オブジェクト指向プログラミング演習課題 – 解答例

この資料では、演習課題の解答例となるPythonコードを掲載しています。 コードを読むだけでなく、なぜこのような作りになっているのか、オブジェクト指向のどの概念が使われているのかを考えながら確認してみてください。

演習1：じゃんけんゲーム ✊✌✋ – 解答例

```
# filepath: janken_game.py
import random

class Player:
    """プレイヤーを表すクラス"""
    def __init__(self, name):
        self.name = name
        self.hand = None  # プレイヤーが出した手（1:グー, 2:チョキ, 3:パー）

    def choose_hand(self):
        """プレイヤーに手を選ばせるメソッド"""
        while True:
            try:
                choice = int(input(f"{self.name}の手を選んでください（1:グー, 2:チョキ, 3:パー): > "))
                if choice in [1, 2, 3]:
                    self.hand = choice
                    return
                else:
                    print("1, 2, 3のいずれかを入力してください。")
            except ValueError:
                print("数値を入力してください。")

class ComputerPlayer(Player):
    """コンピュータプレイヤーを表すクラス（Playerを継承)"""
    def __init__(self, name="コンピュータ"): # 名前を固定
        super().__init__(name)

    def choose_hand(self):
        """コンピュータがランダムに手を選ぶメソッド（オーバーライド)"""
        self.hand = random.randint(1, 3)
        print(f"{self.name}が手を選びました。")


class Game:
```

```
    """じゃんけんゲームを管理するクラス"""
    def __init__(self, player_name="あなた"):
        self.player = Player(player_name)
        self.computer = ComputerPlayer() # Playerクラスを継承したComputerPlayerを使用
        self.hand_map = {1: "グー", 2: "チョキ", 3: "パー"}

    def display_hands(self):
        """両者の手を表示するメソッド"""
        print(f"{self.player.name}: {self.hand_map[self.player.hand]}")
        print(f"{self.computer.name}: {self.hand_map[self.computer.hand]}")

    def judge_winner(self):
        """勝敗を判定するメソッド
        戻り値: 0:あいこ, 1:プレイヤーの勝ち, 2:コンピュータの勝ち
        """
        p_hand = self.player.hand
        c_hand = self.computer.hand

        if p_hand == c_hand:
            return 0  # あいこ
        elif (p_hand == 1 and c_hand == 2) or ¥
             (p_hand == 2 and c_hand == 3) or ¥
             (p_hand == 3 and c_hand == 1):
            return 1  # プレイヤーの勝ち
        else:
            return 2  # コンピュータの勝ち

    def play_round(self):
        """1ラウンドのゲームを実行するメソッド"""
        print("¥n--- じゃんけん勝負！ ---")
        self.player.choose_hand()
        self.computer.choose_hand()
        self.display_hands()

        result = self.judge_winner()
        if result == 0:
            print("あいこです！")
        elif result == 1:
            print(f"おめでとうございます、{self.player.name}の勝ちです！")
        else:
            print(f"残念、{self.computer.name}の勝ちです！")
        return result

    def start(self):
        """ゲームを開始し、繰り返すメソッド"""
        print("じゃんけんゲームを開始します！")
        while True:
            self.play_round()
            while True:
                again = input("¥nもう一度プレイしますか？ (y/n): > ").lower()
                if again in ['y', 'n']:
                    break
                print("yかnで入力してください。")
            if again == 'n':
                break
```

```
        print("ゲームを終了します。")

# ゲームの実行
if __name__ == "__main__":
    game = Game()
    game.start()
```

解説ポイント：

- `Player` クラス: プレイヤーの基本的な情報（名前、出した手）と操作（手を選ぶ）を定義。
- `ComputerPlayer` クラス: `Player` を継承し、`choose_hand` メソッドをコンピュータ用にオーバーライド。
- `Game` クラス: ゲーム全体の進行、プレイヤーとコンピュータのインスタンス管理、勝敗判定ロジックを担当。
- `self` が各インスタンス自身を指していることを意識しましょう。

