Assignment 2 - Mars Rover (10%)

You are to write the client-side networking component for a multiplayer video game using the Python programming language. The client will need to implement the network protocol to communicate correctly to the server.

Rover is a multiplayer game where you control a single rover, explore and interact with the planet. Since this is a beta version of the game, the game focuses on an exploration component.

The planet is defined as a two-dimensional grid of tiles. All tiles are square-shaped and of unit length. Each tile will hold information related to its current coordinates. Planets are spherical and not flat. This means the left-most column of the grid should "meet up" with the right-most column. Similarly, the top-most row should meet up with the bottom-most.

You will need to complete the following:

- Parsing user input and executing commands
- Implement the network protocol for the client side
- Implement client-side functions for drawing and interaction.
- Maintain state information within the client.
- Messages between client and server is ascii and will need to be treated as such
- Utilise simple concurrency within your game by using Python's multiprocess package (alternatively you can use `fork()` and create a shared buffer)

User Input

The user interacts with the game by inputting commands that will update their character on the server. The user will enter commands following a `>` symbol. This symbol denotes when the program is waiting for a user to input a command.

If a user inputs an invalid command, the program must output `invalid command`. Unless otherwise stated, this includes invalid arguments.

If the user inputs a command at a time when they shouldn't, the program must output `cannot invoke the command in this state`.

Network Protocol

You will be implementing a text-based network protocol. Both server requests and responses will begin with a fixed 256-byte message containing the string representation of an unsigned integer number which denotes the number of bytes in the following message.

For server requests, the server will expect the length to be followed by a message of that length containing a command to the server (see **Commands** below).

Format:

```
<length>
<command type> [<command arguments>...]
```

Example:

```
23
login rste3264 gxtztlyt
```

The server response will then begin with the length of the message, followed by either an `ok` or an `error` response (see **Messages from server**). Unless specified, when an error is returned by the server, you should print out the message to stdout.

Commands

The program will print out a `>` character when waiting for user input. Once a user has inputted a command it will be processed by the program and sent to the server. After it has been processed, it will once again wait for user input again.

Example:

```
> connect 127.0.0.1 9000
Connected, please log in
> login rste3264 gxtztlyt
Logged In!
>
```

connect

- `connect <hostname/ip> <port>`

Your client will connect to a server. This command is not part of the messaging protocol; your program must create a socket and connect to the server. You must use TCP transport protocol for communications with the server.

Server Response

If the client can connect to the server, the server will respond with `ok`, accepting the connect request once the client has opened a socket to the server:

```
10
ok connect
```

Client Output

If the client's connection fails or the user has not entered a hostname/ip or port, your client must respond with the following message to the user:

```
Unable to connect to the server, check command arguments
```

If the connection succeeded, your client must respond with

```
Connected, please log in
```

login

- login <ident> <password>

The user is able to specify their identifier and password. This should be the first command executed after a connection has been established. If other commands are executed, the server will respond with an error (see **Messages from the server**).

Server Request

Format:

```
<length>
login <ident> <password>
```

Example:

```
18
login my_id 112233
```

Server Response

The server will respond with an **ok** response to confirm that it has received your message.

```
8
ok login
```

Client Output

In the event that the user has not specified an ident or password, your client must respond with

```
Incomplete login criteria
```

In the event that the server does not accept your login, your client must output. The server will reply with an error message.

```
Invalid login details
```

When the user has logged in, the client must output

```
Logged In!
```

observe

- observe

This will allow the user to get a simple look at the area around the rover. The server will provide a of 7x5 grid of the user's location with the rover at the centre. This grid will be in a linear sequence of tuples parsed from the **ok** response.

The data received from this needs to be stored by the client to be later committed to the server with the `commit` command.

Server Request

```
14
action observe
```

Server Response

After the server has received the message, it will return an **ok** response, followed by a sequence of tuples containing details about the terrain. The tuples will be in row-major order to allow you to easily render each tile.

