



Temasek Junior College
2023/2024 JC H2 Computing
Socket Programming

Section	4	Computer Networks
Unit	4.1	Fundamentals of Computer Networks
Objectives	4.1.6	Implement an iterative server with socket programming. Given the server code, students should be able to implement the client code for a given scenario, and vice-versa e.g. for a tic-tac-toe game.

1 What is a Socket?

Suppose you have two programs running at the same time. How would you send data from one program to the other and vice versa?

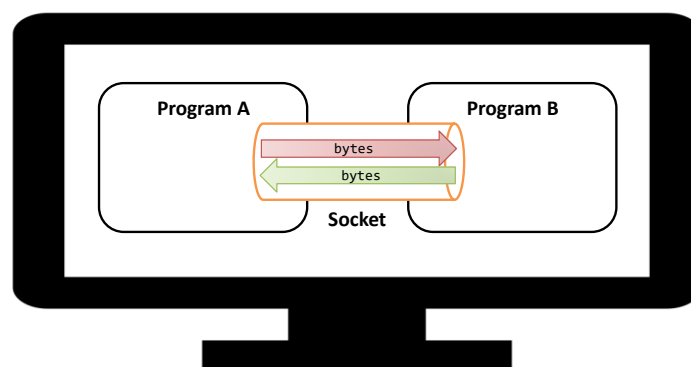
Most operating systems provide a powerful mechanism known as **sockets** to achieve this purpose.

A **socket** is one endpoint of a two-way communication link between two programs running on a computer network, where each endpoint allows for sending and receiving data.

Sockets facilitate data transmission between processes, which may reside on the same machine or on different hosts over a network.

A **socket** is one end of a two-way communication link between two programs running on a computer network. It is an endpoint for sending and receiving data between the two programs. Each socket is associated with an Internet Protocol (IP) address, which identifies the host, and a port number, which identifies a specific application or service on that host.

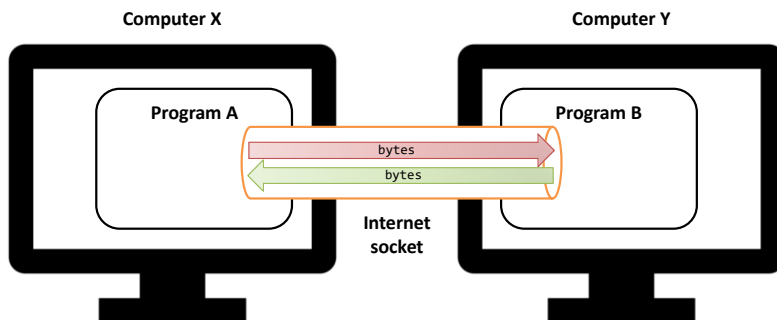
From a layman's perspective, a socket connection can be visualised as a "pipe" running between two programs. The "flow" in the "pipe" is "bidirectional", allowing data (represented by bytes) to be "transported" in both directions.



There are many types of sockets, with the **Internet socket** being the type most discussed.

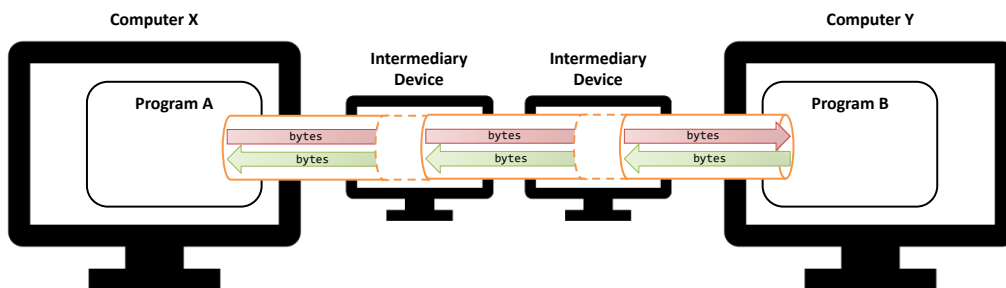
Internally, Internet sockets deliver data using the same Transmission Control Protocol and Internet Protocol (TCP/IP) suite that is used to transmit data over the Internet.

This means that Internet sockets can deliver data between any two programs, even programs that are running on different computers, so long as the two computers can access each other over the network.



For simplicity, an Internet socket can be visualised as a “pipe” attached between two computers.

In reality, data transmitted through an Internet socket may pass through multiple devices before reaching its destination. A more accurate visualisation will be as follows:



During data transmission, there is a risk of data being stolen from or modified without authorisation on any of these devices. Hence it is important to encrypt the data prior to transmission to mitigate the risks.

As networks can become congested, we cannot assume that data sent over Internet sockets will be transmitted instantaneously. For instance, a program may receive only the first half of a message before the second half arrives at a later juncture.

To avoid working with incomplete data, we will need to define a **protocol** (explained later) so that the start and the end of messages can be detected unambiguously.

Question

Which of the following methods for sending data from one Python program to another does NOT work?

- A** One program copies the data onto the system clipboard and the other program reads it from the system clipboard.
- B** One program writes the data into a file (that is readable to everyone) and the other program reads it from the same file.
- C** One programs assigns the data to a Python variable and the other program reads it from the same variable.

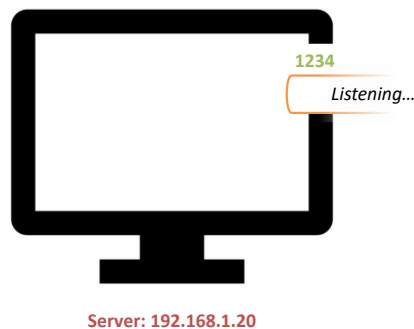
- D The two programs set up a socket connection and transmit the data from one program to the other through the socket.

(C)

2 Creating a Socket Connection

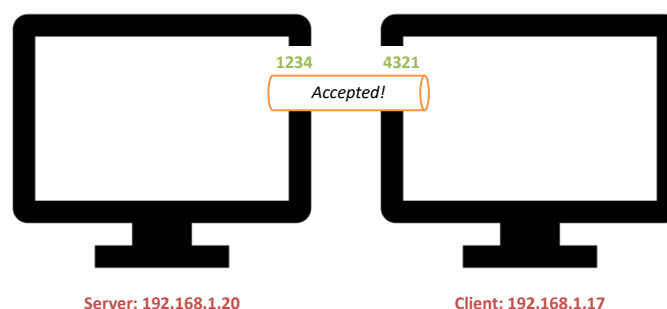
Creating a socket connection is a multi-step process that requires one program to be the **server** and another program to be the **client**. The server's IP address and port number for accepting connections must also be known ahead of time by the client.

