オブジェクト指向プログラミング演習課題 🚀

この演習では、学んだオブジェクト指向の考え方を使って、3つの簡単なCUIゲームを順番に作成していきます。 それぞれのゲームを作ることで、クラス、インスタンス、属性、メソッドといった基本的な概念から、少しずつステップアップしていきましょう！

演習1：じゃんけんゲーム ✊✌️✋

目的：

- クラスとインスタンスの基本的な作り方を学ぶ。
- 簡単なメソッドの定義と呼び出しを体験する。
- プレイヤーとコンピュータの対戦ロジックを考える。

作るもの： プレイヤーが手（グー、チョキ、パー）を選び、コンピュータもランダムに手を出して勝敗を判定するCUIゲームです。

実行イメージ：

```
じゃんけんゲームを開始します！
あなたの手を選んでください（1:グー, 2:チョキ, 3:パー): > 1

あなた: グー
コンピュータ: パー

残念、あなたの負けです！

もう一度プレイしますか？ (y/n): > n
ゲームを終了します。
```

実装ステップのヒント：

1. **`Player` クラスを考えてみよう！**
   - **属性案**:
     - `name`: プレイヤーの名前（例: ”あなた”）

- `hand`: プレイヤーが出した手（例: 1, 2, 3 など数字で管理すると便利）

- **メソッド案**:
  - `__init__(self, name)`: 名前を受け取って初期化する。
  - `choose_hand(self)`: プレイヤーに入力を促し、`self.hand` に選択した手を設定する。

2. **`ComputerPlayer` クラスを考えてみよう！（Playerクラスを継承すると楽かも？）**
   - **属性案**: `Player` クラスと同じものを持たせられる。
   - **メソッド案**:
     - `__init__(self, name="コンピュータ")`: 名前は固定で良いでしょう。
     - `choose_hand(self)`: ランダムに手を選び `self.hand` に設定する（`random.randint(1, 3)` など）。
3. **`Game` クラスを考えてみよう！**
   - ゲーム全体を管理するクラスです。
   - **属性案**:
     - `player`: `Player` クラスのインスタンス。
     - `computer`: `ComputerPlayer` クラスのインスタンス。
     - `hand_map`: 手の数字と文字列（"グー"など）を対応させる辞書（例: `{1: "グー", ...}`）
   - **メソッド案**:
     - `__init__(self)`: `player` と `computer` のインスタンスを作成する。
     - `display_hands(self)`: 両者の手を見やすく表示する。
     - `judge_winner(self)`: `player.hand` と `computer.hand` を比較して勝敗を判定し、結果（例: 0:あいこ, 1:プレイヤー勝ち, 2:コンピュータ勝ち）を返す。
     - `play_round(self)`: 1回のじゃんけんを実行する（手の選択、表示、勝敗判定、結果表示）。
     - `start(self)`: ゲームを開始し、`play_round` を繰り返し実行するループを持つ。
4. **勝敗判定のロジックを考えよう。**
   - `judge_winner` メソッドの中で、if文を使って条件分岐します。
5. **ゲーム全体の流れ（`if __name__ == "__main__":` の中身のイメージ）:**

```
# game = Game()  # Gameオブジェクトを作成
# game.start()   # ゲームを開始
```

演習2：サイコロゲーム 🎲

目的：

- 複数の小さなクラスを連携させる方法を学ぶ。
- オブジェクトが他のオブジェクトを利用する形を体験する。

作るもの： プレイヤーとコンピュータがそれぞれサイコロを振り、出た目の大きさで勝敗を決めるシンプルなゲームです。

実行イメージ：

```
サイコロゲーム！

--- ラウンド 1 ---
あなた がサイコロを振ります... 出た目は 4 です。
コンピュータ がサイコロを振ります... 出た目は 2 です。
あなたの勝ちです！

現在のスコア: あなた 1 - 0 コンピュータ

続けますか？ (y/n): y
```

実装ステップのヒント：

1. `Die`（サイコロ）クラスを作ろう！
   - **属性案**：
     - `sides`：サイコロの面の数（デフォルトは6）
   - **メソッド案**：
     - `__init__(self, sides=6)`：面の数を設定する。
     - `roll(self)`：1から `self.sides` までのランダムな整数を返す（`random.randint(1, self.sides)`）。
2. `Player` クラスを考えよう！
   - **属性案**：
     - `name`：プレイヤーの名前。
     - `score`：プレイヤーの勝利数。
     - `current_roll`：直近で振ったサイコロの目。
   - **メソッド案**：
     - `__init__(self, name)`：名前で初期化、スコアは0。

- `roll_die(self, die_instance)`: 引数で受け取った `Die` オブジェクトの `roll()` メソッドを呼び出し、結果を `self.current_roll` に保存し、その値を返す。

