Program Running Instructions

The program comes in 3 files (not including this README pdf and the Brainstorm pdf), however only **main.py** needs to be run. The program can be run by typing `> python main.py` in the command line, at which point the game immediately begins.

No additional dependencies besides a python interpreter are necessary, the entire program uses no external libraries besides `random`.

Note: The Blackjack Brainstorm pdf is not necessary in understanding how the program functions, it serves more as a collection of random and unorganized notes I used in order to navigate the prompt. I figured it best to attach it with the other files as it documents some of my thought process.

Assumptions / Tackling Unclarity

- Should I ever worry about the deck running out?
  - I decided I didn't, considering the game itself consists of (at most) 2 turns, the player and then the dealer, and we will never actually be able to draw the whole deck.
  - My reasoning was: In the worst case scenario, the max points that can be reached require the player to have 20pts and the dealer to have 21 (given the turn order), for a total of 41 points. The player hitting 21+ points would immediately end the game, and similarly for the dealer. This means that any game where the total points exceeds 41 cannot be possible (as it would automatically end via win or bust). Considering a standard deck of 52 sums to 380 in the worst case (counting Aces as 11 and J,Q,K as 10), and 340 in the best case (counting Aces as 1), we can see that **the game will end far before we reach the end of the deck**.

- Should hands have infinite capacity?
  - Infinite capacity for hands is not necessary, directly influenced by the assumption above.
  - The max amount of cards a hand can have before reaching 21+ points is (4 Aces + 4 Twos + 3 Threes) = 11 cards for 21 points. While a dynamically growing hand could be implemented (for example by doubling its size every time it hits capacity), a static hand of 12 could just as easily work.
  - In the end, python's built in lists allowed me to implement a flexible hand without much trouble.

What I Did Well

I feel something I did well in this assignment was create a straightforward program while leaving a strong and flexible foundation for expansion of mechanics. I also think modularizing my utilities lends itself to the overall scalability of this program. In addition to this, I am proud of this README doc, I hope it proves useful in navigating the program.

Design Choices / Algorithmic Decisions

- **General Design Choices**:
    - There are 4 driving forces in the game, a **player** that takes inputs, a **dealer** with predefined behavior, **message displays**, and a **main system / engine** that enforces the ruleset of the game.
    - Modularization in mind, the different utilities are divided into their respective classes/methods: **Player** (encompassing the input player and the dealer), **Deck** (offering any methods the deck may need), **Display** (offering tools for displaying messages during the game), and the **WinLoseChecker()** method (takes care of win/bust cases). These can all be found in **utils.py**.
    - The game itself can be found in **main.py** as a simple method call to main(). Here, all utilities from the aforementioned file are referenced to create the general flow of the game.

- **Players**
    - Players can be instantiated via the **Player** class, with a name fed in as an argument.
    - Players have **3 primary methods**, along with a few helper methods that are of no concern at the high-level. Players also have **5 attributes** (including the name attribute).
    - **Attributes:**
        1. **self.name -> str**. An identifier for who the player is, input-player or dealer (helps the **Display System** differentiate between players).
        2. **self.hand -> List<str>**. Stores/keeps track of all cards the player has in their hand.
        3. **self.handSum -> int**. Keeps track of the **optimal** overall sum of the player's hand.
        4. **self.numAces -> int**. Indicates how many aces the player has in their hand (helps in finding optimal sum).

5. **self.prevAceVals -> int**. Indicates the previous most optimal sum of aces in the player's hand (helps in finding optimal sum).

- **Methods:**
    1. **__init__(self, name: str) -> Player**. Initializes a player.
    2. **hit(self, deck: Deck) -> None**. Takes the argument deck and draws a card from it, adding it to the player's hand.
    3. **getHandValue(self) -> int**. Returns the optimal sum of the player's hand in O(1) time.
    4. **Helper: addCard(self, card: str) -> None**. Adds the given card to the player's hand. If the player has at least one Ace, check for a new optimal value.
    5. **Helper: recalculateHandValue(self) -> None**. Calculates the new optimal hand sum in O(1) time by keeping track of the amount of Aces as well as the Ace values prior to optimizing. There can be a max of 1 Ace worth 11 points in a hand, meaning there are always at most 2 options for Ace values: all 1s, or a single 11 and the rest as 1s. By subtracting the previous Ace values from the current hand sum, we can avoid re-summing the deck (excluding Aces). Also by keeping track of the number of Aces, we can evaluate our two options; self.numAces in the case of all 1s or (self.numAces - 1) + 11 in the case of a single 11. Summing this to (self.handSum - self.prevAceVals) returns the new optimal hand value.

Cards

- The initial plan was to create a Card class that would specify attributes, but I felt that was overcomplicating things for a project of this scope. **Representing cards as strings** and **accessing their respective values from a str -> int map** proved to be a simple yet efficient implementation.

The Deck

- The Deck class has **2 attributes** and **4 methods**, all fairly straightforward.
- **Attributes:**
    1. **self.deck -> List<str>**. A collection of all 52 cards (4 of each type).
    2. **self.size -> int**. Indicates how big the deck currently is. (Unused)
- **Methods:**
    1. **__init__(self) -> Deck**. Initializes the deck and size.
    2. **shuffle(self) -> None**. Uses the random library to randomize the order of cards in self.deck.

