“Kab’s Amazing Poker” in Python

by Noa Chayer

**Introduction**

While I was unable to secure a spot in introduction to programming during my time at Dawson, I was set on at least making my independent project within that domain. I must thank Jonathon Sumner for making that happen and taking the time to oversee my project. I had a lot of fun and learned a lot while working on this. The final product is a game meant to simulate 4-player poker with the use of python and the pygame library, humbly called “Kab’s Amazing Poker”. The game can be found in the resources section (p.5).



**Breakdown**

The development process can be broken down in 4 major sections: Game logic, Card logic, AI, and finally the UI.

Game Logic:

The only class in the main script is to create player objects. These objects represent each player with every property that needs to exist being attached to them (ex. money, cards, status, id, etc.). This allows for quick access for everything that is needed to simulate money management. Money is calculated using call targets and forwards. If a given player’s forward does not match the call target, they must call and the difference of the two will be forwarded. It is only at the end of the rotation (before the table shows additional cards) that the sum of all player forwards are added to the pot, which is distributed to the winner at the end of the round. The appropriate measures are taken to account for money limitations and blinds.

The play order for each player is first determined by who gets the big blind, respecting the rules of real poker. The program then goes through a “while” loop with its condition being the presence of an empty element in the “status” list. When either the player or a computer decides to call, check, raise, or fold, their status, which is a property of their object, that status gets attached to the global status list as a string. It is only if all elements in the list have an existing value that the rotation of players is done. Obviously, when a player raises, the list (except “Busted” statuses) is reset to continue the rotation further. It is in this rotation loop that decision functions are called for both the player and the computers, which will be touched upon further in the documentation.

And so, the overall logic hierarchy involves a triple-embedded “while” loop. The most parented one is looped every round. It resets every value and resets the decks while containing the appropriate elements to generate the game window. It is the main loop. Then, the second middle loop accounts for game rotation within a single round. Finally, the child loop of the former checks for statuses, as explained previously. These loops, along with many flag checks form the basic skeleton for the game to function.

Card Logic:

This logic encompasses both card distribution and card evaluation. Four lists are defined. The first is a deck list with 52 elements, each representing the 52 cards in a standard deck as strings under the format “SUIT+VALUE”. For example, a queen of hearts is depicted as “HQ”. This makes it easier to later access the suit and value if needed like when creating the visual sprites for the cards. This string list has a parallel list consisting of proper numerical worth for each card. This ensures that the index of one represents the value of the other. And so, a 3 of spades is worth 2 points while a king of spades is worth 12. This is the core idea behind eventually evaluating player hands and cards. The other two lists have the same function but are for the hands (High card to Royal flush).

When the round starts, two cards are given to each player by randomly selecting them in the deck and attributing them to the players’ hands, removing said card from the base deck in the process to avoid duplicates. The same principle applies each time a card is drawn to create the table deck (flop, turn, river).

During a round, each player’s cards are evaluated multiple times to assess their worth and calculate a risk factor that influences the computer’s decisions. Potential straights or flushes are also recognized and reduce the player’s risk factor even further.

At the end of the round, everyone’s cards are evaluated through sorting logic and basic algorithms to determine potential hands and their corresponding values to determine the winner (for an in-depth look, refer to the value\_det() function at line 27 of main.py). When all is done, everyone’s cards are reset to nothing, and the base deck is replenished.

Artificial Player Intelligence:

The decisions that the AI takes in this game are centered around the premise of risk as explained previously. The risk, being a float mainly between 0 and 1 (no positive limit but is highly unlikely to be higher than 1), makes use of “if” conditions along with other factors to determine the action the computer should undertake. However, the risk is not the only numerical factor that is considered, a small random factor is also given to create some form of unpredictability.

In the same vein, certain conditions are implemented to ensure that the computers don’t undertake self-sabotaging plays. For example, if the call target is 0, even if a computer’s risk is above 0.5 (high), they will refrain from folding since they can check and let the game play out. Similarly, if a computer has the big blind at the beginning of a round, even if the risk is high, the computer will check. Additionally, a small random factor can determine if a bot bluffs. There are many more of these small adjustments and alongside the overall risk system, it ensures that the bots play in a pseudo-conservative fashion. Playtests involving other people than myself have shown that the bots are able to win in a long-term scenario, however, this will be touched upon later in the documentation (for a more in-depth look at the AI, please refer to the cpu\_decision() and cpu\_raise() functions at line 390 and 469 respectively in main.py).

