Design Section

**Decomposition of the problem**

1. Login Authentication system
   1. Input for username and password
   2. Hashing and salting of inputs
   3. Checks against server database
   4. Progresses to main menu if correct
   5. Can create a new username and password
2. Account Creator
   1. Ability to add
      1. Username
      2. Password
      3. Password verification (enter it twice)
   2. Block out password so user can’t see it as they type
   3. Creates and stores new Hash and Salt
3. Main menu buttons
   1. View server list
      1. A full refreshable and current list of available games with their status
      2. A back button
      3. The ability to join a game which opens a sperate window
   2. Host a game
      1. A tutorial on port forwarding
      2. A lobby with the option to start/cancel the game at any point
      3. You can see when people join
   3. View statistics
4. Gameplay
   1. An exit button
   2. Visual indicator of what your cards are
   3. Visual indicator of everyone’s balance
   4. Full deck of cards created
      1. Fully random hand shuffling
      2. Each card can only be used once
   5. Prompts to act on your turn
      1. Either call raise or fold
         1. Call. Match the current pot or if its too expensive put you all in
         2. Raise. Allow the user to increase the bet by a certain amount
         3. Fold. Allow the user to exit the game
         4. Record individual players contributions separately
      2. Bets loop until everyone is paid up or folded
      3. A timer that when complete automatically folds inactive players
      4. Calculate the winner
         1. Check cards for hands such as straight, flush etc…
         2. Rank hands from winner to looser
         3. Remove folded hands
         4. Calculate pot allocations
            1. Account for split pots by using the individual contribution and not allowing them to collect more than each player multiplied by this value
   6. Can see other players actions in real time
5. Sql reader
   1. Necessary functions
      1. Append a new value/record to the database as a tuple to prevent sql injection
      2. Replace/overwrite entries when given the new value and the one to be replaced
      3. Read and return a set of records as a 2d array when given the sql code to be executed
      4. Read and write encrypted entries for password security using asymmetrical encryption where a single server has access to the private key

**Structure**

The Program is split into several modules to take advantage of Python’s powerful modular design. This helps me perform isolated tests of individual systems, makes working on the project easier a I don’t have to comb through an extremely large file to find what I’m working on. It also helps with the decomposition of the problem as I can run different aspects in different files. Here you can see the file/module structure of the end user’s application, the right side of the tree represents if the user is hosting the game and the left side represents when the user has joined a game hosted elsewhere. “Client\_A” and “client\_B” handle the majority of the networking and communication during the runtime of the game. “Mainloop” holds the classes used to run the game and all the code associated with it. “Mainprogramclient” holds the user interface and menu code that allow the player to select what they want to do.

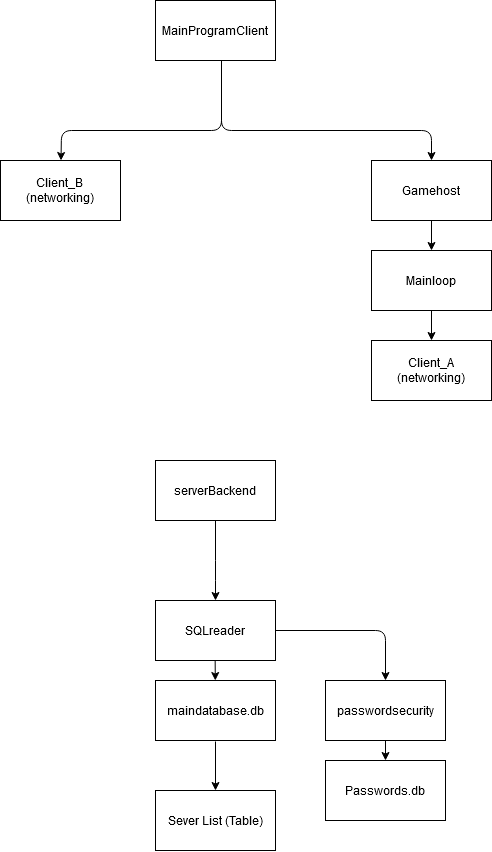


Figure - The file/module structure of the application

Below is the module/file structure for the server. This server is hosted on my network and has all the databases needed for the game as well as handling connecting players together. I used an “SQLreader” module I wrote however it was not as powerful as intended because you can’t insert a variable as the table name which meant the functions could not be as general purpose and reusable as I would have liked and be heavily specialised to its use case.