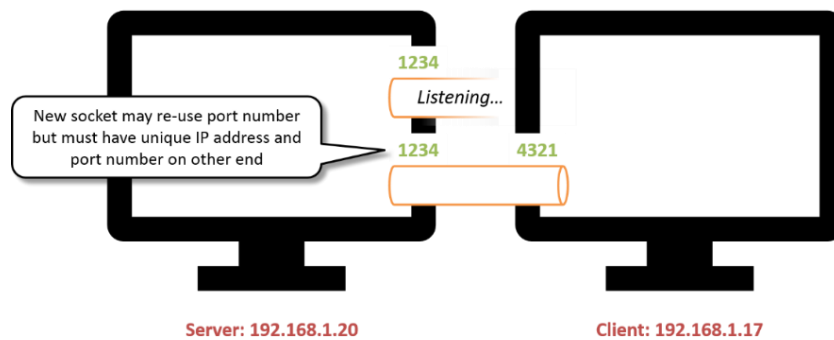
First, the server creates a **passive socket**, binds it to the pre-chosen port number and listens for an incoming connection. The passive socket will not be the eventual socket where the connection will be established on. It merely waits for an incoming connection.



Next, the client initiates a connection request using the server's IP address and port number. If no server is listening on the chosen port, the connection will be refused.

On the other hand, if the connection request reaches an IP address and port number that a server is listening on, the server accepts and establishes a new socket connection for the requesting client using a dynamically assigned port number, which may or may not be the same as the port number where the passive socket was initially listening on. **Regardless, there must be a unique IP address and port number on the client end of the connection.**





The passive socket goes back to listening for new connections while the client and server can now exchange data using the newly established connection.

Note that the newly-created connection is symmetrical: data sent on one end is received on the other end and vice versa. In other words, once a socket is established, it can send data both from the client to the server and from the server to the client.

Question

Which of the following statements about using sockets is FALSE?

- A** Before connecting, the client must know the server's IP address and port number but not vice versa.
- B** For each connection, the client's and server's port numbers must match.
- C** The server must be running before the client can successfully connect.
- D** The server uses a socket solely to listen for connection requests and creates a completely new socket connection each time it accepts a connection.

(B)

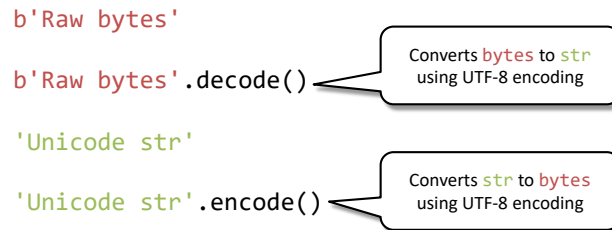
3 Unicode and Encodings

Sockets work at a very basic level. They can only send and receive data in the form of raw bytes. In other words, we must be able to encode the data into a sequence of 8-bit characters using Python's bytes type.

In Python, a string can be converted into bytes using the `str.encode()` method and the transmitted bytes can be decoded using the `bytes.decode()` method.

This encoding and decoding is necessary as internally, a Python string is actually treated as a sequence of numbers called Unicode **code points**. There are over a million possible code points, so it is not always possible to represent each code point using just 8 bits. Instead, the Unicode standard defines an encoding called **UTF-8** so code points can be represented using bytes in a space-efficient and consistent manner.

To enter a sequence of bytes directly in code, we can use a byte literal that starts with the letter **b**, followed by a sequence of bytes (in the form of ASCII characters) enclosed in matching single or double quotes. Note that most escape codes that work for string literals will also work for byte literals.

**Question**

The Chinese character 中 can be written as the string literal `'\u4e2d'` in Python. This uses an escape code that produces a character by specifying its Unicode code point.

Use Python to evaluate `len('\u4e2d')`. What is the result?

- A An error
- B 1
- C 2
- D 3

(B)

Question

Use Python to evaluate `len('\u4e2d'.encode())`. What is the result now?

- A An error
- B 1
- C 2
- D 3

(D)

This shows that the Unicode code point for 中 is represented by 3 bytes in UTF-8.

4 The Python socket Module

You can create and manage sockets in Python by importing the **socket** module and creating **socket** objects. The methods of the **socket** class are summarised below:

Methods	Description
<code>bind((host, port))</code>	Binds socket object to the given address tuple (host , port), where host is an IPv4 address and port is a port number.

<code>listen()</code>	Enables socket to listen for incoming connections from clients.
<code>accept()</code>	Waits for an incoming connection and returns a tuple containing a new socket object for the connection and an address tuple (host , port), where host is the IPv4 address of the connected client and port is its port number.
<code>connect((host, port))</code>	Initiates a connection to the given address tuple (host , port), where host is the IPv4 address of the server and port is its port number.
<code>recv(max_bytes)</code>	Receives and returns up to the given number of bytes from the socket.
<code>sendall(bytes)</code>	Sends the given bytes to the socket.

5 Establishing a Socket Connection

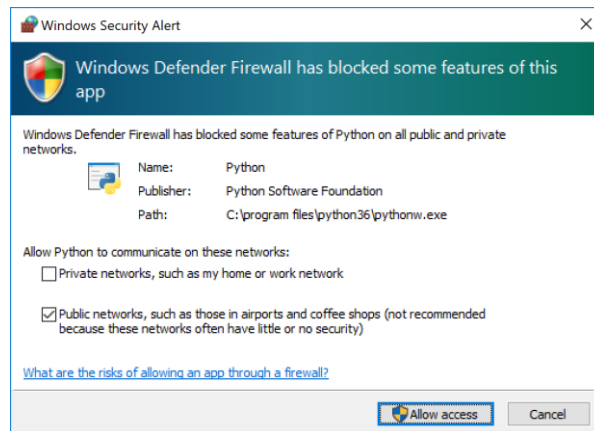
Create the following basic server program that listens for a client on port **12345**, accepts a connection request, sends `b'Hello from server\n'` to the client through the socket, then closes the socket.

