10.014: Computational Thinking for Design
1D Project
"Stress Bridge"

Name	ID
Jon Koo Jia Jun	1006388
Kwok Keith	1006344
Fano Lim Cheng Yi	1006373
Tan Jun Onn	1006234
Ek Hong Sheng Gerald	1006318
Li Junyi	1006081

Table Of Contents

DESCRIPTION……2
CODE DOCUMENTATION……..……....……4
1) *Socket*
……..……4
2) *Serializer*
……15
3) *Data base*
……16
4) *Server logic*
…….…….....……21
5) *GUI*
……...…24
REFERENCE…….……28

Description

We came up with the idea of the game “Stress Bridge” because it was how we felt when making this game especially in the GUI process. By coming up with our own variant of bridge we decided to materialise our very own version of the game with a similar but not the same set of rules as the game “draw-bridge”. Similarly, we included the ability for the game to run as a two player turn based game. However, the main difference of our game would be that cards will be drawn out in random instead of having the players take turn to draw the first card or the second one in order to build their deck.

The purpose of the game is to train up the cognitive ability of the users who are going to be elderly players as well as young children through decision theory coupled with fast paced game theory. With the users' minds being “stressed” during high decision-making processes, we hope to train up the decision-making skills of a whole generation to reduce the likelihood of dementia. (Centre for Healthy Brain Ageing (CHeBA) UNSW, 2016) By improving mental decision making and training it through our game, users will also keep their mind healthy and active while improving emotional age if one were to play from a young age. Such game can help with the mind upkeep and activeness especially for those who have time to spare; most likely, elderly people who have retired. Dementia has seen to be the leading neurological disease amongst the elderly community since the turn of the second millennium. It is more common as one grows older but abnormal to aging as many live up to their 90s without any signs of dementia and or reduce in cognitive ability. (US Department of Health & Human Services, National Institute of Aging, NIH, 2021) Playing simple card games such as Stress Bridge can help in engaging logic processing and sorting ability with its mentally stimulating aspects. (Hospital News Canada, 2018)

Such game can also educate toddlers and children methods of counting and mathematics by monopolising a fun and adrenaline driven narrative in a game. By playing this game, toddlers will begin to imprint mathematics and the difference between numbers into their minds at a young age. As well as identifying the difference between images in a deck of cards with a short span of time when as they will be pressured to make decisions hastily. Therefore, cognitive ability of children will improve by playing this game as it is crucial for them to their education at an early age by memorising simple images and numbers in a deck of cards. The presence of 4 different suits will also introduce different dimensions and levels of the classes of numbers and picture cards. Lastly, Stress Bridge also helps develop emotional intellect through key emotional abilities such as learning patience and the importance in handling losses which are some key areas that must be taught to children. (Roth, 2020) Users of all ages will also be able to use our program whenever they wish to burn off some time as the game is easy to understand and each round of the game takes very little time, allowing users to freely start and stop whenever they want.

The game will first start off by randomly shuffling out 2 hands of 13 cards from a single deck for 2 players. The second player to draw will be the first to start as an advantage. He/she will first deal out any card of their choosing from their hand where the other user must follow with the suit that was played initially by the previous user. The highest value card played will score a point to the user which wins them the round, with 2 being the smallest and Ace being the largest possible card in a suit. One of the key rules is that the player must play the suit card should he have one, regardless of if he will lose that round. Only if the user doesn't not have any cards of that suit is he/she allowed to play any other card; but will still lose that round. Winner gets to start a new round by choosing any card from his hand to play. Thus, the first to win 7 rounds wins the game. However, it is ultimately up to the two players to decide who to start dealing the card to start the round because we like to offer more flexibility to the players and at the same time, allow the younger users learn the importance of integrity and understand fair play. Said set of features will also act as a “friendship test” which will allow users of all ages to identify the true personalities of their friends and even family members. The game would have a persistent saving feature which leverages on a database, (DB) to actively update the total rounds won by respective users during the game as to also keep track for when someone wins the game. The DB also allows users to log their number of wins so that when they log

in again, they can share their records with others which encourages users to play more and to compete with others more, should they have set on a goal to achieve in number of points. Additionally, it stores the encrypted password from server into DB so that the next time the same user enters with their same name and password, they will be able to access their record; else a new account will be created when there is no account associated to the on keyed in on the DB. Thus, they need to remember their account, and this will allow users to feel that they have more ownership in their account and record as we will not be implementing a hint feature for which users forget their passwords and usernames. For the database, we will not be able to steal their passwords and sensitive information as it is even hidden from our eyes because it will be encrypted in server before storing into DB, thus assuring the users that they can confidently set a comfortable password that they can remember instead of a generic one, which also solves the possible problem of them repeating the creation of too many accounts when they forget their password and even possibly lose interest in the game.

Code Documentation (Python 3.8+)

1) Socket (Send messages across network)

Using the socket package available in python's native library, we will use it to communicate between the client and the server.

Below we shall explain the Server side communication…

We will import these libraries:

```
import socket
import threading
import database as db
from server_logic import *
from serializer import serialize, deserialize
```

We then proceed to define these **constants:**

```
HEADER = 64
PORT = 5055
SERVER = "192.168.0.12"  # IPV4 of local server
ADDR = (SERVER, PORT)  # Address to be bound
FORMAT = 'utf-8'
DISCONNECT_MESSAGE = "!DISCONNECT"
KICK_MESSAGE = "!KICK"
VERIFY_MESSAGE = "!VERIFY"
WAIT_MESSAGE = "!WAIT"
START_MESSAGE = "!START"
START_STATE = False
```

Variables	***Function***
HEADER	`int` to indicate the number of binaries needed to be filled when server/clients sends a message
PORT	Server Port
SERVER	Using the IPV4, this defines the server's IP address (based on the device you use to start the server)
ADDR	Creates a tuple using the format (SERVER, PORT)
FORMAT	Character encoding used, 'utf-8'
DISCONNECT_MESSAGE	`str` message received from client to run disconnect sequence for that client
WAIT_MESSAGE	`str` message received from client to get the server to wait for client message
START_MESSAGE	`str` message to send to client to indicate game start

KICK_MESSAGE	`str` message to send to client to indicate wrong password
VERIFY_MESSAGE	`str` message to send to client to indicate correct password
START_STATE	`bool` To indicate if the game has started.

