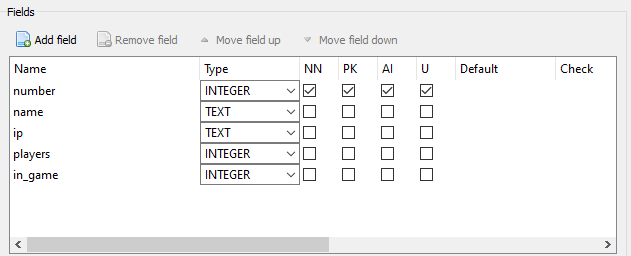
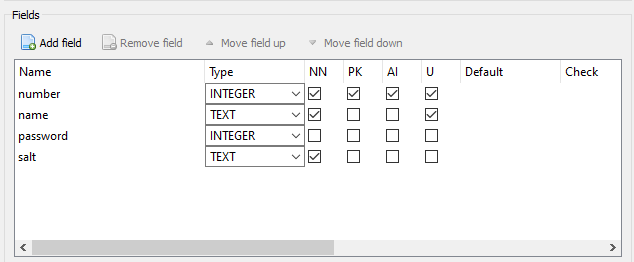
# development section

## server-side database creation

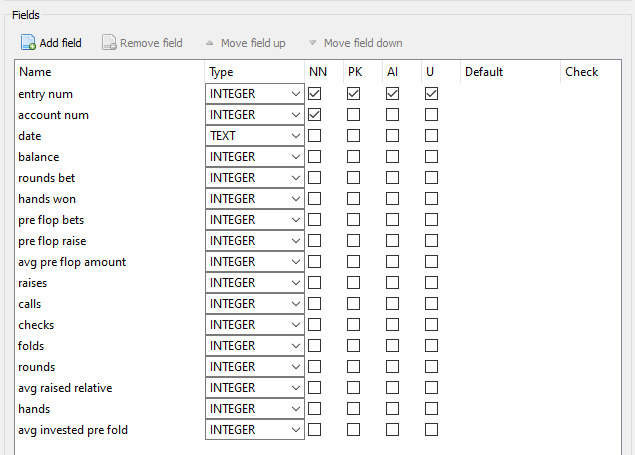
### Server list table creation and settings.



### Accounts table creation and settings.



### Statistics table creation and settings.

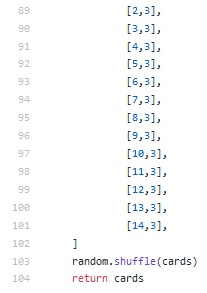
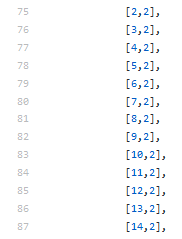
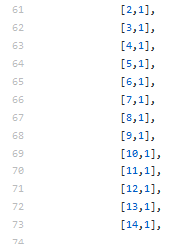
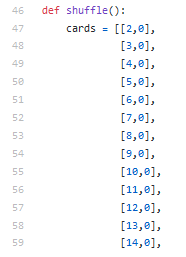


## The command line prototype

First, I made a prototype of the poker game which worked only on one device and was full command line to rapidly test and iterate on the mechanics to make sure it all works as intended.

## dealing the cards

### Creating the deck of cards, the shuffle function.



#### Since the design I made aces high as it makes it leads to fewer edge cases and exceptions

### The flop river and turn functions

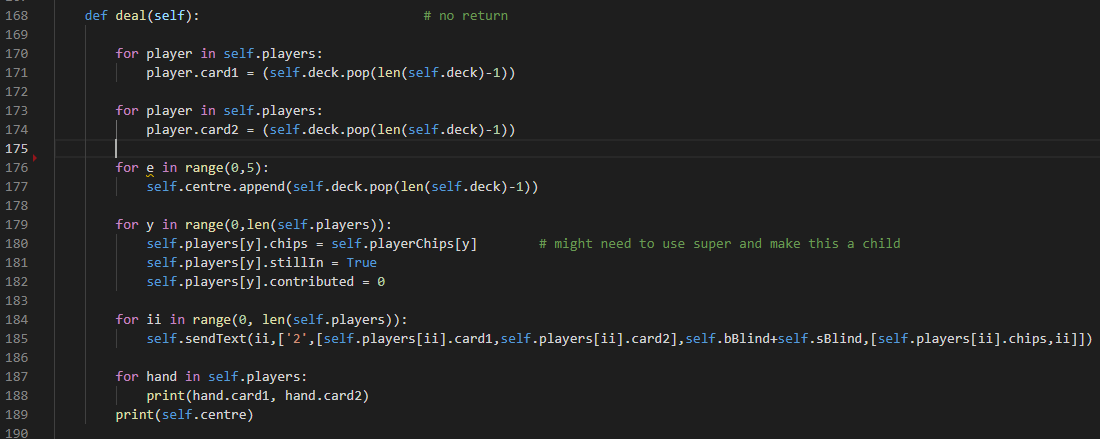
#### The code

#### Evaluation

This code selects the appropriate cards from the centre, combines them with the tag which the user client uses to establish which labels to change. The card is returned but this isn’t used in the final version of the application.

