Socket Programming in Python: Advanced Topics

This 11-video course explores advanced features of Python sockets, including the transfer of large files over sockets, two-way communication, and differences between blocking and nonblocking sockets. You will learn to transfer large files over sockets by breaking them up into chunks, and to transfer images over TCP (transmission control protocol) sockets. Then you will learn how to transfer Python objects by using the pickle module. Next, learn how to create a chat application and use it to transfer several types of data from a server application to a client. Learners continue by exploring how to configure two-way communication over sockets by building a simple chat. This course examines the performance versus reliability trade-off when one uses blocking and nonblocking sockets. You will examine and compare TCP, a connection-oriented protocol, and UDP (Universal Datagram Protocol) which is connectionless. Finally, you will examine the performance versus reliability trade-off with a TCP and UDP, and why TCP is better suited for apps which require high reliability at the other end of the communication line.

Table of Contents

Course Overview

[Video description begins] *Topic title: Course Overview*. [Video description ends]

Hi, and welcome to this course, Advanced Topics in Python Socket Programming. My name is Kishan Iyer and I will be your instructor for this course.

[Video description begins] *Your host for the session is Kishan Iyer. He is a Software engineer and a big data expert*. [Video description ends]

A little about myself first, I have a masters degree in computer science from Columbia University, and have previously worked in companies such as Deutsche Bank and WebMD in New York. I presently work for Loonycorn, a studio for high quality video content. Python is the most popular programming language at the moment. This is partly thanks to its intuitive syntax, but also due to the vast array of libraries available, for tasks ranging from data analysis to networking.

The focus of this learning path, is to get you familiar with Python sockets, which are one of the means by which Python applications can communicate with each other. Whether they lie on the same machine, or whether they can reach each other over a network. The goal of this course, its to cover a breadth of concepts around socket programming in Python. We will take a look at the transfer of large files, including images over TCP sockets.

We also cover the configuration of two way communication over sockets by building a simple chat. We then delve into the performance versus reliability tradeoff when using blocking and non-blocking sockets, and then examine the same tradeoff, when choosing between TCP and UDP connections. By the end of this course, you will have a fairly broad understanding of socket programming in Python.

You will know how to transfer large files over sockets by breaking them up into chunks. You will recognize the differences between blocking and non-blocking sockets, and also the variations in UDP and TCP sockets. You will then have the necessary skills to build your own Python applications, which can communicate data to each other, by means of sockets.

Sending Large Text Files Using Sockets

[Video description begins] *Topic title: Sending Large Text Files Using Sockets. Your host for this session is Kishan Iyer.* [Video description ends]

So far in this learning path, we have covered how sockets can be used in order to transmit small bits of data, from one Python application to another.

[Video description begins] *A Sublime text editor opens. In the Folders pane, under the pythonsockets directory, there are two folders: pythonobjects and socketsintro. In the editor, there are three tabs: product.py, server.py, and client.py. Currently, the client.py tab is open.* [Video description ends]

We have covered the transfer of not just text data, but also Python objects by making use of the pickle module. You will observe within this client.py source file, that at each iteration, we only received 1,024 bytes or 1 kilobyte from the server at each time.

[Video description begins] *He highlights code line 12. It reads: msg = s.recv(1024).* [Video description ends]

This was more than enough, to capture the entire object which was being sent over from the server. However, this will not always be the case. There will be several occasions where you need to transmit, rather large files from one application over to another, and you'll want to use sockets in order to accomplish this. This is exactly what we will implement in this demo. But we won't be modifying any of the existing sources for this purpose.

We will now create a brand new directory. However before that, in the text editor, we can close out the files we currently have open. So I'm going to close down the client, then the server, and also this product.py file, and then heading over to the shell, we can go up one directory level over to the root directory, for this project, and then create a new one called files_and_images.

[Video description begins] *A Python shell window opens. The command prompt is: ~/workspace/pythonsockets/pythonobjects>.* [Video description ends]

[Video description begins] *He types cd . . and presses Enter. The command prompt changes to: ~/workspace/pythonsockets>. He types mkdir files_and_images and presses Enter.* [Video description ends]

As the name suggests, the goal now is to be able to transmit rather large files, including image files, from one Python application over to another. So once the directory is created we cd into it, and heading back now to the text editor, we can see that this directory is now visible here.

[Video description begins] *He types cd files_and_images and presses Enter. The command prompt changes to: ~/workspace/pythonsockets/files_and_images>.* [Video description ends]

[Video description begins] *He shifts to the Sublime text editor. In the Folders pane, a new directory named files_and_images, appears under pythonsockets.* [Video description ends]

However, we are not quite done with the directory creation. For the purpose of our demos, we have both the server and the client applications running from our own machines. That is they're running on the same machine.

So in order to separate the files which are transmitted from the server from those which are downloaded by the client, we will now need to create two additional directories.

[Video description begins] *He shifts to the Python shell window.* [Video description ends]

So we'll now create one called server_files, and within this folder, we will store all of the data, which needs to be transmitted over to the server.

[Video description begins] *He types mkdir server_files and presses Enter.* [Video description ends]

Following that, we'll create a directory called client_files, which will initially be empty. But after the files, have been transmitted from the server to the client, they should be saved by the client application in this location, and we will need to ensure that what was transmitted from the server, is in fact, what has been downloaded by the client.

[Video description begins] *He types mkdir client_files and presses Enter.* [Video description ends]

So with these two directories created, we head back to the text editor, and in the background, I have now downloaded a large text file into the server_files directory.

[Video description begins] *He shifts to the Sublime text editor. In the Folders pane, two new directories appear under files_and_images: client_files and server_files.* [Video description ends]

So the source from which I downloaded this text file is available on the screen in front of you. Specifically, I got it from kaggle.com, though you could use any large text file for this purpose.

[Video description begins] *The text file source is: https://www.kaggle.com/tentotheminus9/religious-and-philosophical-texts/version/1#pg2680.txt.* [Video description ends]

In order to view the contents of the text file, we can expand the server files directory, and you see that there is this file called meditations.txt. Upon opening it, we can see that this is a book called Meditations written by the former Roman Emperor, Marcus Aurelius.

[Video description begins] *He clicks on meditations.txt in the Folders pane. A new tab for the same opens in the editor. The first line reads: The Project Gutenberg EBook of Meditations, by Marcus Aurelius.* [Video description ends]

Again, the contents of this file, is not really important for this demo. What does matter however, is that the size of this file is about 450 kilobytes. So while this file will be transmitted from the server, the client will receive it, in chunks of 4 kilobytes each. We can now begin coding up a new source file. But before that, just make sure that the client_files directory is empty.

[Video description begins] *In the Folders pane, he highlights the client_files directory.* [Video description ends]

So we can now save this new source file, and I'm going to call this server.py, and then begin pasting the code for it.

[Video description begins] *A box appears. It has input fields for Save As and Tags. Save and Cancel buttons are present at the bottom. In the input field for Save As, he types server.py.* [Video description ends]

[Video description begins] *He clicks the Save button. The box closes. In the Folders pane, a new file named server.py appears under the server_files directory. A new tab titled server.py opens in the editor.* [Video description ends]

So just to reiterate, the goal here is for the server to transmit this meditations.txt file over to the client, and the client should download and save the file within the client_files directory. All right, let us begin coding up the server. Just as before, we will import the socket module, and then we will initialize a socket, and then work with the socket object, inside a with block.

[Video description begins] *The following code appears in code line 1: import socket.* [Video description ends]

[Video description begins] *The following code appears in code line 3: with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:.* [Video description ends]

So by specifying the first argument, when initializing the socket object, as socket.AF_INET, we are saying that this socket, will be attached to an address, which is represented by a hostname and a port number, and then we set up a TCP socket, by specifying the second argument as socket.SOCK_STREAM. We begin working with the socket instance, by first binding it, to our own host at port 4571.

[Video description begins] *The following code appears in code line 5: s.bind((socket.gethostname(),4571)).* [Video description ends]

So we invoke the socket objects, gethostname function, in order to get a computer's hostname. We then set up the socket to listen, for up to five different connections, and then print out to the console that the server is up and listening.

[Video description begins] *The following code appears in code line 6: s.listen(5). The following code appears in code line 7: print('Server is up. Listening for connections...\n').* [Video description ends]

Following that, we get the server to block, until a client connection is established, by invoking the socket instance's accept method.

[Video description begins] *The following code appears in code line 9: client, address = s.accept().* [Video description ends]

However, once a connection has been set up, we will capture the instance, of the client socket in the variable client, and then also retrieve its address in the address variable, and we will print these out to the console.

[Video description begins] *The following code appears in code line 10: print('Connection to', address, 'established\n'). The following code appears in code line 11: print('Client object:', client, '\n').* [Video description ends]

Following that, we will open up our meditations.txt file. So for this, we make use the open function in Python, in order to open up the file. So the first argument here, is the location of the file, which is of course in the server_files directory, and then we specify the mode, in which we want to open the file.

[Video description begins] *The following code appears in code line 13: custom_file = open('server_files/meditations.txt', 'rb').* [Video description ends]

Rb means that we are opening the file in read-only mode, and in the binary format. I had mentioned at the beginning of the demo, that when transmitting large files, we should break them up into chunks, and then transmit one chunk at a time, and to do that, we will first read in the first 40 kilobytes of the file by invoking the file objects read method.