Creating a **socket connection:**

```
# Stream data through socket (Using IPV4)
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(ADDR)  # Bind socket to address

# To keep track of which players are connected
# (WARNING: USING ADDRESS OF USER FOR NOW!)
players_ls = []
```

Using socket, we start by defining a socket object.

- server: `socket object` defined as a socket.AF_INET, and the type socket.SOCK_STREAM.

We then bind this socket object to the address declared above. This allows us to create a network communication path for the server.

```
db_players = db.fetch("players")
```

A `list` of `dictionary` is fetched from the database *db*, which contains the verified/registered users and this is passed to the *db_players* variable.

<Function> start(num_players):
This function takes in the 1 argument, *number of players (num_players)*, and start the game server. When this function is ran, the server listens for connections, and threads ***handle_client(conn, addr)*** when there is a connection request. The function will iterate through a *while loop* to get the server to listen in for connection request using the blocking code, *server.accept( )*.

```
server.listen()  # Get server to listen for connections
print(f"[LISTENING] Server is listening on {SERVER}")

while True:
    # Get server to wait for connection, take connection info if present
    conn, addr = server.accept()
```

When a connection comes in, we take in the connection, and address of the user. We then start the thread of the user, using:

```
thread = threading.Thread(target=handle_client, args=(conn, addr))
thread.start()
```

The server will repeat this listening for connection and starting of thread in a while loop()
Again, above we thread the **handle_client(conn, addr)** function which we shall discuss next.

<Function> handle_client(conn, addr):
This function takes in 2 arguments, *IPV4 connection (conn) & address of user (addr)*. This function aims to handle client threading for server functions, primarily to listen for client message.

The function starts off by setting START_STATE to false. This indicates that the game has not started. Once we run the *start(num_players)* function, the server waits for users to connect. When a user connects, we send the IPV4 and address of the user to *handle_client(conn, addr)* function.

```
START_STATE = False
print(f"[NEW CONNECTION] {addr} connected...")

connected = True
```

We proceed to declare a new variable of type `bool` (*connected*). This variable is set to *True* to indicate that the user has connected to the server.

The function then loops through a *while loop* that uses the *connected* variable as a flag. We shall then explain the following while loop and its function:

```
while connected:

    # How many BYTES to receive from the client
    # Wait till receive message from client before continuing

    # Get the length of message
    # (i.e. "124....." -> indicates 124 bytes for message)
    msg_length = conn.recv(HEADER).decode(FORMAT)
```

The function purpose is to handle any messages from the client. Based on the messages received from the client, it would then decide how the game should proceed.

Socket requires us to indicate the amount of characters to receive from the message sent by the user. To get the length of bytes, the message is sent in the following format,

"<message length>…" number of bytes to receive initially is based on HEADER value.

The HEADER value, *64*, contains the `int` number of the bytes we want to initially receive that would hold the value of the length of the actual message. Hence, if the initial string was "124…" <… indicates the appended blank spaces to reach the HEADER value, *64 bytes*>.

The server then receives the actual message from the client, by receiving the number of bytes based on the value obtained in the HEADER of the string.

Since for the first connection, the user would send a string of type *None*. We want to check if the msg_length is *None*. If it is the first connection, we do not need this message sent by the client, hence the server would do nothing.

```
if msg_length:  # Check if message is NONE (First Connection)
{…}
```

```
else:
     pass
```

The following code illustrates the above explanation on how the code to receive client message works as explained above:

```
if msg_length:  # Check if message is NONE (First Connection)
    msg_length = int(msg_length)
    msg = conn.recv(msg_length).decode(FORMAT)  # Get actual message
```

Using this statement, this loop only runs when the game has yet to start:

```
if not START_STATE:
    reply = "."

    if msg == WAIT_MESSAGE:
        if len(players_ls) == num_players:
            START_STATE = True
            reply = START_MESSAGE
        else:
            reply = "Waiting for others to join..."

    # For disconnecting client
    elif msg == DISCONNECT_MESSAGE:
        connected = False

    else:
        players_ls.append(msg)
        game.addPlayer(Player(msg, len(players_ls)-1,0))
```

The loop would then check the message content sent form the client.

If the server receives *WAIT_MESSAGE*, the server would check if the number of connections equals the number of players defined in the *start(num_players)* function.

If the number of connections equals to this number, the game would start.
To start the game, we would change the *START_STATE* variable to *True*. The server also informs the user by sending a message, *START_MESSAGE*.

If the number of connections does not equal to the number of players defined, the game would not start, and the server would wait for more connections.

If the server received *DISCONNECT_MESSAGE*, the server would set the *connected* variable to *False*. Recall that the while loop runs with the flag *connected*. Henceforth, the while loop would break once it has completed this iteration of the loop.

```
else:
    data = deserialize(msg)
    name = data["pname"]
    hashed_pw = data["password"]
```

```
    verifier = {}
    for row in db_players:
        k,v = row["pname"], row["password"]
        verifier[k] = v
    if name in verifier.keys():
        if hashed_pw == verifier[name]:
            print(name,"verified")
            reply = VERIFY_MESSAGE
            players_ls.append(name)
        else:
            reply = KICK_MESSAGE
```

Any other messages not handled above, would be the login for the user. This message is deserialise into a `dictionary` that stores the username *(name)* and password *(hashed_pw)* for the user trying to login.

We declare a new `dictionary` *verifier* that is used to store the username and password in key, value pairs of the registered users. This is done using a for-loop, of the `list` *db_players* that contains the verified players.