### The deal function.

#### The code



#### Evaluation

This function is essentially a constructor which populates the player objects in the players array. It also handles sending information to the players that they will need at the start of each hand.

## the hand checking functions

### The check straight function.

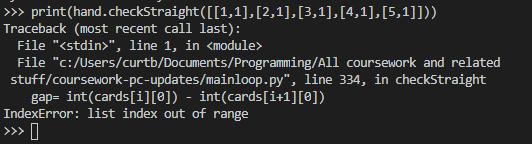
#### Code version 1

#### 

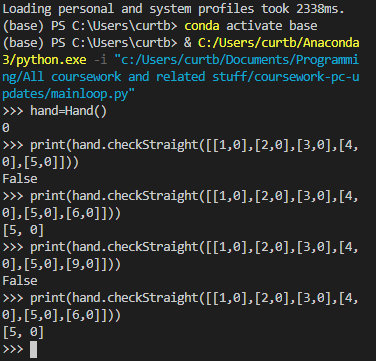
#### Test results

#### 

Dataset was not a straight so the function returns false.



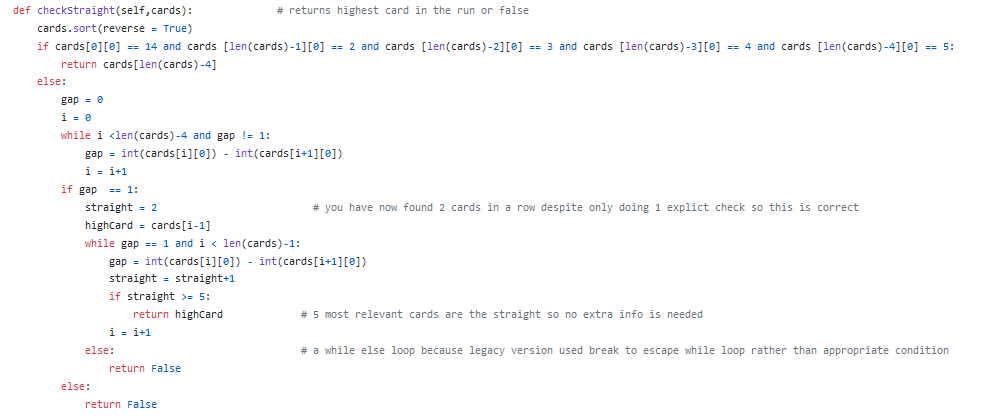
Dataset a straight but the function errored



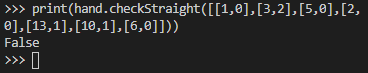
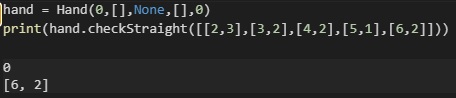
Even once the error was fixed it now needed straights to be runs of 6 rather than the intended 5

The code also has no handling for aces being high numbers which needed to be added after the change was made from the initial design.

#### Revised code



#### Re-testing new solution



returns the highest value from the straight as intended.

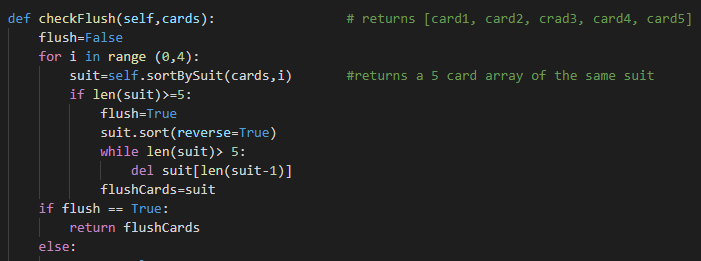
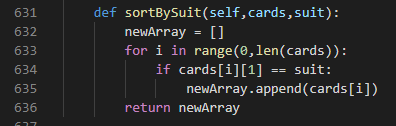
Dataset was not a straight so the function still returns false.

