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）

この資料は、Pythonプログラミングの初学者を対象とし、サイコロゲームを題材に基礎からオブジェクト指向プログラミング（OOP）までを段階的に理解してもらうことを目的としています。 **___や日本語になっている部分を埋めてください**

ステップ12: オブジェクト指向プログラミング（OOP）でクラスを作ってみよう

これまでのプログラムは関数を中心とした構造でしたが、より大きなプログラムでは**オブジェクト指向プログラミング（OOP）**が威力を発揮します。 このステップでは、**クラス（class）**を使って、データとそれを操作する処理をまとめて管理する方法を学びましょう。

考えてみよう（コードの穴埋め）：

```
import random

# 結果の定数を定義
PLAYER_WIN = ”player”
COMPUTER_WIN = ”computer”
DRAW = ”draw”

class Die:
    """サイコロを表すクラス"""
    def __init__(self, sides=6):
        # サイコロの面数を保存するインスタンス変数
        self.sides = ___

    def roll(self):
        # 1からself.sidesまでのランダムな整数を返す
        return random.randint(1, ___)

class GameConfig:
    """ゲームで共有する設定を管理するクラス"""
    def __init__(self, die_sides=6):
```

```
        # サイコロの面数を保存
        self.die_sides = ___
        # メッセージを辞書で保存
        self.messages = {
            ___: "あなたの勝ち！",
            ___: "コンピュータの勝ち！",
            ___: "引き分け！"
        }

class DiceGame:
    """ゲーム共通の機能を持つ基底クラス"""
    def __init__(self, config: GameConfig):
        # 設定オブジェクトを保存
        self.config = ___
        # プレイヤー用とコンピュータ用のサイコロを作成
        self.player_die = ___(config.die_sides)
        self.computer_die = ___(config.die_sides)

    def ask_continue(self, prompt="もう一度振りますか？ (y/n): ") -> bool:
        """続行するかどうかをユーザーに確認するメソッド"""
        while True:
            ans = input(prompt).lower()
            if ans in ("y", "n"):
                return ans == "y"
            print("無効な入力です。'y' または 'n' を入力してください。")

    def print_history(self, history: list[int]):
        """履歴を表示するメソッド"""
        print("これまでの出た目:")
        for i, v in enumerate(history, start=1):
            print(f" {i}回目: {v}")
        print("合計:", sum(history))

class SoloGame(___):
    """ソロプレイを行うクラス（DiceGameを継承）"""
    def play(self):
        """ソロプレイのメインメソッド"""
        history = []
        while True:
            # プレイヤーのサイコロを振る
            v = ___.roll()
            history.append(v)
            print(f"出た目は {v} です。")

            # 続行するかどうか確認
            if not ___("もう一度振りますか？ (y/n): "):
                print("ゲームを終了します。")
                break

        # 履歴を表示
        ___(history)

class VsComputerGame(___):
    """コンピュータ対戦を行うクラス（DiceGameを継承）"""
    def play_one_round(self):
```

```
    """1回分の対戦を行うメソッド"""
    # プレイヤーとコンピュータのサイコロを振る
    player_roll = ___.roll()
    computer_roll = ___.roll()

    print(f"あなた:{player_roll} vs コンピュータ:{computer_roll}")

    # 勝敗を判定
    if player_roll > computer_roll:
        result = ___
    elif player_roll < computer_roll:
        result = ___
    else:
        result = ___

    # 結果メッセージを設定から取得して表示
    print(___.messages[result])
    return player_roll, computer_roll, result

# 対戦履歴の詳細統計を表示する関数
def print_match_statistics(self,match_history):
    if not match_history:
        print("対戦履歴がありません。")
        return

    print("¥n---詳細な対戦履歴---")
    for i, match in enumerate(match_history, start=1):
        player_roll, computer_roll, result = match
        print(f"{i}戦目: あなた:{player_roll}, コンピュータ:{computer_roll}, 結果:{result}")

    print("¥n---統計情報---")

    player_rolls = [match[0] for match in match_history]
    computer_rolls = [match[1] for match in match_history]

    print(f"あなたの出目統計:")
    print(f"  最高出目: {max(player_rolls)}")
    print(f"  最低出目: {min(player_rolls)}")
    print(f"  平均出目: {sum(player_rolls) / len(player_rolls):.2f}")

    print(f"コンピュータの出目統計:")
    print(f"  最高出目: {max(computer_rolls)}")
    print(f"  最低出目: {min(computer_rolls)}")
    print(f"  平均出目: {sum(computer_rolls) / len(computer_rolls):.2f}")

    # 勝敗数を辞書で管理
    win_counts = {PLAYER_WIN: 0, COMPUTER_WIN: 0, DRAW: 0}

    for match in match_history:
        _, _, winner = match
        win_counts[winner] += 1

    print(f"¥n最終結果:")
    print(f"  あなたの勝利: {win_counts[PLAYER_WIN]}回")
    print(f"  コンピュータの勝利: {win_counts[COMPUTER_WIN]}回")
```

```
        print(f"  引き分け: {win_counts[DRAW]}回")


    def play(self):
        """対戦のメインメソッド"""
        match_history = []  # 各対戦の詳細情報を保存するリスト
        while True:
            # 1回分の対戦を実行
            player_roll, computer_roll, result = self.play_one_round()
            match_history.append((player_roll, computer_roll, result))

            # 続行するかどうか確認
            if not ___("もう一度対戦しますか？ (y/n): "):
                print("対戦を終了します。")
                break

        match_historyを引数にprint_match_staticsを呼び出す

# メイン処理
if __name__ == "__main__":
    # ゲーム設定を作成
    game_config = ___(die_sides=6)

    print("モード選択: 1=ソロプレイ, 2=対戦")
    mode = input("選択 (1/2): ")

    if mode == "1":
        # ソロゲームのインスタンスを作成
        game = ___(game_config)
    elif mode == "2":
        # 対戦ゲームのインスタンスを作成
        game = ___(game_config)
    else:
        print("無効な選択です。")
        exit()

    # ゲームを開始
    game.___()
```