[Video description begins] *The following code appears in code line 15: custom_data = custom_file.read(40960).* [Video description ends]

The argument of 40960 represents the first 40 kilobytes of this file. Since when we open it, the cursor of the file object will point to the beginning. Now in order to transfer the file in 40 kilobyte chunks, we will need to make use of a while loop. So at each iteration of the while loop, we will read up to 40 kilobytes of the file, and as long as there is data to read, we will perform a few operations.

[Video description begins] *The following code appears in code line 17: while (custom_data):.* [Video description ends]

First of all, we will send over whatever data has been read from the file, and the maximum size of this data of course is 40 kilobytes.

[Video description begins] *The following code appears in code line 18: client.send(custom_data).* [Video description ends]

Following that we invoke the read function once more in order to read another chunk of data from the file. So each time we invoke the read operation on a file, we move the cursor forward by the specified number of bytes.

[Video description begins] *The following code appears in code line 19: custom_data = custom_file.read(40960).* [Video description ends]

So in this particular while loop, we read 40 kilobytes at a time and then send them over to the connected client. Once we get towards the end of the file, the amount of data to be read will be less than 40 kilobytes. However, this will still be captured, within a custom data variable and then sent over in the next iteration of the while loop.

However, when we invoke the read operation at that point, there will be nothing to read, in which case the custom data variable will be empty, and we break out of the while loop, and once that happens, we will print out the message, that the custom file has been sent over, successfully.

[Video description begins] *The following code appears in code line 21: print('Custom File successfully Sent').* [Video description ends]

So we have now coded up our server application, to send over a large file, to a connected client, via a TCP socket, and it will do so in chunks of 40 kilobytes. In the next video, we will code up the client to receive this file, and then following that, we will run both the server as well as the client to make sure, that the file transfer, does take place as we want it to.

Receiving Large Text Files Using Sockets

[Video description begins] *Topic title: Receiving Large Text Files Using Sockets. Your host for this session is Kishan Iyer.* [Video description ends]

In the previous video, we coded up a server application which reads in a large file and chunks of 40 kilobytes and then transfers it over a TCP socket connection to a client. It is now time for us to code up the client application, for which we can create a new file, and I'm going to call this client.py.

[Video description begins] *A Sublime text editor opens. In the Folders pane, under the server_files directory, a file named server.py is selected. A tab for the same is open in the editor.* [Video description ends]

[Video description begins] *A box appears. It has input fields for Save As and Tags. Save and Cancel buttons are present at the bottom. In the input field for Save As, he types client.py.* [Video description ends]

[Video description begins] *He clicks the Save button. The box closes. In the Folders pane, a new file named client.py appears under the server_files directory. A new tab titled client.py opens in the editor.* [Video description ends]

We begin coding by, of course, importing the socket module, and we follow this up by initializing the socket. So this is going to be a TCP socket as well.

[Video description begins] *The following code appears in code line 1: import socket.* [Video description ends]

[Video description begins] *The following code appears in code line 3: with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:.* [Video description ends]

Within the with block, we first establish a connection to the server socket. So for this we invoke the connect function, pass along the server's host name, and in this case, of course, it's the same as the host name for the client, and the server socket is listening on port 4571.

[Video description begins] *The following code appears in code line 5: s.connect((socket.gethostname(),4571)).* [Video description ends]

Once we connect to the server, we know that it will be sending us a file, so we initialize a file object. So we will save this down in the location client_files, and it's going to be saved as received_file.txt.

[Video description begins] *The following code appears in code line 7: custom_file = open('client_files/received_file.txt', 'wb').* [Video description ends]

Note here that we are opening this file in write mode, as implied by the w, and then it's going to be in the binary format, as indicated by the b. So once we have initialized this file, we have the cursor for the file pointing to the beginning, and we should prepare to receive chunks of the file, from the server.

[Video description begins] *The following code appears in code line 9: while True:.* [Video description ends]

So for this we initialize a while block, and inside this while loop, we start off by receiving 40 kilobytes of data, from the server.

[Video description begins] *The following code appears in code line 11: data = s.recv(40960).* [Video description ends]

So this is going to be captured in the variable data, and we first check whether it is an empty message.

[Video description begins] *The following code appears in code line 13: if not data:.* [Video description ends]

So the received data will be empty, if the server has terminated the connection, and we have set up the server, to close down the socket once there is no more data to send, and when that happens, we will break out of this while loop. But before that we will print out the fact, that there are no more messages from the server.

[Video description begins] *The following code appears in code line 14: print('No messages from the server. Closing the connection...'). The following code appears in code line 15: break.* [Video description ends]

On the other hand, if the received data is not empty, we will write that data down, into the file, which we have just opened up.

[Video description begins] *The following code appears in code line 17: custom_file.write(data). The following code appears in code line 18: print('Batch of data written to file...').* [Video description ends]

So this while loop will keep iterating, as long as the server keeps sending it data.

[Video description begins] *He highlights code lines 9-18.* [Video description ends]

However, once there is no more data to send, the server will terminate its own socket in which case, we break out of the while loop, and at that point, we can close down this file, and we do this by explicitly calling, the close function for the file object.

[Video description begins] *He highlights code line 15.* [Video description ends]

[Video description begins] *The following code appears in code line 20: custom_file.close().* [Video description ends]

Note here, that we can also make use of, a with block, for the file object, and that is something we will explore a little later, in this course.

[Video description begins] *He points at code line 3.* [Video description ends]

For now though, we are done with the coding part for our client application, and it is now time for us to test out the communication between the server and the client, and to make sure, that the entire contents of the file, does get transmitted from the server, and is saved down by the client in the client_files directory. So we can save this client.py and then head over to the shell where I have two terminals up.

[Video description begins] *Two terminal windows open on the screen. The command prompt in both the windows is: ~/workspace/pythonsockets/files_and_images>.* [Video description ends]

In the window on the left, I will run the server, and then on the right, I will run the client. So when initializing the server, it is now listening for connections.

[Video description begins] *In the left terminal window, he types python server.py and presses Enter.* [Video description ends]

So the transfer will only begin, once a client application, has connected to the server. So for that we head over to the right window, and start up our client app, and when we do that, we observe that multiple batches of data, have been copied over and then written to the file, at the client end.

[Video description begins] *In the right terminal window, he types python client.py and presses Enter.* [Video description ends]

The server did close the connection though, since we can see from the client console, that it received an empty message from the server. So the client closed its own end, of the socket as well. We can also confirm from the side of the server, that the file was sent successfully.

Well, this is all well and good, but there is only one true way to confirm, that the file transfer, did in fact take place. So we head over to the text editor, and when we expand the client_files directory, the directory which was previously empty, now contains a file called received_file.txt.

[Video description begins] *In the Folders pane, a new file named received_file.txt appears under the client_files directory.* [Video description ends]

So at least the file was created, but we need to open it up, in order to examine the contents, and the data here seems to indicate, that the file transfer did in fact take place, without any hitches.

[Video description begins] *He clicks on received_file.txt in the Folders pane. A new tab for the same opens in the editor. The first line reads: The Project Gutenberg EBook of Meditations, by Marcus Aurelius.* [Video description ends]

Had there been any issues, we would have seen some garbled text over here. But since this is clearly English, this is a good sign, that we have successfully performed a file transfer, using Python sockets. We can continue with the visual inspection, by bringing up the original file, from the end of the server. So meditations.txt, seems to contain the exact same information.

[Video description begins] *He clicks on meditations.txt in the Folders pane. A new tab for the same opens in the editor. Its first line also reads: The Project Gutenberg EBook of Meditations, by Marcus Aurelius.* [Video description ends]

If you like, you can head back to the received_file and then scroll all the way to the bottom, to make sure that everything looks okay. However, rather than doing a line-by-line comparison manually, we can head over to the terminal and then make use of the diff utility. This will do a line by line comparison, between the server's version of meditations.txt, and the file which was received by the client.

[Video description begins] *He enters three command lines in the left terminal window. The first command is: diff \. The second command is: server_files/meditations.txt \. The third command is client_files/received_file.txt. He presses Enter.* [Video description ends]

On running this, we will be able to see each of the lines, where there is a discrepancy, between the two files. So when we run this, there is absolutely no output, which means that the two files are identical. So we have now successfully transferred over a large file, from our server application over to the client, using a TCP socket, and we did this by breaking up the file into chunks. While in this case, we were able to transfer over a text file, in the next video, we will see how we can use the exact same logic, in order to transfer other kinds of files, as well. Specifically, we will transmit an image file, from the server to the client.

Transferring Image Files with Sockets

[Video description begins] *Topic title: Transferring Image Files with Sockets. Your host for this session is Kishan Iyer.* [Video description ends]

In the previous video, we successfully transferred a large text file, from our server application, over to our client app, using a TCP socket.

[Video description begins] *A Sublime text editor opens. In the Folders pane, under the server_files directory, a file named meditations.txt is selected. In the editor, there are four tabs: server.py, client.py, received_file.txt, and meditations.txt. Currently, the meditations.txt tab is open.* [Video description ends]

We will now make use of the same server and client, and then make a few modifications to them, in order to transfer over an image file. So, we currently have text files within our server files, as well as the client files directory.

[Video description begins] *In the Folders pane, under the client_files directory, he highlights the received_file.txt file. Under the server_files directory, he highlights the meditations.txt file.* [Video description ends]

However, before we get started on the next demo, I'm going to close down these two text files, which are currently opened in my text editor, and then, in the background, I have downloaded an image file into the server files directory.

