別な関数であるインスタンスメソッドを作成する

- ここでは、クラスオブジェクト(インスタンス)が持つインスタンスメソッドを作成する
  - 第1引き数を`self`とする
    - インスタンスメソッドを呼び出す時は`selfは書かない`ことに注意！

```
# dog3.py
class Dog3:
    # コンストラクタ（第1引き数は、selfである）
    def __init__(self, name, age):
        self.name = name  # インスタンス変数（データ）
        self.age = age  # インスタンス変数（データ）

    # クラスオブジェクト(インスタンス)が持つインスタンス・メソッドを作成する
    # 第1引き数は「self」である ➡ 呼び出す時は書かないルールである
    def bark(self):  # メソッド（bark：吠える）
        return f"{self.name}が吠えています：ワンワン！"

    def get_info(self):  # メソッド（info：information,情報）
        return f"{self.name}は{self.age}歳です"
```

```
# dog3_test.py
from dog3 import Dog3

# インスタンスを生成(初期化)する
pochi = Dog3("ポチ", 5)

# インスタンスの属性にアクセスする
print(f"ポチ: {pochi.name}, {pochi.age}")

# インスタンスのメソッドを呼び出す
print(pochi.get_info())
print(pochi.bark())

"""
ポチ: ポチ, 5
ポチは5歳です
ポチが吠えています：ワンワン！
"""
```

## クラスメソッドの使い方

- @classmethod
    - クラスメソッドの前置きとして書く事は必須である
    - クラスメソッドは、クラス名でクラスの外からアクセス出来る
    - 実は、クラスのオブジェクト(インスタンス)からもアクセス出来る
        - これは、オブジェクトのメソッドでもあるという考え方
            - ただし、名前から考えても紛らわしいので避けるのが一般的である
- cls
    selfが、インスタンス自身を表現するのに対して、clsはクラス自身を表現している

```
class MyClass:
    count = 0  # クラス変数

    @classmethod
    def get_count(cls):  # clsは、クラスを表現している：スコープはメソッド内だけ
        print(cls.count)  # 内部でclsを使用


# 外部からのアクセス方法
MyClass.get_count()  # クラス名でアクセス、実行結果：0
obj = MyClass()
obj.get_count()  # インスタンスからもアクセス可能、実行結果：0
```

## クラス変数とクラス変数へのアクセス

```
# Pythonのクラス変数を学ぼう！（plushtoy.py）

# (object): 継承元のクラス
class PlushToy(object):  # plush toy : ぬいぐるみ
    """ぬいぐるみクラス - クラス変数でコレクションの総数を管理"""

    # クラス変数：全てのぬいぐるみで共有される
    total_collection = 0

    def __init__(self, name, animal_type, size="M"):
        """新しいぬいぐるみを作成"""
        self.name = name  # インスタンス変数（個別）
        self.animal_type = animal_type  # インスタンス変数（個別）
        self.size = size  # インスタンス変数（個別）

        # ぬいぐるみが作られるたびに総数を1増やす
        PlushToy.total_collection += 1
        print(
            f"{name}をコレクションに追加しました！ 現在のコレクション数: \
            {PlushToy.total_collection}")

    def introduce(self):
```

```
        """ぬいぐるみの紹介"""
        return f"私は{self.size}サイズの{self.animal_type}、{self.name}です♪"

    @classmethod
    def get_collection_count(cls):
        """コレクションの総数を取得（クラスメソッド）"""
        return cls.total_collection

    # objectクラスのメソッドをオーバーライド　➡　実用的な文字列にする
    #　　クラスの、祖先は、objectクラスなのである
    # print()、str()、f-string で自動的に呼ばれる
    # この例の場合、f-stringの中でインスタンスが呼ばれる時、
    # f-stringが呼ぶ、`__str__(self:つまりインスタンス自身)__`は、
    # オバーライドされているので、このインスタンスにふさわしい文字列になるわけである
    def __str__(self): #  print()、str()、f-string で自動的に呼ばれる
        """ぬいぐるみ情報を文字列で表示"""
        return f"{self.name}({self.animal_type}・{self.size}) - \
        総コレクション数: {PlushToy.total_collection}"
```

```
from plushtoy import PlushToy

# plushtoy_test.py

# 実際に使ってみよう！
print("=== ぬいぐるみコレクション管理システム ===")
# クラス関数：クラス名.get_collection_count()
print(f"最初のコレクション数: {PlushToy.get_collection_count()}")
"""
=== ぬいぐるみコレクション管理システム ===
最初のコレクション数: 0
"""

# ぬいぐるみを作ってみる（コンストラクタを呼ぶとき、selfは書かないルールである）
print("\n")
plush1 = PlushToy("くまちゃん", "クマ", "L")
plush2 = PlushToy("うさぎん", "ウサギ", "S")
plush3 = PlushToy("ぺんぎー", "ペンギン", "M")
"""
くまちゃんをコレクションに追加しました！ 現在のコレクション数: 1
うさぎんをコレクションに追加しました！ 現在のコレクション数: 2
ぺんぎーをコレクションに追加しました！ 現在のコレクション数: 3
"""

print("\n=== コレクションの紹介 ===")
print(plush1.introduce())
print(plush2.introduce())
print(plush3.introduce())
"""
=== コレクションの紹介 ===
私はLサイズのクマ、くまちゃんです♪
私はSサイズのウサギ、うさぎんです♪
```

```
私はMサイズのペンギン、ぺんぎーです♪
"""

print(f"\n=== 現在のコレクション状況 ===")
print(f"総コレクション数: {PlushToy.get_collection_count()}")
print("各ぬいぐるみの情報:")
print(f"  {plush1}")
print(f"  {plush2}")
print(f"  {plush3}")
"""
=== 現在のコレクション状況 ===
総コレクション数: 3
各ぬいぐるみの情報:
  くまちゃん(クマ・L) - 総コレクション数: 3
  うさぎん(ウサギ・S) - 総コレクション数: 3
  ぺんぎー(ペンギン・M) - 総コレクション数: 3
"""

# クラス変数は全インスタンスで共有されることを確認
print(f"\nクラス変数の確認:")
print(f"PlushToy.total_collection = {PlushToy.total_collection}")
print(f"plush1のクラス経由 = {plush1.__class__.total_collection}")
print(f"plush2のクラス経由 = {plush2.__class__.total_collection}")
"""
クラス変数の確認:
PlushToy.total_collection = 3
plush1のクラス経由 = 3
plush2のクラス経由 = 3
"""
```