We check if the user's name exists inside the *verifier* keys. If it exists, we will proceed to check their hashed passwords *(hashed_pw)*. If the *hashed_pw* is equals to the hashed password stored in the verified database, the server will send the *VERIFY_MESSAGE* to the client. If the *hashed_pw* does not equal the hashed password stored in the verified database, the server will send the *KICK_MESSAGE* to the client, which will require the client to relogin.

```
    else:
        print(name,hashed_pw,"is new")
        reply = VERIFY_MESSAGE
        db.insert("players",pname=name,password=hashed_pw,points=0)
        players_ls.append(name)
```

If the player is not found in the *verifier* keys. The server will create a new user in the database. This is done by using:

```
db.insert("players",pname=name,password=hashed_pw,points=0)
```

We will then add this new user to the verified player `list` *players_ls*. We will then add the player into the *game* `object` by constructing a new *player* `object`.

```
players_ls.append(name)
game.addPlayer(Player(name,len(players_ls)-1,0))
```

The server then ends this loop by returning a message to the client via the function *send (conn, reply)*, which we shall introduce later in this documentation.

```
print(players_ls)

print(f"[{addr}] {msg}")

# To get server to send message
send(conn, reply)
```

The following loop is executed when *START_STATE* is *True*. This indicates that the game has started or is starting.

```
else:
        print(msg)
        data = json.loads(msg)
        pname = data["name"]
        extra = data["extra"]
        …
```

When the game starts, the message (*msg*) received by the user is in the form of a dictionary containing the details and actions of the player. Below is the format of this dictionary:

data = { "name": < `str` player's name >, "extra": < `dictionary` extra > }

The name of the player is extracted from the data *`dict`* and passed to the *pname* variable. The extra data (contains the game actions data) is extracted from the data *`dict`* and passed to the *extra* variable.
The code then iterates through each player in the game using the for loop:

```
for player in game.players:
    if player.name == pname:
        if extra["action"] == "query":
            send(conn,player.toJSON())
        elif extra["action"] == "draw":
            game.draw(pname,extra["amt"]) \
                if "amt" in extra.keys() else game.draw(pname)
            send(conn,player.toJSON())
        elif extra["action"] == "discard":
            game.discard(pname,extra["index"])
            send(conn,game.toJSON())
        else:
            send(conn,game.toJSON())
```

We would then check what the client is able to do based on the *extra `dict` ("action" key).* If this key is *"query"*, the player object would be sent to the client.
If this key is "*draw"*, the player object would draw the number of cards based on extra["action"]. It would then send the player object to the client.
If this key is *"discard"* it would discard the card based on the index of the card on the hand.

```
conn.close()
```

The socket connection is disconnected using the *conn.close()* function in the socket package. This only executes when the *connected* variable is *False* which indicates that the user has sent a *DISCONNECT_MESSAGE*.

<Function> send(conn, msg):
This function takes in 2 arguments, *IPV4 connection (conn) & message (msg)*. This function is used to send a message to clients from the server.

```
def send(conn, msg):
    """
    For server to send message to clients.

    :param conn: Connection IPV4
    :param msg: message to send to clients
    :return: NONE
    """

    message = msg.encode(FORMAT)
    msg_length = len(message)  # Get length of message (in bytes)
    send_length = str(msg_length).encode(FORMAT)  # Encode Message Length
    send_length += b' ' * (HEADER - len(send_length))
    conn.send(send_length)
    conn.send(message)
```

To send the message from the server to client and vice versa (the send method is similar in the client side), we have to first encode the message in utf-8. We then grab the length of this encoded message, which would give us the number of bytes we need to receive on the receiving side (client). This length is then encoded to utf-8 and passed to a *send_length* variable. Again, we will use the *HEADER* constant to identify the number of bytes used to store the *send_length* variable. To attain this length of bytes, blank spaces of byte 1 are appended to the end of the *send_length* value. The connection then sends the *send_length* and the *message* over to the receiving user (client).

Below we shall explain the Client-side communication…

We will import these libraries:

```
import socket
import gui
import json
import hashlib
from serializer import serialize, deserialize
from time import sleep
```

These are the **constant variables:**

```
HEADER = 64
PORT = 5055
FORMAT = 'utf-8'
DISCONNECT_MESSAGE = "!DISCONNECT"
WAIT_MESSAGE = "!WAIT"
START_MESSAGE = "!START"
SERVER = "132.147.94.36"
ADDR = (SERVER, PORT)
name = ""
extra = {}

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(ADDR)
```

Variables	***Function***
HEADER	`int` to indicate the number of binaries needed to be filled when server/clients sends a message

PORT	Server Port
SERVER	Using the IPV4, this defines the server's IP address (based on the device you use to start the server)
ADDR	Creates a tuple using the format (SERVER, PORT)
FORMAT	Character encoding used, 'utf-8'
DISCONNECT_MESSAGE	`str` message to send to server to run disconnect sequence for the current client
WAIT_MESSAGE	`str` message to send to server to get the server to wait for current client message
START_MESSAGE	`str` message received from client to indicate game start
KICK_MESSAGE	`str` message to receive by the client to indicate wrong password
VERIFY_MESSAGE	`str` message to receive by client to indicate correct password
extra	`dict` containing the key value pair of game details for that particular user {"action": `str`<action>, "amt": `int`}
data	`dict` containing the player details, user name and the extra game details {"name": `str`<user_name>, "extra": `dict`<extra>}

We proceed to create the socket for the client side using the address *(ADDR)*:

```
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(ADDR)
```

The client only has one function that sends a message to the server side.

<Function> send(conn, msg):

This function handles the sending of message from client to server and takes in 2 arguments, *IPV4 connection (conn) & message (msg)*.

```
def send(msg):
    """ For client to send message to server """
    message = msg.encode(FORMAT)
    msg_length = len(message)  # Get length of message (in bytes)
    send_length = str(msg_length).encode(FORMAT)  # Encode Message Length
    send_length += b' ' * (HEADER - len(send_length))
```

```
    client.send(send_length)
    client.send(message)

    msg_length = client.recv(HEADER).decode(FORMAT)
    if msg_length:
        msg_length = int(msg_length)
        msg = client.recv(msg_length).decode(FORMAT)
        print(f"{msg}")
    return msg
```

Like the server side, we will encode the message with the length of the message to be send in bytes and then the actual message to be sent. The function also handles the receiving of the message from the server and decodes for the user.

```
login = False
while not login:
    name = input("What is your name? ")
    password = input("Password: ").encode('UTF-8')
    hashed_pw = hashlib.sha256(password).hexdigest()
    data = {"pname": name, "password": hashed_pw}
    reply = send(serialize(data))
    if reply == KICK_MESSAGE:
        print("Wrong Password!")
    elif reply == VERIFY_MESSAGE:
        print("Successfully logged in as",name)
        login = True
    else:
        print("Unknown Error")
```

To create a password system to our game to only allow verified users into the network, the above block of code is executed.

A `bool` *login* is set to *False*, the while-loop iterates the login sequence by using the *login* as the conditional flag to exit the loop. This means that the loop will continue to loop until the username and password is verified as queried to the server.

The user is prompted to input the username and the password, and this is passed to the `string` *name* and `string` *password* respectively. The *password* is encoded using UTF-8 and is then hashed using *sha256* from the python standard *hashlib*, this is passed to the *hashed_pw* variable.

A `dictionary` *data* is constructed with the *name* and *hashed_pw* to be passed to the server. The server's response would then be stored in *reply*.

We then check if the server replies with either *KICK_MESSAGE* or *VERIFY_MESSAGE*. If *KICK_MESSAGE*, the while loop is iterated again. If *VERIFY_MESSAGE*, the *login* variable would be set the *True* and the login while loop would terminate.

```
reply = ""
while not reply == START_MESSAGE:
    reply = send(WAIT_MESSAGE)
    sleep(1)
```

Here we set a string variable *reply*, with an empty `string`. Iterating through the while loop with *reply* not equals to the *START_MESSAGE* as the flag, we will then send the WAIT_MESSAGE to

the server using the *send(msg)* function. The server will respond with a message to the client side and this message would be stored to the *reply* variable. If this *reply* variable is equals to the *START_MESSAGE*, then we will exit this loop and start the game on the client side. If not, the client will keep sending this *WAIT_MESSAGE* to the server until this condition is fulfilled (when the server has enough connections to start the game).

```
print("Game Started!")
extra = {"action": "draw", "amt": 26}
data = {"name" : name, "extra" : extra}
hand = deserialize(send(serialize(data)))["hand"]
```

The `dictionary` *extra* contains the "action" and "amt" keys. The "action" key is used to indicate what actions the user is about to perform. The "amt" key is used to store an `int` value for the actions.

The `dictionary` *data* contains the "name" and "extra" keys. The "name" key is used to store the name of the user while the "extra" key is used to store the *extra* `dictionary`.

We then receive the hand of the user by querying the server for the hand of the user using:

```
        hand = deserialize(send(serialize(data)))["hand"]
```

We then handle the graphical user interface (*gui*) of the user, by using *Tkinter*. We will define the class *Application* in the document later. An object, *app*, of class *Application* is created.

```
root = gui.Tk()
root.geometry("1920x1080+0+0")
app = gui.Application(master=root)
app.displayHand(hand)
```

We then use the method from the *Application* class *displayHand(hand)*, that simply displays the hand to the client.

We then want to loop through a while-loop after the game has started to run the rest of the game. Here, we proceed to create a new `dictionary` *data* that contains the *"name"* key and *"extra"* key.

```
while True:
    data = {"name" : name, "extra": {"action":""}}
    if app.selected is not None:
        extra = {"action": "discard", "index": app.selected}
        data = {"name" : name, "extra" : extra}
        reply = send(serialize(data))
        …
```

We firstly check if the user has selected any card using the *selected* attribute of the *app* object. If this attribute is *not None*, it indicates that the user has selected a card and we will proceed to run this loop.

In this loop we declare a new `dictionary` *extra* that contains the *"action"* and *"index"* keys. The *action* key indicates the action that the code should do, and the *index* key indicates the card's index number to discard.

We then declare a new `dictionary` *data* that contains the *"name"* and *"extra"* keys, like above.

The code then sends *data* back to the server and gets a response from the server side.

```
        try:
            game = deserialize(reply)
            players = game["players"]
            for player in players:
                if player["name"] == name:
                    hand = player["hand"]
                    app.displayHand(hand)
            app.selected = None
        except:
            pass
```

Using a try-except loop, we can execute a group of codes and handle any error easily. We use this technique to check if certain data can be deserialised. As such, we proceed to attempt to deserialise the reply from the server. If this deserialization is valid, we proceed to store the `list` *players*, which contains a *`dictionary`* of player information, in the `dictionary` *game* into the *players* variable.

We iterate through each item in `list` *players*. We find the player's hand by matching the *player["name"]* with the *name* variable that contains the name of the current client running this script.

If this is true, we then extract `dictionary` player variable's *"hand"* and pass this value to the *hand* variable. This *hand* variable is then used to create the gui for the hand of the player, by executing *app.displayHand(hand)*.

```
    else:
        reply = send(serialize(data))
        try:
            game = deserialize(reply)
        except:
            pass
```

This else statement is executed when nothing has been selected by the client yet. It simply attempts to deserialise the server's reply.

```
try:
    pile1 = game["pile1"]["cards"]
    pile2 = game["pile2"]["cards"]
    app.displayPileCards(pile1,pile2)
except:
    pass
sleep(0.1)
app.update()
```

After generating the gui for the client, we will then create the gui for the 2 pile of decks in play. We simply, assign *pile1* and *pile2* with the "*cards"* in *game["pile1"]* and *game["pile2"]* dictionary. Using the method *app.displayPileCards(pile1, pile2)*, we generate the gui for the card piles in play.