User Interface (UI):

When first working on this project, it was debated whether a UI alongside a full game window should be incorporated as it would take a lot of time and resources. However, it was decided that it would be for the best as poker is a visual-heavy game as most people who know poker recognize hands and cards not by a string of two characters but by a visual representation of said cards. Pygame was the library of choice for this endeavor as it offers substantial documentation and easy-to-use resources to construct a fully visual game.

\*Every sprite and visual element that can be seen is an object created with its own classes. The tick rate of the game is set to 60 using pygame.clock.\*

The cards are composed of two elements: the suit and the value. A pre-drawn sprite of a card with each suit is present in the files so they can be accessed and instantiated while the actual value of the card is manually drawn onto the base sprite. Originally, it was considered to create 52 sprites, one for each card. However, that would have been excessive and unoptimized.



Fig 1. Empty Card Sprite vs Rendered Card (C9)

The bots all have their unique sprites and animations, which work by creating an array of frames and loading the appropriate image files using an “i” loop. The objects loop through that array to display the next frame. A similar process is used to create the scrolling text. Sequential characters originating from the string of text that needs to be displayed are appended to the displayed text one at a time, incrementing once every 2 frames while playing the “tick” sound effect.

There are mainly two transformative animation definitions: move\_card() and flip\_card(). Using the object that is passed through when calling the function, the former calculates an angle θ that is used to modify the object’s center coordinates over time to create motion while the latter scales the object inward and replaces the current sprite of that card with a stored sprite of the same card when its scale is null. Following that, the card is scaled back up to normal, creating the illusion of flipping the card (for a more in-depth look refer to lines 372 and 411 of UI.py). Side note: it is in those definitions that the sound effect is played.

**Playtesting**

The final game was play tested by multiple people to get insight into the effectiveness of the artificial intelligence.

Methodology:

Players were put in front of the game and were told to play to win for 10 minutes. At the end of that time, the final amount of money for every CPU and the player was recorded at the end of the round in progress. If the play tester was not aware of how to play poker, they would be given a simple explanation before starting. The player’s proficiency in both poker and programming has been noted (refer to fig.2).

Results:

Fig.2 – Tabulated playtest results

A screenshot of a computer

Description automatically generated with medium confidence

(Notes and comments from each playtest can be found in the .xlsx file in the resources)

Given this data, the mean player gain is -4187.50 while the mean CPU gain is 1395.83 over a total of 86.42 minutes. This sets a positive trend for the computer, indicating some form of poker intelligence (more data can be found in the aforementioned .xlsx file.)

However, there was one player with higher poker knowledge and experience that was able to completely wipe out the CPUs of their money. Unfortunately, this piece of data had to be omitted since the game ran into multiple bugs, causing and erroneous summation in which the total amount of money was higher than what was possible. This issue has since been fixed.



\*This data was not included in the mean and median.

**Conclusion**

This project allowed me to learn a lot more about animation and image manipulation through pygame and was both a good logical and artistic exercise. I am content with the final product. The game appears to have no glaring flaws or bugs and the graphics and sounds add a lot to it.

If I had to reproach my code, I cannot help but find that it is somewhat clunky and strenuous on the computer that runs it. Additionally, I had to cut some features to save some time. One of those is the side pot. I had coded the skeleton for it, but it quickly became obvious that it would take a lot of time and would put the rest of the code in jeopardy stability-wise. Other than that, a return function when clicking on “Raise” by accident could have been arranged. Finally, small quality-of-life elements such as the buttons darkening while hovering over them or an indicator showing when it is the player’s turn would have been welcome.

However, despite these minor shortcomings, the goal of creating a fully functional poker game with working and seemingly intelligent artificial intelligence has, in my opinion, been reached.

**Resources**

Game (Instructions included): <https://github.com/Kaboudouo/CE_Poker/tree/master>

Collected data:

