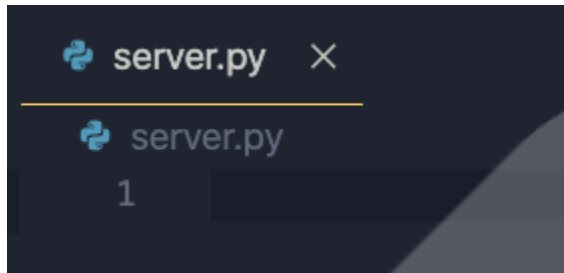


Topic	SOCKETS - SERVER	
Class Description	Students will learn about sockets and get into socket programming with Python	
Class	C-199	
Class time	45 mins	
Goal	<ul style="list-style-type: none"> <li>Understand about what sockets are</li> <li>Understanding about threads</li> <li>Building socket server in Python</li> </ul>	
Resources Required	<ul style="list-style-type: none"> <li>Teacher Resources:               <ul style="list-style-type: none"> <li>Laptop with internet connectivity</li> <li>Earphones with mic</li> <li>Notebook and pen</li> <li>Visual Studio Code</li> </ul> </li> <li>Student Resources:               <ul style="list-style-type: none"> <li>Laptop with internet connectivity</li> <li>Earphones with mic</li> <li>Notebook and pen</li> <li>Visual Studio Code</li> </ul> </li> </ul>	
Class structure	<b>Warm-Up</b> <b>Teacher - led Activity 1</b> <b>Student - led Activity 1</b> <b>Wrap-Up</b>	<b>10 mins</b> <b>10 mins</b> <b>20 mins</b> <b>5 mins</b>
WARM UP SESSION - 10mins		
Teacher Action		Student Action
<i>Hey &lt;student's name&gt;. How are you? It's great to see you!            Are you excited to learn something new today?</i>		<b>ESR:</b> Hi, thanks, Yes I am excited about it!
Q&A Session		

Question	Answer
Which Machine requests to access the data?  A. Server B. Client C. WWW D. None of these	B
DNS, IMPS, FTP, TCP/IP are the examples of _____ ?  A. Port B. Socket C. Data D. Protocol	D
<b>TEACHER-LED ACTIVITY - 10mins</b>	
<b>Teacher Initiates Screen Share</b>	
<b>ACTIVITY</b> <ul style="list-style-type: none"> <li>• Learning about Sockets and Threads</li> <li>• Building socket server with Python</li> </ul>	
Teacher Action	Student Action
In the last class, we saw how sockets can be a combination of an IP Address and a Port!  Do you remember what TCP Ports are?	<b>ESR:</b> Varied!
Sockets allow communication between two different processes (or applications) on the same or different machines.  They are used in a <b>client-server application framework</b> .	
<b>Servers</b> are backend applications (or processes) that perform some functions on a request from the client.	

<p>Remember how we learnt about different HTTP Requests? Can you tell me a few of them?</p>	<p><b>ESR:</b></p> <ul style="list-style-type: none"> <li>• GET</li> <li>• POST</li> <li>• PUT</li> <li>• DELETE</li> </ul>
<p>Great!</p> <p>Now when you surf to a website on the browser, the browser is making a GET request to the server of that website, which returns data. This data could be in the form of a JSON, on HTML content.</p> <p>In this scenario, your browser running in your machine is the <b>Client</b>.</p> <p>Both HTTP and HTTPS use sockets to transfer data back and forth between server and client, therefore whenever your browser makes a request to the website's server, it's establishing a connection through sockets.</p>	
<p>Now do note that the same website can be accessed by multiple browsers running on different machines at the same time.</p> <p>This is one of the attributes of the <b>server</b>. It can establish connections to multiple clients at a time.</p>	
<p>Although websites running on HTTP or HTTPS use sockets behind the scenes, one of the most classic uses of sockets is for building chat apps!</p> <p>That's what we will be building to understand more about how sockets work!</p> <p>In a chat app, it will be required to build a <b>client</b> and a <b>server</b> and we can do it through <b>socket programming</b> in Python</p>	
<p>Let's start with the <b>Server</b>.</p>	

Teacher opens a new Visual Studio Code and creates a file called **server.py**



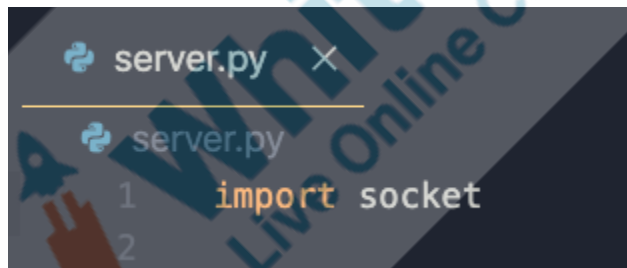
Python has a very famous and widely used library for creating sockets. It is known as **socket**.

