Assignment 1.2: Building Modular Games in Python

Preparation for Monte Carlo Simulations

1 Objective

The goal of this assignment is to strengthen your Python fundamentals, specifically focusing on **Object-Oriented Programming (OOP)** and **Modular Design**.

Before we dive into complex Monte Carlo simulations, we need a robust simulation environment. Given you have gone through basic python material and are confident enough this task will test it all. Your task is to implement a card game (like **Blackjack**), dice game or any other simple terminal based game. The focus is not just on making it "work", but on making the code clean, reusable, and modular. This will help you to build a strong base in python and programming in general.

Why Modularity?

In Monte Carlo simulations, we often run a game millions of times. If your game logic is entangled with your print statements, or if your state management is messy, the simulation will be slow and difficult to analyze. Separation of concerns is key.

2 Sample Architecture(for blackjack)

Here is a sample architecture for a modular implementation of blackjack game in python. It is a sample on how to implement simple logic in modular manner. It might seem overkill but is helpful when you need to do small changes to logic and there wont be a ripple effect of bugs all over your code. Try not write all your code in a single file. Structure your project to separate data representation from game logic.

2.1 The Data Model (`cards.py`)

This module should handle the physical components of the game. It should not know anything about the rules of Blackjack or betting.

- **Enum for Suits:** Use Python's `enum` class to define Card Suits (Spades, Hearts, etc.). This prevents typo-related bugs.
- **Class `Card`:**
  - Should store the `suit` and `value` (rank).
  - Implement the `__str__` dunder method to return a pretty string representation (e.g., "A♠").
- **Class `Deck`:**
  - Should contain a list of `Card` objects.
  - Methods needed: `shuffle()`, `draw()`, and `reset()`.
- **Class `Hand`:**
  - Represents the cards held by a player or dealer.
  - Methods: `add_card()`, `flush()` (remove all cards).
  - **Critical Logic:** Implement a method to calculate the *value* of the hand. In Blackjack, Aces can be 1 or 11. This calculation logic belongs here.

2.2 The Game Engine (`blackjack.py`)

This module controls the flow. It imports from `cards.py`.

- **Class `BlackJack`:**
  - **State:** Maintains the deck, the player's hand, and the dealer's hand.
  - **Actions:** Implement methods for `hit()` and `stand()`.
  - **Rules:** Implement logic to check for "Bust" (over 21), "Blackjack" (21 on first two cards), and comparisons between player and dealer.
  - **The Loop:** A method like `run_match()` that orchestrates a single round.

3 TUI: Text-Based User Interface

Here is a small guide on how to make the terminal output look good, for a more immersive experience. It is completely optional and up to you to use if you like it. A simulation is useless if we cannot visualize it for debugging. However, standard console output flows downward. If you simply `print(card)` inside a loop, your hand will look like a vertical list, taking up too much screen real estate.

You need to engineer a way to print cards **side-by-side**.

3.1 Anatomy of an ASCII Card

Instead of treating a card as a single string, think of it as a list of strings (slices).

Example Target Output:

```
 ___   ___
|A  | |10 |
| ♠ | | ♥ |
|__A| |__0|
```

To achieve this, your `Card` class needs logic to generate the specific lines.

- **Line 1 (Top):** The top border.
- **Line 2 (Rank):** The rank (A, K, 2, etc.) aligned to the left.
- **Line 3 (Suit):** The suit symbol in the center.
- **Line 4 (Bottom):** The bottom border with the rank aligned right.

Tip: Unicode Suits

Python handles Unicode natively. You can make your TUI look professional using these codes:

- Spades: `u'\u2660'` (♠)
- Hearts: `u'\u2665'` (♡)
- Diamonds: `u'\u2666'` (♢)
- Clubs: `u'\u2663'` (♣)

You can always search on google for more symbols, if you think of some common symbol it is highly probable a unicode character exists for it.

3.2 The Horizontal Join Logic

To print cards horizontally, you must stitch the corresponding lines of each card together before printing.

