Socket Programming in Python: Introduction

Learners can explore basic concepts of Python socket programming, and how to communicate small amounts of data between Python applications by using either the same machine or over a network, in this 9-video course. Begin by learning how to use Python language to set up a communication line by creating a socket. Then learn to initialize a simple socket, and use it to transfer text data from one application to another. This course next demonstrates how to create a client app and server app in Python, and how each app uses a socket to communicate. Learners will observe a demonstration of how to transmit a Python dictionary and custom object over a socket connection. You will learn how to use a socket model to set up a simple TCP (transmission control protocol) socket to transfer text between applications. Next, learners will examine other properties of Python sockets, including its use with the context manager and the setting of a time-out for connections. Finally, you will learn to use the Pickle library to convey Python objects over a socket connection.

Table of Contents

Course Overview

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

Hi and welcome to this course, an introduction to socket programming in Python. My name is Kishan Iyer and I will be your instructor for this course.

[Video description begins] *Your host for this 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 Master's 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 among the most popular programming languages at the moment.

This is partly thanks to its intuitive syntax, but also due to the vast array of libraries available for tasks such as data analysis and 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 are on the same machine or connected over a network.

In this course, we begin by looking at a socket module in Python and use it to set up a simple TCP socket in order to transfer some text from one application to another. We then cover some of the properties of Python sockets, such as its use with the with context manager and the setting of a timeout for connections.

We finally cover the use of the Pickle library in order to convey Python objects over a socket connection. By the end of this course, you would be familiar with the basics of socket programming in Python and can build

applications which are able to communicate small bits of data. Whether these are simple text or Python objects, you can then move on to exploring sockets a little further, in order to see how large files can be transferred between Python apps by breaking them into chunks.

Introducing the socket Module

[Video description begins] *Topic title: Introducing the socket Module. Your host for this session is Kishan Iyer.* [Video description ends]

We now begin with labs on socket programming in Python. Sockets are a means at which processes can exchange information.

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

These processes could reside on the same machine, or could even be on different machines spread across a network. One potential network is, of course, the Internet. So sockets effectively provide a communication line between two or more processes.

In this learning path, we will take a look at how exactly we can make use of the Python language in order to set up such a communication line by creating a socket, and then get two different Python processes or Python applications to communicate with each other using that socket.

Before writing any of the source code for this learning path, I've created a workspace directory on my machine. So in my home directory, I have a directory called workspace, which contains a sub directory called pythonsockets. Now inside this directory, the first thing I'm going to do is to create a new sub directory for the first demo of this learning path.

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

So this is going to be a directory called socketsintro, and once this directory has been created, we can cd into it.

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

So this is the directory where we will be creating our first source files. But before we get to that, we need to ensure that we have the sockets package which is available for Python.

[Video description begins] *The following command appears: sudo pip install sockets.* [Video description ends]

This package includes the socket module which gives us access to the BSD socket interface, which is available on most operating systems today. So let us go ahead and install this using the pip package installer for Python packages, and once this installation has taken place, the next thing we need to do is to take a look at the operating system's hosts file.

[Video description begins] *The following command appears: less /etc/hosts.* [Video description ends]

Since I'm on a macOS, this is located within my etc directory in a file called hosts, and the location is going to be different if you're on a different OS. For example, if you're on Windows, you will find the corresponding file in the location C:\Windows\System32\drivers\etc. So let us open up this host file and then take a look at its contents.

[Video description begins] *A Host Database opens. It has IP addresses for the following: localhost, Loony-Macbook-Air.local, and broadcasthost.* [Video description ends]

So you will observe that what this contains is a mapping of hostnames to IP addresses. In my case, I have three different entries within this host file. So the hostname, localhost points to the IP address, 127.0.0.1. Whereas, the hostname, Loony-Macbook-Air.local, points to the exact same IP address.

So both of those hosts names will translate to the same IP address. I have another entry for the broadcasthost hostname. The reason I bring up this host file is to ensure that your computer's hostname does have a corresponding entry within this host file, so that it can be translated to an IP address.

One way to get your computer's hostname is to run the command hostname from your terminal. All right, once you have ensured that there is an entry for your hostname within this host file, we can proceed back to our shell, and then from here, let us bring up the Python shell.