[Video description begins] *In the Folders pane, he collapses the client_files and server_files directories.* [Video description ends]

[Video description begins] *In the editor, he closes two tabs: received_file.txt and meditations.txt. Currently, the client.py tab is open.* [Video description ends]

So this is an image of a dog and the link from which this image was downloaded, is available on the screen in front of you, so it is downloaded from pexels.com.

[Video description begins] *The image file source is: https://www.pexels.com/photo/adorable-animal-breed-canine-356378/.* [Video description ends]

So in order to view the image, we can still make use of the Sublime Text editor. So I expand the server_files directory, and you can see that there is a file called dog.jpg, and we can just click on it to open it up, and this brings up a picture of a rather adorable looking dog, which is clearly quite happy.

[Video description begins] *A third tab titled dog.jpg opens in the editor.* [Video description ends]

Anyway, moving back to the demo, the goal for us, is to make use of a TCP socket, in order to transfer this image, from the server application, over to the client app. So we will have a setup similar to what we had previously, for the file transfer. In fact, we will be making use of, the same source files, and just be modifying it a little bit, so that it transmits this dog image, rather than a text file.

[Video description begins] *In the Folders pane, under the server_files directory, he highlights two files: client.py and server.py.* [Video description ends]

So heading over to the server file, I'm going to start off by importing, the PIL module into our source file, and from that, I will import the image object.

[Video description begins] *He shifts to the server.py tab. The following code appears in code line 2: from PIL import Image. The succeeding code lines shift one line down.* [Video description ends]

So PIL in case you're unfamiliar with it, is a Python imaging library, and is in fact short for pillow. It can be used in order to open, manipulate, and perform a variety of operations on images. This is however, only required if you wish to manipulate the image, in any manner. We don't really need to use it, in this source file. However, what we do need to modify, within our server.py file, is firstly to get rid of all of these lines, which involves the transfer of the text file.

[Video description begins] *He highlights code lines 14-22.* [Video description ends]

But all of the lines above it, where we declare and initialize our server socket, will remain as is.

[Video description begins] *He highlights code lines 4-12. Code line 4 is: with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:. Code line 6 is: s.bind((socket.gethostname(),4571)). Code line 7 is: s.listen(5). Code line 8 is: print('Server is up. Listening for connections...\n'). Code line 10 is: client, address s.accept(). Code line 11 is: print('Connection to', address, 'established\n'). Code line 12 is: print('Client object:', client, '\n').* [Video description ends]

Following that, we include our new code, where we once again open up a file, except this time, we open a dog.jpg file. Once again, this is going to be in read mode, and in the binary format and we reference this as image_file.

[Video description begins] *He deletes code lines 14-22. The following code appears in code line 14: with open('server_files/dog.jpg', 'rb') as image_file:.* [Video description ends]

Just as we did previously with a text file, we invoke our file objects read method and we will be reading in 40 kilobytes at a time.

[Video description begins] *The following code appears in code line 16: image_batch= image_file.read(40960).* [Video description ends]

This will make up an image batch, and as long as the image batch is not empty, just as we did previously, we invoke the client objects, send method in order to transmit this batch of data over to the client.

[Video description begins] *The following code appears in code line 18: while (image_batch):. The following code appears in code line 19: client.send(image_batch).* [Video description ends]

So 40 kilobytes of data get transferred and then following that, we read an additional 40 KB of data from the image.

[Video description begins] *The following code appears in code line 20: image_batch = image_file.read(40960).* [Video description ends]

So we are treating it exactly as we did the text file, given that we opened up the text file, as well as the image in binary format, we are able to treat them in the same manner. So this transmission will take place, as long as their is data to be read from the file, and when each batch of data, is sent, we print out a message confirming this in the console.

[Video description begins] *The following code appears in code line 21: print('One batch sent to client...')*. [Video description ends]

However, once there is no more data to send over, we will in fact break out of the while loop, in which case we convey the message to the console, that the file transmission has taken place.

[Video description begins] *The following code appears in code line 23: print('Image sent successfully')*. [Video description ends]

So you will notice the similarities between the transmission of an image and a text file.

[Video description begins] *He highlights code lines 1-23*. [Video description ends]

In fact, other than the fact that we open up a different file on this occasion, there is no significant change from the previous version of our server app. So we can now save down this file, and then head over to the client and once again, there are no significant changes to make here.

[Video description begins] *In the editor, he shifts to the client.py tab*. [Video description ends]

However, we will introduce a new concept, we will ensure that the file we are saving down the transmitted data into, is not the same text file, which we had created previously. But as I had indicated earlier on, we can in fact, make use of a with block, in order to handle the file.

[Video description begins] *He deletes code line 7. The following code appears in its place: with open('client_files/received_image.jpg', 'wb') as image_file:*. [Video description ends]

So we open up a file called received_image.jpg and this is going to be opened in write mode, and in the binary format, and we will reference this as image_file, and we can make use of the same code which we had before, except I'm now going to select all of this, and then move it all into the with block.

[Video description begins] *He selects code lines 9-20. The code shifts to the with block*. [Video description ends]

There are a few minor changes that we can make, however. For one, since we have changed the name of the file pointer, to image_file, we will change this reference to custom_file, and then write the received data into the image_file instead, and finally, rather than having to explicitly close the file, we can let the with block handle that for us.

[Video description begins] *He edits code line 17. It now reads: image_file.write(data)*. [Video description ends]

So I'm just getting rid of that line entirely.

[Video description begins] *He highlights code line 20. It reads: custom_file.close(). He then deletes it*. [Video description ends]

So you'll observe that we now have, two different with blocks, in our client file. We have one for the socket and then another one, for the file. Functionally though, this client application is exactly the same, as the previous version, where we received a text file. Except this time, all of the received data is saved, into received_image.jpg. All right, we can now save down this file, and then test out our image transfer.

So we head back to the shell, and on the left window, I will bring up the server, which is now up and listening for connections, and to initiate the file transfer, we can now start up the client application, and immediately, we see that the data has been transmitted in two different batches, and from the server end we can see that the file transmission seems to have been a success.

[Video description begins] *Two terminal windows appear. The command prompt for both is: ~/workspace/pythonsockets/files_and_images>*. [Video description ends]

[Video description begins] *In the left terminal window, he types python server.py and presses Enter.* [Video description ends]

[Video description begins] *In the right terminal window, he types python client.py and presses Enter.* [Video description ends]

So the socket was closed at both ends, and now it's time for us to ensure, that the file transfer did in fact take place. So I'm heading back to the Sublime Text Editor, and upon expanding the client files directory, I can see, that this received_image file, is now here, and it wasn't previously.

[Video description begins] *In the Folders pane, a new file named received_image.jpg appears under the client_files directory.* [Video description ends]

So we will first do a visual comparison of the two images. We head over first to the server's version, of this dog image and once again, we can admire this rather happy looking puppy.

[Video description begins] *In the Folders pane, under the server_files directory, he clicks on the dog.jpg file. A third tab titled dog.jpg opens in the editor.* [Video description ends]

However, to confirm that, the received image, was exactly the same as the client end, we head over to the client files directory, and click on received_image, and this one looks exactly the same.

[Video description begins] *A fourth tab titled received_image.jpg opens in the editor. It contains the same image of the dog as in the dog.jpg tab*. [Video description ends]

So we now have visual confirmation, that the image which was sent over by the server, is the exact same image, which was downloaded by the client. However, one final test, is for us to head back to the terminal, and perform a diff operation between these two files, and this returns nothing, which means that the two files are identical.

[Video description begins] *In the left terminal window, he enters three command lines. The first command is: diff \. The second command is: server_files/dog.jpg \. The third command is client_files/received_image.jpg*. [Video description ends]

So whenever you need to transfer large files, from one Python application to another, you can do so using a TCP socket, and you can use the exact same logic, as we saw in this demo.

Using Sockets to Build a Chat Application - Part 1

[Video description begins] *Topic title: Using Sockets to Build a Chat Application - Part 1. Your host for this session is Kishan Iyer.* [Video description ends]

So far in this learning path, we have made use of sockets in Python, in order to transfer different types of data, from a server application over to a client.

[Video description begins] *A Sublime IDE is open*. [Video description ends]

However, all of the transfer so far have been one way. It is now time for us to examine two way communication, on sockets using Python, and we will demonstrate this, by building up a chat application, where a client and server, can both chat with each other. We will set up a situation, where a customer of a company, will be able to contact a support staff, in order to make a purchase.

We will be creating some new source files, for this purpose and those will be placed in a separate directory. To do that, we head back to the terminal and I'm going to go up one directory level, into the root of this project workspace, and then from there, we will create a new directory for our current demo.

[Video description begins] *He opens a terminal window. The command prompt is: ~/workspace/pythonsockets/files_and_images>*. [Video description ends]

[Video description begins] *The following code appears: cd . .. The command prompt changes to: ~/workspace/pythonsockets>*. [Video description ends]

So this is going to be called chat, and then we can cd into it, and I'm just going to do an ls to confirm that this directory is currently empty.

[Video description begins] *The following command appears: mkdir chat. The following command appears: cd chat. The command prompt changes to: ~/workspace/pythonsockets/chat>*. [Video description ends]

All right, heading back to the text editor, the chat directory is now visible, and the first thing we will do is, to create a New File within this.

[Video description begins] *In the Folders pane, a sub-directory named chat appears under pythonsockets directory.* [Video description ends]

[Video description begins] *He right-clicks on the chat sub-directory. A few options appear. He clicks on New File. An untitled tab opens in the editor pane.* [Video description ends]

So I'm going to call this server.py. So as touched upon earlier, we will be simulating a situation, where a customer, gets in touch with, the support staff of a company, and a company's end of the connection will be represented by the server application.

[Video description begins] *A pop-up appears. It has input fields for Save As and Tags. In the input field for Save As, he types: server.py.* [Video description ends]

[Video description begins] *He then clicks the Save button. The pop-up closes. The tab title changes to server.py. A file with the same name appears under the chat sub-directory.* [Video description ends]

In such a situation, it is the company's end of the connection, which needs to be up all the time, and be listening for connections, coming in from a client. So just as before, we will set up the server application to bind to a specific address, and listen for incoming connections from clients. The clients will know exactly which address, the server can be reached at.

Whereas the server will accept connections, from any client application. All right, let us begin coding. We, of course, need to import the socket module, and sticking with the example we have seen previously, we will initialize a TCP socket, and this will bind to an address represented by a host and a port.

[Video description begins] *The following code appears on code line 1: import socket*. [Video description ends]

[Video description begins] *The following code appears on line 3: with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:*. [Video description ends]

So we will reference this servers end of the socket as sock. Once this socket has been instantiated, the first thing we will do is to prompt the server, to input a name.

[Video description begins] *The following code appears on line 5: server_name = input('Enter your name: ').* [Video description ends]

So this is the name of the server, which will appear in all of the chat communications which will follow. So once the user has input the name for the server, this will be stored in the variable, server_name, and we follow this up, by binding this socket to a specific address, represented by the host name and the port number.

[Video description begins] *The following code appears on line 7: sock.bind((socket.gethostname(), 4571)).* [Video description ends]

So all of the clients will know, that the server can be reached, at this address. We then get the socket to listen, for incoming connections, and then we print out the fact, that the server is up and is listening.

[Video description begins] *Two code lines appear on lines 9 and 10. Code line 9 is: sock.listen(5). Code line 10 is : print(server_name, 'is up. Listening for connections...\n').* [Video description ends]

The server process will then block, until a client connects to this, and we implement this of course, by calling the sockets.accept() method.

[Video description begins] *Three code lines appear on lines 12-14. Code line 12 is: client, address = sock.accept(). Code line 13 is: print('Connection to', address, 'established\n'). Code line 14 is: print('Client object:', client, '\n').* [Video description ends]

Where we capture the clients end of the socket, along with the client's address.

[Video description begins] *He highlights code line 12.* [Video description ends]

When we define the client application, we will need to code it up in such a way, that the first piece of information, which it sends to the server via the socket, is the name of the client. So we can receive that name at the server, by invoking the receive method, using the client socket.

[Video description begins] *Three code lines appear on lines 16-18.* [Video description ends]

[Video description begins] *He highlights code line 16. It reads: client_name_raw = client.recv(1024).* [Video description ends]

All of the server applications, which we have built so far, in this learning path, have only made use of the client socket, in order to send data to the client. On this occasion though, because we are implementing two way communication, we will also use this, in order to receive bits of data.

Given that this is a chat application, the size of the data is not going to be too large, so we only receive one kilobyte at a time. The raw data, which is sent over from the client, is going to be in the form of bytes, which will in fact, be an encoded string. In order to retrieve the original string, we will need to invoke, the decode method, on these raw bytes.

[Video description begins] *He highlights code line 17. It reads: client_name = client_name_raw.decode().* [Video description ends]

Once we do that, we receive the text, which has been sent over from the client to the server, which represents the name of the client, and we print out the name of the client, to the console, and that the connection has been established.

[Video description begins] *He highlights code line 18. It reads: print('Client %s has initiated a connection.' %client_name).* [Video description ends]

So at this point, the communication line has been set up, and we have received the client's name. However, the client needs to have the server's name as well. So we now invoke the send method, using the client socket object, where we send over the server name, in an encoded format.

[Video description begins] *The following code appears on line 20: client.send(server_name.encode()).* [Video description ends]

You'll notice here, that we are invoking encode, without specifying, the encoding format. The default format here is going to be UTF-8. This also applies, when we make use of the decode method, where the default format is UTF-8 again. All right, at this point, all of the setup work for our chat, has been completed.

The sockets will have been set up, and the connection between the client and the server sockets, have been made, and the names of the server and client applications have been exchanged. It is now time to begin the interactive chat, and for that, we implement a while loop.

[Video description begins] *The following code appears on line 22: while True:.* [Video description ends]

The point here, is for the server application to be running all the time, until it explicitly chooses to quit. We implement this by setting up a while True loop, where we will define the condition under which, we can break out of the loop inside it. The first thing we will do within the loop though, is to create the message, to be sent from the server to the client.

[Video description begins] *Two lines of code appear on lines 24 and 25. Code line 24 is: send_message = input(server_name +' - '). Code line 25 is: client.send(send_message.encode()).* [Video description ends]

This means that once the connection has been established, the initiation of the chat needs to happen from the server end. We can implement this by prompting the server for an input. But in the console, we will first print out the server name followed by a -. At this point, the server will then input the message which needs to be sent over to the client, and this is captured in the variable, send_message, and then we invoke the client sockets send method, where we will send this message in the form of an encoded bytestream.

Since this is an interactive chat, we now need the server to be able to receive messages from the client. But before we get to that, we need to cover the case, where the server chooses to quit the chat application, and this can be done, if the server types in the message, bye.

[Video description begins] *Two lines of code appear on lines 27 and 28. Code line 27 is: if send_message.lower() == "bye":. Code line 28 is: break.* [Video description ends]

When this happens, the server will break out of this while loop and the servers end of the socket will terminate, and no one will be able to communicate with it anymore. However, when the server does not want to quit, it needs to be able to receive incoming messages from the client, and this is done by invoking the receive method on the client socket.

[Video description begins] *Three lines of code appear on lines 30-32. He highlights code lines 30 and 31. Code line 30 is: message_recv = client.recv(1024). Code line 31 is: message_recv = message_recv.decode().* [Video description ends]

Since this received message is going to be a stream of bytes, we will need to invoke the decode method, in order to get the raw text, and then this is printed out to the servers console, followed by the client name.

[Video description begins] *He highlights code line 32. It reads: print(client_name, '-', message_recv).* [Video description ends]

So the server's console, will be able to see the entire transcript, of the chat. Since all of the messages sent by the server, are first accepted as input, we will be able to view those, in the console, and all of the messages, which are received from the client, will be printed out to the console, thanks to this print statement.

So with that, we have now concluded the definition, of our server application. Just to quickly summarize what we have implemented, the server application, will be up and running and listening for input connections, coming in from client applications.

[Video description begins] *He highlights code lines 5-10.* [Video description ends]

Once a client does connect, the first thing it'll do, is to send over its name to the server, and then the server will send its own name back, to the client.

[Video description begins] *He highlights code lines 12-20.* [Video description ends]

Following that, there will be an interactive chat, where each party can send a message, and then wait for the other one to respond. In the next video, we will code up the clients end of this connection, and then, we'll test out, our chat application.

Using Sockets to Build a Chat Application - Part 2

[Video description begins] *Topic title: Using Sockets to Build a Chat Application - Part 2. Your host for this session is Kishan Iyer.* [Video description ends]

Now that we have coded up, the server's end of this chat application, we will focus now on building, the client's end of this communication.

[Video description begins] *A Sublime IDE is open. The server.py tab is open. It has some code lines.* [Video description ends]

So for that, we will need to create a new source file, which is going to be called client.py.

[Video description begins] *An untitled tab opens in the editor pane. A pop-up appears. It has input fields for Save As and Tags. In the input field for Save As, he types: client.py.* [Video description ends]

But before we start writing any code, let us give some thought, to what exactly we need to do here.

[Video description begins] *He then clicks the Save button. The pop-up closes. The tab title changes to client.py. A file with the same name appears under the chat sub-directory.* [Video description ends]

We know that the server will be listening on a specific address. So it's up to the client, to connect to that address. We have also set up the server in such a manner, that the first details which will be exchanged, once the connection has been established, will be the names of the server and client.

So we need to code the client, to accept a name from the user and then send it over to the server, as soon as the socket connection has been set up, and beyond this, the interactive chat can take place. So we can now begin coding by, of course, importing the socket module, and then we set up a TCP socket, and then the first thing we do here, is to prompt the user of this client application to enter their name.

[Video description begins] *The following code appears on code line 1: import socket.* [Video description ends]

[Video description begins] *The following code appears on line 3: with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:.* [Video description ends]

This is stored in the variable client name, and we then set up this client application, to connect to the server socket.

[Video description begins] *The following code appears on line 5: client_name = input('Enter your name: ').* [Video description ends]

[Video description begins] *Two code lines appear on lines 7 and 8. Code line 7 is: sock.connect((socket.gethostname(), 4571)). Code line 8 is: sock.send(client_name.encode()).* [Video description ends]

The server is running on the same host as the client, and can be reached on port 4571, and the first thing the client needs to do, once a connection has been set up is to send over its own name to the server, and for this, it makes use of the socket object's send method, where the client name is sent in the form of an encoded byte stream.

Once this is done, the server will give its own name in exchange, and we invoke the socket's receive method, in order to get the raw server name which of course needs to be decoded into a string.

[Video description begins] *Three code lines appear on lines 10-12. He highlights code line 10. It reads: server_name_raw = sock.recv(1024).* [Video description ends]

[Video description begins] *He highlights code line 11. It reads: server_name = server_name_raw.decode().* [Video description ends]

Once that is done, we will print out the fact that we have now connected to the server, and once all of the setup work has been done, we'll get down to implementing the interactive chat, and just as with the server, we will make use of a while loop.