We get the server to sleep before updating to get the gui to be responsive to the mouse events as it needs time to draw all the resources. Hence, we get the game to refresh at 10Hz before updating the gui:

```
sleep(0.1)
app.update()k
```

2) Serializer (To serialise python objects into json)

The serialiser is used to change the python object into a JSON (JavaScript Object Notation) to send it across the socket.

We will import the json library,

```
import json
```

We define a class Serializable:

```
class Serializable:
    def toJSON(self): # python obj -> JSON
        return json.dumps(self, default=lambda o: o.__dict__,
                          sort_keys=True, separators=(',', ':'))
```

This class would be inherited by other classes so that the python objects can be serialised to JSON.

<Method> toJSON(self):
This method is part of the `class` *Serializable*, it returns the JSON converted version of a python object.

```
def serialize(d): # python list/dict -> JSON
    return json.dumps(d,separators=(',', ':'))

def deserialize(jsonString): # JSON -> python dict
    return json.loads(jsonString)
```

<function> serialize(d):
This function takes in 1 argument, *d*, that is a python `list` or `dictionary`. It then converts this argument into a *JSON string*.

<function> deserialize(jsonString):
This function takes in 1 argument, *jsonString*, that is a `JSON`. It then deserialises this JSON into a python `dictionary` using the function *json.loads(jsonString).*

3) Database (Create, modify databases)

This section would discuss the functions defined to handle databases using the sqlite3 package.

We import the sqlite3 library:

```
import sqlite3
```

We then create a sqlite3 connection:

```
con = sqlite3.connect("data.db",check_same_thread=False)
```

This connection connects to the database, “data.db”. If this database does not exist, this connection would create a new database “data.db”.

A cursor object is created in the connection to allow us to navigate the database,:

```
c = con.cursor() # cursor object
```

We now have the basic sqlite3 objects that will enable us to do the following functions:

<function> create(table_name, **kwargs):

This function takes in 2 arguments, *table_name & **kwargs.* *table_name* is the name of the table that the user wishes to create.***kwargs* expects a dictionary of the following key, value format:

key=<column name>, value=<SQL data type>
i.e.

create(“playerdata”, name=”TEXT”, hp=”INTEGER”)

OR

columns = (“name”: “TEXT”, “hp”: “INTEGER”)
*create(“playerdata”, **columns)*

***kwargs* allows us to receive any number of arguments in key, value pairs. In the latter example, ***columns* helps us unpack the *dictionary columns*, to its key, value pairs (same format as the former).

The c.execute(`string` SQL) function, helps to execute an SQL statement. The first statement executed in the function is as follows:

```
c.execute(f" SELECT count(name) FROM sqlite_master WHERE type='table' AND
name='{table_name}' ")
```

All sqlite3 databases automatically generate a schema table when first created, referred to as *sqlite_master*, which stores a description of all objects present in the database, namely tables, indexes, views and triggers. A schema table contains several rows pertaining to the properties of each object stored in the table as shown below:

CREATE TABLE sqlite_schema(
type text,
name text,
tbl_name text,
rootpage integer,
sql text
);

Therefore, the SQL statement references the *sqlite_master* schema table and returns the number of objects in the *name* row that are tables and have the same name as the argument *table_name*.

The following code checks if there is already a table existing:

```
if c.fetchone()[0]== 0:
# if number of tables is 0 (table does not exist), create table
```

The function *c.fetchone()* is generally used to return a single row as a tuple. In this case, it will either return *(1,)* if a table exists and *(0,)* if not. *c.fetchone()[0]* returns the first value of the tuple and if no tables are present, *c.fetchone()[0] == 0* would evaluate to *True*, executing the *if-loop* below:

```
    if len(kwargs) > 0:
       msg = f"CREATE TABLE {table_name} "
       cols = ""
       for k,v in kwargs.items():
          cols += f"{k} {v}, "
       cols = f"({cols[:-2]})"
       c.execute(f"{msg}{cols};")
    else:
       print(f"Table {table_name} cannot be created (no columns specified)")
```

In this *if-loop*, we first check if the keyword arguments passed by the user at least have 1 key, value pair. If there is, we proceed to use the key, value pairs passed by the user and generate the columns and values of the table to be created using this portion of the abovementioned *if-loop*:

```
       for k,v in kwargs.items():
          cols += f"{k} {v}, "
```

We then slice the *cols* variable accordingly to remove the additional comma and spacing at the end of the variable, preventing syntax errors:

```
       cols = f"({cols[:-2]})"
```

We will proceed to create the table using the SQL statement:

```
       c.execute(f"{msg}{cols};")
```

If there are no keyword arguments passed, the loop would print out an error function in the console:

```
   else:
       print(f"Table {table_name} cannot be created (no columns specified)")
```

If there is an existing table, we would not want the function to create another table. As such we would handle this by skipping the if-loop that creates the columns of the table. Instead, we will get the loop to print out an error function in the console as follows:

```
else:
    print(f"Table {table_name} already exists")
```

We finally end this function with the *con.commit()* method that sends a COMMIT statement to the MySQL server. This is done as Python does not autocommit after every modification to the table data.

<function> drop(table_name):
This function takes in 1 argument, *table_name*. Based on *table_name,* it would then delete this table completely from the database, *data.db*.