ヒント:

- `self` は、そのクラスのインスタンス（実体）自身を指します。
- `__init__` メソッドは、クラスのインスタンスを作成するときに自動的に呼ばれる特別なメソッドです。
- クラスを継承すると、親クラスのメソッドを子クラスで使うことができます。
- `Die` クラスのインスタンスを作成するには `Die(面数)` と書きます。
- メソッドを呼び出すには `オブジェクト.メソッド名()` の形式を使います。

解答例:

```
import random

# 結果の定数を定義
PLAYER_WIN = ”player”
COMPUTER_WIN = ”computer”
DRAW = ”draw”

class Die:
    """サイコロを表すクラス"""
    def __init__(self, sides=6):
        # サイコロの面数を保存するインスタンス変数
        self.sides = sides

    def roll(self):
        # 1からself.sidesまでのランダムな整数を返す
        return random.randint(1, self.sides)

class GameConfig:
    """ゲームで共有する設定を管理するクラス"""
    def __init__(self, die_sides=6):
        # サイコロの面数を保存
        self.die_sides = die_sides
        # メッセージを辞書で保存
        self.messages = {
            PLAYER_WIN: ”あなたの勝ち！”,
            COMPUTER_WIN: ”コンピュータの勝ち！”,
            DRAW: ”引き分け！”
        }

class DiceGame:
    """ゲーム共通の機能を持つ基底クラス"""
    def __init__(self, config: GameConfig):
        # 設定オブジェクトを保存
        self.config = config
        # プレイヤー用とコンピュータ用のサイコロを作成
        self.player_die = Die(config.die_sides)
        self.computer_die = Die(config.die_sides)

    def ask_continue(self, prompt=”もう一度振りますか？ (y/n): ”) -> bool:
        """続行するかどうかをユーザーに確認するメソッド"""
        while True:
            ans = input(prompt).lower()
            if ans in (”y”, ”n”):
                return ans == ”y”
            print(”無効な入力です。’y’ または ’n’ を入力してください。”)

    def print_history(self, history: list[int]):
        """履歴を表示するメソッド"""
        print(”これまでの出た目:”)
        for i, v in enumerate(history, start=1):
            print(f” {i}回目: {v}”)
        print(”合計:”, sum(history))

class SoloGame(DiceGame):
```

```
    """ソロプレイを行うクラス（DiceGameを継承）"""
    def play(self):
        """ソロプレイのメインメソッド"""
        history = []
        while True:
            # プレイヤーのサイコロを振る
            v = self.player_die.roll()
            history.append(v)
            print(f"出た目は {v} です。")

            # 続行するかどうか確認
            if not self.ask_continue("もう一度振りますか？ (y/n): "):
                print("ゲームを終了します。")
                break

        # 履歴を表示
        self.print_history(history)

class VsComputerGame(DiceGame):
    """コンピュータ対戦を行うクラス（DiceGameを継承）"""
    def play_one_round(self):
        """1回分の対戦を行うメソッド"""
        # プレイヤーとコンピュータのサイコロを振る
        player_roll = self.player_die.roll()
        computer_roll = self.computer_die.roll()

        print(f"あなた:{player_roll} vs コンピュータ:{computer_roll}")

        # 勝敗を判定
        if player_roll > computer_roll:
            result = PLAYER_WIN
        elif player_roll < computer_roll:
            result = COMPUTER_WIN
        else:
            result = DRAW

        # 結果メッセージを設定から取得して表示
        print(self.config.messages[result])
        return player_roll, computer_roll, result

    # 対戦履歴の詳細統計を表示する関数
    def print_match_statistics(self,match_history):
        if not match_history:
            print("対戦履歴がありません。")
            return

        print("¥n---詳細な対戦履歴---")
        for i, match in enumerate(match_history, start=1):
            player_roll, computer_roll, result = match
            print(f"{i}戦目: あなた:{player_roll}, コンピュータ:{computer_roll}, 結果:{result}")

        print("¥n---統計情報---")

        player_rolls = [match[0] for match in match_history]
        computer_rolls = [match[1] for match in match_history]
```

```
        print(f"あなたの出目統計:")
        print(f"  最高出目: {max(player_rolls)}")
        print(f"  最低出目: {min(player_rolls)}")
        print(f"  平均出目: {sum(player_rolls) / len(player_rolls):.2f}")

        print(f"コンピュータの出目統計:")
        print(f"  最高出目: {max(computer_rolls)}")
        print(f"  最低出目: {min(computer_rolls)}")
        print(f"  平均出目: {sum(computer_rolls) / len(computer_rolls):.2f}")

        # 勝敗数を辞書で管理
        win_counts = {PLAYER_WIN: 0, COMPUTER_WIN: 0, DRAW: 0}

        for match in match_history:
            _, _, winner = match
            win_counts[winner] += 1

        print(f"¥n最終結果:")
        print(f"  あなたの勝利: {win_counts[PLAYER_WIN]}回")
        print(f"  コンピュータの勝利: {win_counts[COMPUTER_WIN]}回")
        print(f"  引き分け: {win_counts[DRAW]}回")の詳細統計を表示する関数



    def play(self):
        """対戦のメインメソッド"""
        match_history = []  # 各対戦の詳細情報を保存するリスト

        while True:
            # 1回分の対戦を実行
            player_roll, computer_roll, result = self.play_one_round()

            # 履歴に追加
            match_history.append((player_roll, computer_roll, result))

            # 続行するかどうか確認
            if not self.ask_continue("もう一度対戦しますか？ (y/n): "):
                print("対戦を終了します。")
                break

        # 結果を表示
        self.print_match_statistics(match_history)


# メイン処理
if __name__ == "__main__":
    # ゲーム設定を作成
    game_config = GameConfig(die_sides=6)

    print("モード選択: 1=ソロプレイ, 2=対戦")
    mode = input("選択 (1/2): ")

    if mode == "1":
        # ソロゲームのインスタンスを作成
```

```
    game = SoloGame(game_config)
elif mode == "2":
    # 対戦ゲームのインスタンスを作成
    game = VsComputerGame(game_config)
else:
    print("無効な選択です。")
    exit()

# ゲームを開始
game.play()
```