Format:

```
<length>
ok observe (<x coordinate>,<y coordinate>,<elevation>,<rover on tile>,<message on tile>)
...
```

Example:

```
430
ok observe (0,0,0,0,0) (1,0,0,0,0) (2,0,0,0,0) (3,0,-3,0,0) (4,0,2,0,0) (5,0,0,0,1) ...
```

- `<x coordinate>` and `<y coordinate>` are the absolute x and y coordinates of the tile on the grid.
- `<elevation>` is the absolute elevation of the tile.
- `<rover on tile>` will be 0 if there are no rovers on the tile and 1 if there is at least 1 rover on the tile.
- `<message on tile>` will be 0 if there is no message on the tile and 1 if there is a message on the tile.

Client Output

Afterwards, your program must display a representation of the grid to the user where each tile is represented by two characters and separated by `|` symbols.

Example:

```
|R |  |  | -| 2|  |  |
|  |M |  |  |  |  |  |
|  | 1|  |RR|  |  |  |
|  |  |  |  |  |  |R2|
|  |  |  | 5|  |  |  |
```

Elevation will appear on the grid as a number on the right column of the cell. The displayed elevation of each tile is relative to the current elevation of the rover. e.g. if a rover is on a tile of elevation of 1 and a tile retrieved from an `observe` command has an elevation of 3, it will appear as an elevation of 2. The server sends an absolute elevation value for each tile.

Any tile which has a relative elevation of > 9 will only show the maximum elevation (9). Any tile which has a relative elevation of < 0 will show a '-' symbol on the right column. If the tile's relative elevation is 0, only a space will be shown. If the relative elevation is > 0 and <= 9, it will show the elevation in the right column of the cell.

The left column of the cell will contain a 'R' symbol if there is a rover on that tile, or a 'M' symbol if there is a message there. If there is both a rover and a message, the 'R' symbol will appear. If there is neither, it will contain a space.

move

- move (north | west | east | south)

This will move the user's rover one space (in north, west, east or south). This can be abbreviated to `m` followed by the first character of the direction.

Server Request

Format:

```
<length>
action move <direction>
```

Example:

```
16
action move west
```

Server Output

After the server has received this message, it will return an **ok** response with the new position of your rover.

Format:

```
<length>
ok move (<x coordinate>, <y coordinate>)
```

Example:

```
11
ok move (2,3)
```

stats

- stats

Show the current stats of the player. It will return the current number of tiles that have been explored and the current position of the rover.

Server Request

```
12
action stats
```

Server Response

The server will respond with **ok** followed by a tuple containing the current position of the rover and the number of tiles which have been explored.

Format:

```
<length>
ok stats (<x position>,<y position>,<number of tiles explored>)
```

Example:

```
17
ok stats (2,3,56)
```

Client Output

Format:

```
Number of tiles explored: <number of tiles explored>
Current position: (<x coordinate>,<y coordinate>)
```

Example:

```
Number of tiles explored: 56
Current position: (2,3)
```

inspect

The inspect command will allow the user to inspect an adjacent tile and gather information from it. This can reveal messages which have been left on tiles with the `note` command.

Server Request

Format:

```
<length>
action inspect <direction>
```

Example:

```
19
action inspect west
```

Server Response

If a message has been left on the tile, the server will respond with the message that has been left

Format:

```
<length>
ok inspect (<message>)
```

Example:

```
30
ok inspect (Curiosity Wake Up)
```

If no message is left on the tile, the server will respond with

```
10
ok inspect
```

Client Output

If a message was found on the tile, the client must output:

Format:

```
You found a note: <message>
```

Example:

```
You found a note: Curiosity Wake Up
```

If no message was found, the client must output:

```
Nothing interesting was found here
```

note

- note

A message will be placed on the rover's current tile. This allows for simple communication within the game itself. If a message is placed on a tile which already has a message, the server will overwrite the old message.