You can find it's documentation in Student Activity 1

Teacher opens [Teacher Activity 1](#)

Student opens [Student Activity 1](#)

Let's start by importing that -



Next, we will create a simple socket -

```
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Here, we are creating a socket with the help of **socket.socket()** function.

This is a function that takes 2 main arguments -

1. **address family**

## 2. *socket type*

**address family** is the family of addresses that the socket can communicate with. Classic examples of 2 of the most famous address families are **IPv4** and **IPv6**.

**AF\_INET** represents **IPv4** while **AF\_INET6** represents **IPv6**.

**AF\_INET** is also the default value of the first argument, if not provided. That's because **IPv4** is still widely used while **IPv6** is relatively new.

For this type, we are using **SOCK\_STREAM**. It is the default value (if not provided) and it is used to create a **TCP Socket**. We could also use **SOCK\_DGRAM** which is used to create a **UDP Socket**, however use of it case specific.

Some of the major differences between a **TCP** and a **UDP** is -

TCP (Transmission Control Protocol)	UDP (User Datagram Protocol)
Tries to resend the packets that are lost	Doesn't resend the packets that are lost
Adds a sequence number to the packets and reorders them to ensure the packets do not arrive in wrong order	Packets can arrive in any order.
Slower, as it manages everything	Faster due to lack of features
Heavy to use as OS keeps track of ongoing communication sessions between clients and servers	Lighter to use on the machine
TCP is used by - <ul style="list-style-type: none"> <li>• HTTP</li> <li>• HTTPS</li> <li>• SMTP</li> <li>• FTP</li> <li>• etc.</li> </ul>	UDP is used by - <ul style="list-style-type: none"> <li>• DNS</li> <li>• DHCP</li> <li>• etc.</li> </ul>

As we can see in the table above, UDP doesn't have many use cases. In our case, since we are building a chat app, we do not want the messages to be received in the wrong order, or to not receive a few messages at all, therefore we are going with **SOCK\_STREAM** instead of **SOCK\_DGRAM**.

Next, we can define the IP Address and the Port we are using -

```
ip_address = '127.0.0.1'
port = 8000
```

Since our localhost IP Address is **127.0.0.1**, we will be using that here and we can go with port **8000**, however this port can be any number. Just make sure that it is not anything lower than **1,024**, since those are reserved ports.

Next, we want to bind our server with the IP Address and the Port that we want to use and then we are ready to listen for any incoming requests from the clients!

```
server.bind((ip_address, port))
server.listen()
```

Our **server** is the **socket** that we created earlier with the **socket.socket()** function and we are using the **bind()** function that takes a **tuple** with **ip\_address** and **port** in it.

Once our server is binded, we can start listening on this server socket with the **listen()** function.

Now multiple clients can connect on our server, therefore we would want to maintain a list of all the clients that are connected to the server at any given time, so let's create a list for that -

```
clients = []
```

In this list, we can store all the clients that are connected with the server at any given time.

We can also add a print statement to know that our server has started successfully -

```
print("Server is running...")
```

Our code until now looks like this -

```
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

ip_address = '127.0.0.1'
port = 8000

server.bind((ip_address, port))
server.listen()

clients = []

print("Server is running...")
```

Great! So now a server is created. Let's run this code in the terminal to see what we get?

`python3.8 server.py`

*Teacher runs the code in terminal*

**Server has started...**

We noticed that the server was started but the script was over right after it! Now, if we are to use our server to manage clients, it needs to keep running!

The job of our server is to listen to clients trying to make a request for as long as it's running. How can it be achieved?

**ESR:**  
By using a **while** loop

Excellent! Let's add a while loop then -

```
print("Server has started...")  
  