ポイント:

- **クラス（class）**: 関連するデータ（属性）と処理（メソッド）をまとめた設計図のようなもの。
- **インスタンス**: クラスから作られた実際のオブジェクト。`Die(6)` でDieクラスのインスタンスを作成。
- `__init__` **メソッド**: クラスのインスタンスを作成するときに自動で呼ばれる特別なメソッド（コンストラクタ）。
- `self`: そのインスタンス自身を指す特別な変数。メソッド内でインスタンス変数にアクセスするときに使用。
- **継承**: `class SoloGame(DiceGame):` のように書くことで、親クラス（DiceGame）の機能を子クラス（SoloGame）が受け継ぐ。
- **カプセル化**: 関連するデータと処理をクラス内にまとめることで、コードの整理と再利用性が向上。

> コラム **オブジェクト指向の3大要素**
>
> 1. **カプセル化**: 関連するデータと処理をまとめる（今回学習）
> 2. **継承**: 既存のクラスの機能を受け継いで新しいクラスを作る（今回学習）
> 3. **ポリモーフィズム**: 同じメソッド名でも、クラスによって異なる動作をする
>
> 今回のコードでは、`SoloGame` と `VsComputerGame` が両方とも `play()` メソッドを持っていますが、それぞれ異なる動作をします。これがポリモーフィズムの例です！

> コラム **なぜクラスを使うのか？**
>
> 関数だけでもプログラムは作れますが、クラスを使うことで：
>
> - **コードの整理**: 関連する機能がまとめられる
> - **再利用性**: 同じクラスから複数のインスタンスを作成可能
> - **拡張性**: 継承により既存の機能を活用して新機能を追加しやすい
> - **保守性**: 変更する場所が明確になり、バグを減らせる

ステップ13: プライベート変数・メソッドでクラスの安全性を高めよう

ステップ12でクラスを作成しましたが、実は現在のコードには大きな問題があります。 このステップでは、その問題を発見し、**プライベート変数・メソッド**と**@propertyデコレータ**を使って解決する方法を学びましょう。

まず問題を発見してみよう

現在のコードには、外部から勝手にクラスの内部データを変更できてしまうという問題があります。 例えば、以下のような「いじわる」なコードを書くことができてしまいます：

```
# メイン処理で、ゲーム開始前に...
game = VsComputerGame(game_config)

# いじわる：プレイヤーのサイコロを100面にして、コンピュータのサイコロを1面にする
game.player_die = Die(sides=100)  # プレイヤーは常に大きな数
game.computer_die = Die(sides=1)  # コンピュータは常に1

game.play()  # これではゲームが成り立たない！
```

このように、クラスの外部から重要なデータを自由に変更できてしまうと、ゲームの公平性が保てません。 そこで、**外部からアクセスすべきでない変数やメソッドを隠す**必要があります。

