Pythonオブジェクト指向をはじめよう！ 

プログラミングの新しい考え方

これまでのプログラミングと何が違うの？🤔

- プログラムが大きくなると…
  - 「このデータはどこで使ってる？」
  - 「この処理とあの処理、どう関係してる？」
  - データと処理のつながりが複雑に… 😭
- **オブジェクト指向** なら…
  - 現実世界の「モノ」のように、データと処理を **ひとまとめ** に！
  - スッキリ整理整頓 ✨

オブジェクト指向って何？💡

現実世界の「モノ」（オブジェクト）をイメージしよう！

- **例：スマートフォン** 📱
  - **データ（属性）**: 機種名、画面サイズ、バッテリー残量
  - **処理（メソッド／振る舞い）**: 電話をかける、アプリを起動する

プログラムの世界でも、このように「モノ」で考えるのがオブジェクト指向です。

クラス：オブジェクトの設計図 🛠️

- **クラス** ＝ オブジェクトの「設計図」や「型」
  - 例：たい焼きの「たい焼きの型」🐟
- 同じ種類のオブジェクトが共通して持つ「データ」や「処理」を定義します。
- Pythonでは `class` キーワードで作ります。

```
# 犬クラスの設計図（まだ空っぽ）
class Dog:
    pass
```

インスタンス：設計図から作られた実物 ✨

- **インスタンス** ＝ クラス（設計図）から作られた具体的な「モノ」
  - 例：「たい焼きの型」から作られた、あんこ味やクリーム味の「たい焼き」一つひとつ
- クラス名の後ろに `()` をつけて作ります。

```
# Dogクラスからインスタンス（具体的な犬）を作成
pochi = Dog() #  「ポチ」という犬インスタンス
hachi = Dog() #  「ハチ」という犬インスタンス
```

`__init__` メソッド：はじめまして！の儀式 👶

- インスタンスが作られるとき、**自動的に呼び出される特別なメソッド**。
- 別名：**コンストラクタ** (建設する人)
- ここで、インスタンスが最初に持つべきデータ（名前、年齢など）を設定します。

```python
class Dog:
    def __init__(self, name_param, age_param):
        print(f"{name_param}を作ります！")
        self.name = name_param  # インスタンス自身のname
        self.age = age_param    # インスタンス自身のage

pochi = Dog("ポチ", 3) # このとき __init__ が呼ばれる
print(f"{pochi.name}は{pochi.age}歳です。") # 出力: ポチは3歳です。
```

`self` って何者？🤔👉「インスタンス自身」

- `__init__` や他のメソッドの **最初の引数** に書く `self` 。
- これは、**作られようとしているインスタンスや、メソッドを呼び出したインスタンスそのもの** を指します。
- `self.name = "ポチ"` は「そのインスタンス自身の `name` に `"ポチ"` を設定する」という意味。
- メソッドを定義するときは `self` を書きますが、呼び出すときは書きません (Python が自動でやってくれます！)。

```python
class Dog:
    def __init__(self, name): # self は作られる犬自身
        self.name = name

    def introduce(self): # self はこのメソッドを呼んだ犬自身
        print(f"僕の名前は{self.name}です。")

pochi = Dog("ポチ") # Pythonがpochiをselfとして__init__に渡す
pochi.introduce() # Pythonがpochiをselfとしてintroduceに渡す
```

属性：モノが持つデータ 📊

- インスタンスが持つ、個別のデータ（状態）のこと。
  - 例： `pochi` の名前は「ポチ」、年齢は「3歳」
- `__init__` メソッドの中で `self.属性名 = 値` のように設定。
- 外からは `インスタンス名.属性名` でアクセスしたり、値を変更したりできます。

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

pochi = Dog("ポチ", 3)
print(pochi.name)  # 出力: ポチ
pochi.age = 4      # ポチの年齢を変更
print(f"{pochi.name}は{pochi.age}歳になりました。")
```

メソッド：モノができる操作 

- クラスに定義された、インスタンスが行うことができる操作（関数のようなもの）。
  - 例：「犬が吠える」「犬が歳をとる」
- メソッド定義の最初の引数は、必ず `self` （そのメソッドを呼び出したインスタンス自身）。
- メソッドの中で `self.属性名` を使って、そのインスタンス自身のデータを使えます。

```python
class Dog:
    def __init__(self, name):
        self.name = name
    def bark(self): # 吠えるメソッド
        print(f"{self.name}がワン！と吠えました。")

pochi = Dog("ポチ")
pochi.bark()  # 出力: ポチがワン！と吠えました。
```

OOPのすごいところ①：カプセル化 

情報を守り、整理する！

- **カプセル化**：オブジェクトの内部データ（属性）を隠し、決められた操作（メソッド）を通してのみアクセスするようにすること。
- **なぜ？**
  - **データ保護**: 間違った値が入るのを防ぐ (例: HPがマイナスにならないように)
  - **利用の単純化**: 使う側は内部構造を知らなくてもOK
  - **変更に強く**: 内部の仕組みを変えても、外への影響を少なくできる
- **Pythonでの工夫**:
  - 属性名の前に "__" (アンダースコア2つ) を付ける(外からアクセスしにくくする)