while True:
```

Inside this loop, the first thing that we want to do is to accept any connection request made by a client to our server. Now our **server** is a Socket object that we created above, and it has a special method called **accept()**.

This **accept()** method accepts any connection request made to the server and returns 2 parameters -

1. The socket object of the client that is trying to connect
2. Their IP Address and Port number in the form of a tuple

Let's add some code so our server can accept requests from clients -

```
while True:  
    conn, addr = server.accept()  
    list_of_clients.append(conn)  
    print (addr[0] + " connected")
```

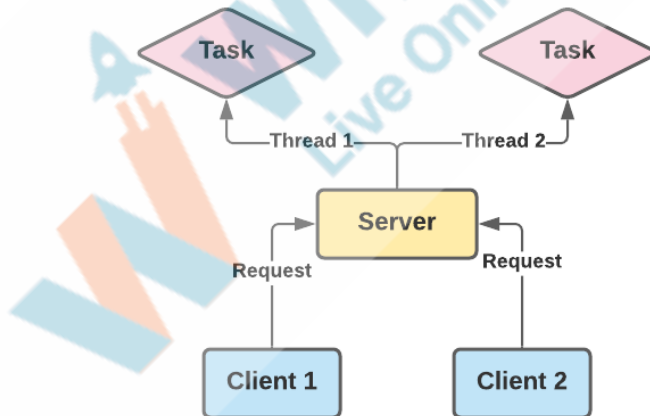
Here, first we are accepting any connection requests made to the server with **server.accept()** function, and then we are saving the 2 parameters in returns in **conn**, which is the socket object of the client that is trying to connect and **addr**, which is a tuple containing IP Address and Port Number of the client.

Since we have a client that tried connecting with the server, we can append it's socket object **conn** to our **list\_of\_clients** to use later.

Awesome! Now one more thing yet to be handled is multiple client connections! Let's think about it.

Multiple clients can connect with the same server at the same time! This way, we will have just one server instance running for multiple clients.



<p>Imagine a scenario where there are thousands of users connected to the same server, making a request that takes 5 seconds to execute. As for how Python runs the code, it will complete the request for the first user, and only then move to the second user. How slow would the service be!</p> <p>How do you think anyone can tackle this?</p>	<p><b>ESR:</b> Varied!</p>
<p>We can use threading! It allows different parts of the programs to run concurrently using threads where threads are a separate flow of execution.</p> <p>In the same scenario above, the function that takes 5 seconds to execute can be executed concurrently for each of the requests made on a separate thread.</p> <p>That way, all the requests would be handled separately on a different thread and can execute simultaneously!</p> <p>Take a look at the image below to understand it better -</p>	
<div data-bbox="393 1138 1039 1551" data-label="Diagram">  <pre> graph TD     C1[Client 1] -- Request --&gt; S[Server]     C2[Client 2] -- Request --&gt; S     S -- Thread 1 --&gt; T1{Task}     S -- Thread 2 --&gt; T2{Task} </pre> </div> <p>Here, we can see that if <b>Client 1</b> and <b>Client 2</b> make a <b>Request</b> at the same time to the <b>Server</b>, then our server can perform the task in 2 different Threads simultaneously to save time, without having to wait for the previous tasks to get finished.</p> <p>In our chat application, since we are dealing with multiple clients, we can use threads -</p>	

```
from threading import Thread
```

We will start by importing **Thread** from the **threading** module at the very top and then -

```
while True:
    conn, addr = server.accept()
    list_of_clients.append(conn)
    print (addr[0] + " connected")

    new_thread = Thread(target= clientthread, args=(conn, addr))
    new_thread.start()
```

We will create a variable **new\_thread** inside which we will use **Thread()**.

Now the **Thread()** uses 2 arguments, **target** and **args**.

**target** here, is the name of the function that we want this thread to execute, and **args** are the parameters that you may want to pass into that function.

In our case, we are calling a function **clientthread** with arguments (**conn, addr**). Don't worry, we are yet to create the function **clientthread**.

To start this thread, we can simply call the **start()** function on our newly created thread.

Now, let's create the function **clientthread()**.

In this function, the first thing that we'd like to do is to welcome the client to the chat app -