3. **draw(self) -> str**. Picks and returns a card from self.deck by popping the 0th element, ideally done AFTER the deck is shuffled at least once (at the start of the game).
4. **size(self) -> int**. Returns the current size of the deck. (Unused)

- **Display System**
    - The **Display System**, which encompasses all visual feedback for the game, is a class with **8 methods**, one being a helper function. They are all straightforward, being simple print statements that visualize what occurs in the game.
    - **Methods:**
        - **displayHand(self, player: Player, hiddenCards=0: int, hideTotal=False: bool) -> None**. Displays the hand of the given player in the command line. The **hiddenCards and hideTotal arguments are optional**, hiding no cards or the total by default, but can hide the specified amount of cards/total by displaying them as **'?'** in the command line.
        - **displayInputPromt(self) -> None**. Displays the input options during the input-player's turn.
        - **displayHit(self, player: Player) -> None**. Displays a message stating the given player has hit.
        - **displayStandHand(self, player: Player) -> None**. Displays a message stating the given player has stood, along with their current hand.
        - **displayWin(self, player: Player) -> None**. Displays the given player as the winner of the game, and their winning hand.
        - **displayBust(self, player: Player) -> None**. Displays the given player as the loser of the game, and their losing hand.
        - **displayTie(self) -> None**. Displays a tie message.
        - **handToStr(self, player: Player) -> str**. Converts the given player's hand into string format.

- **Rule Enforcer**
    - The rules are very simple for this game, all based on the value of their hand:
        1. If any player **reaches 21**, they automatically **win**.
        2. If they go **above 21** they automatically **lose**.
        3. Any amount of points **less than 21** means they are free to **Hit/Stand** as desired.

- Given these three cases, I found it sufficient to create a method **WinLoseChecker(player: Player, display: Display) -> bool**, which checks what case the given player falls under and (if necessary) communicates with the display to show the corresponding message in the command line. If the case is game-ending (21 or bust), the method returns true, otherwise false.

- **Game Structure**
  1. **Phase 1:**
     The **Display System** and **Deck** are **initialized** along with both **Players** (input and dealer), the **Deck** is shuffled. Players are then **given 2 cards each** from the **Deck** and **their hands are displayed**. There is a case where a player gets (10/J/Q/K + Ace) as their starting hand, which is 21, thus we have a **WinLose Check** at the end of this phase.
  2. **Phase 2:**
     This is the **input player's turn**. The player may **Hit** or **Stand** whenever by **entering H/h or S/s** respectively in the **command line**. This was implemented with a while(true) loop that checks for inputs. Any time the player **Hits**, the updated hands are displayed and a **WinLose Check** is done. When the player **Stands**, the while(true) loop breaks and the **dealer's turn** begins.
  3. **Phase 3:**
     This is the **dealer's turn**. As predefined by the spec, the **dealer** will **Hit until** their hand value is **17 or above**. Every time the **dealer Hits**, the updated hands are displayed and a **WinLose Check** is done.
  4. **Phase 4:**
     This is the final phase. If no player got 21 or above during their turn, both **scores are compared** and the player with the **highest score wins** (since it's below 21), and their victory is displayed in the command line. In the case of a **tie**, a tie message is displayed.

- **Ace optimality**
  - Refer to **Design Choices / Algorithmic Decisions -> Players -> Methods -> 5.** .

Tradeoffs While Programming

The biggest tradeoffs I kept running into while programming were how scalable I wanted my program to be. At nearly every point of implementation I would think to myself “Should I make this mechanic more general?” / “Is this something I might want to expand on?”. It was a bit difficult finding a balance of intuitive yet scalable implementations, I often had to take the time constraint into account and disallow myself from creating an overly complicated mechanism. Overall, however, I feel I hit a healthy balance of simplicity and scalability. Isolating the different components of the game into classes formatted my code in such a way that expanding on any given aspect doesn’t require navigating a tangled path of function calls. Things are also modularized in intuitive ways and can be modified without disturbing the other components.

Improvements Given Time

Given more time, my first step would be to do a full revision of the program and polish what I currently have. The second and biggest thing I would improve on is the components themselves, adding more and interesting mechanics to the game. One example of this, which ties back into the tradeoffs portion, is how I dealt with turns. Of course, in the spec you see that there are only really 2 turns to account for, one for the input player and another for the dealer. But at some point I was heavily considering creating a more nuanced turn system that could account for more actions from players as well as accommodating more players in general. This addition would make the game feel more dynamic, and being able to play with other input-based players would up the ante.

Another mechanic I had in mind that goes hand-in-hand with Blackjack is a betting system. This system would let players bet some amount of currency on a hand, returning increased rewards if successful. This mechanic would be trickier to implement and would definitely take more time, but I think it would be a fun addition.

A final improvement I would try to work on given more time is a better display system, in other words a better GUI. This would really add to the overall integrity of the game.

Manual Tests I Ran

Many of the tests I didn’t get to automate were moved to this manual portion. This included working through potential edge cases for methods on scratch paper, as well as manually following the logic of a method line by line with examples to evaluate its integrity. A lot of manual testing consisted of breaking down the game logic into smaller portions to evaluate.

Automated Tests

Although I was unable to spend much time creating automated tests, I was able to create a few. These can be found in the **test.py** file, sectioned off with comments indicating what portion the tests relate to. Tests are **organized in terms of what component they are testing**, and they can all be called in a single function call to **allTests(n: int) -> None**, where n dictates how many times the deck is shuffled and drawn from, as well as how many times the player hits, all in their respective testing categories.

Tests can also be run in terms of components. In each section, divided by comments, you can see the topmost method calls all tests in that section. These can be called if you only want to test a specific portion of the program. Equally so, you could call specific tests via their methods.