YOUR TASK

Your task is to add multiplayer functionality to the game provided

1. Share block placements over the network
2. Share the positions of both players over the networks
3. Edit the program to allow an unlimited number of users simultaneously

HOW THE GAME WORKS

You can move with WASD, place blocks with left click, remove with right click and swap your selected block by scrolling

All of the state for the game, is stored in ***state*** which is an instance of ***GameState***

```
17 class Player:
16     def __init__(self, colour: int):
15         self.player_pos = rl.Vector2(0, 0)
14         self.player_vel = rl.Vector2(0, 0)
13         self.player_grounded = False
12         self.colour = colour
11
10
 9 class GameState:
 8     def __init__(self):
 7         self.spritesheet = None
 6         self.grid = [0 for _ in range(TILES_X * TILES_Y)]
 5         self.player = Player(COLOUR)
 4         self.other_players = []
 3         self.other_players.append(Player(1))
 2         self.updated_tiles: dict[tuple[int, int], int] = {}
 1         self.selected_tile = 1
```

other_players contains a list of players that belong to other players. **It is populated with a single player to start with for you to use to hold data for received player data**

You can update the grid by using setTile

```
11 def set_tile(state: GameState, x, y: int, tile: int):
10     state.grid[y * TILES_X + x] = tile
```

Tile	**#**
Air	**0**
Stone	**1**
Dirt	**2**
Grass	**3**

Each frame the program calls ***send_state*** and ***receive_state***. You should complete these functions.

Send_state should send a packet for each block change and the current position (Task 2). To do this, you will need to encode your fields into a binary format (**bytes**) then call the function ***send_data*** which will send data to all destination ports.

receive_state should receive these packets and then update the state. This will then be drawn on the screen. To accomplish this, you need to retrieve all of the received data using ***receive_data*** then extract the variables out of the binary format (**bytes**)

I have stored all of the changes to blocks in a dictionary *updated blocks* which uses *x, y* of the tile as the key and the block type of the value. **You can iterate through it like this**

```
9 def send_state(state):
8     for pos, block in state.updated_tiles.items():
7         x = pos[0]
6         y = pos[1]
```

I have also completed the first part of ***receive_state*** for you

```
4 def receive_state(state):
3     for data in receive_data():
2         pass
```

For each data you will need to retrieve the data out of the binary format, then update the state.

YOU DO NOT NEED TO RETURN THE STATE

RUNNING THE PROGRAM

You should run multiple instances of the program at the same time

To run the program on the lab computers you need to enter the **Python Virtual Environment**

>> source /venv/activate/bin

You can then run the program but **you need to set it up correctly**

>> python block_game.py --port [PORT] --connect [PORT OF OTHER PROGRAM] --colour [COLOUR]

For port number, I would suggest 5005 (default for this program) but you could use a different one if you want, **the other port needs to be different**. The colour is the colour of the sprite (you should try to send this across the network)

Colour	**#**
Red	**0**
Blue	**1**
Yellow	**2**
Green	**3**

FIRST PROGRAM

>> python block_game.py --port 5005 --connect 5006 --colour 0

SECOND PROGRAM

>> python block_game.py --port 5006 --connect 5005 --colour 1

SENDING DATA TO OTHER PLAYERS

Data is sent to all specified ports; This is stored in the *DESTINATION_PORTS* global variable

The function *send_data* will do this for you. **However, it requires you to format your data as 'bytes' first (see last page)**

```
# Sends a bytestream to all of the specified destination ports
def send_data(data: bytes) -> None:
    sock = SOCK
    if not hasattr(send_data, "buffer"):
        send_data.buffer = []
    global DESTINATION_PORTS
    current_time = time.monotonic()
    send_data.buffer.append((data, current_time))

    remaining = []

    for packet_data, timestamp in send_data.buffer:
        if current_time - timestamp >= DELAY / 1000:
            for port in DESTINATION_PORTS:
                sock.sendto(packet_data, (IP, port))
                print("sent message")
        else:
            remaining.append((packet_data, timestamp))
    send_data.buffer = remaining
```

You should call *send_data* each frame. **You can delay sending data**. You can make this delay longer or shorter by changing the global variable ***DELAY***. Setting this to zero will mean that packets are sent instantly. Sending this to 1000 means that each message will be delayed by a second.

RECEIVING DATA

You will receive data from any instance of the game that is specified to send data (using the --connect flag). You should assume that all data is valid and that you are not receiving anything malicious or erroneous.

```
22 # Returns a list of bytes, may be empty, each item is its own transmission
21 def receive_data() -> list:
20     sock = SOCK
19     return_data = []
18     while True:
17         try:
16             data, addr = sock.recvfrom(1024)
15             return_data.append(data)
14         except BlockingIOError:
13             break
12     return return_data
```

Each frame, you should call the function **receive_data** which will return a list of bytes that you have not received previously. **Each packet is its own item.** If the list is empty, there is no data to process (since you last called receive_data)

To retrieve data out of bytes you will have to unpack it! (see last page)

CONVERTING INTO and OUT OF BYTES (packing and unpacking)

For converting a string (NOT RECOMMENDED):

```
def pack_data(message: str) -> bytes:
    return str.encode(message)
    # OR
    return message.encode("utf-8")


def unpack_data(data: bytes) -> str:
    return bytes.decode(data)
```

Using structs (RECOMMENDED APPROACH):

```
def pack_data(var1, var2, var3) -> bytes:
    return struct.pack("!HHH", var1, var2, var3)


def unpack_data(data: bytes) -> tuple[int, int, int]:
    return struct.unpack("!HHH", data)
```

Structs allow you to pack multiple values of different types into a single byte-stream that you can send over the network. To do this, you need a specified format that you use for both encoding and decoding

The first character '!' is the encoding format: big-endian (standard for networking)

The following characters indicate a specific data type of the data stored in that order

(so HHH stores three unsigned shorts; if var1 was a float it would be dHH)

Char	Data Type	Size (bytes)	Char	Data Type	Size (bytes)
b	Signed char	1	B	Unsigned char	1
h	Signed short	2	H	Unsigned short	2
i	Signed int	4	I	Unsigned int	4
f	Float	4	d	Double	8

The **unpack** function returns a n-tuple where n is the number of variables specified, you can easily assign each value to a different variable as shown below

```
x, y, colour = unpack_data(data)
```