We will first check if the table exists using the following SQL statement which references the *sqlite_master* schema table and returns the number of objects in the *name* row that are tables and have the same name as the argument *table_name:*

```
c.execute(f" SELECT count(name) FROM sqlite_master WHERE type='table' AND
name='{table_name}' ")
```

Using *c.fetchone()[0]*, we check if the table is present. If so we continue with the *if-loop*:

```
if c.fetchone()[0]:
    c.execute(f"DROP TABLE {table_name}")
    con.commit()
    print(f"Table {table_name} suceessfully dropped")

else:
    print(f"Table {table_name} not found")
```

Executing the following SQL statement within the *if-loop* simply drops the table based on the *table_name* variable. We then commit this statement using the *con.commit()* method.

```
    c.execute(f"DROP TABLE {table_name}")
    con.commit()
```

If there is no such table present, we will proceed to run the following code which gets the loop to print out an error message in the console:

```
else:
    print(f"Table {table_name} not found")
```

<function> fetch(table_name, count=0, delete=False):
This function takes in 3 arguments, *table_name, count (default=0) and delete (default=False)* and returns up to the *count* number of rows from the *table_name* as a `list` of `dictionaries`.

If *count* = 0, it will return all rows of the *table_name*.

If *delete = True*, the function will delete the rows from the *table_name* after retrieving the rows.

The function begins by creating an empty list. If an empty argument for *count* is passed to the function, an empty string will be assigned to *limit_query*. Else, the *LIMIT* statement is assigned to *limit_query* instead. *query* selects a set number of rows while *query2* deletes the same number of rows from the top of *table_name*, with the number of rows being specified by *limit_query*.

```
lst = []
limit_query = "" if not count else f"LIMIT {count}"
query = f"SELECT * FROM {table_name} {limit_query}"
query2 = f"DELETE FROM {table_name} {limit_query}"
```

We will execute the first query to fetch the rows from the table:

```
rows = c.execute(query)
```

We then iterate through each row. The name of each row, *rows.description[i][0]*, and the value of the row, *row[i]*, are stored as keys and values in the dictionary *d* respectively. *d* is appended to *lst* for each new row, resulting in a list of dictionaries.

```
for row in rows:
    d = {}
    for i in range(len(rows.description)):
        k,v = rows.description[i][0], row[i]
        d[k] = v
    lst.append(d)
```

By executing *query2*, the fetch function will delete the rows selected when executing *query*.

```
c.execute(query2) if delete else None
con.commit()
return lst
```

<function> insert(table_name, **kwargs):

This function takes in 2 arguments, one *table_name* & ***kwargs (accepts any number of keyword arguments)*. This function inserts a new row into the table stated by *table_name*. The kwargs is a `dictionary` (key=<column name>, value=<cell data>), similar to that of the *create()* function above.

We first declare two variables which we will store the name of the column (*cols*) and the value (*vals*) of that column.

```
cols = ""
vals = ""
```

After which, we will iterate through the key, value pairs defined by the user, combining all keys into a single string separated by commas and doing the same for all values.

```
for k, v in kwargs.items():
    cols += k + ", "
    vals += f"'{v}'" + ", "
```

We then slice the *cols* and *vals* variables accordingly to remove the additional comma and spacing at the end of the *cols* and *vals* string variables, preventing syntax errors.

```
cols = f"({cols[:-2]})"
vals = f"({vals[:-2]})"
```

We then execute this SQL statement that inserts the *cols* as the values under the first column and *vals* as the value of *cols* for a specific row into the table stated by *table_name* and commit this SQL statement to the database.

```
c.execute(f"INSERT INTO {table_name} {cols} VALUES {vals}")
con.commit()
```

<function> update(table_name, column, where_clause, **kwargs):
This function takes in 4 arguments, *table_name*, *column*, *where_clause* and ***kwargs*. The function updates all rows in *table_name*, whenever cell value of specified column matches *where_clause*.

table_name – table name to be updated
column – column to be updated
where_clause – Query statement to find the value in that column (Only for TEXT type SQL)
***kwargs* – key, value pairs that contains the <column name>=<value to edit>

Examples of usage of this function:
update("playerdata", "name", "James", hp=5)
OR
*update("playerdata", "name", "James", **columns)*

The function firstly defines a string variable that would hold the query, *set_query*. It also defines a string variable that would hold the query (condition to find the value) to the database, *where_query*.

```
set_query = ""
where_query = f"WHERE {column}='{where_clause}'"
```

Using the key, value pairs arguments, we will then create a query as follows. This updates the value of the column stated for those values that fulfills the *where_clause* query.

```
for k,v in kwargs.items():
    set_query += f"{k}='{v}', "
set_query = f"{set_query[:-2]}"
query = f"UPDATE {table_name} SET {set_query} {where_query}"
```

The query is then executed and committed to the database.

```
c.execute(query)
con.commit()
```

<function> deleteall(table_name):
This function takes in an argument, *table_name*, and deletes all the rows in that table. It does not delete the entire table, which is the function of *drop(table_name)*.

The function uses the SQL statement to delete all the columns from the stated *table_name*.

```
query = f"DELETE FROM {table_name}"
```

The query is then executed and committed to the database.

```
c.execute(query)
con.commit()
```

4) server_logic (Cards, Decks, Players and Game)

This section would discuss the classes constructed which is required in order to run our game.

We start by importing 3 libraries, namely random, json and serializer which would be used primarily for shuffling the deck, and to send over objects from the server to the client when the game runs.

```
import random, json, serializer
```

<class> Card(serializer.Serializable):

The Card class is constructed which in itself is an empty list in the beginning. It takes in 1 argument, *serializer.Serializable*. This allows any object in the card class to now be able seralize into a json string. *Self.value* and *self.suit* binds the attribute to value and suit and then *return f"* allow us to make string interpolation much simpler since we intent to return both the card value and suit as a string together whenever the *<function> getName* is being called.

```
class Card(serializer.Serializable):
    def __init__(self, value, suit):
        self.value = value
        self.suit = suit
    def getName():
        return f"{self.value}{self.suit}"
```

<class> Deck(serializer.Serializable):