考えてみよう（コードの穴埋め）：

```
import random

# 結果の定数を定義
PLAYER_WIN = ”player”
COMPUTER_WIN = ”computer”
DRAW = ”draw”

class Die:
    """サイコロを表すクラス"""
    def __init__(self, sides=6):
        # プライベート変数：外部から直接変更されたくない
        self._sides = ___  # sidesの前に_を付けてプライベート化

    def roll(self):
```

```
        # プライベート変数を使用
        return random.randint(1, ___)

    @property
    def sides(self):
        """面数を取得（読み取り専用）"""
        # プライベート変数の値を返す
        return ___

class GameConfig:
    """ゲームで共有する設定を管理するクラス"""
    def __init__(self, die_sides=6):
        # プライベート変数にする
        self._die_sides = ___
        self._messages = {
            PLAYER_WIN: "あなたの勝ち！",
            COMPUTER_WIN: "コンピュータの勝ち！",
            DRAW: "引き分け！"
        }

    @property
    def die_sides(self):
        """サイコロの面数を取得（読み取り専用）"""
        return ___

    @property
    def messages(self):
        """メッセージ辞書を取得（読み取り専用）"""
        return ___

class DiceGame:
    """ゲーム共通の機能を持つ基底クラス"""
    def __init__(self, config: GameConfig):
        # すべてプライベート変数にする
        self._config = ___
        self._player_die = ___(config.die_sides)
        self._computer_die = ___(config.die_sides)

    def _ask_continue(self, prompt="もう一度振りますか？ (y/n): ") -> bool:
        """プライベートメソッド：外部から直接呼ばれたくない内部処理"""
        while True:
            ans = input(prompt).lower()
            if ans in ("y", "n"):
                return ans == "y"
            print("無効な入力です。'y' または 'n' を入力してください。")

    # この関数はどこから呼ばれるべきでしょうか？ private or public?
    def ___(self, history: list[int]):
        """プライベートメソッド：内部でのみ使用する履歴表示処理"""
        print("これまでの出た目:")
        for i, v in enumerate(history, start=1):
            print(f" {i}回目: {v}")
        print("合計:", sum(history))

class SoloGame(DiceGame):
```

```
    """ソロプレイを行うクラス（DiceGameを継承）"""
    def play(self):
        """パブリックメソッド：外部から呼び出されることを想定"""
        history = []
        while True:
            # プライベート変数を使用
            v = ___.roll()
            history.append(v)
            print(f"出た目は {v} です。")

            # プライベートメソッドを呼び出し
            if not ___("もう一度振りますか？ (y/n): "):
                print("ゲームを終了します。")
                break

        # プライベートメソッドを呼び出し
        ___(history)

class VsComputerGame(DiceGame):
    """コンピュータ対戦を行うクラス（DiceGameを継承）"""
    def _play_one_round(self):
        """プライベートメソッド：1回分の対戦処理（内部でのみ使用）"""
        # プライベート変数を使用
        player_roll = ___.roll()
        computer_roll = ___.roll()

        print(f"あなた:{player_roll} vs コンピュータ:{computer_roll}")

        # 勝敗を判定
        if player_roll > computer_roll:
            result = PLAYER_WIN
        elif player_roll < computer_roll:
            result = COMPUTER_WIN
        else:
            result = DRAW

        # プライベート変数を使用してメッセージを取得
        print(___.messages[result])
        return player_roll, computer_roll, result

    def _print_match_statistics(self, match_history):
        """プライベートメソッド：対戦統計を表示（内部でのみ使用）"""
        if not match_history:
            print("対戦履歴がありません。")
            return

        print("¥n---詳細な対戦履歴---")
        for i, match in enumerate(match_history, start=1):
            player_roll, computer_roll, result = match
            print(f"{i}戦目: あなた:{player_roll}, コンピュータ:{computer_roll}, 結果:{result}")

        print("¥n---統計情報---")

        player_rolls = [match[0] for match in match_history]
        computer_rolls = [match[1] for match in match_history]
```

```
        print(f"あなたの出目統計:")
        print(f"  最高出目: {max(player_rolls)}")
        print(f"  最低出目: {min(player_rolls)}")
        print(f"  平均出目: {sum(player_rolls) / len(player_rolls):.2f}")

        print(f"コンピュータの出目統計:")
        print(f"  最高出目: {max(computer_rolls)}")
        print(f"  最低出目: {min(computer_rolls)}")
        print(f"  平均出目: {sum(computer_rolls) / len(computer_rolls):.2f}")

        win_counts = {PLAYER_WIN: 0, COMPUTER_WIN: 0, DRAW: 0}

        for match in match_history:
            _, _, winner = match
            win_counts[winner] += 1

        print(f"¥n最終結果:")
        print(f"  あなたの勝利: {win_counts[PLAYER_WIN]}回")
        print(f"  コンピュータの勝利: {win_counts[COMPUTER_WIN]}回")
        print(f"  引き分け: {win_counts[DRAW]}回")

    def play(self):
        """パブリックメソッド：外部から呼び出されることを想定"""
        match_history = []

        while True:
            # プライベートメソッドを呼び出し
            player_roll, computer_roll, result = ___()
            match_history.append([player_roll, computer_roll, result])

            # プライベートメソッドを呼び出し
            if not ___("もう一度対戦しますか？ (y/n): "):
                print("対戦を終了します。")
                break

        # プライベートメソッドを呼び出し
        ___(match_history)

# メイン処理
if __name__ == "__main__":
    # ゲーム設定を作成
    game_config = GameConfig(die_sides=6)

    print("モード選択: 1=ソロプレイ, 2=対戦")
    mode = input("選択 (1/2): ")

    if mode == "1":
        game = SoloGame(game_config)
    elif mode == "2":
        game = VsComputerGame(game_config)
    else:
        print("無効な選択です。")
        exit()
```

```
    # もう「いじわる」はできない！
    # game.player_die = Die(sides=100)  # エラー：_player_dieはプライベート
    # game._player_die = Die(sides=100)  # 技術的には可能だが「触るな」の意味

    # 読み取り専用でアクセスはできる
    print(f"サイコロの面数: {game._config.die_sides}")

    game.play()
```

ヒント:

- プライベート変数・メソッドには名前の前に `_` を付けます。
- `@property` デコレータを使うと、メソッドを変数のようにアクセスできる読み取り専用プロパティになります。
- プライベートメソッドは `self._メソッド名()` で呼び出します。（クラス内部でのみ）
- プライベート変数は `self._変数名` でアクセスします。（クラス内部でのみ）