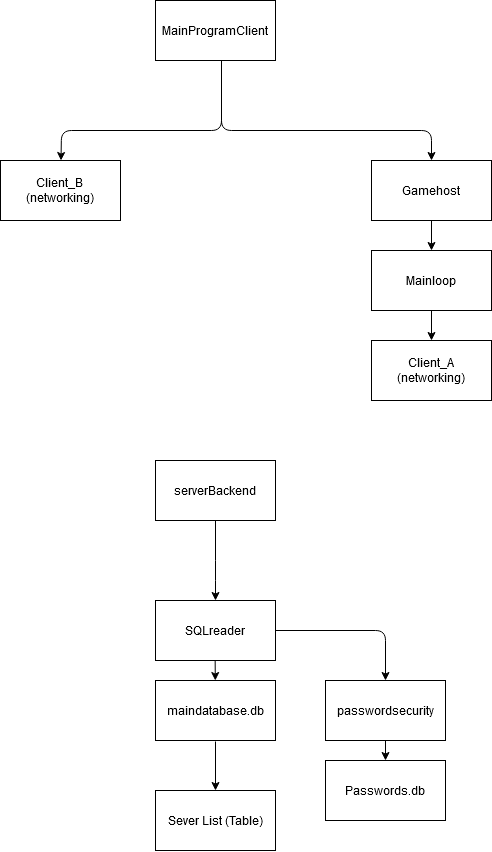


Figure - The File/module structure of the server

Figure - The error message given when attempting to use variables as table names

**Choosing my resources**

**Choosing an encryption library and algorithm**

* I initially planned to use the “pycrypto” library as it seemed simple and lightweight while still being powerful enough to protect my users. However after some research I discovered that this library is unmaintained since 2013 and is therefore not safe for use. This prompted a switch to “cryptography” a more complex encryption library.
* For the purpose of sending sensitive fata to and from a downloadable piece of software a symmetrical encryption algorithm would not be sufficient meaning I had to use an asymmetrical algorithm. I settled on rsa due to its faster validation times and similar security (when using a 2048 character modulus) to dsa while being very well documented.
* Cryptography also supports all major methods of hashing and salting which are necessary for password security so that I am in accordance of the data protection act

**Selecting a database manager**

The database manager used is extremely important as you will spend a lot of time working on it so it must be easy for you to develop on while also being powerful enough to perform the tasks required of it. I decided to compare “mySQL” to “SQLite”, two very popular, python compatible database managers.

**mySQL**

* Industry standard, very well documented and supported
* Allows multiple simultaneous edits which may be useful but due to my limited use of the threading module its use case is limited
* Cross language making it useful as a skill to learn and be familiar with
* More advanced and flexible
* Inbuilt server capabilities

**SQLite**

* I am already intimately familiar with the SQLite browser and python module
* SQLite is, as the name implies, very lightweight and simple which is very useful with the time limitations and scope of this project
* I have already written useful code I can simply reuse
* I have to hack a server together using subpar techniques

Due to the tight time constraints of this project I elected to use SQLite as despite being worse than mySQL I am already very familiar with it and can reuse code/functions from previous projects whereas mySQL has a steep learning curve and would require significant time investment to learn. Furthermore all the functionality of mySQL is possible by combining modules such as socket, pickle and SQLite.

**Selecting a data serialiser**

To send data over a socket in python it must be the “bytes” datatype, for strings this is achieved using the .encode() method however for more complex objects such as arrays and dictionaries a data serialised is needed. I compared the two most popular solutions, json and pickle.

**JSON**

* Human readable
* Cross language (don’t plan on using multiple anyway)
* Industry standard, better documented and supported
* Much safer, as I’m passing ip addresses etc may be an advantage
* Can’t deal with tuples etc and makes non string keys in dictionaries into strings, this causes issues with my mainloop.py using integer keys on a dictionary

**Pickle**

* Faster
* Can send classes
* Allows for data serialization of more data types and objects and better support for pythonic use cases due to being built from the ground up for python

Note: Could be faster using cpickle but this is not supported in python 3.x yet

Due to the better python support for pickle I elected to use it for my data serialiser