The Deck class is constructed using *self.cards = []* which in itself is an empty list in the beginning. It also takes in 1 argument, *serializer.Serializable*. This allows any object in the card class to now be able seralize into a json string.

```
class Deck(serializer.Serializable):
    def __init__(self):
        self.cards = []
```

<function> build(self):

A deck is built by utilising 2 for loops. Values from 2 to Ace are constructed and represented by a range of values of 1 but not limited to 14 respectively as well as being assigned a suit of clubs, diamonds, hearts and spades. Within the for loop, the card objects are made by assigning those 1 to 13 numbers with their suits in *card= Card(value,suit)*. The cards constructed are then added into the list by *self.card.append*(card). Now, the deck has been constructed which will then be shuffled by the *self.shuffle()* command.

```
def build(self):
    for value in range(1,14):
        for suit in ["c","d","h","s"]:
            card = Card(value,suit)
            self.cards.append(card)
    self.shuffle()
```

<function> shuffle(self):

The shuffle function only calls *random.shuffle()* to the list of cards in the deck. By utilizing the random library, we minimized the need to make our own *random.shuffle()* function which could be predictable to the users as they play the game when the deck is shuffled.

```
def shuffle(self):
    random.shuffle(self.cards)
```

<class> Player(serializer.Serializable):

The player class is constructed which gives the Player attributes of a name, position and points. *Self.position* assigns a positional attribute of 1 or 0 to a player which will assign players dealt cards. Similarly, self.name assigns names to the players and self.hand creates a list which will be the hand of the player. *Self.tricks* of a player is initially at 0 as it will increase based on the amount of rounds the player wins in a game. *Self.points* is then the number of points a player has won after winning a number of games which is determined by the integer value in *self.tricks*,

```
class Player(serializer.Serializable):
    def __init__(self,name,position,points):
        self.position = position
        self.name = name
        self.hand = []
        self.points = points
        self.tricks = 0
```

<class> Game(serializer.Serializable):

The Game class is constructed first by giving Game attributes such as *self.deck, self.pile1, self.pile2, self.players and self.deck.build(). self.deck* constructs deck from <class> Deck and then 2 piles of decks, namely *self.pile1 and self.pile2* also construct deck same way. *Self.players* creates an empty list that will later be used in <*function*> *addPlayer*. Lastly, *self.deck.build() calls* the function build in <*class*> *Deck* to construct build cards into the empty deck.

```
class Game(serializer.Serializable):
    def __init__(self):
        self.deck = Deck()
        self.pile1 = Deck()
        self.pile2 = Deck()
        self.players = []
        self.deck.build()
```

<function> addPlayer(self,player):

The addPlayer function only contains self.players.append(player) which adds player names into *self.players* list constructed in the Game class.

```
def addPlayer(self,player):
    self.players.append(player)
```

<function> draw(self, name, count=1):

The draw function contains default parameters of the name of the player which was assigned previously in *Class Player* to draw the card to, which is represented by *name* and the parameter of *count=1* which specifies the default number of cards to draw to the player. In *draw(), len(self.deck.cards) >= count:* makes it such that the program will only run when the number of cards in the deck is greater or equal to the number of cards assigned to be taken out of the deck

which is *count=1:* The for loop in *for i in range(count):* selects the number of cards to be drawn which by default, should be 1. with *card=self.deck.cards.pop()* the last card of the deck list could be removed from the list and then assigned to the object *card.*
for player in self.players: calls out the name of the player in *self.players* and the *if player.name == name* checks for the player that had drawn the card as mentioned in the *name* parameter to assign the card to the players list hand by *player.hand.appened(card).*

```
def draw(self, name, count=1):
    if len(self.deck.cards) >= count:
        for i in range(count):
            card = self.deck.cards.pop()
            for player in self.players:
                if player.name == name:
                    player.hand.append(card)
```

<function> discard(self, name, index =0):
The discard function contains default parameters of the name of the player which was assigned in *Class Player* to discard the card to, which is represented by *name* and the parameter of *index = 0* specifies the default number of cards to discard. In *discard(), for player in self.players:*, the program will scan through player names through *for player in self.players:* and if it matches using *if player.name ==name:*, the card at the hand index will be removed from the line *card = player.hand[index]* and *plaer.hand.remove(card)*. Depending on the position of the player, the card will the go to the correct pile using the condition, *pile = self.pile2.cards if player.position else self.pile1.cards* and added into that pile through *pile.append(card).*

```
def discard(self, name, index = 0):
    for player in self.players:
        if player.name == name:
            card = player.hand[index]
            player.hand.remove(card)
            pile = self.pile2.cards if player.position else self.pile1.cards
            pile.append(card)
```

5) GUI

This section would discuss the class constructed which is required in order to generate the graphical user interface (GUI) using the *tkinter* library.

We will import the following libraries:

```
import time
from tkinter import *
```

Then we define a class named *Application (Frame)*:

```
class Application(Frame):

    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.pile1 = Label(self.master)
        self.pile1.place(x=750,y=200)
        self.pile2 = Label(self.master)
        self.pile2.place(x=600,y=200)
        self.photos = {}
        self.hand = []
        self.selected = None
        for value in range(1,14):
            for suit in ["c","d","h","s"]:
                name = str(value)+suit
                p = PhotoImage(file=f"res/{name}.png").subsample(5)
                self.photos[name] = p
```

This class contains the following attributes which can be created using the constructor. This class inherits, is the child class, of the class *master*. It also contains the above attributes.