[Video description begins] *He highlights code line 12. It reads: print('You have connected to the server %s' %server_name).* [Video description ends]

[Video description begins] *The following code appears on line 14: while True:.* [Video description ends]

Inside the server's while loop, we have set it up in such a manner that it is the server which initiates the communication, and you'll need to code up the client, in order to receive that communication from the server.

[Video description begins] *Three lines of code appear on lines 16-18. He highlights code line 16. It reads: message_recv = sock.recv(1024).* [Video description ends]

So it does this by invoking the receive method and the incoming message is decoded and then printed out to the console.

[Video description begins] *He highlights code line 17. It reads: message_recv = message_recv.decode(). He highlights code line 18. It reads: print(server_name, '-', message_recv).* [Video description ends]

Once this is done, it is the client's turn to send a message over to the server, and to do this, it prompts the user for an input, and then this input is collected, and then sent over to the server using the socket's send method.

[Video description begins] *Two lines of code appear on lines 20 and 21.* [Video description ends]

[Video description begins] *He highlights code line 20. It reads: send_message = input(client_name +' - ').* [Video description ends]

[Video description begins] *He highlights code line 21. It reads: sock.send(send_message.encode()).* [Video description ends]

So we coded up the server, to first send a message and then receive one, and we have done the inverse at the client end. Now it should be possible for us to break out of this while loop, and now we set up the condition.

[Video description begins] *Two lines of code appear on lines 23 and 24. Code line 23 is: if send_message.lower() == "bye":. Code line 24 is: break.* [Video description ends]

So when the client sends the message, bye, that is the point at which, the client terminates his own end of the socket. This happens when we break out of the while loop, and we exit the with block for the socket connection. So with that, we have now concluded the definition of our client application. So just to quickly summarize what will happen in our chat, the server needs to be up, and listening for incoming client connections.

Once such a client does connect to the server, they will first exchange their names, and an interactive chat will begin. We have coded up both the server and the client in such a manner that once they have sent a message over to the other end, they'll need to wait until a message gets back to them.

So it's always a case of send a message and then receive a message. It's not possible for either end of this application to send multiple messages without receiving one in turn. So with that done, we can now save down this client application, and it's time to test out our communication. So we head over to the terminal, and on the left, I will bring up the server, and then we'll run the client from the right.

[Video description begins] *Two terminal windows appear side by side. The command prompt for each is: ~/workspace/pythonsockets/chat>*. [Video description ends]

So we start up the server application, and just as we had defined it in the code, the first thing which happens, is that we are prompted to enter a name.

[Video description begins] *In the left terminal window, the following command appears: python server.py.* [Video description ends]

So let's just say, we enter the name, Support. So we now get the message that the server is up and listening for incoming connections. It is at this point where we can start up a client. So we head over to the right window, and the client application also prompts the user to enter a name.

[Video description begins] *In the right terminal window, the following command appears: python client.py.* [Video description ends]

So let's just say the user calls themselves Customer. As soon as we hit Enter, the message has made its way from the client over to the server, and we can see that the connection is now complete. Focusing on the server's window, we get the message that the client Customer, has initiated a connection, and now the server is prompted to enter some input. However at the client's end, you will see that the client, does not see a prompt.

So it is now up to the server, to initiate the communication. So let's just say, this support staff, just ask the client, how exactly they can help them, and once we hit Enter, this message has made its way over to the client. Once this happens, it is the client who has the prompt, and the server now has to wait for the client to send over a message.

So the customer, can now type in their own message, and once we hit Enter, this makes its way to the server, and it's now up to the server, to send over a message. So you can continue playing around with this chat application, and then simulate some back and forth communication. So the customer wishes to purchase a phone, and then the support staff, asks them some follow-up questions.

So we can keep going along with this scenario, until at some point, the customer is happy, with the service they have received, and there will come a point, where they are ready, to terminate the communication. So we have coded up our applications in such a way, that the customer and also the server, can close down their end of the communication, once they enter the text, bye. So we can now type in this text, at the end of the customer, and then when we hit Enter, we see that the client application has now exited.

However, the server is still up and running, and the server itself can break out of this socket, by typing in bye, and on hitting Enter, the server application, has terminated as well. So what we have demonstrated, in this particular application, is a very basic chat. While a real chat application, will have many more features, and will also have the ability, for each end to send multiple messages, over to the other.

The underlying infrastructure, will be about the same, as what we have implemented here, where we will make use of TCP sockets, have the server bind to a specific address, and get client applications, to connect to it. The back and forth communication, can also be implemented, by using the send and receive function calls, which we have used. So this will hopefully give you a fairly good idea, of how you can implement a chat, using Python sockets.

Sockets in Blocking Mode

[Video description begins] *Topic title: Sockets in Blocking Mode. Your host for this session is Kishan Iyer.* [Video description ends]

So far in this learning path, we have created many different sockets, in order to transfer various kinds of data, from one Python application to another.

[Video description begins] *A Sublime IDE is open.* [Video description ends]

One of the properties, which is common to all of the sockets we have created so far, though, is that they have been set to run in blocking mode. To understand blocking mode and non-blocking mode, you need to recognize that each time you invoke a send function in order to send data over a socket, what goes on under the hood is that the data first gets written into a buffer, which is linked to the socket.

The data is then transferred from this buffer over to the application at the other end of the socket. Blocking mode is when the sending application waits for the buffer to have space before it writes data into it, and non-blocking mode is when it does not. In this demo, we will see exactly how this can affect, the performance of an application.

[Video description begins] *A terminal window appears. The command prompt is: ~/workspace/pythonsockets/chat>.* [Video description ends]

So we begin by heading first over to the shell, and then we first navigate into the root directory of the project.

[Video description begins] *The following command appears: cd . .. The command prompt changes to: ~/workspace/pythonsockets>.* [Video description ends]

So from here, we will create a new folder in order to demonstrate blocking and non-blocking modes.

[Video description begins] *The following command appears: mkdir blocking_and_nonblockingmodes.* [Video description ends]

So when the folder has been created, we cd into it, and then I'll just run an ls to show that it is empty, and now heading over to the text editor, we confirm that this newly created directory is visible here, and then following that, we can create a new file within this.

[Video description begins] *The following code appears: cd blocking_and_nonblockingmodes. The command prompt changes to: ~/workspace/pythonsockets/ blocking_and_nonblockingmodes>.* [Video description ends]

[Video description begins] *In the Folders pane, a sub-directory named blocking_and_nonblockingmodes appears under the pythonsockets directory.* [Video description ends]

[Video description begins] *He right-clicks on the blocking_and_nonblockingmodes sub-directory. A few options appears.* [Video description ends]

So this is going to be called server.py.

[Video description begins] *He clicks on New File. An untitled tab opens in the editor pane. A pop-up appears. It has input fields for Save As and Tags. In the input field for Save As, he types: server.py.* [Video description ends]

This application will of course have a corresponding client.py file. But unlike the previous client and servers which we have created, in this case, we will be transferring data over from the client application over to this server. So let us begin coding this up.

[Video description begins] *The pop-up closes. The tab title changes to server.py.* [Video description ends]

We start off by importing the datetime module, and specifically the datetime object within it, and then of course, we import the socket module, and we follow this up by initializing a TCP socket.

[Video description begins] *A code appears on line 1. It reads: from datetime import datetime.* [Video description ends]

[Video description begins] *The following code appears on line 2: import socket.* [Video description ends]

[Video description begins] *The following code appears on line 4: with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:.* [Video description ends]

The first thing we do inside the with block is to bind our socket to our host and get it to listen on port 4571.

[Video description begins] *Two code lines appear on lines 6 and 7. Code line 6 is: s.bind((socket.gethostname(), 4571)). Code line 7 is: s.listen(5).* [Video description ends]

We then print out the fact that the server is up and listening, and then we get the server to wait for incoming connections by invoking the socket's accept method, and once a connection has been made, we will retrieve the socket, which can be used to communicate with the other end of the socket, and also the address of the client connecting to this.

[Video description begins] *The following code appears on line 9: print('Server is up. Listening for connections...\n').* [Video description ends]

[Video description begins] *Three lines of code appear on lines 11-13. Code line 11 is: client, address = s.accept(). Code line 12 is: print('Connection to', address, 'established\n'). Code line 13 is: print('Client object:', client, '\n').* [Video description ends]

All right, so far, this application looks similar to what we have seen previously. We bring up a server, and get it to listen for incoming connections. On this occasion, though, we will be measuring how much time it takes for data to be transferred over from the client over to the server.

[Video description begins] *The following code appears on line 15: start_time = datetime.now().* [Video description ends]

This is done to demonstrate the fact, that when we initialize a socket to run in blocking mode, transfer of large pieces of data can take significantly longer than if we use non-blocking mode. So in order to measure the time for the transfer, before we receive any data at this end, we initialize this start time variable to the current time, which is accessible as datetime.now. It is following that where we invoke the socket's receive method, in order to receive a 1,024 bytes of data.

[Video description begins] *Two lines of code appear on lines 17 and 18. He highlights code line 17. It reads: data = client.recv(1024).* [Video description ends]

So this is captured in the variable data, and then we have a variable called total_received_size, which will be used in order to capture, the total size of the data, which we receive at this server end.

[Video description begins] *He highlights code line 18. It reads: total_received_size = len(data).* [Video description ends]

This is initialized to the value of the first chunk of data which we receive on the socket. We will be incrementing this value in just a little bit, though. We then create another variable called i, and this represents the number of batches of data, which the server has received.