演習2：サイコロゲーム 🎲 - 解答例

```
# filepath: dice_game.py
import random

class Die:
    """サイコロを表すクラス"""
    def __init__(self, sides=6):
        self.sides = sides  # サイコロの面の数

    def roll(self):
        """サイコロを振って、1から面の数までのランダムな整数を返すメソッド"""
        return random.randint(1, self.sides)

class Player:
    """プレイヤーを表すクラス"""
    def __init__(self, name):
        self.name = name
        self.score = 0    # プレイヤーの現在のスコア（勝利数）
        self.current_roll = 0 # 直近で振ったサイコロの目

    def roll_die(self, die_instance):
        """指定されたサイコロを振って、出た目を記録するメソッド"""
        self.current_roll = die_instance.roll()
        print(f"{self.name} がサイコロを振ります... 出た目は {self.current_roll} です。")
        return self.current_roll

class Game:
    """サイコロゲームを管理するクラス"""
    def __init__(self, player_name="あなた", computer_name="コンピュータ"):
```

```
        self.player = Player(player_name)
        self.computer = Player(computer_name) # Playerクラスをコンピュータにも使用
        self.die = Die()  # 6面のサイコロを1つ作成

    def play_round(self):
        """1ラウンドのゲームを実行するメソッド"""
        print(f"¥n--- ラウンド開始 ---")
        player_roll = self.player.roll_die(self.die)
        computer_roll = self.computer.roll_die(self.die)

        if player_roll > computer_roll:
            print(f"{self.player.name}の勝ちです！")
            self.player.score += 1
        elif computer_roll > player_roll:
            print(f"{self.computer.name}の勝ちです！")
            self.computer.score += 1
        else:
            print("引き分けです！")

    def show_scores(self):
        """現在のスコアを表示するメソッド"""
        print(f"¥n現在のスコア: {self.player.name} {self.player.score} - {self.computer.score} {self.computer.name}")

    def start(self):
        """ゲームを開始し、繰り返すメソッド"""
        print("サイコロゲーム！")
        while True:
            self.play_round()
            self.show_scores()
            while True:
                again = input("¥n続けますか？ (y/n): > ").lower()
                if again in ['y', 'n']:
                    break
                print("yかnで入力してください。")
            if again == 'n':
                break
        print("ゲームを終了します。")

# ゲームの実行
if __name__ == "__main__":
    game = Game()
    game.start()
```

解説ポイント：

- `Die` クラス: サイコロという「モノ」を表現。`roll()` という振る舞いを持つ。
- `Player` クラス: プレイヤーの情報と、`Die` オブジェクトを使ってサイコロを振るという操作を持つ。
- `Game` クラス: `Player` オブジェクト2つと `Die` オブジェクト1つを**属性として持ち**（オブジェクトのコンポジション）、ゲームを進行させる。

演習3：テキストベースRPG風バトル ⚔️ - 解答例

```python
# filepath: rpg_battle.py
import random

class Character:
    """キャラクターの基本となる親クラス"""
    def __init__(self, name, hp, attack_power):
        self.name = name
        self.max_hp = hp  # 最大HP
        self.hp = hp      # 現在のHP
        self.attack_power = attack_power
        self.is_defeated = False # 倒されたかどうかのフラグ

    def attack(self, target):
        """対象キャラクターを攻撃するメソッド"""
        if self.is_defeated:
            print(f"{self.name}は倒れているため攻撃できない！")
            return
        if target.is_defeated:
            print(f"{target.name}は既に倒れている！")
            return

        damage = random.randint(self.attack_power // 2, self.attack_power) # ダメージに少し幅を持たせる
        print(f"{self.name}のこうげき！")
        target.take_damage(damage)

    def take_damage(self, damage):
        """ダメージを受けるメソッド"""
        if self.is_defeated:
            return # 既に倒れていれば何もしない

        self.hp -= damage
        print(f"{self.name}に {damage} のダメージ！")
        if self.hp <= 0:
            self.hp = 0
            self.is_defeated = True
            print(f"{self.name}はたおれた...")
        else:
            print(f"[{self.name} HP: {self.hp}/{self.max_hp}]")

    def show_status(self):
        """簡単なステータスを表示するメソッド"""
        if self.is_defeated:
            print(f"[{self.name} HP: 0/{self.max_hp} (戦闘不能)]")
        else:
            print(f"[{self.name} HP: {self.hp}/{self.max_hp}]")


class Hero(Character):
    """勇者を表す子クラス (Characterを継承)"""
```

```
    def __init__(self, name, hp, attack_power, magic_power):
        super().__init__(name, hp, attack_power) # 親クラスの__init__を呼び出す
        self.magic_power = magic_power
        self.mp = 50 # 仮のMP

    def cast_spell(self, target):
        """魔法で攻撃するメソッド (Hero独自のメソッド)"""
        if self.is_defeated:
            print(f"{self.name}は倒れているため魔法を唱えられない！")
            return
        if target.is_defeated:
            print(f"{target.name}は既に倒れている！")
            return

        spell_cost = 10
        if self.mp >= spell_cost:
            self.mp -= spell_cost
            damage = random.randint(self.magic_power, self.magic_power + 10)
            print(f"{self.name}は魔法「ファイア」をとなえた！ (MP:{self.mp})")
            target.take_damage(damage)
        else:
            print(f"{self.name}はMPが足りない！")
            self.attack(target) # MPが足りなければ通常攻撃

    # attackメソッドをオーバーライドして、少しメッセージを変えることもできる
    # def attack(self, target):
    #     super().attack(target) # 親のattackを呼び出しつつ
    #     print(f"勇者の一撃が決まった！")


class Monster(Character):
    """モンスターを表す子クラス (Characterを継承)"""
    def __init__(self, name, hp, attack_power, special_attack_name="とくしゅこうげき"):
        super().__init__(name, hp, attack_power)
        self.special_attack_name = special_attack_name

    def special_attack(self, target):
        """特殊攻撃をするメソッド (Monster独自のメソッド)"""
        if self.is_defeated:
            print(f"{self.name}は倒れているため攻撃できない！")
            return
        if target.is_defeated:
            print(f"{target.name}は既に倒れている！")
            return

        # 30%の確率で特殊攻撃
        if random.random() < 0.3:
            damage = random.randint(self.attack_power, self.attack_power * 2)
            print(f"{self.name}の「{self.special_attack_name}」！")
            target.take_damage(damage)
        else:
            # 通常攻撃
            super().attack(target) # 親のattackメソッドを呼び出す

    # Monsterはattackメソッドをオーバーライドせず、親のものをそのまま使うか、
```

```
    # もしくはspecial_attackの中で通常攻撃の代わりに呼び出す形にする。
    # ここでは、special_attackの中で確率で通常攻撃も行うようにしてみる。


def battle_loop(player_char, enemy_char):
    """バトルを進行するメインループ"""
    turn = 0
    print("¥n--- バトル開始！ ---")
    player_char.show_status()
    enemy_char.show_status()

    while not player_char.is_defeated and not enemy_char.is_defeated:
        turn += 1
        print(f"¥n--- ターン {turn} ---")

        # プレイヤーのターン
        if not player_char.is_defeated:
            print(f"¥n{player_char.name}のターン:")
            action_valid = False
            while not action_valid:
                action = input("コマンド？ (1:こうげき, 2:まほう ※Heroのみ): > ")
                if action == "1":
                    player_char.attack(enemy_char)
                    action_valid = True
                elif action == "2" and isinstance(player_char, Hero): # Heroインスタンスかチェック
                    player_char.cast_spell(enemy_char)
                    action_valid = True
                elif action == "2" and not isinstance(player_char, Hero):
                    print("あなたはそのコマンドを使えない！")
                else:
                    print("正しいコマンドを入力してください。")
            if enemy_char.is_defeated:
                print(f"¥n{enemy_char.name}をたおした！")
                break

        # 敵のターン
        if not enemy_char.is_defeated:
            print(f"¥n{enemy_char.name}のターン:")
            if isinstance(enemy_char, Monster) and hasattr(enemy_char, 'special_attack'):
                enemy_char.special_attack(player_char) # Monsterならspecial_attackを試みる
            else:
                enemy_char.attack(player_char) # それ以外なら通常のattack

            if player_char.is_defeated:
                print(f"¥n{player_char.name}はたおれてしまった...")
                break
        player_char.show_status()
        enemy_char.show_status()


    print("¥n--- バトル終了！ ---")
    if player_char.is_defeated:
        print("ゲームオーバー...")
    elif enemy_char.is_defeated:
        print(f"{player_char.name}の勝利！")
```

```
# ゲームの実行
if __name__ == "__main__":
    hero = Hero("アベル", 100, 15, 20)  # 名前, HP, 攻撃力, 魔力
    slime = Monster("スライム", 40, 10, "溶解液") # 名前, HP, 攻撃力, 特殊攻撃名
    # goblin = Monster("ゴブリンリーダー", 70, 18, "強打")

    battle_loop(hero, slime)
    # battle_loop(hero, goblin) # 別のモンスターとも戦える
```

