Pythonプログラミング補足資料３：オブジェクト指向はじめの一歩（新題材版）

この資料は、「Pythonサイコロゲームで学ぶ！基礎からオブジェクト指向プログラミング（3/3)」の演習（オブジェクト指向プログラミング）を進める上で重要な、OOPの基本的な概念について、サイコロゲームとは異なる題材を用いて解説します。

1. オブジェクト指向プログラミング（OOP）とは？

オブジェクト指向プログラミング（Object-Oriented Programming, OOP）は、プログラムを現実世界の「モノ（オブジェクト）」のように捉え、それらの「モノ」が持つ情報（データ）と、それらが行う操作（処理）をひとまとめにして扱う考え方です。

- **オブジェクトとは？**
  - 例えば、「車」「本」「従業員」など、具体的な「モノ」をプログラム上で表現したものです。
  - 各オブジェクトは、自身に関する**データ（属性）**（例：車の色、本のタイトル、従業員の名前）と、そのデータを操作したり、オブジェクト自身が行う**処理（メソッド）**（例：車が走る、本を開く、従業員が働く）を持っています。
- **なぜOOPを使うのか？**
  - **整理整頓:** 関連するデータと処理を「クラス」という設計図にまとめることで、コードが部品化され、見通しが良くなります。
  - **再利用性:** 一度作ったクラス（設計図）から、同じ種類のオブジェクト（実体）を効率的に何個も作れます。
  - **保守性:** 機能の変更や修正が、関連するクラスの内部に限定されやすいため、プログラム全体への影響を抑えられます。
  - **拡張性:** 既存のクラスを元にして新しい機能を持つクラスを作りやすいため、プログラムの成長に対応しやすくなります。

2. クラスとインスタンス

- **クラス（Class):**

- オブジェクトの「設計図」や「型枠」です。どのような属性を持ち、どのようなメソッド（操作）ができるかを定義します。
- 例：`Car` クラスは、「車」という種類のオブジェクトが共通して持つべき特徴（色、最高速度など）や操作（加速する、停止するなど）を定義します。

```
class Car: # ”Car”という名前のクラスを定義
    # クラスの属性（クラス変数） - このクラスの全ての車に共通
    maker_country = ”Japan”

    def __init__(self, color, max_speed): # コンストラクタ
        # インスタンスの属性（インスタンス変数） - 各車ごとに異なる
        self.color = color
        self.max_speed = max_speed
        self.current_speed = 0 # 初期速度は0

    def accelerate(self, amount): # メソッド
        if self.current_speed + amount <= self.max_speed:
            self.current_speed += amount
        else:
            self.current_speed = self.max_speed
        print(f”{self.color}の車が {self.current_speed} km/h に加速しました。”)

    def brake(self): # メソッド
        self.current_speed = 0
        print(f”{self.color}の車が停止しました。”)
```

- **インスタンス（Instance）:**
  - クラスという設計図に基づいて実際に作られた「モノ」の実体です。オブジェクトとも呼ばれます。
  - `my_car = Car(”Red”, 180)` のようにして作られた `my_car` は、`Car` クラスのインスタンス（赤い色で最高速度180km/hの具体的な車）です。

```
# Carクラスのインスタンスを作成
car1 = Car(”Blue”, 200)
car2 = Car(”Silver”, 150)

car1.accelerate(50) # Blueの車が 50 km/h に加速しました。
car2.accelerate(30) # Silverの車が 30 km/h に加速しました。
print(f”car1の色: {car1.color}, car2のメーカー国: {car2.maker_country}”)
# 出力: car1の色: Blue, car2のメーカー国: Japan
```

- **`__init__` メソッド（コンストラクタ）:**
  - クラスのインスタンスが作成されるときに、**自動的に呼び出される特別なメソッド**です。
  - 主に、インスタンスが持つべき初期データ（属性）を設定するために使われます。
- **`self`:**

- クラスのメソッド内で、**そのメソッドを呼び出しているインスタンス自身**を指す特別な第一引数です。
- `self.属性名` や `self.メソッド名()` のようにして、インスタンスの属性や他のメソッドにアクセスします。

- **インスタンス変数（属性 Attribute）:**
  - 各インスタンスが個別に持つデータのことです（例: `car1` の `self.color` は "Blue"、`car2` の `self.color` は "Silver"）。
- **メソッド (Method):**
  - クラスに属する関数で、インスタンスの属性を操作したり、特定の処理を行ったりします（例: `accelerate()` メソッド）。

3. 継承（Inheritance）

- **継承とは？**
  - 既存のクラス（**親クラス**、基底クラス、スーパークラス）の特性（属性やメソッド）を引き継いで、新しいクラス（**子クラス**、派生クラス、サブクラス）を作成する仕組みです。
  - 子クラスは親クラスの機能を再利用しつつ、独自の機能を追加したり、親の機能を上書き（オーバーライド）したりできます。
  - 例：`ElectricCar` クラスが `Car` クラスを継承し、バッテリー容量という属性や充電メソッドを追加する。

```
class Vehicle: # 親クラス
    def __init__(self, brand):
        self.brand = brand
        self.is_moving = False

    def start_engine(self):
        print(f"{self.brand}のエンジンが始動しました。")
        self.is_moving = True

    def stop_engine(self):
        print(f"{self.brand}のエンジンが停止しました。")
        self.is_moving = False

class ElectricCar(Vehicle): # Vehicleクラスを継承する子クラス
    def __init__(self, brand, battery_capacity):
        super().__init__(brand) # 親クラスの__init__を呼び出す
        self.battery_capacity = battery_capacity # 子クラス独自の属性

    def charge(self):
        print(f"{self.brand}（バッテリー: {self.battery_capacity}kWh）を充電中です。")

    # 親クラスのメソッドをオーバーライド（上書き）
    def start_engine(self):
        print(f"{self.brand}の電気モーターが起動しました。静かです！")
        self.is_moving = True
```

```
my_ev = ElectricCar(”Tesla”, 75)
my_ev.start_engine() # Teslaの電気モーターが起動しました。静かです！
my_ev.charge()       # Tesla（バッテリー: 75kWh）を充電中です。
my_ev.stop_engine()  # Teslaのエンジンが停止しました。（親クラスのメソッドを利用）
```

- **super()**: 子クラスから親クラスのメソッドを呼び出す際に使います。特に `__init__` で親クラスの初期化処理を呼び出すのによく使われます。

4. カプセル化とアクセス制御

- **カプセル化（Encapsulation）:**
  - オブジェクトの内部データ（属性）を外部から直接アクセスできないように隠蔽し、データの操作は公開されたメソッドを通じてのみ行うようにする考え方です。
  - これにより、データの不正な変更を防ぎ、オブジェクトの整合性を保ちます。
- **Pythonにおける「プライベート」の慣習:**
  - 属性名やメソッド名の前に **アンダースコア `_` を1つ** 付ける（例: `_balance`）:
    - 「これは内部用なので、外部から直接変更しないでください」という開発者間の目印（紳士協定）。技術的にはアクセス可能。
  - 属性名やメソッド名の前に **アンダースコア `__` を2つ** 付ける（例: `__secret_code`）:
    - Pythonが名前を自動的に変更（名前マングリング）し、外部からその名前で直接アクセスしにくくする。より強い隠蔽。

```
class BankAccount:
    def __init__(self, owner_name, initial_deposit):
        self.owner_name = owner_name
        self._balance = initial_deposit # アンダースコア1つで「保護された」属性を示す

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            print(f”{amount}円入金しました。残高: {self._balance}円”)
        else:
            print(”入金額は0より大きい値を指定してください。”)

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
            print(f”{amount}円出金しました。残高: {self._balance}円”)
        else:
            print(”出金額が不正か、残高が不足しています。”)

    def get_balance(self): # 残高確認用の公開メソッド
```

```
        return self._balance

account = BankAccount("山田太郎", 50000)
# account._balance = -10000 # 直接変更すべきではない（できてしまうが非推奨）
account.deposit(10000)
print(f"{account.owner_name}さんの残高: {account.get_balance()}円")
```

5. プロパティ（`@property`）

- **`@property` デコレータ:**
  - メソッドを、まるで属性（インスタンス変数）のようにアクセスできるようにするための仕組みです。
  - 主に、属性へのアクセスを制御したい場合（例: 読み取り専用にする、値を取得する際に計算を挟む）に使います。

```
import math

class Circle:
    def __init__(self, radius):
        if radius <= 0:
            raise ValueError("半径は正の値でなければなりません。")
        self._radius = radius # 実際の半径は _radius に保存

    @property # これを付けると radius メソッドが属性のように扱える
    def radius(self):
        """円の半径（読み取り専用）"""
        return self._radius

    @property
    def diameter(self):
        """円の直径（計算によって求められる読み取り専用プロパティ）"""
        return self._radius * 2

    @property
    def area(self):
        """円の面積（計算によって求められる読み取り専用プロパティ）"""
        return math.pi * (self._radius ** 2)

    # 半径を変更するためのセッターメソッド（オプション）
    # @radius.setter
    # def radius(self, value):
    #     if value <= 0:
    #         raise ValueError("半径は正の値でなければなりません。")
    #     self._radius = value

my_circle = Circle(5)
print(f"半径: {my_circle.radius}")     # my_circle.radius() ではなく属性のようにアクセス
print(f"直径: {my_circle.diameter}")   # 計算結果が返る
print(f"面積: {my_circle.area:.2f}")
```

```
# my_circle.radius = 10 # @propertyだけでは書き込み不可（セッターがない場合）
# my_circle.diameter = 20 # エラー！プロパティには直接代入できない
```

`@property` を使うことで、メソッド呼び出しの `()` を書かずに済むため、コードがより自然に見えることがあります。また、内部の実装を変更しても、外部からのアクセス方法を変えずに済む利点もあります。

6. オブジェクト指向設計の恩恵

オブジェクト指向でプログラムを設計すると、特にプログラムが大きくなったり、機能が複雑になったりした場合に、以下のような多くの利点があります。

- **変更の局所化:**
  - ある機能の変更が、関連するクラスの内部に留まりやすく、他の部分への予期せぬ影響を減らせます。
  - 例：`BankAccount` クラスの利息計算ロジックを変更しても、他のクラス（例えば `Customer` クラス）には影響が出にくい。
- **コードの再利用:**
  - `Vehicle` クラスを一度作れば、それを継承して `Truck` クラス、`Bus` クラスなど、様々な種類の乗り物クラスを効率的に作れます。
- **責任の明確化:**
  - 各クラスが特定の役割や責任を持つため、プログラム全体の構造が理解しやすくなります。「何をするクラスなのか」が明確になります。
- **拡張性:**
  - 新しい種類の「モノ」や新しい「機能」を追加する際に、既存のクラスを継承したり、新しいクラスを既存のクラスと連携させたりすることで、柔軟に対応できます。
  - 例：オンラインショップシステムで、`Product` クラスを基に新しい商品カテゴリ（`BookProduct`, `ElectronicsProduct`）を追加するのが容易になる。

これらの概念は、より複雑で大規模なプログラムを構築するための基礎となります。演習を通じて、オブジェクト指向の考え方に慣れていきましょう。