#### Evaluation

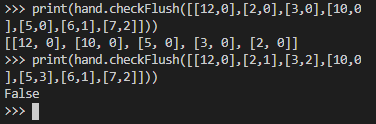
The code now works and returns the highest value from the straight which is what can be used to compare to other straights later in the development.

### The check flush function

#### Code



#### Tests

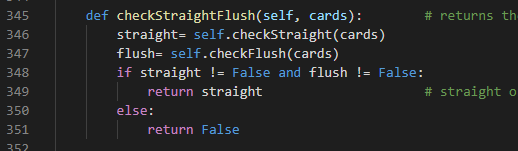


#### Evaluation

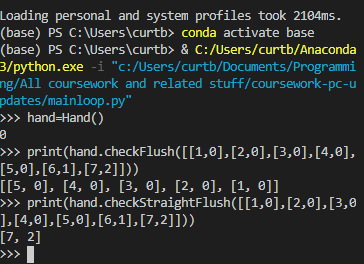
The code returns the full 5 cards if there is a flush which is important as comparing 2 flushes could go down to the lowest card if the four highest are identical. The code returned false on non-flush hands which is intended

### The check straight flush function

#### Code version 1

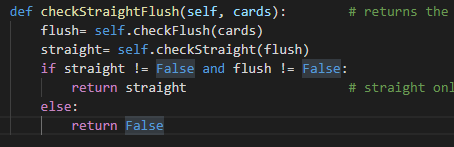


#### Test 1

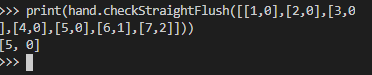


Here a flush and a straight made up of 2 different sets of 5 cards is registered as a straight flush which shouldn’t happen.

#### Code version 2



#### Test 2

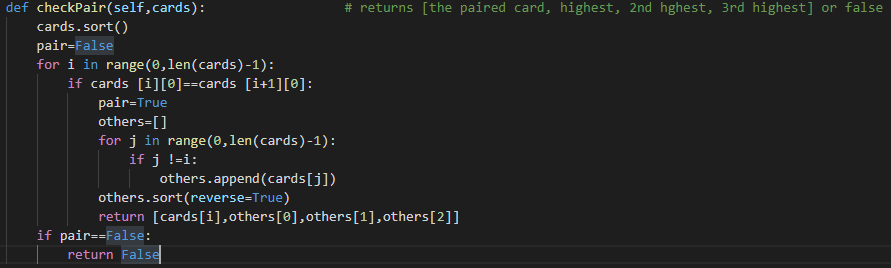


#### Evaluation

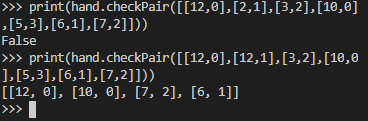
The code now performs as desired by only passing the 5 flush cards into the straight.

