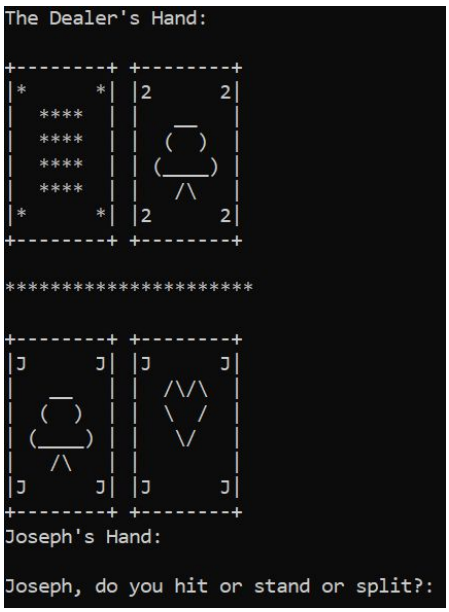
# Blackjack Design Document

## Instructions for running:

On a Unix-based operating system like Linux or MacOS, simply navigate to the directory of the 3 files in the terminal and type: `python Blackjack.py`
If using Windows, navigate to the directory of the 3 files in the command prompt and type: `py Blackjack.py`

Sample output from the game is shown below:




## Explanation of design choices:

The Blackjack game is divided into 3 files, each one having a specific, unified purpose which allows maximum modularity and usability. For example, if we wanted to create a different card game, we could reuse the `Cards.py` file as it holds classes for a deck of cards and a player, which is a construct that can be used in more than just blackjack.

### Cards.py

As previously explained, `Cards.py` holds classes for a deck of cards and a player. The deck of cards subclasses from the “MutableSequence” class found in the standard library collections.abc. This choice was made because, through Python’s generous duck-typing, the deck of cards can work easily with functions like random.shuffle(deck), deck.pop(), and other, iterator specific methods/functions. Note: the __repr__ methods for both classes are unnecessary in a practical sense, but are kept in the file because it allows easy debugging using the print() function, which is good practice.

### terminalCards.py

`terminalCards.py`, like `Cards.py`, holds code that is independent of blackjack specifically. `terminalCards.py` enables console output of the cards in the game using ascii art. The choice of using the basic print() function to output this ascii art of the cards could be controversial. One could, quite reasonably, claim that I should have used sys.stdout.write(), or perhaps a library like “curses” which would allow beautiful, console-based screen painting. I would personally argue that, while curses would allow a more visually appealing, easily used user interface, it should not be used in this context because it only works on Unix-based machines. While the prompt for the challenge did only specify it needed to work Unix-based machines, portability should still be maintained just in case a user did wish to run it on a Windows machine. My commitment to portability can also be seen in my clearTerminal() function which works on both Unix-based consoles and Windows command prompt. I would also argue that sys.stdout.write() would needlessly complicate such a simple task, bordering on over-engineering, thus the humble print() works quite well here.

### Blackjack.py

`Blackjack.py`, holds all the blackjack specific code for this game. It extends the regular player class found in `Cards.py` to have blackjack specific methods and attributes. It also implements all the rules and gameplay of blackjack using a well factored main() function which allows the reuse of `Blackjack.py` as a module. A particularly astute reader of the code may notice that no getters and setters are made, which, at first glance is a cardinal sin (no data encapsulation) when creating object oriented code. However, this is not the case in good Python programming, because, if a getter and setter would ever need to be added, we would simply use a “property” decorator and our code would be easily changed without the need for any messy getters and setters.

## Choice of tooling:

Python 3.7 was used for this code, partially because it is my strongest language, but also because it allows extremely quick programming (allowing this code to be written in under 3 hours) and is common enough to be portable to most machines.

The only modules used other than the ones we created were the standard “random” and “time” modules. By only using standard modules we make our program as portable as possible (meaning external libraries don’t need to be installed to run the code), decreasing the burden on the user.

No tests were written for the code because of time and the lack of complexity, however, if tests were to be written, unittest would be used because it is a standard module in Python 3.