```python
# カプセル化の例
class Player:
    def __init__(self, name, hp):
        self.name = name
        self.__hp = hp # __で始まる属性は外から直接触りにくい

    def get_hp(self): # HPを取得するゲッター
        return self.__hp

    def set_hp(self, new_hp): # HPを設定するセッター（値のチェックも可能）
        if new_hp < 0:
            self.__hp = 0
        else:
            self.__hp = new_hp
        print(f"{self.name}のHPは{self.__hp}に。")

player1 = Player("勇者", 50)
# player1.__hp = -100 # 直接は良くない！
player1.set_hp(-10) # 出力: 勇者のHPは0に。
print(player1.get_hp()) # 出力: 0
```

OOPのすごいところ②：継承 👥

性質を引き継ぎ、拡張する！

- **継承**：既存のクラス（親クラス）の属性やメソッドをまるごと引き継いで、新しいクラス（子クラス）を作ること。
- **なぜ？**
  - **コードの再利用**: 親の機能をそのまま使える！楽ちん！
  - **階層構造**: 「動物」クラス → 「犬」クラス、「猫」クラスのように整理
  - **機能の追加・変更**: 子クラス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**）したりできる
- `super()`：子クラスから親クラスのメソッドを呼び出すときに使う。

```python
# 継承の例
class Animal: # 親クラス
    def __init__(self, name):
        self.name = name
    def eat(self):
        print(f"{self.name}が食事中。")

class Dog(Animal): # Animalを継承する子クラス
    def __init__(self, name, breed):
        super().__init__(name) # 親の__init__を呼び出す
        self.breed = breed     # Dog独自の属性
    def bark(self):
        print(f"{self.name}({self.breed})がワン！")
    def eat(self): # 親のeatをオーバーライド
        super().eat() # 親のeatも呼びつつ
        print(f"{self.name}はドッグフードが好き！")

my_dog = Dog("ポチ", "柴犬")
my_dog.eat() # Dogのeatが呼ばれる
my_dog.bark()
```

OOPのすごいところ③：ポリモーフィズム（多態性）🎭

同じ指示で、異なる振る舞い！

- **ポリモーフィズム**：「多くの形を持つ」という意味。同じ名前のメソッドを呼び出しても、オブジェクトの種類によって実行される処理が異なること。
- **なぜ？**
  - **柔軟性向上**: コードがスッキリ！「もし犬なら…もし猫なら…」が不要に。
  - **拡張性向上**: 新しい種類のオブジェクトが増えても、既存コードの変更が少ない。
- 継承とオーバーライドが土台！

```
# ポリモーフィズムの例 (Animal, Dog, Catクラスは前述のものとする)
class Cat(Animal):
    def __init__(self, name, toy):
        super().__init__(name)
        self.toy = toy
    def eat(self): # Catもeatをオーバーライド
        print(f"{self.name}は魚をカリカリ。")
    def speak(self):
        print(f"{self.name}がニャー！")

animals = [Dog("ポチ", "柴犬"), Cat("タマ", "ねこじゃらし")]

for animal in animals:
    animal.eat() # 同じ animal.eat() でも、犬と猫で動きが変わる！
    # animal.speak() # speakメソッドも同様 (DogとCatに定義されていれば)
```

実践してみよう！簡単なRPGキャラクター作り ⚔️🛡️

1. **「モノ」は？**: キャラクター、モンスター
2. **属性は？**: 名前、HP、攻撃力
3. **メソッドは？**: 攻撃する、ダメージを受ける
4. **継承は？**: `Character` クラス(親) → `Hero` クラス、 `Monster` クラス(子)
5. **カプセル化は？**: HPを不正な値にされないように
6. **ポリモーフィズムは？**: 「攻撃」で勇者とモンスターの動きを変える

```python
# RPGキャラクターのイメージ（詳細はoop.md参照）
class Character:
    def __init__(self, name, hp, attack_power): # ...略...
    def attack(self, target): # ...略...
    def take_damage(self, damage): # ...略...

class Hero(Character):
    def cast_spell(self, target): # 勇者独自の魔法
        print(f"{self.name}が魔法！") # ...略...
    def attack(self, target): # 勇者の攻撃はちょっと違う（オーバーライド）
        print(f"勇者{self.name}の渾身の一撃！") # ...略...

class Monster(Character):
    def roar(self): # モンスター独自の咆哮
        print(f"{self.name}がグオオオ！") # ...略...

hero = Hero("アベル", 100, 15, 10)
slime = Monster("スライム", 30, 8, "やくそう")
hero.attack(slime) # Heroのattackが呼ばれる
slime.attack(hero) # Characterのattackが呼ばれる
```

身近なオブジェクト指向 普段使いのPythonにも！

- **文字列（`str`）**

```
my_string = "hello"
print(type(my_string)) # <class 'str'>
print(my_string.upper()) # .upper() はstrオブジェクトのメソッド
```

- **リスト（`list`）**

```
my_list = [1, 2, 3]
print(type(my_list)) # <class 'list'>
my_list.append(4)    # .append() はlistオブジェクトのメソッド
```

まとめ：オブジェクト指向でプログラミングをもっと楽しく！ 🎉

- **部品を組み立てる** ようにプログラムを作れる！
- 複雑なものを **スッキリ整理** できる！
- **現実世界の「モノ」** のように考えられる！

オブジェクト指向をマスターして、もっと面白いプログラム作りに挑戦しよう！