[Video description begins] *The following code appears on line 20: i = 1.* [Video description ends]

Given the single invocation of the receive method, this value is now initialized to 1, and just as with the total_received_size variable, we will be incrementing this as well, and we will be doing all of that within a while block. Since the client application will send over many chunks of data, we will get this while loop to process those chunks, until there is no more data for the client to send over.

[Video description begins] *The following code appears on line 22: while data:.* [Video description ends]

So inside the while loop definition, we first print out the data, which has been received, from the other end of the socket, and then we invoke the receive function once more, in order to receive the next 1,024 bytes of data.

[Video description begins] *Two lines of code appear on lines 23 and 24. He highlights code line 23. It reads: print (data.decode('utf-8')).* [Video description ends]

[Video description begins] *He highlights code line 24. It reads: data = client.recv(1024).* [Video description ends]

This of course will go on, until there is no more data, for the client to send. But when we receive one chunk of data, we will be incrementing the two variables, which we have just created.

[Video description begins] *Two code lines appear on lines 26 and 27. Code line 26 reads: total_received_size += len(data). Code line 27 reads: i += 1.* [Video description ends]

The total_received_size, is going to be incremented by the size of the data chunk received, and the variable i, representing the number of batches of data, is also going to be incremented by 1. The reason we do this, is to compare how much data, is received when we run our application, in blocking mode, versus non-blocking mode, and we will soon see, that there is a difference, between the two different modes. We have now concluded the definition, of our while loop, at which point all of the data, has been received. We can then print out, the number of batches of data which came in, and then we can also close down the socket connection.

[Video description begins] *Two code lines appear on lines 29 and 30. He highlights code line 29. It reads: print ('All data received in %i batches' %i).* [Video description ends]

[Video description begins] *He highlights code line 30. It reads: client.close().* [Video description ends]

We will then once again invoke datetime.now in order to calculate the current time, and then print out the differences, between end_time and start_time, which will give us a measure, of how much time the entire data transfer took, and finally, we print out the total size of the data, which has been received by the server application.

[Video description begins] *Two lines of code appear on lines 32 and 33. He highlights code line 32. It reads: end_time = datetime.now().* [Video description ends]

[Video description begins] *He highlights code line 33. It reads: print('Duration: ',end_time - start_time).* [Video description ends]

[Video description begins] *The following code appears on line 35: print('Size of data received: %i bytes' %total_received_size).* [Video description ends]

So this concludes the definition of our server.py. But before we proceed to the client, you can take a quick look, at the metrics which we are calculating at this end. In the variable i, we have the number of batches of data, which are received, and we also calculate the total size of the data, which has come in. Beyond that, we also measure the total time taken, for the data transfer.

All of these measures can vary significantly, depending on whether we have set up a socket to use blocking mode or non-blocking mode, and this is a configuration you will take care of, in the client application, which is the end of the application, which is sending data over. All right, so with that, we can save down this server.py, and in the next video, we will code up the client and then execute both of these apps, first with the server set to run in blocking mode, and then in non-blocking mode.

Sockets in Non-Blocking Mode

[Video description begins] *Topic title: Sockets in Non-Blocking Mode. Your host for this session is Kishan Iyer.* [Video description ends]

In the previous video, we coded up the receiving end of our socket connection, which is the server.py file.

[Video description begins] *A Sublime IDE opens. Currently, the server.py tab is open. It contains a few code lines.* [Video description ends]

It is now time for us to develop the sender, which is the client. So I'll now go ahead and create a new source file, which is called client.py. Now before we write any code, let us quickly summarize what we which to accomplish.

[Video description begins] *A pop-up appears. It has input boxes for Save As and Tags. In the input box for Save As, he types: client.py.* [Video description ends]

[Video description begins] *A new tab titled client.py opens.* [Video description ends]

The goal here is to demonstrate the differences between blocking and non-blocking modes of a socket connection. The metrics which we are using in order to show these differences are the total time taken for the data transfer, the number of batches, as well as the total size of the data being transmitted. In order for there to be a meaningful difference in these values, we need to make sure that the size of data being sent over is significant. All right, so let us now begin coding. We import the socket module, and we follow this up by initializing a TCP socket.

[Video description begins] *The following code appears in line 3: with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:.* [Video description ends]

We then connect this end of the socket over to the servers end, and this of course is accessible on the same host at Port 4571, and then following that, we configure this socket, to run in blocking mode.

[Video description begins] *The following code appears in line 5: s.connect((socket.gethostname(), 4571)).* [Video description ends]

So for that, we invoke the socket's setblocking method, and then we pass this an argument of True. Note here, that the default mode for any socket is blocking. So this particular function call is not really required.

But, for the sake of our demonstration, we will set this explicitly, and then later on, we will set this value to false, in order to demonstrate the differences, between these two modes. All right, so with our socket initialized and configured, it is now time for us to begin the data transfer, and the large chunk of data which we send over, is going to be the text, hello server, multiple times. To be precise, the product of a 1,024*1,024*10 is about 10.4 million.

[Video description begins] *He adds new code on lines 9 and 10. Code line 9 reads: data = bytes('Hello Server\n', 'utf-8') *1024*1024*10.* [Video description ends]

That combined with the fact, that the text, Hello Server, with the new line character at the end, contains 13 bytes. Means that we are sending over, a 136 megabytes of data, and we will print out this total size of data being sent over, to the console, and then following that, we invoke the send method of the socket in order to transmit this data over to the other end, which is going to be our server application.

[Video description begins] *He highlights code line 10. It reads: print('Size of data sent: %i bytes' %len(data)).* [Video description ends]

To ensure that there are no errors in the sending process, we make use of the assert keyword.

[Video description begins] *A new code appears on line 12. It reads: assert s.send(data).* [Video description ends]

So note here, that when running a socket in blocking mode, an invocation of the send method, will make sure that, there is space available in the right buffer, associated with this socket. If the buffer is full, then the application will block, until there is room available. In which case, the send method will return, and then send over the remaining bits of data.

In non-blocking mode, the send method will return, whether or not the buffer is full. So, let us save down this client.py, and now run our applications in blocking mode. So I have these two terminal windows up, and from the left I'm going to run the server application which is now up and running, and then heading over to the right window, we'll start up the client.py app.

[Video description begins] *Two terminal windows appear. The command prompt in both the terminal windows is: ~/workspace/pythonsockets/blocking_and_nonblockingmodes>.* [Video description ends]

[Video description begins] *In the left terminal window, he enters the following command: python server.py.* [Video description ends]

[Video description begins] *In the right terminal window, he enters the following command: python client.py.* [Video description ends]

All right, with that done, we can see that the data transfer has begun. From the client console, you can see that this data transfer involves a 136 MB. So let us just wait for the server to receive all of this information, and you'll observe that while this is going on, the client is suspended. That is, it is in a blocking state, until all of the data has been received, and we can see all of the statistics which we have printed out at the server end.

Since the server received the data, in batches of a 1024 bytes each, a total of 133,000 batches were received. All of this took about 22 seconds, and the total size of the data received, matches the total size of the data sent from the client which is a little over 136 megabytes. The specific metric I would like to draw your attention to here, is the fact that this transfer took about 21.9 seconds, and while this was going on, the client was blocking.

That is, it was in a state of suspended animation, and could not really work on anything else. Now this is clearly not the most efficient way to run a program, and this can be avoided, if we set our socket to run, in nonblocking mode. This however does come with a few caveats, and to demonstrate that, we can head back to the text editor, and from here, we can leave the server.py file, in exactly its current form where it receives data in batches of a 124 bytes, and then prints out the three different metrics.

[Video description begins] *He highlights code line 24. It reads: data = client.recv(1024).* [Video description ends]

[Video description begins] *He highlights code lines 29, 33, and 35. Code line 29 reads: print('All data received in %i batches' %i). Code line 33 reads: print('Duration: ', end_time - start_time). Code line 35 reads: print('Size of data received: %i bytes' %total_received_size).* [Video description ends]

However, we will make one tiny change to a client source. So when we head to client.py, we will once again leave things pretty much as they are, except for the call to the sockets set blocking method. We set the argument now to False. What this means, is that the socket will not synchronize with the right buffer. In order to ensure that the data which is being sent over does get written to that buffer. Just to recap what I had said a little earlier on.

Each time we invoke the send method, the data gets written into a write buffer linked to the socket, and then transferred over to the receiving end of the socket. In the case of blocking mode, the application ensures that there is room in the buffer for the data to be written to, and, if the write buffer is full, then the application will wait or block until there is space available. This can significantly slow down, a large data transfer such as the one we have here, where a total of 136 MB is being sent over.

However, it does ensure, that the data does get written to the buffer itself. So what happens if you were to turn blocking mode off? Well, we can test it out by saving down this file, and then heading back to the terminal. So once again, I'm going to start up the server application, and then from the right window, we'll start up the client, and as soon as we hit enter, we see that the data transfer is complete.

[Video description begins] *In the left terminal window, he enters the following command: python server.py.* [Video description ends]

[Video description begins] *In the right terminal window, he enters the following command: python client.py.* [Video description ends]

However, there are a few different things for us to take a look at. You will observe that from the client end, a total of a 136 megabytes was sent. However, at the end of the server, only 261 kilobytes have been received. This was done in 257 batches, and significantly, the total time for this transfer, was just a fraction of a second. So what does all of this mean?