```
def clientthread(conn, addr):
    conn.send("Welcome to this chatroom!".encode('utf-8'))
```

Here, we have defined the function **clientthread()** that we were using in the thread, and we have passed **conn** and **addr** to the function.

Now, the client's socket object **conn** has a special function called **send()**, which can send any message to that client.

We are using this function to welcome the user to the chatroom. Do note that we are encoding the message with **utf-8**, since we are sending the data from server to the client.

Now, this function **clientthread()** will only be called once, that's when the client will try to make a connection request with the server.

The main use of this function is to receive the messages sent by the clients, and send those messages to the other clients connected.

For example, if you are sending a message to your friend, then it's the server's responsibility to receive the message first, and then send it to your friend.

Now since this will be an active chat, the messages will be sent and received at any time, therefore it only makes sense to add another **while True** loop here -

```
def clientthread(conn, addr):  
    conn.send("Welcome to this chatroom!".encode('utf-8'))  
    while True:  
        try:  
  
        except:  
            continue
```

Here, we have added the **while True** loop for the same reason, and inside this loop we have added a **try - except** block to ensure that the server doesn't break in case of an error.

Inside the **try** block, the first thing that we want to do is to receive any messages sent by the client. Let's do that -

```
while True:
    try:
        message = conn.recv(2048).decode('utf-8')
    except:
        continue
```

Just as we have the **send()** function to send a message, we have a **recv()** function to receive a message. **2048** is the maximum length of the message that can be received by the server. Any message exceeding **2048** characters will be stripped down to **2048** characters max.

Also, just how we were **encoding** our messages with **utf-8**, this time, we will be decoding the messages with the **decode()** function.

Now one thing to note is that if the user closes the **client** app, then too, the server will receive a message but there would be no content in the message.

That tells us that we will have to check if the message is there or not. We can do that with an **if-else** block -

```
def clientthread(conn, addr):
    conn.send('Welcome to this chatroom!'.encode('utf-8'))
    while True:
        try:
            message = conn.recv(2048).decode('utf-8')
            if message:
                # ... (code to handle the message)
            else:
                remove(conn)
        except:
            continue
```

Here, if the message is there, then we will perform something otherwise we will want to remove the client's socket object stored with us in our **client\_list**!

For that, let's assume we have a **remove()** function to do that. We will create it later.

Now, **if** the message is there, we will want to send this message to all the other clients connected with our server.

For example, if there are 3 clients connected to the server - Client 1, Client 2 and Client 3, and Client 2 sends a message, then we want to send this message to both Client 1 and Client 3.

For this, we can have a function called **broadcast()**, that can mass broadcast the message sent by any client -

```
def clientthread(conn, addr):  
    conn.send("Welcome to this chatroom!".encode('utf-8'))  
    while True:  
        try:  
            message = conn.recv(2048).decode('utf-8')  
            if message:  
                print("<" + addr[0] + "> " + message)  
  
                message_to_send = "<" + addr[0] + "> " + message  
                broadcast(message_to_send, conn)  
            else:  
                remove(conn)  
        except:  
            continue
```

Here, we are first printing the message that was sent by the client, we are then calling a **broadcast()** function with the message and client's socket object that sent the message.

Do note that we are yet to create the **remove()** and the **broadcast()** functions.

So far so good! Now, you will work on the other 2 functions **broadcast()** and **remove()** that we have used but not created.

Teacher Stops Screen Share	
STUDENT-LED ACTIVITY - 20 mins	
<ul style="list-style-type: none"> <li>Ask the student to press the ESC key to come back to the panel.</li> <li>Guide the student to start Screen Share.</li> <li>The teacher gets into Fullscreen.</li> </ul>	
<p align="center"><b><u>ACTIVITY</u></b></p> <ul style="list-style-type: none"> <li>Create function to broadcast messages</li> <li>Create function to remove client</li> </ul>	
Teacher Action	Student Action
<i>Guide the student to get the boilerplate code from Student Activity 2</i>	<i>Student clones the code from Student Activity 2</i>
<p>Alright, now let's start with the <b><i>broadcast()</i></b> function. What do you think should happen in this function? We have the message that was sent, and the client's socket object who sent that message.</p> <p>How we should do this is that we should iterate over all the client socket objects that we have stored in the list, and send the message to them which are not the client socket object that sent the message.</p> <p><i>Teacher helps the student in writing the code</i></p>	<p><b>ESR:</b> Varied!</p> <p><i>Student writes the code</i></p>
<pre>def broadcast(message, connection):     for clients in list_of_clients:         if clients!=connection:             try:                 clients.send(message.encode('utf-8'))             except:                 remove(clients)</pre>	

Here, we can see that inside the **broadcast()** function, we are iterating over all the client socket objects we have in **list\_of\_clients** list.

Inside this **for** loop, we have an **if** condition that checks if the socket object is the same as the one which sent the message or not. If not, then we have a **try-except** block where inside the **try** statement, we are trying to send the message with **utf-8** encoding using the **send()** function.

Now the only case that this will throw an error is if it's not able to find the client, which might be the case if the client closes their application.

Then, the code will go in the **except** statement where we are calling the **remove()** function.

Now all we have to do is to create the **remove()** function.

This is a very simple function in which we just have to remove the client's socket object from our list of clients.

*Teacher helps the student in writing the code*

*Student writes the code*