Program 1: basic_server.py	
1	<code>import socket</code>
2	
3	<code># Create socket object and bind it to the given IP address and port</code>
4	<code>my_socket = socket.socket()</code>
5	<code>my_socket.bind(('127.0.0.1', 12345))</code>
6	
7	<code># Listen for connection</code>
8	<code>my_socket.listen()</code>
9	
10	<code># Establish new socket connection</code>
11	<code>new_socket, addr = my_socket.accept()</code>
12	
13	<code>print(f'Connected to {str(addr)}')</code>
14	
15	<code># Send data to client</code>
16	<code>new_socket.sendall(b'Hello from server\n')</code>
17	
18	<code># Terminate current active socket</code>
19	<code>new_socket.close()</code>
20	
21	<code># Close the passive listening socket</code>
22	<code>my_socket.close()</code>

Instead of **12345** on line 5, any large port number can be used. This number must be decided ahead of time for the client (written later) to use when attempting to establish a connection.

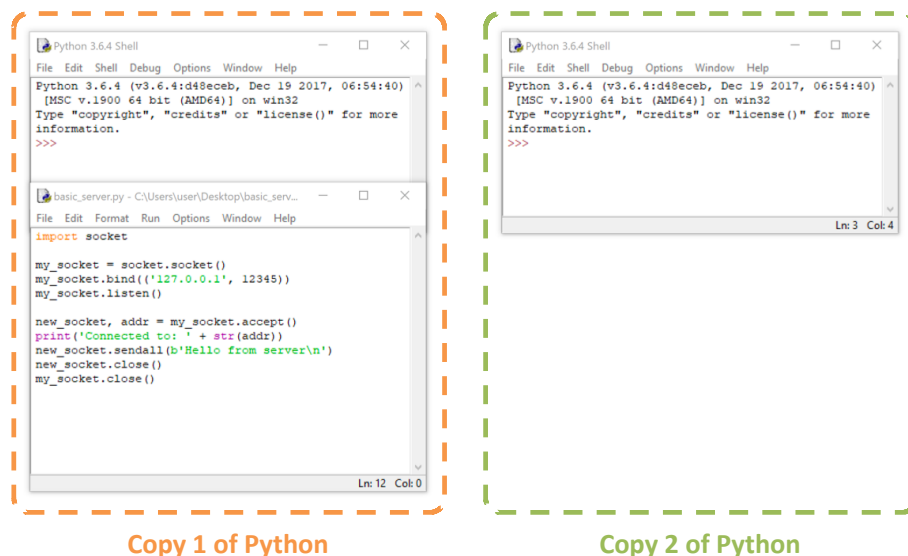
In line 11, `socket.accept()` returns a tuple of the newly created socket and a nested address tuple. Both the new socket and the address tuple will be stored in two variables **new_socket** and **addr** respectively, where **new_socket** is the socket used to send and receive data.

Run this program. If a firewall is running and has not been configured, you may be asked to grant Python network access at this point. Click "Allow access" if you are an administrator and wish to accept connection requests from other computers. Otherwise, click "Cancel".



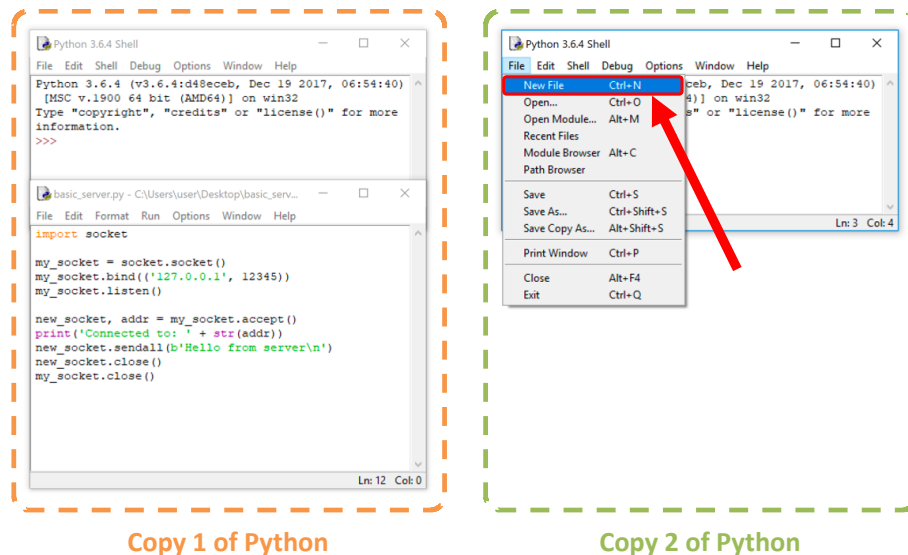
If everything is working correctly, the server should appear “stuck” shortly after it is started. This is because the `socket.accept()` method is **blocking**¹ the program and preventing it from continuing until a connection request is received.

To create a client that can connect to this server, start a second copy of Python. For instance, if you use IDLE on Windows, open the Start Menu and run IDLE again. A good practice is to move any windows from the first copy of Python to one side so the two copies of Python are clearly separated.



Create a new Python program using the second copy of Python. If you use IDLE, select "New File" using the shell window that is not running the server. You will need to know how to differentiate between the two Python shell windows.

¹ A "blocked" process means that it is waiting for some event to occur.



In the window that appears, enter the following basic client program that asks for the server's IP address and port number, requests for a connection, receives and prints at most 1024 bytes from the server, then closes the socket.

Program 2: basic_client.py

```

1 import socket
2
3 # Create new socket object and specify IP and port for connecting to
4 my_socket = socket.socket()
5 address = input('Enter IPv4 address of server: ')
6 port = int(input('Enter port number of server: '))
7
8 # Request for connection
9 my_socket.connect((address, port))
10
11 # Receive data from server
12 print(my_socket.recv(1024))
13
14 # Terminate connection
15 my_socket.close()

```

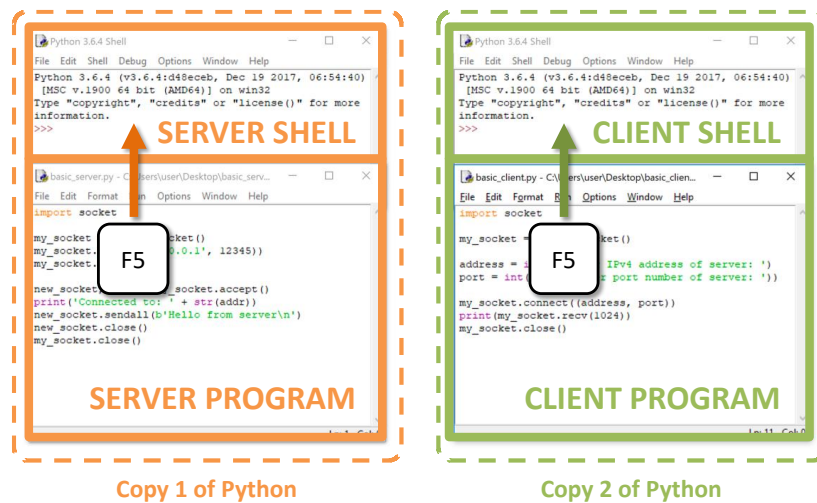
The argument for `socket.recv()` is required and should be set to a relatively small power of 2. In this case, we use the value of 1024 (2^{10}).