Well, when we configure a socket to run in nonblocking mode, it means that, the sending application, does not need to sync up, with the write buffer, in order to ensure, that it has space, when it invokes ascend. This allows the sending application, to proceed with some other work, without having to wait, for the data transfer. However, this comes at a cost of potentially losing some data, if there was no space in the buffer, for it to be written to. So in fact, if we rerun this application, we are likely to see a different amount of data being transferred, than on the previous run.

[Video description begins] *He clears both the terminal windows.* [Video description ends]

[Video description begins] *In the left terminal window, he enters the following command: python server.py.* [Video description ends]

So I'm bringing up the server application here, and then we'll rerun the client as well, and once again, the applications run in just a fraction of a second.

[Video description begins] *In the right terminal window, he enters the following command: python client.py.* [Video description ends]

But also, only a fraction of the data has actually made its way from the client over to the server. Also, previously, a little over 250 batches of data, made its way from the client to the server, and on this run, it is 576 batches.

[Video description begins] *He clears both the terminal windows.* [Video description ends]

In fact, we can rerun this for a third time. So bringing up the server and then the client. On this run, once again, this has only taken about a fifth of a second. However, 624 batches have been transferred over. So in most real applications, in order to get the best performance, you would probably want to set your socket to use nonblocking mode.

But, as we have just observed, this comes with a few caveats. So when we configure a socket, to use non blocking mode, and we are transferring over a large chunk of data, which is likely, to overrun the write buffer, then the amount of data, which actually does make its way, to the recipient is unpredictable, and this is something which we need to account for.

Using Python to Subscribe to RSS Feeds

[Video description begins] *Topic title: Using Python to Subscribe to RSS Feeds. Your host for this session is Kishan Iyer.* [Video description ends]

So far in this learning path, we have covered how socket connections can be used in order for two different Python applications, to communicate with one another.

[Video description begins] *A Sublime IDE opens*. [Video description ends]

We will now take a look at a special case where you want your Python application, to subscribe to a specific data feed. Specifically, to an RSS feed. RSS feeds are a special type of data feed which is used by many different websites, in order to communicate up-to-date information to the subscribers of the feed.

This is often used by news websites, in order to communicate the latest headlines, and a summary of those latest stories, and RSS feeds can also be used to communicate, the latest stock market data. The latest scores for various sports, and so on.

In this demo, we will get our Python application to subscribe to an RSS feed. So to demonstrate that, we will head over to the terminal, and then I'll head over first to the root directory, of the project workspace, and then from there, we'll create a new folder, to demonstrate RSS feeds.

[Video description begins] *The command prompt in the terminal window is: ~/workspace/pythonsockets/blocking_and_nonblockingmodes>*. [Video description ends]

[Video description begins] *He enters the cd . . command. The command prompt changes to ~/workspace/pythonsockets>*. [Video description ends]

So this is called rss_feed, and once we cd into it, the first thing we're going to do, is to install a Python library, which will be required, in order to subscribe to such a feed.

[Video description begins] *He enters the following command: mkdir rss_feed*. [Video description ends]

[Video description begins] *He enters the following command: cd rss_feed. The command prompt changes to ~/workspace/pythonsockets/ rss_feed>*. [Video description ends]

This is known as feedparser. So note here, that what we will be implementing, is not strictly socket programming. But, this is to demonstrate the transfer of data, from an RSS website, over to your own Python application. So we make use of, the pip package manager, in order to install feedparser, and since I already have this, I get the notification that the requirement is already satisfied.

[Video description begins] *He enters the following command: sudo pip install feedparser.* [Video description ends]

If you don't have feedparser already, you'll likely see the installation taking place right now. However, once you do have feedparser, let us put it to use, and before we create any application, which subscribes to the RSS feed, we can take a look at some of the mechanics around the feedparser.

[Video description begins] *He clears the terminal window.* [Video description ends]

For that I'm going to bring up a Python Shell, and once there, let us first import the feedparser module. If it has been installed correctly on your system, you will not see any error here, and this is very much the case for me. So I can now make use of it in order to subscribe to an RSS feed, and the way to do it is to invoke the feedparser.parse method. The argument to this is the URL for an RSS feed.

In this case, I'm subscribing to one, which is provided by the United Nations, specifically for news regarding economic development. RSS feeds are typically available, in the form of an XML file. This will contain the latest items, in the feed and will get periodically updated, by the provider. So each time you wish to get the latest item from an RSS feed, you can invoke feedparser.parse, and this will return an object containing all of the latest data for the feed. Now, let us examine the type of this object, and the output shows us, that this is of type FeedParserDict.

[Video description begins] *He enters the following command: type(feed).* [Video description ends]

So this is a special type of dictionary, containing all of the feed data. If you want to view all of the fields and the functions associated with this object, we can make use of Python's dir method, and from the output, we can gather that there are a lot of different fields and functions, which are applicable to regular Python dictionaries.

[Video description begins] *He enters the following command: dir(feed).* [Video description ends]

Such as items, keys, values, and there are also a few additional attributes. Since this is a dictionary, it will contain a few key and value pairs, and we will now try to access the value corresponding to the key href. Note here that we are treating this feed object as a dictionary here, since we're using the exact same syntax, and the value which is returned, is in fact the URL, for our RSS feed.

[Video description begins] *He enters the following command: feed['href'].* [Video description ends]

So this is very much a dictionary. However, we can also make use of the dot notation in order to get the same value. So when invoking feed.href, the same RSS URL is returned. Now the reason we subscribe to this RSS feed, is to get the latest news related to economic development, supplied by the United Nations. So how exactly can we get to that? For this we access the value corresponding to the key entries, and we make use of the dot notation to access this, and all of the news data is now visible in the screen.

[Video description begins] *He enters the following command: feed.entries.* [Video description ends]

Of course, there is a lot of information here, and to get exactly how many new stories, have been provided within this feed, we can calculate the lenght of feed.entries, and this confirms that a total of 30 articles are present. Now, in order to examine the data which is supplied for each individual article, we will access the first element of this feed.entries array, and even from this output, we can see that a lot of information is provided. At the very top, you can see that there is a key called title. So in order to access that, we use the dot notation to get the title of this news article, and now we can see exactly what the headline for this article is.

[Video description begins] *He enters the following command: feed.entries[0].title.* [Video description ends]

Similarly, if you want the link to the full article, we can access the link field, and this gives us the URL, for this particular article, which is hosted on un.org.

[Video description begins] *He enters the following command: feed.entries[0].link.* [Video description ends]

Now, in order to get a summary, of this news article, we can use the summary field, and this gives us a few lines, summarizing what the article is about.

[Video description begins] *He enters the following command: feed.entries[0].summary.* [Video description ends]

So in this particular RSS feed, we have seen that we are provided with a news headline. A summary of the news article, along with a link to the full article. It is possible for you to code up your Python application, to subscribe to an RSS feed, and then retrieve the latest information, which is published by the issuer, and then potentially use parts of it within your own application.

If you're running a website for example, you can subscribe to this specific United Nations news feed, and then print out the headline, and summary of a handful of these articles. Let's see how you might use this in a Python application, though. So for that we can exit out of the Python shell and then head over to our text editor. The RSS feed directory has been created and it's currently empty. So we will now create a new source file, which I'm going to call client.py.

[Video description begins] *A pop-up box appears. It has input boxes for Save As and Tags. In the input box for Save As, he types: client.py.* [Video description ends]

We just need this one Python source for this demo, and we start off by first importing the feedparser module, and in order to periodically refresh the news feed, we will also import the time module.

[Video description begins] *A tab titled client.py opens in the editor.* [Video description ends]

We then initialize our feed just as we did from the Python shell, where we invoke feedparser.parse, and we are subscribing to the same United Nations RSS feed, for economic development news.

[Video description begins] *He enters new code on lines 4 and 5. Code line 4 reads: feed = feedparser.parse('https://news.un.org/feed/subscribe/\. Code line 5 reads: en/news/topic/economic-development/feed/rss.xml').* [Video description ends]

Following that, we print out the total number of stories in this RSS feed, and we then set up this application to run indefinitely, until it is explicitly terminated with a keyboard interrupt, and at each iteration, we will iterate over each of the articles in the RSS feed, and specifically we will print out the title, or the headline of the article, along with the link to the full article.

[Video description begins] *He enters the following code on line 7: print('Number of RSS posts : ', len(feed.entries), '\n').* [Video description ends]

[Video description begins] *He enters the following code on line 9: while True:.* [Video description ends]

[Video description begins] *He enters new code on lines 10-12. Code line 10 reads: print('######### Begin Feed #########\n\n'). Code line 11 reads: for post in feed.entries:. Code line 12 reads: print(post.title,':\n', post.link,'\n\n').* [Video description ends]

We will then print out the fact, that this is the end of the feed, and we'll then iterate over the articles once again after 30 seconds.

[Video description begins] *He enters new code on lines 14 and 15. Code line 14 reads: print('######### End Feed #########\n\n'). Code line 15 reads: time.sleep(30).* [Video description ends]

If there are any new news articles, they will be available within our feed. So we can now save down this source file, and in order to run this, we head over to the shell and then execute client.py, and when we do this, the 30 news articles are available in the console, and then rather than wait the full 30 seconds, I'm just going to fast forward a little bit, and once the feed has refreshed, I'll just invoke the KeyboardInterrupt by hitting Ctrl+C, and just scrolling further up, we can see that there was a second iteration, of our RSS news feed. So if you'd like to subscribe to an RSS feed, in order to get the latest information, which is supplied by some kind of data provider. You can make use of the Python feedparser module.