---

**意地悪な問題例**（というより、難易度が高い問題）

```
# on_class_var.py

class MyClass:
    count = 0

    def get_count(self):
        print(self.count)
```

```
# on_class_var_test.py

from on_class_var import MyClass

# `print(MyClass.count)` は、何を表示しますか
obj = MyClass()
```

```
print(obj.count)  # 0
obj.count = 10
print(obj.count)  # 10
obj.get_count()  # 10

print(MyClass.count)
```

### 解答

- `print(MyClass.count)` の値は、`0` でした。

### 何故こうなるのか

- `obj.count = 10` は、objオブジェクトに新しいインスタンス変数を生成しているわけです
  - 以下にコードで説明します

```
# on_class_var_test.py

from on_class_var import MyClass

# `print(MyClass.count)` は、何を表示しますか

# MyClassのインスタンス生成
obj = MyClass()

# クラス変数countを表示
# 元はと言えば、これが出来てしまうのが、落とし穴の原因になる
# 自分でコードを書く時は、~~この書き方を止めた方が良いと思う~~
print(obj.count)  # 0    これが出来るのには、理由があるようである。
                         理由もなく出来るような甘い言語設計をするはずもないだろう。
# インスタンスobj に、インスタンス変数countを10で生成する
obj.count = 10

# 以下の表示は、インスタンスobj の、インスタンス変数なのである
print(obj.count)  # 10

# MyClassから生成した`objオブジェクトに`、
# インスタンス変数count が生成され、その値10 を表示する
obj.get_count()  # 10

# クラス変数countは、変わっていない
print(MyClass.count)  # 0
```

### self とは

- classのコンストラクタ`__init__(self, a, b, ...)`の第一引数
- 当該のインスタンス自身を表現している
- なお、このインスタンスメソッドを呼ぶときには、書かないルールになっている

```
class Member:

    def __init__(self, no, name):  # selfは当該のオブジェクト自身
        self.no = no
        self.name = name

    def show(self):  # show()
        print(f"No.{self.no}: {self.name}")


# インスタンスメソッドを呼び出す時には、self は使わない！
tom = Member(15, "Tom Smith")
lucy = Member(37, "Lucy Jones")

# show()
tom.show()  # No.15: Tom Smith
lucy.show()  # No.37: Lucy Jones
```