For more information, see: <https://docs.python.org/3/library/socket.html#socket.socket.recv>

Run this program using the second copy of Python and make sure the server you started previously is still running.

For instance, if you use IDLE, check that there are two shell windows running two different programs simultaneously. Otherwise, it is likely that you had accidentally stopped the server when starting the client.

If this happens, close the client, restart the server, then reopen the client using the second shell window. If things are set up correctly, each program should affect a different shell window when it is run (e.g., by pressing F5).



At this point, the client should be prompting you for the address and port number of the server. Use **127.0.0.1** (local machine) as the address and **12345** as the port number.

The client should successfully connect to the server and print out the bytes that were received. At the same time, the server program should become “unstuck” and end normally.

The left screenshot shows the server program output in a Python 3.6.4 Shell window. It displays the restart command, the connection to ('127.0.0.1', 50831), and the successful execution of the server program. The right screenshot shows the client program output in a Python 3.6.4 Shell window. It displays the restart command, the prompts for the server address and port, the user input of '127.0.0.1' and '12345', and the successful execution of the client program which receives 'b'Hello from server\n'.

Exercise 1

Recall that once a socket connection is established, it is symmetrical and can transfer data in both directions. In learning how to establish a socket connection, we have successfully sent data from the server to the client.

Now, write your own server and client to demonstrate that data can be sent in the opposite direction. Specifically, the client should send `b'Hello from client\n'` to the server and the server should print out any bytes that are received from the client.

Keep in mind that `socket.recv()` will block the program by default and prevent it from continuing until at least 1 byte is received.)

Program 3: practice_server.py

```

1 import socket
2
3 my_socket = socket.socket()
4 my_socket.bind(('0.0.0.0', 12345))
5 my_socket.listen()
6
7 new_socket, addr = my_socket.accept()
8 print(f'Connected to: {str(addr)}')
9
10 # Receive data from client
11 print(new_socket.recv(1024))
12
13 new_socket.close()
14
15 my_socket.close()

```

Program 4: practice_client.py

```

1 import socket
2
3 my_socket = socket.socket()
4 address = input('Enter IPv4 address of server: ')
5 port = int(input('Enter port number of server: '))
6
7 my_socket.connect((address, port))
8
9 # Send data to server
10 my_socket.sendall(b'Hello from client\n')
11
12 my_socket.close()

```

6 Designing a Protocol

The `basic_server.py` and `basic_client.py` programs from the previous section have a hidden flaw: when using the basic server program to send longer sequences of bytes, only part of the data may be successfully transmitted even if we increase the maximum number of bytes that `socket.recv()` can receive.

To understand why, we consider the situation where the sequence of bytes being sent is long enough such that it needs to be broken down to be sent in multiple packets.

We can simulate this by breaking the sequence into two pieces and calling `socket.sendall()` twice, once for each piece.

To simulate a busy network that may delay transport of the second packet, we also import the `time` module and call `time.sleep()` before sending the second piece.

Program 5: basic_server_split.py

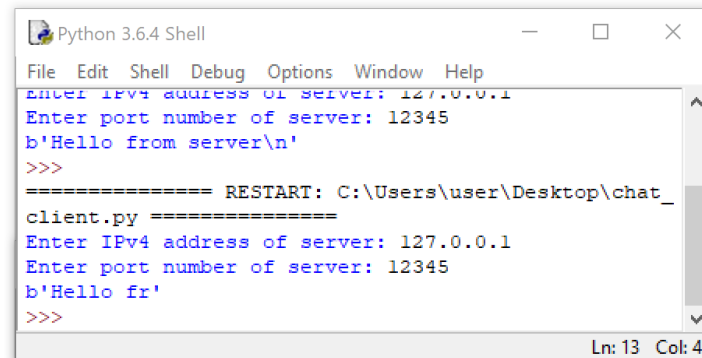
```

1 import socket
2 import time
3
4 my_socket = socket.socket()
5 my_socket.bind(('127.0.0.1', 12345))
6 my_socket.listen()
7
8 new_socket, addr = my_socket.accept()
9
10 # Send first part of message
11 new_socket.sendall(b'Hello fr')
12
13 # Simulate delay of 0.1 seconds
14 time.sleep(0.1)
15
16 # Send second part of message
17 new_socket.sendall(b'om server\n')
18
19 new_socket.close()
20
21 my_socket.close()

```

Run this version of the server. Then run the client such that both programs run simultaneously on the same machine. Use **127.0.0.1** as the address and **12345** as the port number when prompted.

This time, the client should receive only the first piece of data. If the client has closed the socket, the server may also produce an error when trying to send the second piece of data.



```

Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Enter IPv4 address of server: 127.0.0.1
Enter port number of server: 12345
b'Hello from server\n'
>>>
===== RESTART: C:\Users\user\Desktop\chat_
client.py =====
Enter IPv4 address of server: 127.0.0.1
Enter port number of server: 12345
b'Hello fr'
>>>
Ln: 13 Col: 4

```

This example illustrates that, in general, we should never assume that **socket.recv()** will receive all the bytes that were sent over at one go.

The only way to be certain that any received data is complete is to agree beforehand on a **protocol** or set of rules for how communication should take place.

For instance, we can agree beforehand that any data we transmit will always end with a newline character **\n** and that the data itself will never contain the **\n** character. This very simple protocol allows us to detect the end of a transmission easily by just searching for the **\n** character.

The following program updates the client so that it uses the `\n` character to detect when the message ends. This new client calls `socket.recv()` continuously and appends the received bytes to a variable named `data` until the `\n` character is encountered.