Variables	Functions
master	Master class object
pile1	*`label`* of the *master* class object (1st card pile in the play area)
pile2	`label` of the *master* class object (2nd card pile in the play area)
photos	dictionary` store all the key, value pair of the photo's name and the directory of the photo
and	`list` store the hand of the player
selected	`bool` that indicates if the first move of the game has been played

This *for-loop* creates the card deck, it uses iterates through the numbers *from 1 up to 13*. Since we have 4 types of suits of the card, we used the list *["c", "d", "h", "s']* to iterate through each of

this suit. We will then use a `string` *name* variable, to create a string that is formed up by using the number and the suit.
After creating this `string` *name*, we then use the *PhotoImage*(file=<*file directory*>) function which we will pass the *name* variable to create a *tkinter `PhotoImage`* variable.

```
for value in range(1,14):
    for suit in ["c","d","h","s"]:
        name = str(value)+suit
        p = PhotoImage(file=f"res/{name}.png").subsample(5)
        self.photos[name] = p
```

After creating the *`PhotoImage`* variable, we will pass this variable to the *photos* attribute of the *Application* class.

<function> displayPileCards(self, p1, p2):
This function takes in 3 arguments, *self, p1 & p2*. *self* indicates the object of the *Application* class. *p1 & p2*, indicates the first and second pile of the play area respectively.

```
if len(p1) > 0:
    card1 = p1[-1]
    value1, suit1 = card1["value"], card1["suit"]
    name1 = str(value1)+str(suit1)
    w1 = self.pile1
    p1 = self.photos[name1]
    w1.configure(image=p1)
```

We have 2 card piles in play, if there is a card in the pile, the if condition *len(p1) > 0* would evaluate *True*. This executes the if-statement, *p1 & p2*, is the piles of the game area. It can also be seen as the piles for player 1 and player 2 respectively.
We will then take in the last value (card) of the list *p1*. This would be passed to a variable `dictionary` *card1*. We proceed to unpack the `dictionary` *card1's "value"* and *"suit"* attributes into the variables *value1 & suit1*.
We will create a `string` *name1* that stores the *value1* and *suit1* concatenated.
A `label` variable *w1* is declared taking in the value of the attribute *pile1* from the *Application* class.
A `string` variable *p1* is declared with the photo directory by passing the `string` *name1* to the method of the *Application* class, *photos [name1]*.
We then display the `photo` *p1* by changing the `label` *w1* using the function:

```
w1.configure(image=p1)
```

The same block of code is used for the *p2*, pile 2 of the game area.

```
if len(p2) > 0:
    card2 = p2[-1]
    value2, suit2 =  card2["value"], card2["suit"]
    name2 = str(value2)+str(suit2)
    w2 = self.pile2
    p2 = self.photos[name2]
    w2.configure(image=p2)
```

<function> displayHand(self, cards):
This function takes in 2 arguments, *self & cards*. *self* indicates the object of the *Application* class. A `list` *cards* argument is passed to the function. This function displays the hand of the

client/player.

Iterating through the `list` of `labels` *self.hand*, we *destroy* all the labels of *self.hand* to start a new `list` *self.hand*.

```
for l in self.hand:
    l.destroy()
```

`list` *self.hand* is declared with an empty `list`.

```
self.hand = []
```

These are the declared constants, that would be used later to position the cards.

```
x = 300
dx = 70
y = 600
```

We iterate through the `list` *cards*, and pass each value to the `dictionary` *card* variable. A `string` *name* variable is used to create the `string` required for the photo directory. The `photo` *img* variable finds the photo of the card.

We create a new *tkinter `label` label* variable with the image of the card, *img*. We place the *label* variable in the *x* and *y* of the *master tkinter canvas* using the *place(x=x, y=y)*. We then bind the label to 3 trigger events, *<Enter>*, *<Button-1>* & *<Leave>*. We append this label to the *hand* attribute of the *Application object*.

We then offset the *x* coordinate by *dx* for the next card.

```
for card in cards:
    name = str(card["value"])+card["suit"]
    img = self.photos[name]
    label = Label(self.master,image=img)
    label.place(x=x,y=y)
    label.bind("<Enter>",self.enter)
    label.bind("<Button-1>",self.click)
    label.bind("<Leave>",self.leave)
    self.hand.append(label)
    x += dx
```

These are the 3 trigger functions of the labels:

```
def enter(self,event):
    y = int(event.widget.place_info()["y"])
    event.widget.place(y=y-35)

def click(self,event):
    self.selected = self.hand.index(event.widget)

def leave(self,event):
    y = int(event.widget.place_info()["y"])
    event.widget.place(y=y+35)
```

<function> enter (self, events):
The function moves the card position up, to improve the GUI of the user when the user hovers over the card label.

<function> click (self, events):
The function changes the *selected* attribute to the index of the card that was chosen.

<function> leave (self, events):
The function moves the card position down, to improve the GUI of the user when the user's cursor leaves the card label.

References

1. datacamp. 2020. *Inner Classes in Python*. [online] Available at: <https://www.datacamp.com/community/tutorials/inner-classes-python> [Accessed 6 December 2021].

2. National Institute on Aging. 2021. *What Is Dementia? Symptoms, Types, and Diagnosis*. [online] Available at: <https://www.nia.nih.gov/health/what-is-dementia> [Accessed 7 December 2021].

3. Hospital News. 2021. *Activities and games for patients with Alzheimer's disease - Hospital News*. [online] Available at: <https://hospitalnews.com/activities-and-games-for-patients-with-alzheimers-disease/> [Accessed 7 December 2021].

4. Centre for Healthy Brain Ageing (CHeBA). 2016. *Reaction Time Test Predicts Risk of Dementia*. [online] Available at: <https://cheba.unsw.edu.au/news/reaction-time-test-predicts-risk-dementia#:~:text=Researchers%20from%20the%20Centre%20for,within%20the%20next%20four%20years.> [Accessed 7 December 2021].

5. Youth.worldbridge.org. 2021. *Benefits of playing card games with children | Youth World Bridge*. [online] Available at: <http://youth.worldbridge.org/benefits-of-playing-card-games-with-children/> [Accessed 7 December 2021].

6. Vector-playing-card. 2014. A full set of poker playing cards created using vector graphics. The .SVG source for each card is available as well as a high resolution rasterized .PNG version. These images are released into the public domain. Available at: <https://opengameart.org/content/playing-cards-vector-png>