UDP Sockets in Python

[Video description begins] *Topic title: UDP Sockets in Python. Your host for this session is Kishan Iyer.* [Video description ends]

So far in this learning path, all of the sockets which we have created have been TCP sockets.

[Video description begins] *A Sublime IDE opens.* [Video description ends]

Now, just to summarize the differences between TCP and UDP. TCP is a connection-oriented protocol, whereas UDP is connectionless. TCP, as a result, is better suited for applications which require high reliability, since it ensures that any data which is sent is actually received at the other end of the communication line. UDP on the other hand, is not as reliable since it does not look for this confirmation.

However, it involves significantly less overhead, and is better suited for applications, where performance is more important than reliability. We will now take a look at how we can set up sockets to make use of UDP. So for that, I'm heading over to the terminal, and we'll head over to the root directory of the project workspace, and create a new folder called udp_sockets.

[Video description begins] *The command prompt in the terminal window is: ~/workspace/pythonsockets/ rss_feed>. He enters the cd . . command. The command prompt changes to ~/workspace/pythonsockets>.* [Video description ends]

[Video description begins] *He enters the following command: mkdir udp_sockets.* [Video description ends]

So I'll just cd into it, and in order to develop the client and server applications, for this demo, we'll head back to the text editor, and then inside this newly created udp_sockets folder, we'll create a new source file.

[Video description begins] *He enters the following command: cd udp_sockets.* [Video description ends]

[Video description begins] *The command prompt changes to: ~/workspace/pythonsockets/udp_sockets>.* [Video description ends]

[Video description begins] *He enters the ls command.* [Video description ends]

This is going to be called server.py.

[Video description begins] *In the Folder pane, he right-clicks on the udp_sockets sub-directory. A few options appear. He clicks on New File. A pop-up box appears. It has input fields for Save As and Tags. In the input field for Save As, he types: server.py. A tab titled server.py opens in the editor.* [Video description ends]

So we can start coding this up, where we import the socket module, and can also make use of the time module, and when we initialize a socket on this occasion, just as we have with the previous sockets, we see that the type of address, the socket will be linked with, will contain a hostname and a port.

[Video description begins] *He enters the following code in line 4: with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:.* [Video description ends]

This is indicated by the first argument, which we set to socket.AF_INET. However, the difference between this socket and the previous ones, is in the second argument. Whereas previously for a TCP socket, we set the socket type to socket.SOCK_STREAM.

In order to configure the socket to use UDP instead, we set this value to socket.SOCK_DGRAM. With our socket initialized, we can go ahead, and begin coding up, what we will do with the socket, and the first thing we will do, is to set a timeout value.

[Video description begins] *He enters the following code in line 6: s.settimeout((0.2).* [Video description ends]

We have seen previously in this learning path, that with a TCP socket, if there is no incoming connection to the server, an exception is raised, once the timeout has expired. However, this is not the case when we use UDP sockets, as we will soon see. So we now set the server socket to bind to the specific address. So it's once again going to run on port 4571, and beyond that, we will keep account of the number of messages, which are sent over from the server, over to the client at the other end of the socket.

[Video description begins] *He enters the following code in line 8: s.bind((socket.gethostname( ),4571)).* [Video description ends]

So this is initialized to a value of 1, and now, before we begin the transmission of data, we set it up within a while loop, and inside this loop, we first create a message.

[Video description begins] *He enters the following code in line 10: i = 1.* [Video description ends]

[Video description begins] *He enters the following code in line 12: while True:.* [Video description ends]

[Video description begins] *He enters the following code in line 14: message = bytes('Message #' + str(i), 'utf-8').* [Video description ends]

This is going to be in the form of bytes, and these bytes are generated, by encoding a string using the utf-8 format. So the string itself contains the text Message, followed by the # sign, followed by a number represented by the value of i. Once we do this, we will send over this message to the recipient, and note here that we are invoking, the sendto function of the socket object.

[Video description begins] *He enters the following code in line 16: s.sendto(message, ('localhost', 37020)).* [Video description ends]

This is a method which is invoked, when we make use of a UDP socket. So here, we are not really sending this data over to an application at the other end of a socket. But we're just sending it to a specific address, whether or not there is an application listening on that address.

So this address is represented by a combination, of a hostname and a port, and specifically this will be sent over to, anything which happens to be listening on the same host as the server on port 37020. Following that, we will print out the fact that the message has been sent, and then we will get this application to sleep for 5 seconds.

[Video description begins] *He enters two lines of code in lines 18 and 19. Code line 18 is: print ('Message sent!'). Code line 19 is: time.sleep(5).* [Video description ends]

We then increment the value of i, and then the while loop will repeat.

[Video description begins] *He enters the following code on line 21: i += 1.* [Video description ends]

So just to quickly summarize what we have done here. We have gotten the server application, to bind itself to a specific port. However, it doesn't really require another client to connect to it, which is why we don't invoke the accept method which we have seen when using TCP sockets. Instead, we send messages over to port 37020 of the local host. While these messages are sent, there is no guarantee that they will actually be received by an application listening on that address, and this is the feature of UDP communication.

There is very little overhead involved since we don't really need to ensure that the data gets transmitted. However, this comes at the cost of reliability, since we have no information on whether the data was actually received by anyone. This concludes the definition of our server application, and we can now turn our attention towards the client, which is going to be in the source file client.py.

[Video description begins] *A pop-up box appears. It has input fields for Save As and Tags. In the input field for Save As, he types: client.py.* [Video description ends]

We, of course, begin by importing the socket module, and even at this end, we set the socket to make use of UDP, which is why the second argument for the socket type here is SOCK_DGRAM, and at this end, as well, we get the socket to bind to the given address.

[Video description begins] *A tab titled client.py opens in the editor.* [Video description ends]

[Video description begins] *He enters the following code in line 3: with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:*. [Video description ends]

[Video description begins] *He enters the following code in line 5: s.bind(("",37020))*. [Video description ends]

So this is the difference when it comes to UDP communications as compared to TCP sockets. When using TCP sockets, we got one end of the connection to bind to the specified address, and then got the other end to connect to that address. That is what allowed the socket connection to be reliable, where we could ensure that, any data which was getting sent, was in fact received by the other end of the socket.

In the case of UDP, we don't really need to ensure this end-to-end communication. However, the server application did send some data over to port 37020 of the local host, and now, in order for this client to receive this data, we need to make sure that it is listening on that same address, which is why we get it to bind to this address. Note here that the first argument here is a blank string, which will get this socket to bind to the local host.

So both a client as well as the server applications will be bound to specific ports on our local host, and while we coded the server application to send over some data, we will now set up this client to receive that information. So for that we implement a while loop, and inside this, we invoke the socket object's recvfrom function.

[Video description begins] *He enters the following code in line 7: while True:*. [Video description ends]

[Video description begins] *He enters the following code in line 9: data, addr =s.recvfrom(1024)*. [Video description ends]

So this is the complementary function to sendto, which we invoke from the server end, and this will receive any data which has been sent, to the address to which the socket is bound. In fact, invoking recvfrom will return two different objects. The first of these is the data which has been sent to the address in the form of bytes, and the second is the address from which the data has been sent. So in our case, we will print out the received data to the console.

[Video description begins] *He enters the following code in line 10: print ('Received message: ', data)*. [Video description ends]

So we can now save down our client as well, and it is now time for us to test out this transfer of data using UDP. So we head over to the terminal, and from the left, I will start up the server, and while with a TCP socket, the server would wait for an incoming connection.

[Video description begins] *Two terminal windows are open side by side. The command prompt in both the terminal windows is: ~/workspace/pythonsockets/udp_sockets>*. [Video description ends]

[Video description begins] *In the left terminal window, he enters the following command: python server.py*. [Video description ends]

In the case of a UDP connection, the server doesn't really care whether the data is getting sent to a recipient. It is simply sending out the information. So at this point, three different messages have been sent over to the client's address, but the client application is not yet up. In fact, I'll just go ahead and start it, and you can see that, from Message number 5 onwards, the client is now receiving data.

[Video description begins] *In the right terminal window, he enters the following command: python client.py.* [Video description ends]

So this is the difference between TCP and UDP sockets. The sender simply sends the data whether or not there is any recipient listening for the messages, and once a connection has been established, the recipient, only receives all of the new data which has been sent over. All of the previous data is lost.

So I'll just terminate the server now, by hitting control Ctrl+C, and you'll observe that the client is still up and running. This is because, just like with the server, we had bound the client to a specific address. So it is still up and listening for any data, which is sent over to port 37020. In fact, if you were to go ahead and restart the server, the server begins sending messages once more from number 1, and these messages make their way over to the client.

[Video description begins] *In the left terminal window, he enters the following command: python server.py.* [Video description ends]

Now, I'll just go ahead and terminate both of these applications, and that brings us to the end of this demo, on UDP sockets in Python.

Course Summary

[Video description begins] *Topic title: Course Summary.* [Video description ends]

In this course, we cover a wide range of features of Socket programming in Python. We started off by transferring large files, including images over TCP Sockets, which we handled by breaking up these files into smaller chunks. We also set up two-way communication, over sockets by building a simple chat application.

We then looked into the performance versus reliability trade off, when using blocking and non-blocking sockets, and then examined the same trade-off, when choosing between TCP and UDP sockets. At this point, you will have a fairly broad understanding, of Socket Programming in Python. You will know how to transfer large files over sockets, and you now have the necessary skills, to build your own Python applications, which can communicate data to each other, using sockets.