### Check pair function

#### Code



#### Testing

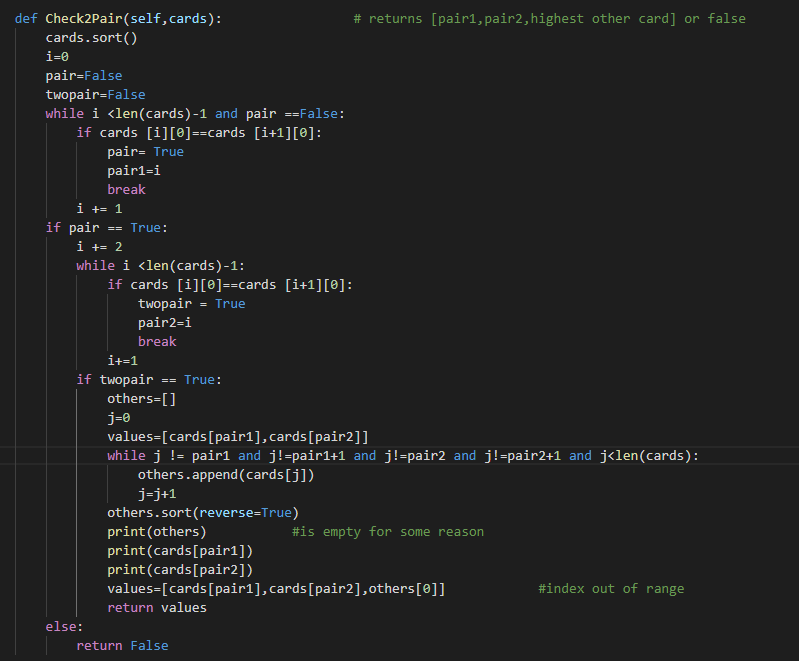


#### Evaluation

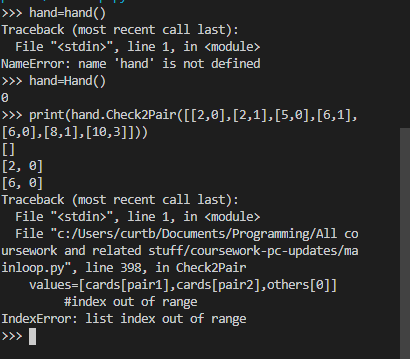
The code returns the 3 non paired cards as well as one of the pairs so if people have opposing pairs the comparison can go down to the third kicker if necessary.

### The check 2 pair function

#### Code version 1



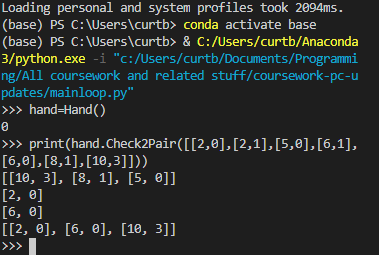
#### Test 1



#### Code version 2



#### Test 2

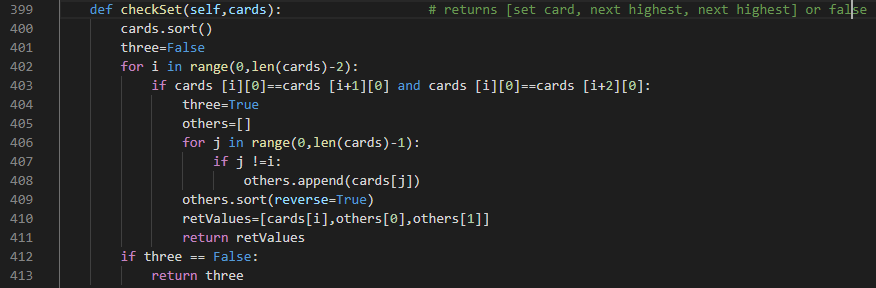


#### Evaluation

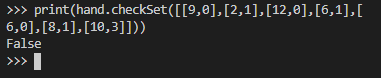
It returns the pairs and the kicker, using the while loop with poorly designed exit clauses lead to index out of range issues, using a for loop allowed for greater robustness and better exit clauses.

### Check Set function

#### Code



#### Tests

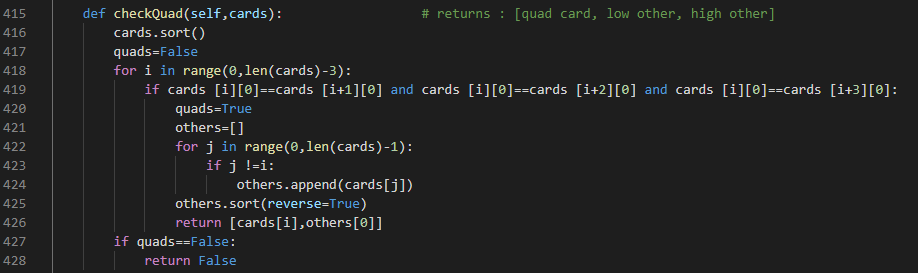


#### Evaluation

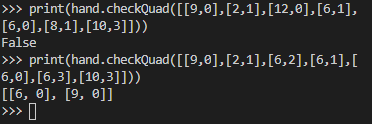
It performs as expected however with retrospect the kickers are.t necessary as even if 2 people have a set they can’t both have an equal set meaning final comparisons however returning these values won’t break anything or affect the end user in any meaningful way.

### The Check quad function

#### Code



#### Tests



#### Evaluation

It performs as expected however with retrospect the kickers are.t necessary as even if 2 people have four of a kind, they can’t both have an equal card meaning final comparisons however returning these values won’t break anything or affect the end user in any meaningful way.

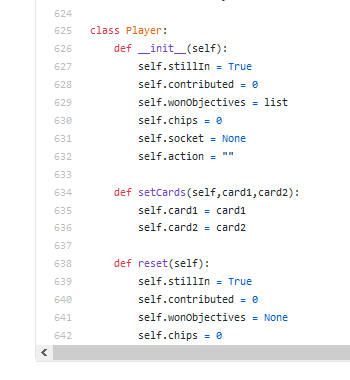
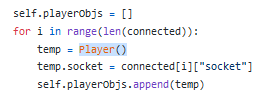
## Switching from dictionary to custom objects

Early in development I used a dictionary which was keyed using player numbers making it functionally no different to an array, each item then had an array which contained useful information, this array was ambiguous and hard to actually understand and should have been dictionary or a custom object. This was a change I then made replacing the original system with an array of player objects.