解答例:

```
import random

# 結果の定数を定義
PLAYER_WIN = "player"
COMPUTER_WIN = "computer"
DRAW = "draw"

class Die:
    """サイコロを表すクラス"""
    def __init__(self, sides=6):
        # プライベート変数：外部から直接変更されたくない
        self._sides = sides

    def roll(self):
        # プライベート変数を使用
        return random.randint(1, self._sides)

    @property
    def sides(self):
        """面数を取得（読み取り専用）"""
        # プライベート変数の値を返す
        return self._sides

class GameConfig:
    """ゲームで共有する設定を管理するクラス"""
    def __init__(self, die_sides=6):
        # プライベート変数にする
        self._die_sides = die_sides
        self._messages = {
            PLAYER_WIN: "あなたの勝ち！",
            COMPUTER_WIN: "コンピュータの勝ち！",
```

```
            DRAW: "引き分け！"
        }

    @property
    def die_sides(self):
        """サイコロの面数を取得（読み取り専用）"""
        return self._die_sides

    @property
    def messages(self):
        """メッセージ辞書を取得（読み取り専用）"""
        return self._messages

class DiceGame:
    """ゲーム共通の機能を持つ基底クラス"""
    def __init__(self, config: GameConfig):
        # すべてプライベート変数にする
        self._config = config
        self._player_die = Die(config.die_sides)
        self._computer_die = Die(config.die_sides)

    def _ask_continue(self, prompt="もう一度振りますか？ (y/n): ") -> bool:
        """プライベートメソッド：外部から直接呼ばれたくない内部処理"""
        while True:
            ans = input(prompt).lower()
            if ans in ("y", "n"):
                return ans == "y"
            print("無効な入力です。'y' または 'n' を入力してください。")

    def _print_history(self, history: list[int]):
        """プライベートメソッド：内部でのみ使用する履歴表示処理"""
        print("これまでの出た目:")
        for i, v in enumerate(history, start=1):
            print(f" {i}回目: {v}")
        print("合計:", sum(history))

class SoloGame(DiceGame):
    """ソロプレイを行うクラス（DiceGameを継承）"""
    def play(self):
        """パブリックメソッド：外部から呼び出されることを想定"""
        history = []
        while True:
            # プライベート変数を使用
            v = self._player_die.roll()
            history.append(v)
            print(f"出た目は {v} です。")

            # プライベートメソッドを呼び出し
            if not self._ask_continue("もう一度振りますか？ (y/n): "):
                print("ゲームを終了します。")
                break

        # プライベートメソッドを呼び出し
        self._print_history(history)
```

```
class VsComputerGame(DiceGame):
    """コンピュータ対戦を行うクラス（DiceGameを継承）"""
    def _play_one_round(self):
        """プライベートメソッド：1回分の対戦処理（内部でのみ使用）"""
        # プライベート変数を使用
        player_roll = self._player_die.roll()
        computer_roll = self._computer_die.roll()

        print(f"あなた:{player_roll} vs コンピュータ:{computer_roll}")

        # 勝敗を判定
        if player_roll > computer_roll:
            result = PLAYER_WIN
        elif player_roll < computer_roll:
            result = COMPUTER_WIN
        else:
            result = DRAW

        # プライベート変数を使用してメッセージを取得
        print(self._config.messages[result])
        return player_roll, computer_roll, result

    def _print_match_statistics(self, match_history):
        """プライベートメソッド：対戦統計を表示（内部でのみ使用）"""
        if not match_history:
            print("対戦履歴がありません。")
            return

        print("¥n---詳細な対戦履歴---")
        for i, match in enumerate(match_history, start=1):
            player_roll, computer_roll, result = match
            print(f"{i}戦目: あなた:{player_roll}, コンピュータ:{computer_roll}, 結果:{result}")

        print("¥n---統計情報---")

        player_rolls = [match[0] for match in match_history]
        computer_rolls = [match[1] for match in match_history]

        print(f"あなたの出目統計:")
        print(f"  最高出目: {max(player_rolls)}")
        print(f"  最低出目: {min(player_rolls)}")
        print(f"  平均出目: {sum(player_rolls) / len(player_rolls):.2f}")

        print(f"コンピュータの出目統計:")
        print(f"  最高出目: {max(computer_rolls)}")
        print(f"  最低出目: {min(computer_rolls)}")
        print(f"  平均出目: {sum(computer_rolls) / len(computer_rolls):.2f}")

        win_counts = {PLAYER_WIN: 0, COMPUTER_WIN: 0, DRAW: 0}

        for match in match_history:
            _, _, winner = match
            win_counts[winner] += 1

        print(f"¥n最終結果:")
```

```
        print(f"  あなたの勝利: {win_counts[PLAYER_WIN]}回")
        print(f"  コンピュータの勝利: {win_counts[COMPUTER_WIN]}回")
        print(f"  引き分け: {win_counts[DRAW]}回")

    def play(self):
        """パブリックメソッド：外部から呼び出されることを想定"""
        match_history = []

        while True:
            # プライベートメソッドを呼び出し
            player_roll, computer_roll, result = self._play_one_round()
            match_history.append([player_roll, computer_roll, result])

            # プライベートメソッドを呼び出し
            if not self._ask_continue("もう一度対戦しますか？ (y/n): "):
                print("対戦を終了します。")
                break

        # プライベートメソッドを呼び出し
        self._print_match_statistics(match_history)

# メイン処理
if __name__ == "__main__":
    # ゲーム設定を作成
    game_config = GameConfig(die_sides=6)

    print("モード選択: 1=ソロプレイ, 2=対戦")
    mode = input("選択 (1/2): ")

    if mode == "1":
        game = SoloGame(game_config)
    elif mode == "2":
        game = VsComputerGame(game_config)
    else:
        print("無効な選択です。")
        exit()

    # もう「いじわる」はできない！
    # game.player_die = Die(sides=100)  # エラー：_player_dieはプライベート
    # game._player_die = Die(sides=100)  # 技術的には可能だが「触るな」の意味

    # 読み取り専用でアクセスはできる
    print(f"サイコロの面数: {game._config.die_sides}")

    game.play()
```

ポイント:

- **プライベート変数**: 名前の前に `_` を付けることで、「外部から直接アクセスすべきでない」ことを示します。
- **プライベートメソッド**: 内部でのみ使用される処理には `_` を付けて、外部インターフェースと区別します。
- **@propertyデコレータ**: メソッドを変数のようにアクセスできる読み取り専用プロパティを作成できます。

- **カプセル化の実現**：クラスの内部実装を隠蔽し、外部には必要最小限のインターフェースのみを公開します。
- **アクセス制御**：重要なデータを不正な変更から保護し、プログラムの安全性を高めます。
- **インターフェースの明確化**：外部から呼び出すべきメソッド（`play()`）と内部処理用メソッド（`_play_one_round()`）が明確に区別されます。

> コラム **Pythonにおける2つのプライベート化手法**
>
> 1. **シングルアンダースコア（`_`）**：慣習的なプライベート化
>   - 技術的にはアクセス可能だが、「触らない」という約束
>   - 開発者同士の暗黙の了解
> 2. **ダブルアンダースコア（`__`）**：名前マングリング
>   - Pythonが自動的に名前を変更し、外部からのアクセスを困難にする
>   - より強力だが、デバッグ時に分かりにくくなることも
>
> 一般的には、シングルアンダースコアがよく使われます。

> コラム **@propertyの威力**
>
> `@property` を使うことで：
>
> ```
> # 通常のメソッド呼び出し
> print(die.get_sides())  # メソッドとして呼び出す
>
> # @propertyを使った場合
> print(die.sides)  # 変数のようにアクセス
> ```
>
> より自然で読みやすいコードが書けます。また、将来的に内部実装を変更しても、外部のコードに影響を与えません。

> コラム **なぜプライベート化が重要なのか？**
>
> 1. **データの整合性**：重要なデータが勝手に変更されることを防ぐ
> 2. **バグの防止**：想定外の使われ方によるバグを減らす
> 3. **保守性の向上**：内部実装を変更しても外部に影響しない
> 4. **使いやすさ**：外部の人は公開されたメソッドだけに注意すればよい

大きなプログラムになるほど、この重要性は増していきます。

ステップ14: オブジェクト指向の恩恵：難易度設定機能を簡単に追加しよう

ステップ13でプライベート化によってクラスの安全性を高めました。 このステップでは、**オブジェクト指向プログラミングの真価**を体感しましょう。既存のコードをほとんど変更することなく、新しい機能（難易度設定）を追加してみます。