The Logic:

1. Create an empty list of strings to hold the combined output rows (e.g., 4 empty strings).

2. Iterate through every card in the `Hand`.
3. For each card, break its string representation into lines (using `.split('\n')`).
4. Append Line 1 of the current card to Line 1 of your output buffer; append Line 2 to Line 2, etc.
5. Finally, join the output buffer with newlines to print.

Conceptual Implementation:

```python
# Inside your Hand class __str__ method:

lines = ['', '', '', ''] # Buffer for 4 rows of text

for card in self.cards:
    # Assume str(card) returns the 4 lines separated by \n
    card_rows = str(card).split('\n')

    for i in range(4):
        # Append the specific row of this card + some spacing
        lines[i] += card_rows[i] + "␣␣␣"

# Return the full block joined by newlines
return '\n'.join(lines)
```

3.3 more examples

Simple die

```
 ___  ___  ___
|   ||0  ||0 0|
| 0 || 0 ||0 0|
|___||__0||0_0|
```

Advanced die

```
┌───────┐
│ ●   ● │
│   ●   │
│ ●   ● │
└───────┘
```

4 Using Randomness

Since we are heading toward Monte Carlo simulations, understanding the `random` module is essential.

- **Reproducibility:** While testing, you might want the same deck order. Look into `random.seed()`.
- **Shuffling:** Use `random.shuffle()` for the deck.
- **Selection:** Use `random.choice()` if you need to pick random attributes (like creating an arbitrary card for testing).

5 Reference Material

Refer to "*The Big Book of Small Python Projects*" (Project #4: Blackjack) for game rules and ASCII art inspiration.

Note: The book uses procedural code (global variables and simple functions). Your assignment is to elevate this to a **Modular, Object-Oriented Architecture**.

6 More Ideas

If you prefer not to implement Blackjack, you may choose a game from the list below based on your confidence level. All projects should adhere to the same Modular OOP standards.

6.1 Level 1: The Essentials (Beginner)

Recommended if you are new to Python classes or TUI Logic.

- **The Game of Pig (Dice Game):**
  - **Objective:** A push-your-luck game where you roll a die to accumulate points but lose turn-points if you roll a 1.
  - **Key Classes:** `Die`, `Player`, `Game`.
  - **TUI Challenge:** Drawing the 6 faces of a die using ASCII characters.
- **War (Card Game):**

  - **Objective:** A pure simulation where players compare top cards. Higher card wins the pile.
  - **Key Classes:** Same as Blackjack (`Card`, `Deck`, `Hand`).
  - **TUI Challenge:** Displaying two cards side-by-side and indicating the winner visually.

6.2 Level 2: The Standard (Intermediate)

Similar complexity to Blackjack.

- **Shut the Box:**
  - **Objective:** A board has tiles numbered 1–9. Roll dice and "shut" tiles that sum to the roll value.
  - **Key Classes:** `Dice`, `Board` (manages the state of tiles), `Player`.
  - **TUI Challenge:** Displaying the state of the tiles (e.g., `[1] [X] [3] ...`).
- **Roulette:**
  - **Objective:** Place bets on where a ball will land on a spinning wheel.
  - **Key Classes:** `Wheel`, `Table` (manages the grid of numbers), `Bet` (polymorphism is useful here for different bet types).
  - **TUI Challenge:** designing a betting table layout.

6.3 Level 3: The Challenge (Advanced)

For those who want a significant coding challenge.

- **Video Poker (Five Card Draw):**
  - **Objective:** You are dealt 5 cards. Choose which to discard. Redraw. Best hand wins.
  - **Challenge:** You must write a `HandEvaluator` class that can algorithmically detect Pairs, Straights, Flushes, and Full Houses.
- **Minesweeper:**
  - **Objective:** Clear a grid of hidden mines using numeric clues.
  - **Challenge:** Managing a 2D Grid state and recursive clearing (flood fill) when zero-neighbor tiles are clicked.