### Dictionary format



### Player class

## comparing the hands

### After creating all of the functions to compare the hands I then needed a function which could use the returned values and evaluate the winners of the hand, it needed to be able to deal with any number of players folded or not and give an output that would allow another function to then use it to allocate the chips to the players. The function comprises of 2 major loops, the first finds out what you have in your hand using the above functions and the second compares these and ranks the hands

### Part one, evaluating the hands

#### The code

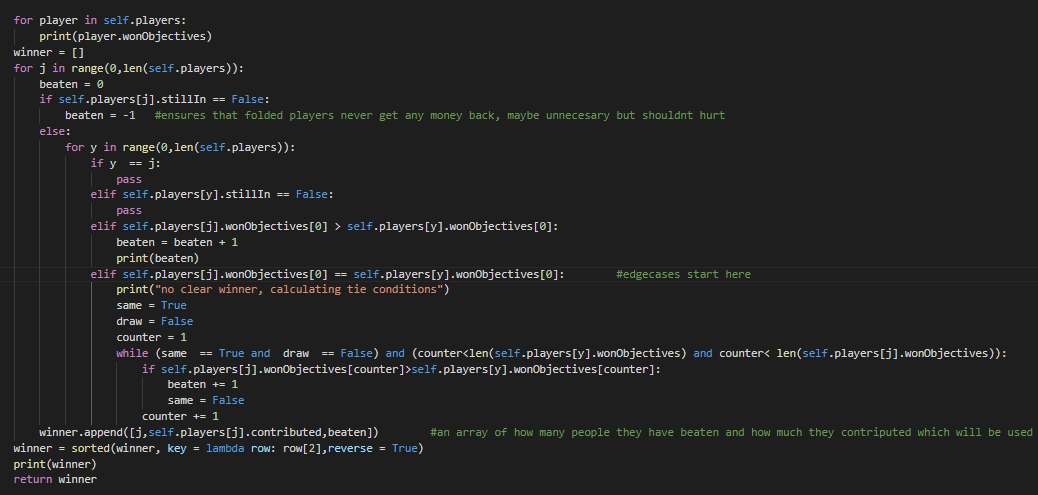


#### Evaluation

This segment calls the functions on the players cards + the centre cards in descending order meaning it will only recognise the most valuable attribute, this is then assigned to the eon objectives property of the player and used in the next loop.

### Comparing the hands

#### The code



#### Evaluation

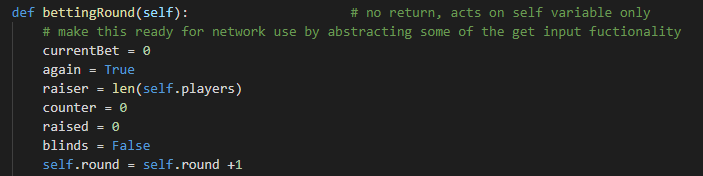
This section of the code comprises primarily of a nested for loop iterating over the players array and finding who has the highest number. If the numbers are the same it goes through the kicker cards and comparing it against each other. This array is then sorted into reverse order so that the allocate chips function can use it to allocate winnings accordingly, more information than just the winner is needed in case of split points were the winner in ineligible for a portion of the bid chips.

## Betting rounds

### Explanation

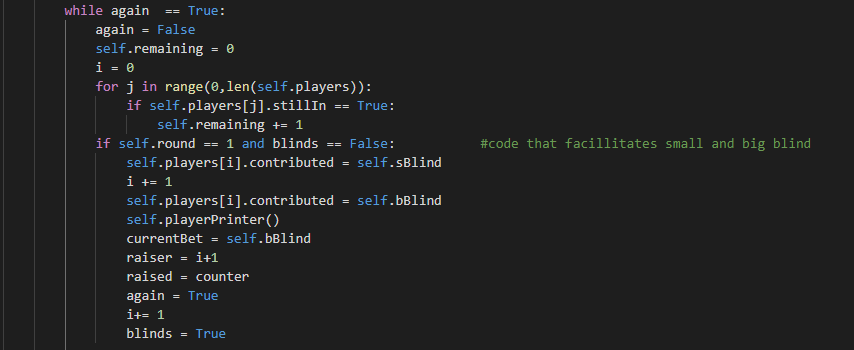
The betting round function processes all of the inputs needed for the handling of peoples actions during the game. It is the most complex function of the game loop itself and works on properties that have essentially been passed by reference rather than on local variable which it then returns. This allows for it to make changes to multiple properties and means any function it calls during execution have the most up to date values for those properties.