追加したい機能

ゲームに**3つの難易度**を追加したいと思います：

- **初級（Easy）**：プレイヤーは1〜9面サイコロ、コンピュータは1〜6面サイコロ（プレイヤー有利）
- **中級（Normal）**：両方とも1〜6面サイコロ（通常のゲーム）
- **上級（Hard）**：プレイヤーは1〜4面サイコロ、コンピュータは1〜6面サイコロ（プレイヤー不利）

関数ベースの実装では多くの関数を修正する必要がありますが、オブジェクト指向では最小限の変更で実現できます！

考えてみよう（コードの穴埋め）：

```
import random

# 結果の定数を定義
PLAYER_WIN = ”player”
COMPUTER_WIN = ”computer”
DRAW = ”draw”

# 難易度レベルの定数を定義
EASY = ”easy”
NORMAL = ”normal”
HARD = ”hard”

class Die:
    """サイコロを表すクラス"""
    def __init__(self, sides=6):
        self._sides = sides

    def roll(self):
        return random.randint(1, self._sides)
```

```
    @property
    def sides(self):
        """面数を取得（読み取り専用）"""
        return self._sides

class GameConfig:
    """ゲームで共有する設定を管理するクラス"""
    def __init__(self, difficulty=NORMAL):
        # 難易度に応じてサイコロの面数を設定
        self._difficulty = ___
        self._difficulty_settings = {
            ___: {"player_sides": 9, "computer_sides": 6, "name": "初級（プレイヤー有利）"},
            ___: {"player_sides": 6, "computer_sides": 6, "name": "中級（通常）"},
            ___: {"player_sides": 4, "computer_sides": 6, "name": "上級（プレイヤー不利）"}
        }

        # 現在の難易度設定を取得
        current_setting = ___[difficulty]
        self._player_die_sides = current_setting["player_sides"]
        self._computer_die_sides = current_setting["computer_sides"]
        self._difficulty_name = current_setting["name"]

        self._messages = {
            PLAYER_WIN: "あなたの勝ち！",
            COMPUTER_WIN: "コンピュータの勝ち！",
            DRAW: "引き分け！"
        }

    @property
    def difficulty(self):
        """現在の難易度を取得"""
        return ___

    @property
    def difficulty_name(self):
        """難易度の日本語名を取得"""
        return ___

    @property
    def player_die_sides(self):
        """プレイヤーのサイコロ面数を取得"""
        return ___

    @property
    def computer_die_sides(self):
        """コンピュータのサイコロ面数を取得"""
        return ___

    @property
    def messages(self):
        """メッセージ辞書を取得（読み取り専用）"""
        return self._messages

class DiceGame:
```

```
    """ゲーム共通の機能を持つ基底クラス"""
    def __init__(self, config: GameConfig):
        self._config = config
        # 難易度に応じて異なる面数のサイコロを作成
        self._player_die = ___(config.player_die_sides)
        self._computer_die = ___(config.computer_die_sides)

    def _ask_continue(self, prompt="もう一度振りますか？ (y/n): ") -> bool:
        """プライベートメソッド：外部から直接呼ばれたくない内部処理"""
        while True:
            ans = input(prompt).lower()
            if ans in ("y", "n"):
                return ans == "y"
            print("無効な入力です。'y' または 'n' を入力してください。")

    def _print_history(self, history: list[int]):
        """プライベートメソッド：内部でのみ使用する履歴表示処理"""
        print("これまでの出た目:")
        for i, v in enumerate(history, start=1):
            print(f" {i}回目: {v}")
        print("合計:", sum(history))


class SoloGame(DiceGame):
    """ソロプレイを行うクラス（DiceGameを継承）"""
    def play(self):
        """パブリックメソッド：外部から呼び出されることを想定"""
        # ゲーム情報を表示
        ___()

        history = []
        while True:
            v = self._player_die.roll()
            history.append(v)
            print(f"出た目は {v} です。")

            if not self._ask_continue("もう一度振りますか？ (y/n): "):
                print("ゲームを終了します。")
                break

        self._print_history(history)

class VsComputerGame(DiceGame):
    """コンピュータ対戦を行うクラス（DiceGameを継承）"""
    def _play_one_round(self):
        """プライベートメソッド：1回分の対戦処理（内部でのみ使用）"""
        player_roll = self._player_die.roll()
        computer_roll = self._computer_die.roll()

        print(f"あなた:{player_roll} vs コンピュータ:{computer_roll}")


        # 勝敗を判定
        if player_roll > computer_roll:
            result = PLAYER_WIN
```

```
        elif player_roll < computer_roll:
            result = COMPUTER_WIN
        else:
            result = DRAW

        print(self._config.messages[result])
        return player_roll, computer_roll, result

    def _print_match_statistics(self, match_history):
        """プライベートメソッド：対戦統計を表示（内部でのみ使用）"""
        if not match_history:
            print("対戦履歴がありません。")
            return

        print("¥n---詳細な対戦履歴---")
        for i, match in enumerate(match_history, start=1):
            player_roll, computer_roll, result = match
            print(f"{i}戦目: あなた:{player_roll}, コンピュータ:{computer_roll}, 結果:{result}")

        print("¥n---統計情報---")

        player_rolls = [match[0] for match in match_history]
        computer_rolls = [match[1] for match in match_history]

        print(f"あなたの出目統計:")
        print(f"  最高出目: {max(player_rolls)}")
        print(f"  最低出目: {min(player_rolls)}")
        print(f"  平均出目: {sum(player_rolls) / len(player_rolls):.2f}")

        print(f"コンピュータの出目統計:")
        print(f"  最高出目: {max(computer_rolls)}")
        print(f"  最低出目: {min(computer_rolls)}")
        print(f"  平均出目: {sum(computer_rolls) / len(computer_rolls):.2f}")

        win_counts = {PLAYER_WIN: 0, COMPUTER_WIN: 0, DRAW: 0}

        for match in match_history:
            _, _, winner = match
            win_counts[winner] += 1

        print(f"¥n最終結果:")
        print(f"  あなたの勝利: {win_counts[PLAYER_WIN]}回")
        print(f"  コンピュータの勝利: {win_counts[COMPUTER_WIN]}回")
        print(f"  引き分け: {win_counts[DRAW]}回")


    def play(self):
        """パブリックメソッド：外部から呼び出されることを想定"""
        # ゲーム情報を表示
        ___()

        match_history = []

        while True:
            player_roll, computer_roll, result = self._play_one_round()
```

```
            match_history.append([player_roll, computer_roll, result])

            if not self._ask_continue("もう一度対戦しますか？ (y/n): "):
                print("対戦を終了します。")
                break

        self._print_match_statistics(match_history)

def select_difficulty():
    """難易度を選択する関数"""
    print("難易度を選択してください:")
    print(" 1: 初級 (プレイヤー有利 - あなた1〜9面 vs コンピュータ1〜6面)")
    print(" 2: 中級 (通常 - 両方とも1〜6面)")
    print(" 3: 上級 (プレイヤー不利 - あなた1〜4面 vs コンピュータ1〜6面)")

    while True:
        choice = input("選択 (1/2/3): ")
        if choice == "1":
            return ___
        elif choice == "2":
            return ___
        elif choice == "3":
            return ___
        else:
            print("無効な選択です。1、2、3のいずれかを入力してください。")

# メイン処理
if __name__ == "__main__":
    # まず難易度を選択
    difficulty = ___()

    # 選択した難易度でゲーム設定を作成
    game_config = ___(difficulty=difficulty)

    print(f"¥n選択された難易度: {game_config.difficulty_name}")
    print("モード選択: 1=ソロプレイ, 2=対戦")
    mode = input("選択 (1/2): ")

    if mode == "1":
        game = SoloGame(game_config)
    elif mode == "2":
        game = VsComputerGame(game_config)
    else:
        print("無効な選択です。")
        exit()

    game.play()
```