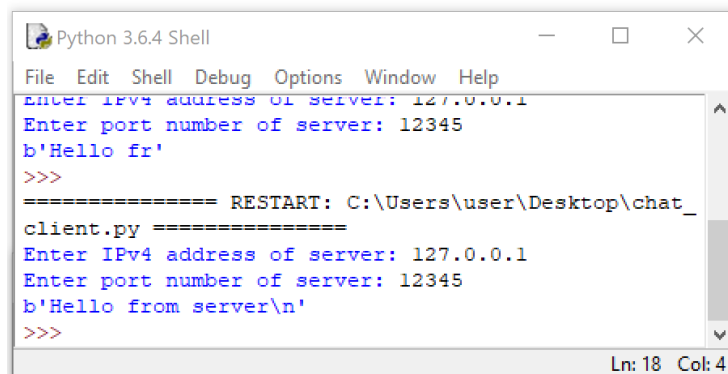
Program 6: basic_client_protocol.py

```

1  import socket
2
3  my_socket = socket.socket()
4  address = input('Enter IPv4 address of server: ')
5  port = int(input('Enter port number of server: '))
6
7  my_socket.connect((address, port))
8
9  # Start empty b-string to store data received
10 data = b''
11
12 # Loop to continuously receive data until '\n' is received
13 while b'\n' not in data:
14     data += my_socket.recv(1024)
15
16 # Output data received
17 print(data)
18
19 my_socket.close()

```

With this new client, all the data sent by the server up to and including the `\n` character is successfully received and printed.



```

Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Enter IPv4 address of server: 127.0.0.1
Enter port number of server: 12345
b'Hello fr'
>>>
===== RESTART: C:\Users\user\Desktop\chat_
client.py =====
Enter IPv4 address of server: 127.0.0.1
Enter port number of server: 12345
b'Hello from server\n'
>>>
Ln: 18 Col: 4

```

Question

A remote control program provides movement instructions to a robot program using a socket connection. The two programs communicate using a protocol where bytes are sent in one direction only from the remote control to the robot.

There are only four valid instructions that can be sent from the remote control. These instructions (in bytes literal form) are:

1. `b'FORWARD\n'`
2. `b'LEFT\n'`
3. `b'RIGHT\n'`
4. `b'END\n'`

According to the protocol, the `b'FORWARD\n'`, `b'LEFT\n'` and `b'RIGHT\n'` instructions can be sent in any order and repeated any number of times. The `b'END\n'` instruction, on the other hand, must always be the last instruction sent by the remote control, after which the socket connection must be terminated by both sides.

Assuming that the protocol is followed exactly, which sequence of bytes (in literal form) may be received by the robot when `socket.recv()` is called?

- A** `b'ND\nLEFT\nLEFT\nFORWARD\nRIGHT\nRIGHT'`
- B** `b'WARD\nFORWARD\nFORWARD\nRIGHT\nEND\n'`
- C** `b'RIGHT\nBACK\nFORWARD\nRIGHT\nRIGHT\nE'`
- D** `b'\nLEFT\nLEFT\nFORWARD\nRIGHT\nLEFTFOR'`

()

7 Iterative and Concurrent Servers

Currently, the server program exits immediately after it finishes working with a client.

However, in reality, we often want the server program to run continuously so that it is always listening and available for multiple clients to send connection requests. We can do this by putting the code that deals with a client in an infinite loop.

Program 7: basic_server_iterative.py

```

1  import socket
2
3  my_socket = socket.socket()
4  my_socket.bind(('127.0.0.1', 12345))
5  my_socket.listen()
6
7  while True:
8      new_socket, addr = my_socket.accept()
9      new_socket.sendall(b'Hello from server\n')
10     new_socket.close()

```

To interrupt a program that is running in an infinite loop, press **Ctrl** + **C**. In IDLE, we can also restart the shell using **Ctrl** + **F6**.

Internally, the server's passive socket keeps a queue of connection requests that have been received. A request is removed from this queue each time **socket.accept()** is called to create a connection.

If the queue is empty, **socket.accept()** will block the program until a connection request is received, as expected.

As **socket.accept()** is called each time the infinite loop repeats, the program will be able to handle multiple clients by processing them one at a time. This means that the program will work as an **iterative server**. Iterative servers are easy to write but limited as they can only handle one client at a time.

Alternatively, the server can be coded such that it starts a **thread** that runs simultaneously with the main program each time a client tries to connect. This makes the program more complicated to write but will be able to handle multiple clients at the same time, hence making it a **concurrent server**.

In 9569 H2 Computing, we shall deal only with the creation of iterative servers.

7.1 Sample Iterative Server Creation 1 – Writing a Chat Program

In this section, we shall create a simple chat client and server where two users can take turns to send single lines of text to each other. One user would be the server and the other the client.

Since each message is restricted to a single line, the newline character `\n` will never be part of a message. This means that we can use `\n` to detect the end of a message as a protocol.

We shall use a different port number, **6789**, to create the following chat server program that repeatedly prompts the user for some text, sends that text to the client (after encoding it into bytes), then receives and prints out the client's response.

Program 8: chat_server.py

```

1  import socket
2
3  listen_socket = socket.socket()
4  listen_socket.bind(('127.0.0.1', 6789))
5  listen_socket.listen()
6
7  chat_socket, addr = listen_socket.accept()
8
9  while True:
10     data = input('INPUT SERVER: ').encode()
11     chat_socket.sendall(data + b'\n')
12
13     print('WAITING FOR CLIENT...')
14
15     data = b''
16     while b'\n' not in data:
17         data += chat_socket.recv(1024)
18     print('CLIENT WROTE: ' + data.decode())

```

The client program is similar, except the order of sending and receiving is reversed.

Program :9 chat_client.py

```

1  import socket
2
3  chat_socket = socket.socket()
4
5  address = input('Enter IPv4 address of server: ')
6  port = int(input('Enter port number of server: '))
7
8  chat_socket.connect((address, port))
9
10 while True:
11     print('WAITING FOR SERVER...')
12
13     data = b''
14     while b'\n' not in data:
15         data += chat_socket.recv(1024)
16     print('SERVER WROTE: ' + data.decode())
17
18     data = input('INPUT CLIENT: ').encode()
19     chat_socket.sendall(data + b'\n')

```

Run the server and client using two different copies of Python. Once again, since the server is running on the same machine as the client, we can use 127.0.0.1 as the server's IPv4 address and 6789 as the port number.

```

Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Type Copyright, Credits or license() for more
information.
>>>
===== RESTART: C:\Users\user\Desktop\chat_
server.py =====
INPUT SERVER: How are you doing?
WAITING FOR CLIENT...
CLIENT WROTE: Great!

INPUT SERVER:

import socket

listen_socket = socket.socket()
listen_socket.bind(('127.0.0.1', 6789))
listen_socket.listen()

chat_socket, addr = listen_socket.accept()
while True:
    data = input('INPUT SERVER: ').encode()
    chat_socket.sendall(data + b'\n')
    print('WAITING FOR CLIENT...')
    data = b''
    while b'\n' not in data:
        data += chat_socket.recv(1024)
    print('CLIENT WROTE: ' + data.decode())
Ln: 7 Col: 0