**データを隠す方法（ただし、裏技で見ることは出来るが普通は無意味なことである）**

- 変数名に`__`を先頭に付けることで実現
  - 外部から読み込むためには、`_クラス名__変数名`となる

```
class Member:

    def __init__(self, no, name, weight):
        self.no = no
        self.name = name
        self.__weight = weight  # `__変数`にすると、隠したい意図を表現できる

    def show(self):  # show()
        print(f"No.{self.no}: {self.name}, {self.__weight}kg")


# インスタンスメソッドを呼び出す時には、self は使わない！
tom = Member(15, "Tom Smith", 70)
lucy = Member(37, "Lucy Jones", 50)

print(tom.name)  # 隠していないので、当然見える：Tom Smith
# print(tom.weight)  # アクセスできない
# print(tom.__weight)  # 実はこれでもアクセス出来ない
# インスタンスのデータに直接アクセスするには、`_クラス名__変数名`
#   だが、これをするぐらいだったら何のために隠したのかということになる
```

```
print(tom._Member__weight)  # アクセス出来るが隠す意味がない：70

# show()
# tom.show()  # No.15: Tom Smith
# lucy.show()  # No.37: Lucy Jones
```

### 継承、オーバーライド、isinstance関数

- 継承は、あるクラスを元にして設計したい時に使う
  - ここでは、「画像生成アプリ」を継承して「画像生成アプリ・プラス」を作成しています。
  - ~~super()**init**(引数s)~~ super().__init__(引数s)：引数sの中に、`self`は含めないこと！
    - ~~ここでの`self(インスタンス自身)`は、**init**()で書かれているから。~~ ← そうではなく構文ルールでした
    - この意味は、親クラスの設定を引き継ぐのが目的である
- オーバーライドは、継承先で、そのクラスに適したメソッドに上書きすることを言う
  - ここでは、「generate_image()メソッド」をオーバーライドして、`上書き`しています
- `isinstance関数`
  - isinstance(あるオブジェクト, あるクラスまたはその派生クラス)の形式
    - 「あるオブジェクト」が
      「あるクラス/またはその派生クラス」の、オブジェクトであれば`True`
      - 派生クラス：その子供クラスや、孫クラスのように継承しているクラス
        - 「あるクラス/またはその派生クラス」はタプルで複数指定できる

```
# inheritance.py

# (object)：基盤クラスである objectクラス を継承：当たり前として書かない人も多い
class ImageGenerator(object):
    def __init__(self, price):
        self.price = price

    def generate_image(self, image):
        print(f"{image}の画像を作成しました")


# ImageGeneratorクラスを継承している ↓ (ImageGenerator)
class ImageGeneratorPlus(ImageGenerator):
    def __init__(self, price, three_d=False):
        # 元になるクラスのコンストラクタを使う方法：ここではImageGeneratorクラス
        # ここでは、selfを書かない（暗黙で使われている）
        super().__init__(price)  # ★ selfを、書いてはいけない！
        # 継承クラス独自の引数
        self.three_d = three_d

    # オーバーライド
    # `@override` は必須ではないが付けると分かりやすい
    def generate_image(self, image):
        if self.three_d == True:
```

```
            print(f"立体の{image}画像を生成しました")
        else:
            print(f"2Dの{image}画像を生成しました")
```

```
# inheritance_test.py

from inheritance import ImageGenerator, ImageGeneratorPlus

ig1 = ImageGenerator(1000)
print(f"このイメージ生成アプリは{ig1.price}円です")
print("デモンストレーションします")
ig1.generate_image("ライオン")

ig2 = ImageGeneratorPlus(2000, three_d=True)
print(f"このイメージ生成アプリは{ig2.price}円です")
print("デモンストレーションします")
ig2.generate_image("ライオン")

# isinstance関数で、継承関係を確認する
print()
print("isinstanceメソッドで継承関係をチェックする")

# ig1オブジェクトは、ImageGeneratorクラスまたはその派生クラスの、オブジェクトか調べている
# あるクラスの、派生クラスは、派生元のクラスと`is-a関係`があるという
#   つまり、派生クラス is a 派生元のクラス : 派生クラスは、派生元のクラスに等しい
#       ただし、それはこの継承という角度ど見ると、等しいと言う意味である
print(
    f"ig1 is-a ImageGenerator？：{isinstance(ig1, ImageGenerator)}"
)
print(
    f"ig1 is-a ImageGeneratorPlus？：{isinstance(ig1, ImageGeneratorPlus)}"
)

"""
このイメージ生成アプリは1000円です
デモンストレーションします
ライオンの画像を作成しました
このイメージ生成アプリは2000円です
デモンストレーションします
立体のライオン画像を生成しました

isinstanceメソッドで継承関係をチェックする
ig1 is-a ImageGenerator？：True
ig1 is-a ImageGeneratorPlus？：False
"""
```

END