ヒント:

- 難易度の定数（`EASY`, `NORMAL`, `HARD`）を辞書のキーとして使用します。
- `GameConfig` クラスのコンストラクタで、難易度に応じて異なるサイコロ面数を設定します。

- 既存のメソッド（`_print_game_info()` など）は `self._メソッド名()` で呼び出します。
- `@property` デコレータで作られたプロパティは、`self._config.プロパティ名` でアクセスできます。

解答例:

```
import random

# 結果の定数を定義
PLAYER_WIN = "player"
COMPUTER_WIN = "computer"
DRAW = "draw"

# 難易度レベルの定数を定義
EASY = "easy"
NORMAL = "normal"
HARD = "hard"

class Die:
    """サイコロを表すクラス"""
    def __init__(self, sides=6):
        self._sides = sides

    def roll(self):
        return random.randint(1, self._sides)

    @property
    def sides(self):
        """面数を取得（読み取り専用）"""
        return self._sides

class GameConfig:
    """ゲームで共有する設定を管理するクラス"""
    def __init__(self, difficulty=NORMAL):
        # 難易度に応じてサイコロの面数を設定
        self._difficulty = difficulty
        self._difficulty_settings = {
            EASY: {"player_sides": 9, "computer_sides": 6, "name": "初級（プレイヤー有利）"},
            NORMAL: {"player_sides": 6, "computer_sides": 6, "name": "中級（通常）"},
            HARD: {"player_sides": 4, "computer_sides": 6, "name": "上級（プレイヤー不利）"}
        }

        # 現在の難易度設定を取得
        current_setting = self._difficulty_settings[difficulty]
        self._player_die_sides = current_setting["player_sides"]
        self._computer_die_sides = current_setting["computer_sides"]
        self._difficulty_name = current_setting["name"]

        self._messages = {
            PLAYER_WIN: "あなたの勝ち！",
            COMPUTER_WIN: "コンピュータの勝ち！",
            DRAW: "引き分け！"
        }
```

```
    @property
    def difficulty(self):
        """現在の難易度を取得"""
        return self._difficulty

    @property
    def difficulty_name(self):
        """難易度の日本語名を取得"""
        return self._difficulty_name

    @property
    def player_die_sides(self):
        """プレイヤーのサイコロ面数を取得"""
        return self._player_die_sides

    @property
    def computer_die_sides(self):
        """コンピュータのサイコロ面数を取得"""
        return self._computer_die_sides

    @property
    def messages(self):
        """メッセージ辞書を取得（読み取り専用）"""
        return self._messages

class DiceGame:
    """ゲーム共通の機能を持つ基底クラス"""
    def __init__(self, config: GameConfig):
        self._config = config
        # 難易度に応じて異なる面数のサイコロを作成
        self._player_die = Die(config.player_die_sides)
        self._computer_die = Die(config.computer_die_sides)

    def _ask_continue(self, prompt="もう一度振りますか？ (y/n): ") -> bool:
        """プライベートメソッド：外部から直接呼ばれたくない内部処理"""
        while True:
            ans = input(prompt).lower()
            if ans in ("y", "n"):
                return ans == "y"
            print("無効な入力です。'y' または 'n' を入力してください。")

    def _print_history(self, history: list[int]):
        """プライベートメソッド：内部でのみ使用する履歴表示処理"""
        print("これまでの出た目:")
        for i, v in enumerate(history, start=1):
            print(f" {i}回目: {v}")
        print("合計:", sum(history))


class SoloGame(DiceGame):
    """ソロプレイを行うクラス（DiceGameを継承）"""
    def play(self):
        """パブリックメソッド：外部から呼び出されることを想定"""
```

```
        history = []
        while True:
            v = self._player_die.roll()
            history.append(v)
            print(f"出た目は {v} です。")

            if not self._ask_continue("もう一度振りますか？ (y/n): "):
                print("ゲームを終了します。")
                break

        self._print_history(history)

class VsComputerGame(DiceGame):
    """コンピュータ対戦を行うクラス（DiceGameを継承）"""
    def _play_one_round(self):
        """プライベートメソッド：1回分の対戦処理（内部でのみ使用）"""
        player_roll = self._player_die.roll()
        computer_roll = self._computer_die.roll()

        print(f"あなた:{player_roll} vs コンピュータ:{computer_roll}")



        # 勝敗を判定
        if player_roll > computer_roll:
            result = PLAYER_WIN
        elif player_roll < computer_roll:
            result = COMPUTER_WIN
        else:
            result = DRAW

        print(self._config.messages[result])
        return player_roll, computer_roll, result

    def _print_match_statistics(self, match_history):
        """プライベートメソッド：対戦統計を表示（内部でのみ使用）"""
        if not match_history:
            print("対戦履歴がありません。")
            return

        print("¥n---詳細な対戦履歴---")
        for i, match in enumerate(match_history, start=1):
            player_roll, computer_roll, result = match
            print(f"{i}戦目: あなた:{player_roll}, コンピュータ:{computer_roll}, 結果:{result}")

        print("¥n---統計情報---")

        player_rolls = [match[0] for match in match_history]
        computer_rolls = [match[1] for match in match_history]

        print(f"あなたの出目統計:")
        print(f"  最高出目: {max(player_rolls)}")
        print(f"  最低出目: {min(player_rolls)}")
        print(f"  平均出目: {sum(player_rolls) / len(player_rolls):.2f}")

        print(f"コンピュータの出目統計:")
```

```
        print(f"  最高出目: {max(computer_rolls)}")
        print(f"  最低出目: {min(computer_rolls)}")
        print(f"  平均出目: {sum(computer_rolls) / len(computer_rolls):.2f}")

        win_counts = {PLAYER_WIN: 0, COMPUTER_WIN: 0, DRAW: 0}

        for match in match_history:
            _, _, winner = match
            win_counts[winner] += 1

        print(f"¥n最終結果:")
        print(f"  あなたの勝利: {win_counts[PLAYER_WIN]}回")
        print(f"  コンピュータの勝利: {win_counts[COMPUTER_WIN]}回")
        print(f"  引き分け: {win_counts[DRAW]}回")



    def play(self):
        """パブリックメソッド：外部から呼び出されることを想定"""

        match_history = []

        while True:
            player_roll, computer_roll, result = self._play_one_round()
            match_history.append([player_roll, computer_roll, result])

            if not self._ask_continue("もう一度対戦しますか？ (y/n): "):
                print("対戦を終了します。")
                break

        self._print_match_statistics(match_history)

def select_difficulty():
    """難易度を選択する関数"""
    print("難易度を選択してください:")
    print(" 1: 初級（プレイヤー有利 - あなた1〜9面 vs コンピュータ1〜6面）")
    print(" 2: 中級（通常 - 両方とも1〜6面）")
    print(" 3: 上級（プレイヤー不利 - あなた1〜4面 vs コンピュータ1〜6面）")

    while True:
        choice = input("選択 (1/2/3): ")
        if choice == "1":
            return EASY
        elif choice == "2":
            return NORMAL
        elif choice == "3":
            return HARD
        else:
            print("無効な選択です。1、2、3のいずれかを入力してください。")

# メイン処理
if __name__ == "__main__":
    # まず難易度を選択
    difficulty = select_difficulty()

    # 選択した難易度でゲーム設定を作成
```

```
game_config = GameConfig(difficulty=difficulty)

print(f"¥n選択された難易度: {game_config.difficulty_name}")
print("モード選択: 1=ソロプレイ, 2=対戦")
mode = input("選択 (1/2): ")

if mode == "1":
    game = SoloGame(game_config)
elif mode == "2":
    game = VsComputerGame(game_config)
else:
    print("無効な選択です。")
    exit()

game.play()
```