```

```

Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
===== RESTART: C:\Users\user\Desktop\chat_
client.py =====
Enter IPv4 address of server: 127.0.0.1
Enter port number of server: 6789
WAITING FOR SERVER...
SERVER WROTE: How are you doing?

INPUT CLIENT: Great!
WAITING FOR SERVER...

import socket

chat_socket = socket.socket()

address = input('Enter IPv4 address of server: ')
port = int(input('Enter port number of server: '))

chat_socket.connect((address, port))
while True:
    print('WAITING FOR SERVER...')
    data = b''
    while b'\n' not in data:
        data += chat_socket.recv(1024)
    print('SERVER WROTE: ' + data.decode())
    data = input('INPUT CLIENT: ').encode()
    chat_socket.sendall(data + b'\n')
Ln: 12 Col: 0

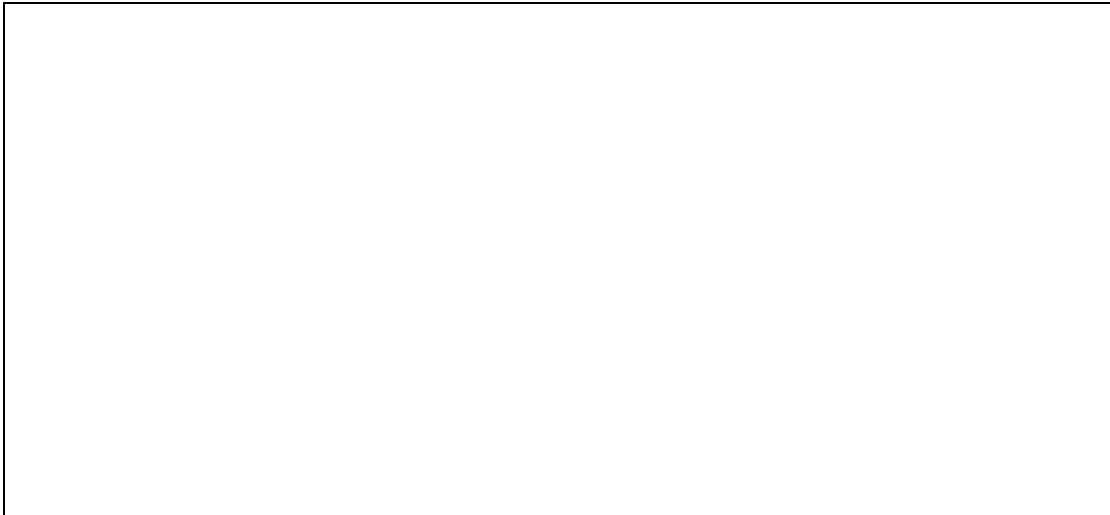
```

- 8 Currently, there is no way to exit our chat programs other than to press Ctrl-C or to restart the shell (in IDLE).

For this question, modify chat_server.py and chat_client.py so that both programs exit once the message 'quit' is sent by any user. Remember to make sure that all sockets are closed properly before exiting.

Program: chat_quit_server.py

Program: chat_quit_client.py



Writing a Turn-Based Game

So far, we have been responsible for writing both the server and client programs. Sometimes, however, both server and protocols designs may be based on an existing standard or developed by someone else. To write a client that can communicate with an existing server, we need to study its code and follow the expected protocol.

Conversely, sometimes the client may be developed by someone else and we need to write a server to communicate with it. In either case, it is important to start by understanding the protocol being used.

To demonstrate how to do this, let us examine the server program for a simple turn-based 2-player game of Tic-Tac-Toe. First, we create a simple library that defines some constants and a TicTacToe class to handle the game logic:

Program 8: tictactoe.py

```

1  N = 3                                # Size of grid
2  WIDTH = len(str(N ** 2))             # Width for each cell
3  PLAYERS = ('O', 'X')                 # Player symbols
4
5  class TicTacToe:
6
7      def __init__(self):
8          self.board = []
9          for i in range(N):
10             self.board.append([None] * N)
11
12     def render_row(self, row_index):
13         start = row_index * N + 1
14         row = self.board[row_index].copy()
15         for column_index in range(N):
16             if row[column_index] is None:
17                 cell = str(start + column_index)
18             else:
19                 cell = PLAYERS[row[column_index]]

```

```

20         if len(cell) < WIDTH:
21             cell += ' ' * (WIDTH - len(cell))
22             row[column_index] = ' ' + cell + ' '
23         return '|'.join(row) + '\n'
24
25     def render_board(self):
26         rows = []
27         for row_index in range(N):
28             rows.append(self.render_row(row_index))
29         divider = '-' * ((WIDTH + 3) * N - 1) + '\n'
30         return divider.join(rows)
31
32     def make_move(self, player_index, cell_index):
33         cell_index -= 1
34         self.board[cell_index // N][
35             cell_index % N] = player_index
36
37     def is_valid_move(self, cell_index):
38         if cell_index < 1 or cell_index > N ** 2:
39             return False
40         cell_index -= 1
41         return self.board[cell_index // N][
42             cell_index % N] is None
43
44     def is_full(self):
45         for row_index in range(N):
46             for column_index in range(N):
47                 if self.board[row_index][
48                     column_index] is None:
49                     return False
50         return True
51
52     def get_winner(self):
53         # Check diagonals
54         if self.board[0][0] is not None:
55             found = True
56             for i in range(N):
57                 if self.board[0][0] != self.board[i][i]:
58                     found = False
59                     break
60             if found:
61                 return self.board[0][0]
62         if self.board[0][N - 1] is not None:
63             found = True
64             for i in range(N):
65                 if self.board[0][N - 1] != self.board[i][
66                     N - i - 1]:
67                     found = False
68                     break
69             if found:
70                 return self.board[0][N - 1]
71
72         # Check rows and columns
73         for i in range(N):

```

```

74         if self.board[i][0] is not None:
75             found = True
76             for j in range(N):
77                 if self.board[i][0] != self.board[i][j]:
78                     found = False
79                     break
80             if found:
81                 return self.board[i][0]
82         if self.board[0][i] is not None:
83             found = True
84             for j in range(N):
85                 if self.board[0][i] != self.board[j][i]:
86                     found = False
87                     break
88             if found:
89                 return self.board[0][i]
90
91         # No matching lines were found, so no winner
92         return None

```

The following is a summary of the methods in TicTacToe class:

Methods	Description
render_row(row_index)	Returns a string representation of the specified row, such as: 1 2 3
render_board()	Returns a string representation of the entire board, such as: 1 2 3 ----- 4 5 6 ----- 7 8 9
make_move(player_index, cell_index)	Modifies the board such that the specified cell is marked with the symbol for the specified player
is_valid_move(cell_index)	Returns whether the specified cell is currently blank
is_full()	Returns whether the entire board has been filled up
get_winner()	Returns winning player for the current board or None if there is no winner

Using this library, we create a server program that creates a TicTacToe object on line 9 to store information about the Tic-Tac-Toe board:

Program 9: game_server.py

```

1  import socket
2  import tictactoe
3
4  listen_socket = socket.socket()
5  listen_socket.bind(('127.0.0.1', 3456))
6  listen_socket.listen()
7
8  game_socket, addr = listen_socket.accept()
9  game = tictactoe.TicTacToe()
10 while True:
11     # Display current Tic-Tac-Toe board
12     print(game.render_board())
13
14     # Check if client player won
15     if game.get_winner() is not None:
16         print('Opponent wins!')
17         print()
18         break
19
20     # Check if board is full
21     if game.is_full():
22         print('Stalemate')
23         print()
24         break
25
26     # Prompt for move from server player
27     move = -1
28     while move != 0 and not game.is_valid_move(move):
29         move = int(input('Server moves ' +
30                         '(0 to quit): '))
31     print()
32     if move == 0:
33         game_socket.sendall(b'END\n')
34         print('You quit, opponent wins!')
35         print()
36         break
37     game.make_move(0, move)
38     game_socket.sendall(b'MOVE' +
39                       str(move).encode() + b'\n')
40
41     # Display current Tic-Tac-Toe board
42     print(game.render_board())
43
44     # Check if server player won
45     if game.get_winner() is not None:
46         print('You win!')
47         print()
48         break
49
50     # Check if board is full
51     if game.is_full():
52         print('Stalemate')
53         print()
54         break

```

```

55
56     # Receive move from client player
57     received = b''
58     while b'\n' not in received:
59         received += game_socket.recv(1024)
60     if received.startswith(b'MOVE'):
61         move = int(received[4:])
62         print('Client moves: ' + str(move))
63         print()
64         game.make_move(1, move)
65     elif received.startswith(b'END'):
66         print('Opponent quits, you win!')
67         print()
68         break
69
70 game_socket.close()
71 listen_socket.close()

```

Analysing this server code, we see that communications with the client is divided into several steps that repeat in an infinite loop:

1. Display current Tic-Tac-Toe board
2. Check if opponent has won, and if so, end game with opponent winning
3. Check if the board is full, and if so, end game with a stalemate
4. Prompt for input from player; if player makes a valid move, update game board accordingly, then send b'MOVE' followed by the chosen cell number and b'\n' to the opponent; if player chooses to quit, send b'END\n' to the opponent and end game with the opponent winning
5. Display current Tic-Tac-Toe board again
6. Check if player has won, and if so, end game with player winning
7. Check if the board is full, and if so, end game with a stalemate
8. Receive opponent's action via the socket; if the action is b'MOVE' followed by a cell number and b'\n', update game board accordingly; if the action is b'END\n', end game with the player winning

As written, the server player always starts first. This means that our client code should start by receiving and processing the server's result. We also know that Tic-Tac-Toe is a symmetrical game (other than the choice of starting player), so we deduce that the client code should be similar to the server code except that "client" and "server" are exchanged and the last step is moved to the front:

1. Receive opponent's action via the socket; if the action is b'MOVE' followed by a cell number and b'\n', update game board accordingly; if the action is b'END\n', end game with the player winning
2. Display current Tic-Tac-Toe board
3. Check if opponent has won, and if so, end game with opponent winning
4. Check if the board is full, and if so, end game with a stalemate
5. Prompt for input from player; if player makes a valid move, update game board accordingly, then send b'MOVE' followed by the chosen cell number and b'\n' to the opponent; if player chooses to quit, send b'END\n' to the opponent and end game with the opponent winning
6. Display current Tic-Tac-Toe board again
7. Check if player has won, and if so, end game with player winning

8. Check if the board is full, and if so, end game with a stalemate

A client program that does this is as follows:

Program 10: game_client.py

```

1  import socket
2  import tictactoe
3
4  game_socket = socket.socket()
5  game_socket.connect(('127.0.0.1', 3456))
6
7  game = tictactoe.TicTacToe()
8  while True:
9      # Receive move from server player
10     received = b''
11     while b'\n' not in received:
12         received += game_socket.recv(1024)
13     if received.startswith(b'MOVE'):
14         move = int(received[4:])
15         print('Server moves: ' + str(move))
16         print()
17         game.make_move(0, move)
18     elif received.startswith(b'END'):
19         print('Opponent quits, you win!')
20         print()
21         break
22
23     # Display current Tic-Tac-Toe board
24     print(game.render_board())
25
26     # Check if server player won
27     if game.get_winner() is not None:
28         print('Opponent wins!')
29         print()
30         break
31
32     # Check if board is full
33     if game.is_full():
34         print('Stalemate')
35         print()
36         break
37
38     # Prompt for move from client player
39     move = -1
40     while move != 0 and not game.is_valid_move(move):
41         move = int(input('Client moves ' +
42                         '(0 to quit): '))
43     print()
44     if move == 0:
45         game_socket.sendall(b'END\n')
46         print('You quit, opponent wins!')
47         print()
48         break

```

```
49     game.make_move(1, move)
50     game_socket.sendall(b'MOVE' +
51                         str(move).encode() + b'\n')
52
53     # Display current Tic-Tac-Toe board
54     print(game.render_board())
55
56     # Check if client player won
57     if game.get_winner() is not None:
58         print('You win!')
59         print()
60         break
61
62     # Check if board is full
63     if game.is_full():
64         print('Stalemate')
65         print()
66         break
67
68     game_socket.close()
```

Run the server and client using two different copies of Python on the same machine to verify that the game works as expected. A sample run is also provided below:

```

===== RESTART: C:/Users/user/Desktop/game_
server.py =====
 1 | 2 | 3
-----
 4 | 5 | 6
-----
 7 | 8 | 9

Server moves (0 to quit): 8

 1 | 2 | 3
-----
 4 | 5 | 6
-----
 7 | 0 | 9

Client moves: 4

 1 | 2 | 3
-----
 X | 5 | 6
-----
 7 | 0 | 9

Server moves (0 to quit): 2

 1 | 0 | 3
-----
 X | 5 | 6
-----
 7 | 0 | 9

Client moves: 6

 1 | 0 | 3
-----
 X | 5 | X
-----
 7 | 0 | 9

Server moves (0 to quit): 5

 1 | 0 | 3
-----
 X | 0 | X
-----
 7 | 0 | 9

You win!

>>>
Ln: 70 Col: 4