Server Request

Format:

```
<length>
action note <message>
```

Example:

```
24
action note hello world!
```

Server Response

```
7
ok note
```

message

- message

Similar to the `note` command, this allows the user to send a message directly to another rover.

Server Request

Format:

```
<length>
action message <ident> <message>
```

Example:

```
36
action message ffde2211 hello world!
```

Server Response

```
10
ok message
```

commit

- commit

When the user has finished playing, they can commit their exploration data to the server. The client will need to maintain all the terrain information which has been retrieved from `observe` commands.

Server Request

The client must send a sequence of tuples containing the coordinates of every tile which has been observed during the session.

Format:

```
<length>
action commit (<x coordinate>, <y coordinate>)...
```

Example:

```
250
action commit (0,0) (1,0) (2,0) (3,0) (4,0) (5,0) ...
```

Server Response

After all tuples have been committed, the server will send an ok to the client, noting that it has received the client's tiles.

```
9
ok commit
```

quit

- quit

The quit command will close the session and exit the game. If you have not committed your data to the server, it will be lost. When the user has issued the quit command, it must wait for an **ok** from the server before terminating the connection.

Server Request

```
4
quit
```

Server Response

```
7
ok quit
```

Messages from server

ok

- ok

When a message is sent to the server, the server will return an **ok** or **error** message to confirm the message has been received and accepted by the server.

Format:

```
<length>
ok <command type> [<additional data>]
```

Example:

```
8
ok stats
```

See **Commands** for other examples.

error

- error

An error message can be sent to your client when it has been unable to parse your command or the command has been misused. This error will be followed by an error message as to what command or protocol message was considered erroneous.

Format:

```
<length>
error <message>
```

Example:

```
28
error unable to parse action
```

When an error is sent to the client, the client should output to standard output

```
Error: <message>
```

Example:

```
Error: unable to parse action
```

event

The client will receive events from the server and show what is currently going on. An event can happen at any time. These events can be what other players perform in relation to you, actions you have performed, and any server notification.

Format:

```
<length>
event <type> [<data>]
```

You may receive the following event types sent to your client, **notify** or **message**. This kind of message can be sent to your client at anytime (assuming the client is connected to the server and logged in).

message events are sent by other players on the server.

Format:

```
<length>
event message <ident> <message>
```

Example:

```
35
event message ffde2211 Hello There!
```

This should output the following to standard output

Format:

```
<ident>: <message>
```

Example:

```
ffde2211: Hello There!
```

notify type events are to be treated these as messages from the server.

Format:

```
<length>
event notify <message>
```

Example:

```
65
event notify We will be going into maintenance mode in 15 minutes
```

The client outputs the following to standard output:

```
Server: <message>
```

Example:

```
Server: We will be going into maintenance mode in 15 minutes
```

Testing

You are expected to write a number of test cases for your program. These should be simple input/output tests. An example test will be included in the scaffold. You are expected to test every execution path of your code.

You're suggested to write a simple testing script in Bash to run your input/output tests to simplify and automate the testing process.

Submitting your code

An Ed assessment workspace will be available soon for you to submit your code. Your final submission is due on the **10/11/2019 at 11:59PM AEST**

Public and hidden test cases will be released in batches until Sunday the 4th of November. Additionally, there will be a set of unreleased test cases which will be run against your code after the due date.

Any attempt to deceive the automatic marking system will be subject to academic dishonesty proceedings.

Marking details

- You will receive up to a maximum of 5 marks for passing all automatic test cases. There will be no new automatic test cases to be released after **04/11/2019 at 11:59 AEST**.
- You will receive up to a maximum of 2 marks for constructing test cases that cover all execution paths, corner cases and common cases within your program.
- You will receive up to a maximum of 2 mark that demonstrates your program can operate in real-world environment via manual inspection and usage.
- You will receive up to a maximum of 1 mark for good code style and implementation.

Academic Declaration

By submitting this assignment you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically

acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.