ポイント:

- **最小限の変更で大きな機能追加**: 既存のクラス構造を活用し、主に `GameConfig` クラスの修正だけで難易度機能を実現しました。
- **設定の一元管理**: 難易度情報を辞書で管理することで、新しい難易度の追加も簡単になります。
- **@propertyの活用**: 設定値への安全なアクセスを提供し、外部から設定を変更される心配がありません。
- **カプセル化の効果**: 各クラスの責任が明確で、設定変更の影響範囲を限定できます。
- **拡張性**: 新しい難易度（例：「超上級」）を追加する場合も、辞書に1行追加するだけで済みます。

> コラム **関数ベースの実装と比較してみよう**
>
> もし関数ベースの実装で同じ機能を追加しようとすると：
>
> ```
> # すべての関数に難易度パラメータを追加する必要がある
> def roll_die(sides=6, is_player=True, difficulty="normal"):
>     # 難易度判定のロジックを各所に追加
>
> def play_vs_computer(difficulty="normal"):
>     # 難易度に応じた分岐処理を多数追加
>
> def print_match_statistics(match_history, difficulty="normal"):
>     # 統計表示も難易度に応じて変更
> ```
>
> オブジェクト指向では、設定クラスに集約することで、コードの重複を避け、変更を最小限に抑えられます。

> コラム **さらなる拡張の可能性**

このクラス設計なら、以下のような機能も簡単に追加できます：

1. **カスタム難易度**：ユーザーが自由にサイコロ面数を設定
2. **ハンディキャップ機能**：プレイヤーに+1ボーナスなど
3. **特殊サイコロ**：特定の面が出やすいサイコロ
4. **ゲーム履歴の保存**：過去の成績を記録・表示

オブジェクト指向の設計により、機能追加が容易になっています！

> コラム **オブジェクト指向プログラミングの恩恵まとめ**

今回の難易度機能追加を通じて、以下のOOPの恩恵を実感できました：

1. **変更の局所化**：主に `GameConfig` クラスだけを変更すれば済む
2. **コードの再利用**：既存のクラス・メソッドをそのまま活用
3. **責任の分離**：各クラスが明確な役割を持ち、理解しやすい
4. **拡張の容易性**：新機能の追加が既存コードに与える影響を最小化
5. **保守性の向上**：バグ修正や改良の際に影響範囲が明確

大規模なプログラム開発では、これらの恩恵はさらに重要になります！