解説ポイント：

- `Character` クラス: 全てのキャラクターに共通する属性（`name`, `hp`, `attack_power`）とメソッド（`attack`, `take_damage`）を定義。これが **親クラス**。
- `Hero` クラスと `Monster` クラス: `Character` を **継承** し、それぞれ独自の属性（例: `Hero` の `magic_power`）やメソッド（例: `Hero` の `cast_spell`, `Monster` の `special_attack`）を追加。
- `super().__init__(...)`: 子クラスの `__init__` から親クラスの `__init__` を呼び出し、共通の初期化処理を行っている。
- `isinstance(obj, ClassName)`: オブジェクトが特定のクラスのインスタンスであるかを確認するのに便利（例: プレイヤーが魔法を使えるかどうかの判定）。
- `hasattr(obj, 'attribute_name')`: オブジェクトが特定の属性やメソッドを持っているか確認するのに便利。
- この例では、`Monster` の `special_attack` メソッド内で、確率で親の `attack` メソッドを `super().attack(target)` で呼び出す代わりに、`self.attack(target)`（もしMonsterクラスでattackをオーバーライドしていなければ親のものが呼ばれる）または直接 `Character.attack(self, target)` のように呼び出すことも考えられますが、ここでは `Monster` が独自の攻撃パターンを持つことを優先しています。
- **ポリモーフィズムの要素**: `battle_loop` 内で `player_char.attack(enemy_char)` や `enemy_char.attack(player_char)` のように同じ `attack` メソッドを呼び出していますが、もし `Hero` や `Monster` が `attack` メソッドをオーバーライドしていれば、それぞれのクラスで定義された独自の攻撃処理が実行されます（この解答例ではHeroのコメントアウト部分やMonsterのspecial_attackの分岐でその片鱗が見えます）。

これらの解答例はあくまで一例です。もっと良い方法や、異なるアプローチもたくさんあります。 ぜひ、ご自身で色々試しながら、オブジェクト指向プログラミングの理解を深めていってください。