### The code – part 1



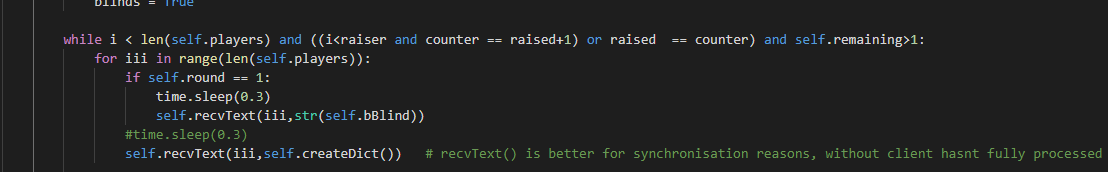
The first section of the function creates the local variables needed at gives them their starting values as well as incrementing which round the game is on by one.

### The code – part 2



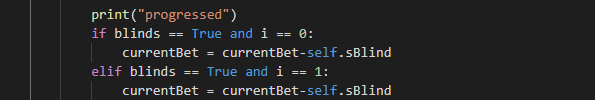
The rest of the function takes using a nested while loop, one may only be ran once but is used if someone raises and therefor you need to go round the table for more people to call. The part pictured here deals with the blinds and is therefor only ran if it is round one and the blinds have not yet been taken.

### The code – part 3



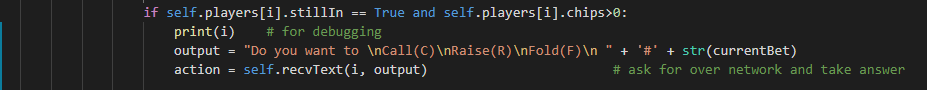
This is the start of the second while loop which loops until it has handled every player or not gotten round to whoever raised that round or until everyone but one person has folded. Here I had to use my send and receive function as I was getting issues with transmissions not being received by the client so requiring a confirmation helps, as does adding a brief sleep period otherwise the client is still processing previous requests before this one arrives. The full networking system will be addressed later.

### The code – part 4



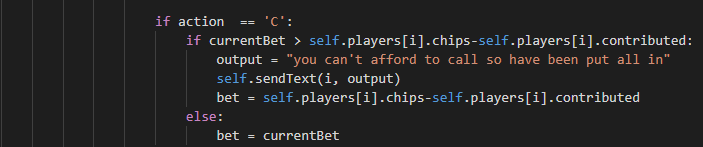
This section actually subtracts the earlier calculated blinds from the first and second player.

### The code – part 5



This is the main section of the function that begins by checking the current player is eligible to make an action by checking if they are folded and if they have any chips to bet. It then takes an input from the user, the actual test being sent in never seen in this version but was used both in the single player and multiplayer command line versions. This return value is then one of 3 things, C, F, or R standing for call, fold and raise each of these then have their own processing else/if statement.

### The code – part 6



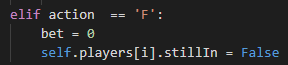
This section processes what happens if a user wants to check or call by adding the current bet to the players bet local variable which once the loop has completed will be added to the player’s contributed attribute. It also checks if the player can afford it and if not notifies them and puts them all in.

### The code – part 7



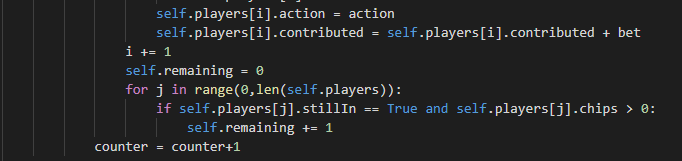
This part of the code handles the raising of bets, after receiving the decision to raise it then needs to ask by how much along with the tag 6 which tells the client that they need to open the raise ui. It also has several checks for if they raise by too much or can’t afford it etc. Many of these became irrelevant as the user interface doesn’t let you enter illegal values but keeping them for insurance in case the ui has an issue is a good idea.

### The code – part 8



This is a short part that handles folding, it has no checks as a player can fold at any time.

### The code – part 9



This is the final section of the loop which updates the attributes to reflect the changes of the local variables throughout the function. It also increments the local counters