```

```

===== RESTART: C:/Users/user/Desktop/game_
client.py =====
Server moves: 8

 1 | 2 | 3
-----
 4 | 5 | 6
-----
 7 | 0 | 9

Client moves (0 to quit): 4

 1 | 2 | 3
-----
 X | 5 | 6
-----
 7 | 0 | 9

Server moves: 2

 1 | 0 | 3
-----
 X | 5 | 6
-----
 7 | 0 | 9

Client moves (0 to quit): 6

 1 | 0 | 3
-----
 X | 5 | X
-----
 7 | 0 | 9

Server moves: 5

 1 | 0 | 3
-----
 X | 0 | X
-----
 7 | 0 | 9

Opponent wins!

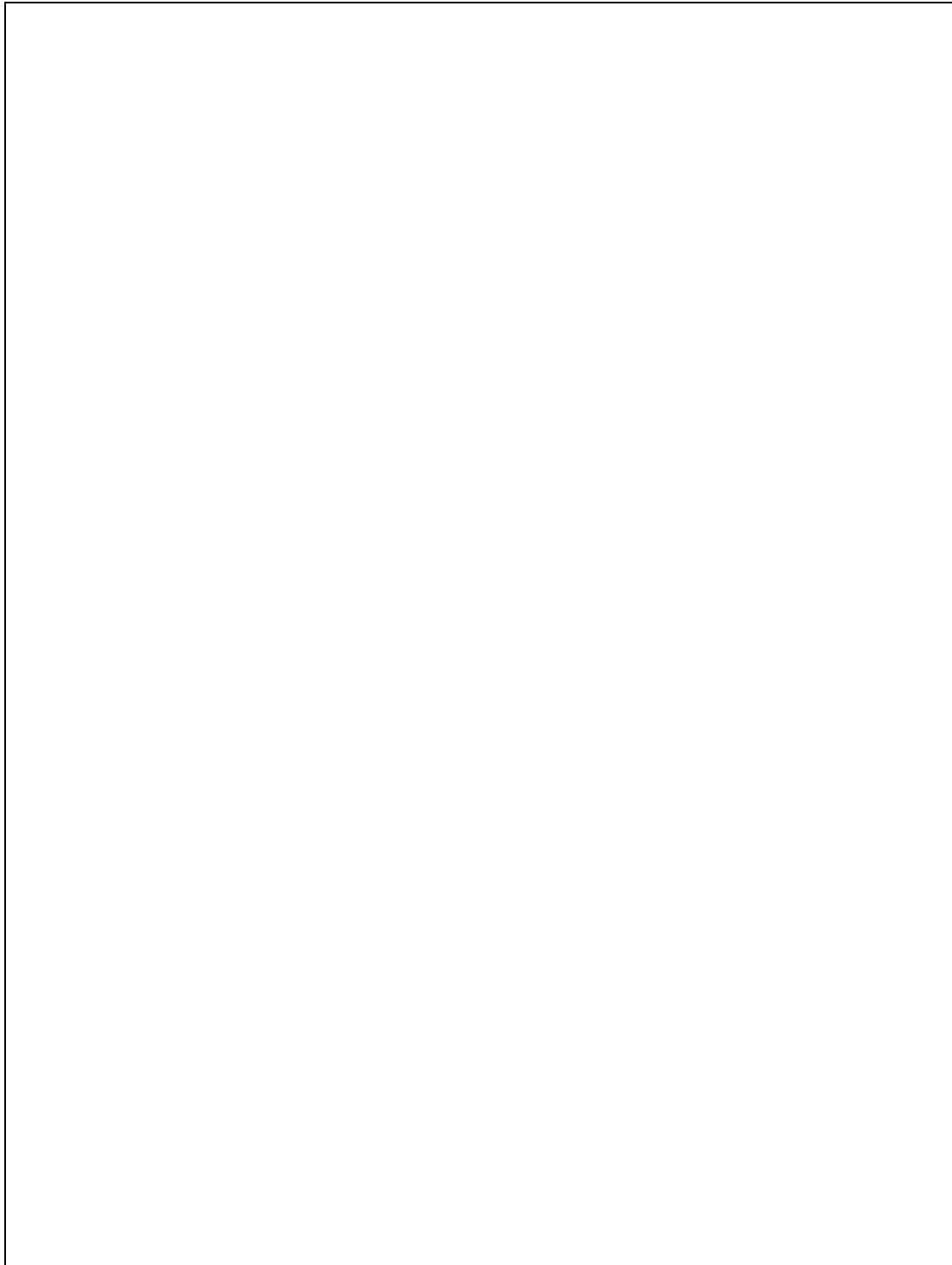
>>>
Ln: 59 Col: 11

```


- 9** Currently, the Tic-Tac-Toe game is written such that the server takes its turn first. For this question, modify `game_server.py` and `game_client.py` so that the client takes its turn first instead.

Program: game_server_alternative.py

Program: game_client_alternative.py



- 10** While Tic-Tac-Toe is a symmetrical game where the rules are the same for both players, other games may be asymmetrical and thus require the two players to behave differently from each other.

The following is a client program for an asymmetric guess-the-number game where a server generates a random number from 1 to 100 and a client tries to guess it within 5 tries. After each incorrect guess, the server returns whether the guess is greater than or less than the required number:

Program 11: `guess_client.py`

```
1 import socket
2
3 s = socket.socket()
4 s.connect(('127.0.0.1', 9999))
5
6 data = b''
7 while True:
8     while b'\n' not in data:
9         data += s.recv(1024)
10    received = data[:data.find(b'\n')]
11    data = data[len(received) + 1:]
12    if received == b'LOW':
13        print('Your guess is too low.')
14    elif received == b'HIGH':
15        print('Your guess is too high.')
16    elif received == b'GUESS':
17        guess = int(input('Enter guess (1-100): '))
18        s.sendall(str(guess).encode() + b'\n')
19    elif received == b'WIN':
20        print('You win!')
21        break
22    elif received == b'GAMEOVER':
23        print('You ran out of tries! Game over.')
24        break
25
26 s.close()
```

Write the corresponding server program.

Program: `guess_server.py`