**Design Mock-ups**

**Explanation of algorithms**

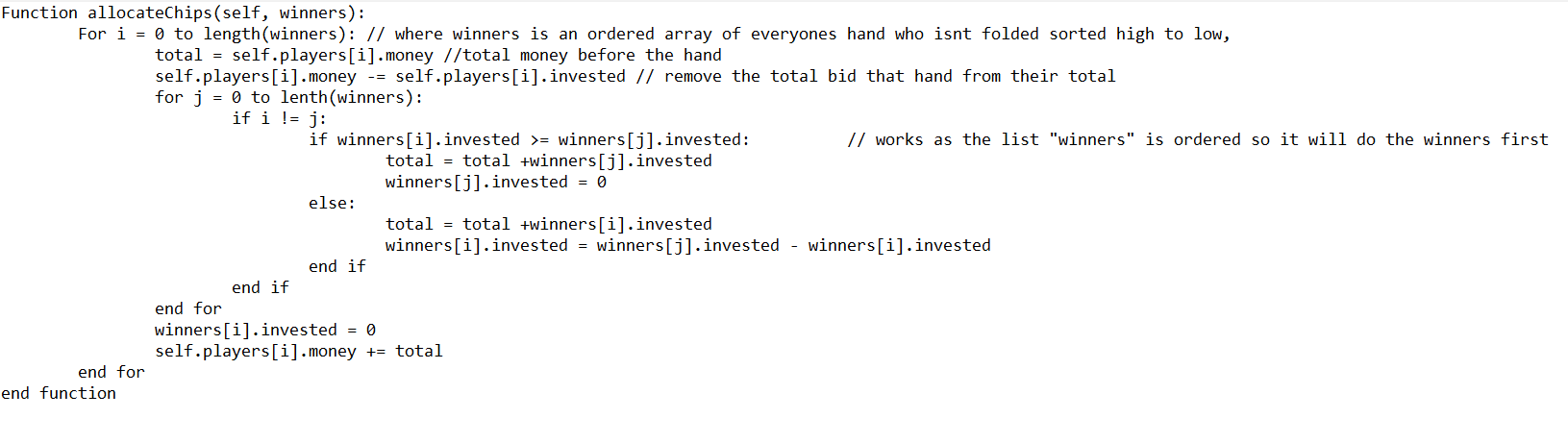
**Hand checking functions etc**

I can reuse several functions inside of other functions as things like a full house are made up of a set and a pair etc.

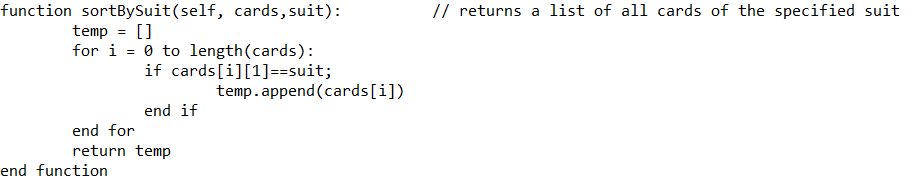
Cards are represented by 2 element arrays where the first element is a number representing the score of the card and the second is the suit (also an integer)

First all 7 available cards to that player are sorted by the first element, this means cards of the same value will be next to each other as will straights. This means loops going through this list can check if the next element has the same value or the value+1.

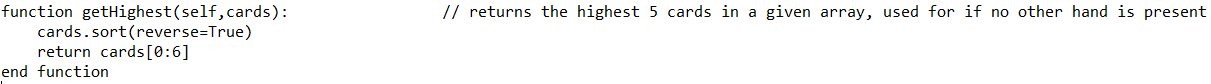
Pot allocation pseudo-code:



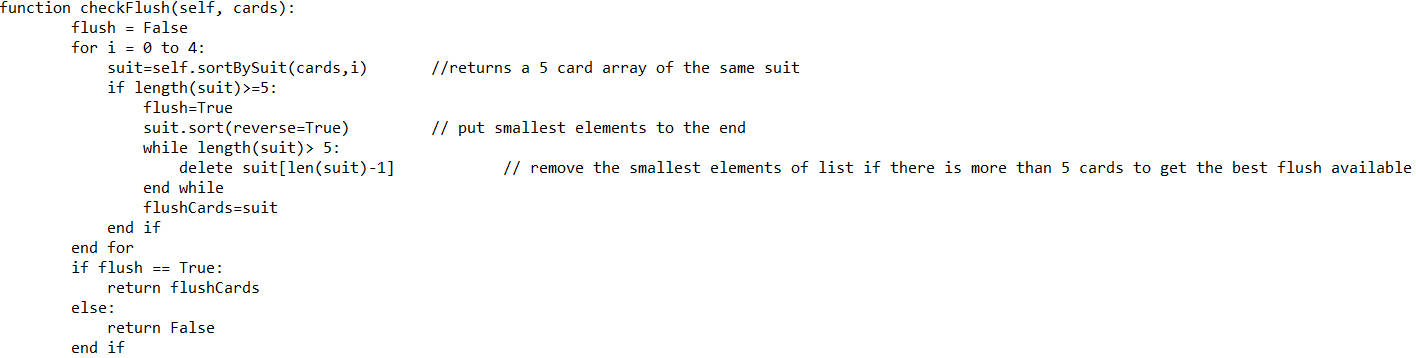
Suit filter function pseudo code:



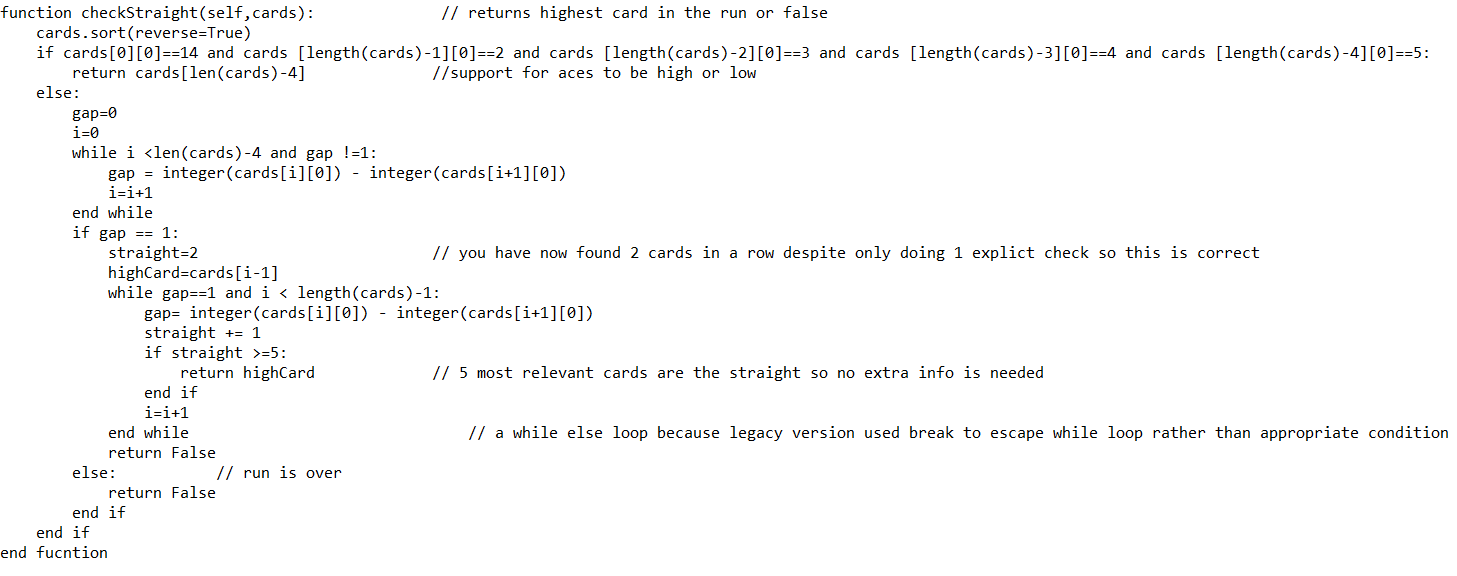
High card pseudo code:



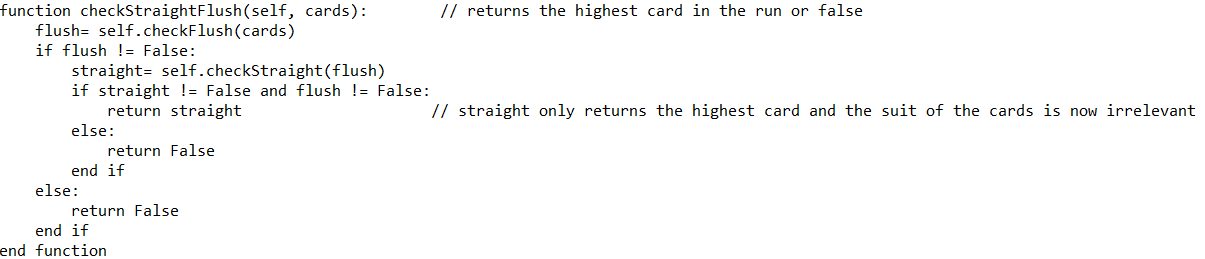
Check Flush pseudo code:



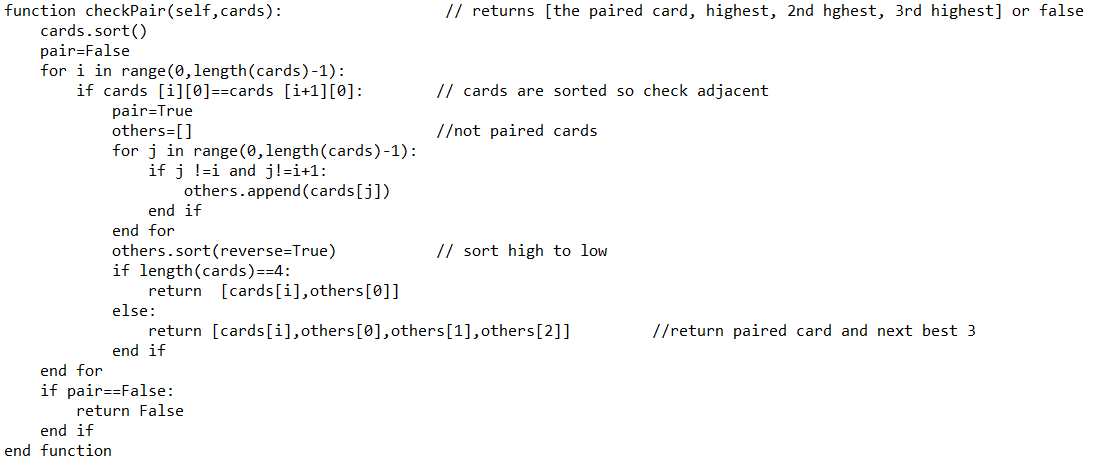
Check Straight pseudo code:



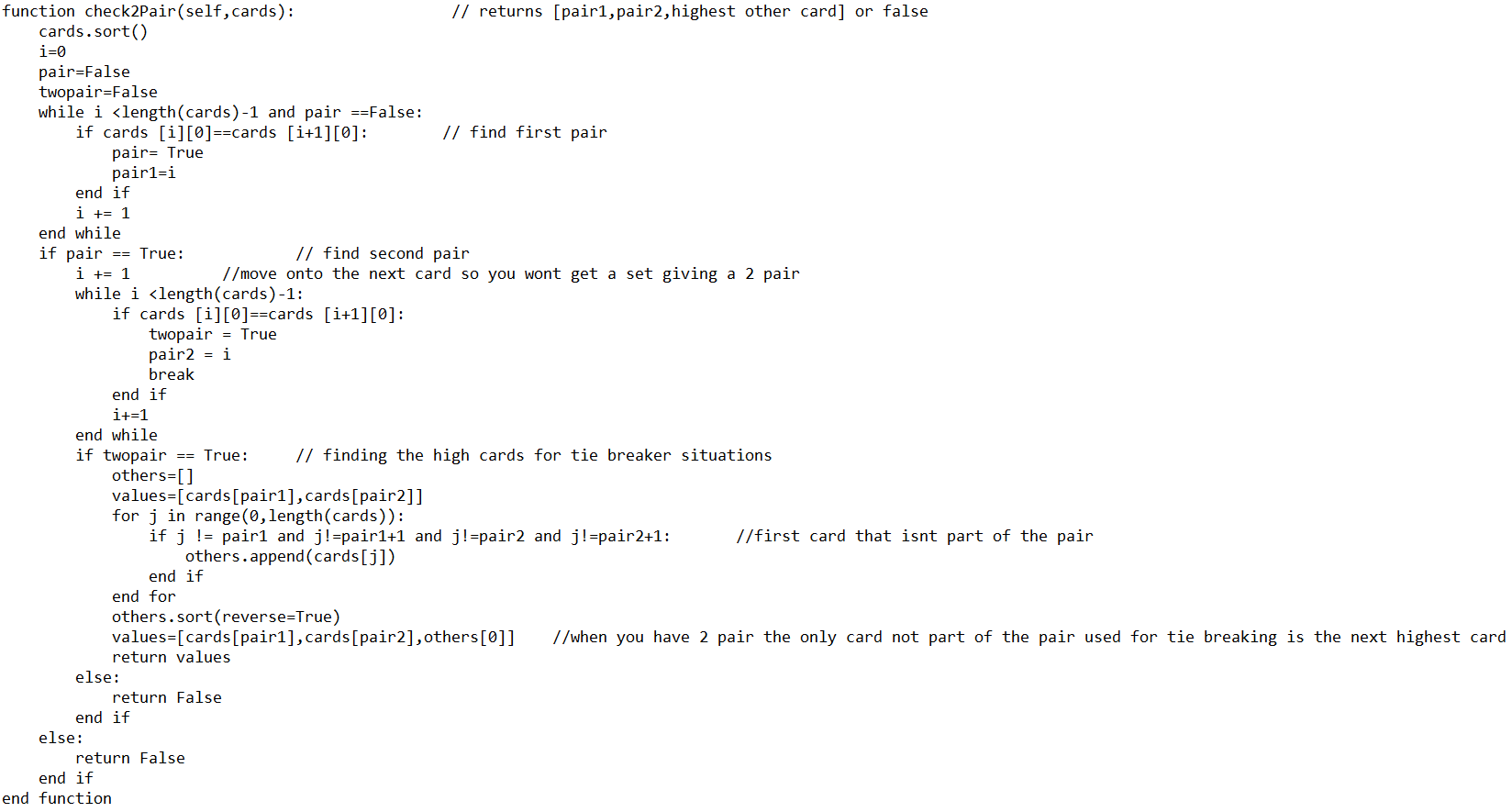
Check straight flush pseudo code:



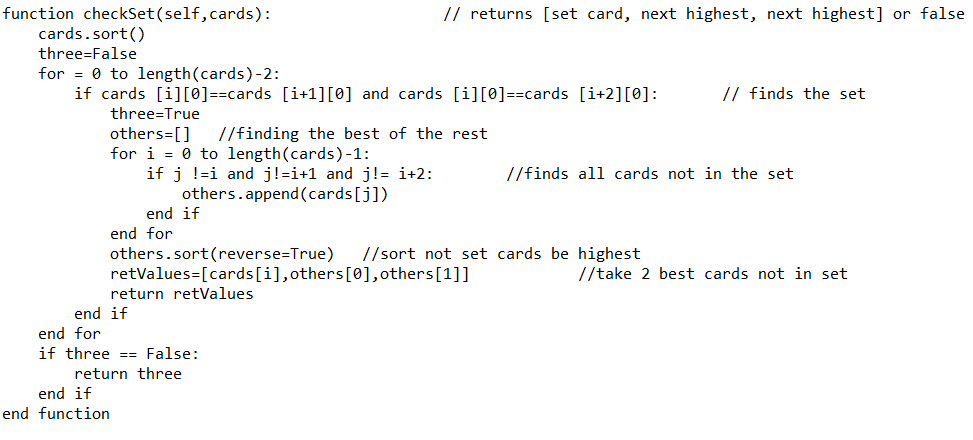
Check pair pseudo code:



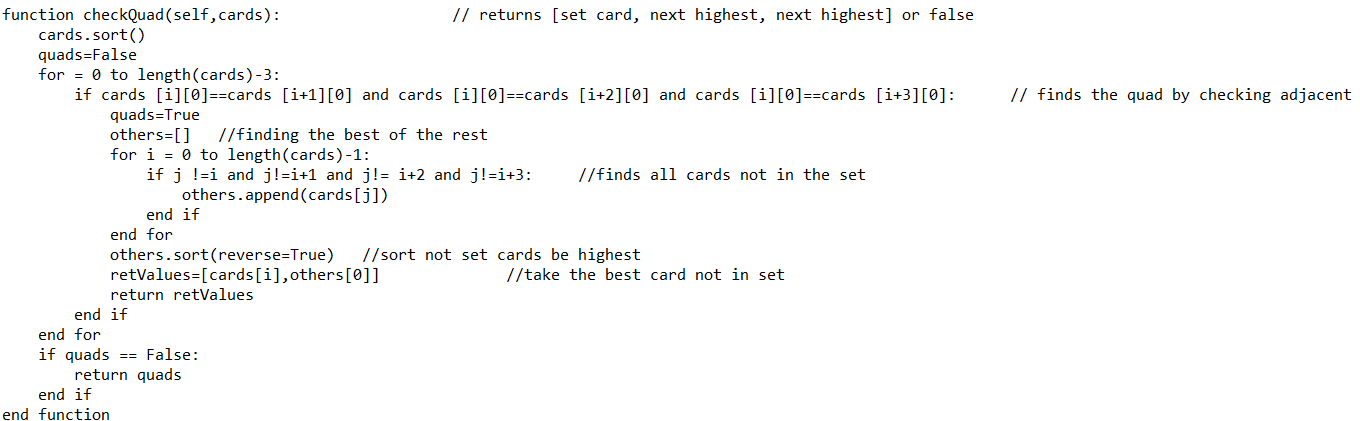
Check 2 pair pseudo code:



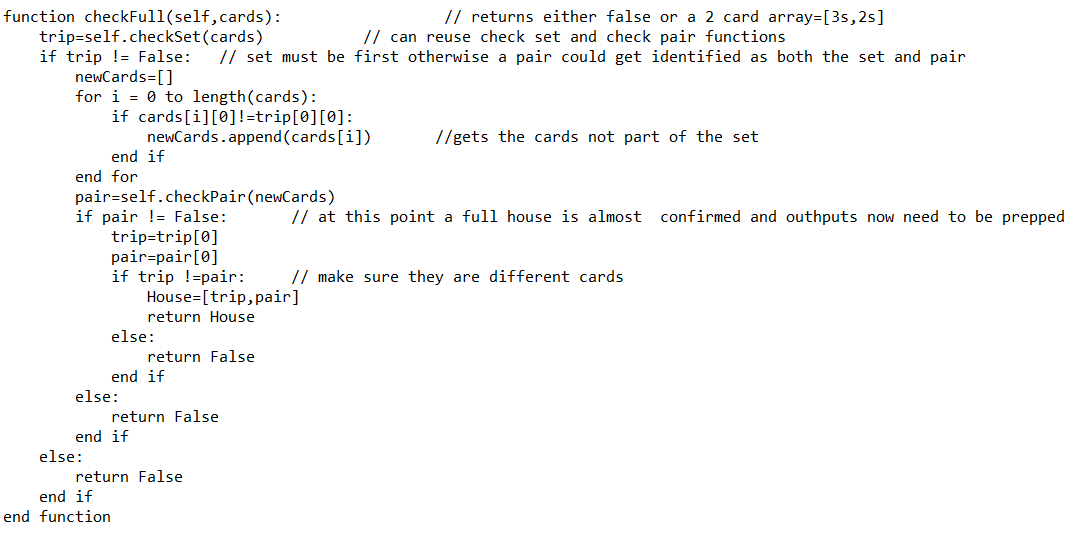
Check set pseudo code:



Check quad pseudo code



Check full house pseudo code



**Explanation of networking**

The processing for the actual game itself will almost all be server side processing. This is because with the random element of shuffling and dealing keeping everything synced up among multiple systems is inefficient. This host will then send via sockets information to each player and receive their response the same way. Due to this essentially being peer to peer where the host isn’t a dedicated server either I will need to use some form of udp/tcp hole punching, which is a method where you force a port open by exploiting outgoing packets to a known server, or I ask the host to port forward their router to their pc manually.

**Class diagram**

**Limitations**

**Test plan**

Testing hand detection functions:

These function always operate on a 2d array containing the cards, because of this other data types do not need testing, instead I must test if, when given a set of cards they correctly detect the hand they are looking for and do not give false positives.

|  |  |  |  |
| --- | --- | --- | --- |
| Condition | Data 1 | Data 2 | Data 3 |
| Pair | Cards with a pair present | Cards with no pair present | Not needed |
| 2 pair | Cards with 2 pairs | Cards with a pair | Cards without a pair |
| 3 of a kind (set) | Cards with a set | Cards without a set |  |
| 4 of a kind | Cards with a 4 of a kind | Cards without a 4 of a kind |  |
| Check straight | Cards with a straight | Cards without a straight | Cards with a run of 6(to ensure it selects the best 5) |
| Check straight flush | Cards with a straight flush | Cards with just a straight | Cards with just a flush |
| Check flush | Cards with a flush | Cards without a flush | Cards with a flush of 6(to ensure it selects the best 5) |
| Check full house | With a full house | Without a full house | Just a pair/set |

For a lot of these I can save testing time by having the functions be run in the most efficient order and stop when a condition is met. This means, for example, I do not need to make sure a pair isn’t actually a set as a set would already have been detected and then pair wouldn’t need to be ran.

**Solution description**