Blackjack Program Analysis

Zachary Avant

To play a card game, one needs cards first, so I figured a good place to start was generating the deck. There are 52 cards in a standard deck, and that would be quite a lot of typing and double-checking. Much easier is breaking a card down into its relevant properties and generating a deck based on those. In Blackjack, only the suite and face matter. The suite is important in creating the correct number of unique cards, while the face carries the card’s actual value. Color doesn’t matter (and suite only matters inasmuch as it ensures uniqueness). There are 13 faces and 4 suites; a nested loop can be used to generate all 52 possible combinations from those 17 components.

Initially, I used a string to store these cards and extracted the value from the first or second characters (second in one case because 1 and 10 share the same first character). However, as we learned in lab, a tuple would be a more efficient container for a card. Tuples don’t have to store the ‘of’ (as in ‘face of suite’) and, in the case of the integer faces, don’t need to convert types. Versus lists, tuples don’t have the memory overhead that comes from mutability. Therefore, a list of tuples instead of a list of strings was used.

Making the initial deal -- the next part of the game -- requires drawing from the deck. Using the random module would be crucial. There is a shuffle function, but it seemed like that might be unnecessary processing. The card-tuples are stored in a list, which is indexed. Drawing the first element is just as fast as drawing the middle one. Reorganizing the entire list seems like it’ll be less efficient than just looking at a random index (but given that a random index has to be generated with each draw, this might be less efficient in the long run -- I have no idea how “fast” creating a random integer actually is). The item at that index - a card in tuple form - can be removed from the deck of cards and added to a hand (initially set to empty). Should the deck run out, a new one is generated (and it should be generated in-place). I didn’t want to use global variables in the program, so the deck has to be passed between functions several times. In retrospect, deck would’ve been a good candidate for global.

Card values are very straightforward, apart from one case. All cards are worth their number, if they have one (eg: an 8 of spades is worth 8 points). “Named” cards are worth 10 points (eg: a king of clubs is worth 10 points), except for the ace. An ace can be either an 11 or a 1, whichever is most beneficial to the holder. Figuring out how to represent this is probably the most difficult part of the program. For a hand with just one ace, it’s easy enough; if the value of the hand goes over 21 from the ace being valued at 11, then the ace should be a 1 instead. If not, the ace should be an 11. Because the content of a hand could change, the value of the ace has to be updated each time a new card is added to avoid putting the hand over 21 points. However, a hand could potentially have more than one ace -- as many as 4 in the “worst” case (worse for computations at least. 4 aces could be a winning hand!).

I considered several solutions to a hand with multiple aces, but trying recursion was the most fruitful. It didn’t work well, but it did help me think about breaking the problem down into various cases, starting at a base case. At one ace, the hand is valued as described above. When a second ace is added, it cannot be 11. Two aces can never be 11 at the same time, as that puts the value over 21. Since only one ace can ever be 11, the rest must necessarily be valued at 1. Only one ace needs to be evaluated then, and all the rest can be held at 1 and added to the hand’s non-ace value. Since the ace has only two possible values, there’s only two possible cases: in one, the hand’s value is 10 or less, and adding 11 will not cause a loss. In the other, the hands value is 11 or more, and the ace must be a 1.

Since the cards are stored as tuples, there needs to be a way to convert them into string form for prettier printing. Elements of the tuple are copied into their appropriate place within a string, separated by “ of ” (as in “7 of diamonds”). Whenever the user needs to see a card, this will be used to make it more readable.

For betting, I decided to try using object oriented programming. I created a Wallet class that stores the user’s money and allows for gambling simulation. The customizable nature of it made it very easy to implement into the broader program despite being a bit of a later addition. As betting is an optional feature, it’s included as a separate program.

The primary body of the Blackjack program should have multiple return statements to avoid having to run through the whole thing in the average case. Points need to be calculated at key parts of the program, and if they meet or exceed 21 in accordance with the rules of Blackjack, the program exits and the results are passed to a frame function that allows for repeat play and score tallying. It’s unclear which of these is really “helping” which, but I’ll be labeling the primary body, where most of the game occurs, as blackjack\_helper due to its lower position on the hierarchy of functions. There should be a hit/stand/quit user input loop. The user can enter “stand” at any time to exit the loop and continue the program. The user could enter “quit” to exit the entire program. The user could also enter “hit” to draw another card indefinitely (but practically limited by the 21 point rules). The dealer may then draw until the hand’s value is 17 or more (as is optimal for the dealer). If the program hasn’t exited already, points must be compared, and a winner declared (no ties allowed -- the dealer wins a tie).