```
def remove(connection):
    if connection in list_of_clients:
        list_of_clients.remove(connection)
```

Here, we just have an **if condition** to check if the client that we want to remove is in the list or not? If it is, we are simply calling the **remove()** function of the lists to remove the client from it.

That's it! Our server is ready.

**Teacher Guides Student to Stop Screen Share**


**WRAP UP SESSION - 5 Mins**

**Quiz time - Click on in-class quiz**



**Question**

**Answer**



<p>Which type of two arguments do we need to pass to make a socket function?</p> <p>A. Address Family, Socket Type          B. Receive, Listen          C. Bind, Collect          D. TCP,UDP</p>	<b>A</b>
<p>What does bind method do in socket?</p> <p>A. Assign IP Address          B. Assign Port Number          C. Link IP Address &amp; Port          D. None of the above</p>	<b>C</b>
<p>Which one is the right way to receive messages from the client with a character limit of 4096?</p> <p>A. conn.receive(4096)          B. conn.recv()          C. conn.receive()          D. conn.recv(4096)</p>	<b>D</b>
<b>End the quiz panel</b>	
<p align="center"><b>FEEDBACK</b></p> <ul style="list-style-type: none"> <li>● <b>Appreciate the students for their efforts in the class.</b></li> <li>● <b>Ask the student to make notes for the reflection journal along with the code they wrote in today's class.</b></li> </ul>	
<b>Teacher Action</b>	<b>Student Action</b>
<p>You get Hats off for your excellent work!</p> <p>In the next class</p>	<p><i>Make sure you have given at least 2 Hats Off during the class for:</i></p> <div>  <div>           Creatively Solved Activities           <div>+10</div> </div> </div>



	<div>Great Question  +10</div> <div>Strong Concentration  +10</div>
<b>Project Discussion</b>	
Teacher Clicks	<div>✕ End Class</div>
<b>ADDITIONAL ACTIVITIES</b>	
<p><b>Additional Activities</b></p> <p><i>Encourage the student to write reflection notes in their reflection journal using markdown.</i></p> <p>Use these as guiding questions:</p> <ul style="list-style-type: none"> <li>• What happened today?             <ul style="list-style-type: none"> <li>◦ Describe what happened.</li> <li>◦ The code I wrote.</li> </ul> </li> <li>• How did I feel after the class?</li> <li>• What have I learned about programming and developing games?</li> <li>• What aspects of the class helped me? What did I find difficult?</li> </ul>	<p><i>The student uses the markdown editor to write her/his reflections in the reflection journal.</i></p>

ACTIVITY LINKS		
Activity Name	Description	Link
Teacher Activity 1	Socket Library	<a href="https://docs.python.org/3/library/socket.html">https://docs.python.org/3/library/socket.html</a>
Teacher Activity 2	Reference Code	<a href="https://github.com/pro-whitehatjr/PRO_199_TeacherReferenceCode">https://github.com/pro-whitehatjr/PRO_199_TeacherReferenceCode</a>
Student Activity 1	Socket Library	<a href="https://docs.python.org/3/library/socket.html">https://docs.python.org/3/library/socket.html</a>
Student Activity 2	Boilerplate Code	<a href="https://github.com/pro-whitehatjr/PRO_C199_SA_Boilerplate">https://github.com/pro-whitehatjr/PRO_C199_SA_Boilerplate</a>