3. **`Game` クラスを考えよう！**
   - **属性案**:
     - `player1`: `Player` クラスのインスタンス（人間プレイヤー用）。
     - `player2`: `Player` クラスのインスタンス（コンピュータプレイヤー用）。
     - `die`: `Die` クラスのインスタンス（ゲームで使うサイコロ）。
   - **メソッド案**:
     - `__init__(self)`: `player1`, `player2`, `die` のインスタンスを作成する。
     - `play_round(self)`:
       1. `player1` と `player2` がそれぞれ `die` を使ってサイコロを振る（`roll_die` メソッドを呼び出す）。
       2. 出た目を比較して勝敗を判定し、勝者のスコアを増やす。
       3. 結果を表示する。
     - `show_scores(self)`: 現在のスコアを表示する。
     - `start(self)`: ゲームを開始し、`play_round` と `show_scores` を繰り返し実行するループを持つ。
4. **ゲーム全体の流れ（`if __name__ == "__main__":` の中身のイメージ）：**

```
# game = Game()  # Gameオブジェクトを作成
# game.start()   # ゲームを開始
```

演習3：テキストベースRPG風バトル ⚔️

目的：

- クラスの継承とメソッドのオーバーライドを体験する。
- より複雑なオブジェクト間の相互作用を実装する。
- カプセル化の考え方を少し意識してみる。

作るもの： プレイヤーキャラクターとモンスターが交互に攻撃しあい、どちらかのHPが0になるまで戦うCUIバトルゲームです。

実行イメージ：

```
--- バトル開始！ ---
[勇者 アベル HP: 100/100] vs [スライム HP: 30/30]

勇者 アベルのターン:
1: こうげき
コマンド？ > 1

勇者 アベルのこうげき！
スライムに 15 のダメージ！
[スライム HP: 15/30]

スライムのターン:
スライムのこうげき！
勇者 アベルに 8 のダメージ！
[勇者 アベル HP: 92/100]

...（戦闘継続）...

スライムをたおした！
```

実装ステップのヒント：

1. **`Character`（キャラクター）という親クラスを作ろう！**
   - **属性案**：
     - `name`：キャラクターの名前。
     - `max_hp`：最大HP。
     - `hp`：現在のHP。
     - `attack_power`：攻撃力。
     - `is_defeated`：倒されたかどうかのフラグ（True/False）。
   - **メソッド案**：
     - `__init__(self, name, hp, attack_power)`：各属性を初期化。`is_defeated` は `False` に。
     - `attack(self, target)`：`target`（別の `Character` オブジェクト）に攻撃する。ダメージ計算（例: `random.randint(self.attack_power // 2, self.attack_power)`）を行い、`target` の `take_damage` メソッドを呼び出す。攻撃前に自分や相手が倒れていないか確認すると良い。
     - `take_damage(self, damage)`：`damage` を受けて `self.hp` を減らす。HPが0以下になったら `self.hp = 0` とし、`self.is_defeated = True` にする。HPが0未満にならないように注意（カプセル化の初歩）。
     - `show_status(self)`：現在のHPなどを表示する。
2. **`Hero`（勇者）クラスを作ろう！（`Character` クラスを継承）**
   - **属性案**：
     - （親クラスの属性に加えて）`mp`：魔法ポイント。

- `magic_power`: 魔法攻撃力。

- **メソッド案**:
  - `__init__(self, name, hp, attack_power, mp, magic_power)`: 親クラスの `__init__` を `super().__init__(name, hp, attack_power)` で呼び出し、追加で `mp` と `magic_power` を初期化。
  - `cast_spell(self, target)`: 勇者独自の魔法攻撃。MPを消費し、`magic_power` に基づいたダメージを `target` に与える。MPが足りない場合の処理も考える。
  - `attack(self, target)`（任意でオーバーライド）: 親の `attack` とは少し違う攻撃メッセージにしたり、効果を変えたりできる。`super().attack(target)` で親の処理を呼び出すことも可能。

3. **`Monster`（モンスター）クラスを作ろう！（`Character` クラスを継承）**
   - **属性案**:
     - （親クラスの属性に加えて）`special_attack_name`: 特殊攻撃の名前。
   - **メソッド案**:
     - `__init__(self, name, hp, attack_power, special_attack_name)`: 親クラスの `__init__` を呼び出し、`special_attack_name` を初期化。
     - `special_attack(self, target)`: モンスター独自の特殊攻撃。通常の `attack` より強力なダメージを与えたり、追加効果があったりするかもしれない。
4. **バトルを進行するメインの処理（関数または `Game` クラスのメソッドとして）を書こう！**
   - `Hero` と `Monster` のインスタンスを作成する。
   - どちらかの `is_defeated` が `True` になるまでループする。
   - ループの中で、プレイヤーのターンとモンスターのターンを交互に実行する。
     - プレイヤーのターン: コマンド入力を受け付け、`Hero` の `attack` や `cast_spell` を呼び出す。
     - モンスターのターン: `Monster` の `attack` や `special_attack` を呼び出す。
   - 各ターン終了後にキャラクターのステータスを表示すると分かりやすい。
5. **ゲーム全体の流れ（`if __name__ == "__main__":` の中身のイメージ）:**

```
# hero = Hero("勇者", 100, 15, 50, 20)
# monster = Monster("スライム", 30, 8, "体当たり")

# # バトルループを実行する関数を呼び出す（例）
# def battle(player_char, enemy_char):
#     # ...（上記4のメイン処理）...
#     pass
# battle(hero, monster)
```

まずは演習1から挑戦してみましょう！行き詰まったら、スライド資料を見返したり、少しずつ動かしながら確認したりするのがおすすめです。頑張ってください！