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

So by running Python from the command line, the Python shell is up and you'll observe that in my case, I have version 3.7.4 of Python on my system. It is recommended that you have at least version 3.5 of Python installed on your system, this is because, there were a lot of changes made to the socket module, in that version, and it is possible that some of the code, we will now go through during the labs in this course, will not be compatible with an earlier version of Python.

All right, so now that we are within the Python shell, it is time for us to run a few commands in order to take a look at the socket module itself. So to use that, we will first import the module into our source and then following that, we will invoke the get hostname function within the module, and from the output, you observe that in my case, the name Loony-Macbook-Air.local appears.

[Video description begins] *The following command appears: import socket.* [Video description ends]

[Video description begins] *The following command appears: socket.gethostname().* [Video description ends]

So this particular hostname, does have an entry within my host file. This is important if you want to be able to set up a socket by making use of the computer's hostname. So we now know that the socket module can be used to get the hostname of your computer.

We will in fact be making use of this function when we initialize sockets later on in this learning path. The hostname however, will need to be translated to an address, and in order to get the address corresponding to the hostname, we can make use of another socket function, which is gethostbyname.

[Video description begins] *The following command appears: socket.gethostbyname(socket.gethostname()).* [Video description ends]

The argument of this is going to be the hostname, and this will return the address, which corresponds to that host. So on running this, I get the output of 127.0.0.1. Throughout the labs in this learning path, I will be running Python applications within my own local host, and when we initialize sockets from those applications, they will be binding to this IP address. We now continue with our exploration of the socket module, this time by invoking the gethostbyaddress function.

[Video description begins] *The following command appears: socket.gethostbyaddr('127.0.0.1').* [Video description ends]

When we run this, what we are returned is, in fact, a triple.

[Video description begins] *The following output appears: ('localhost' , ['1.0.0.127.in-addr.arpa'], ['127.0.0.1']).* [Video description ends]

The first value here is the hostname, and this is the primary hostname, corresponding to the given IP address. In my case, this translates to localhost, and we saw this hostname within my host file previously. The second value

here, is a list of possible aliases for the hostname, and the last value is a list of IP addresses, which are connected with this host, in my case, there is just a single value for this.

Before we exit this Python shell, and then begin developing the source code for our demos, I would like to make one more function invocation. Specifically by calling socket.gethostbyname, by specifying an empty string as argument, this returns a value of 0.0.0.0. So this represents all of the IP addresses for the local machine.

Out of these functions, the only one we will be using during the demos of this path is socket.gethostname, which we will use in order to initialize a socket connection. The other functions, however, should give you an idea of how the socket interface can be used in order to translate IP addresses to hostnames and vice versa. So we can now exit from the Python shell, and from the next video, we will create our first Python socket, and then set up two Python applications to communicate using that socket.

Using Sockets in Client and Server Applications

[Video description begins] *Topic title: Using Sockets in Client and Server Applications. Your host for this session is Kishan Iyer.* [Video description ends]

In this demo, we will be creating two Python applications, a client app and a server app, which will communicate with each other using a socket. This is going to be a hello world of socket programming in Python, where all we will do is to send one line of text from a server to a client.

[Video description begins] *A Sublime IDE opens. There is a Folders pane on the left which contains a directory named pythonsockets. It expands to show a sub-directory named socketsintro. Currently, an untitled file is open in the editor pane.* [Video description ends]

So let us begin coding. You'll observe that I have brought up a text editor on my machine, so I'm making use of the Sublime text editor in order to develop the source files. You may choose any IDE or text editor which you are comfortable with. None of the source files which you will create during this learning path will be too long.

So the choice of text editor or IDE is not all that important. You will observe here that I have brought up the Python sockets directory in this text editor and you will also observe that the sockets intro folder which I created within that is visible. This folder is of course empty. But I'm now going to create a new source file within this. So this is going to be a server application, which I'm going to name server.py.

[Video description begins] *A pop-up appears. It has input fields for Save As and Tags. Cancel and Save buttons are present at the bottom.* [Video description ends]

[Video description begins] *In the input field for Save As, he types server.py.* [Video description ends]

So once the file has been created, let us begin coding. The first thing we're going to do here is to import the socket module and with that, we will use this in order to create our first socket.

[Video description begins] *He clicks the Save button. The pop-up closes. A file named server.py appears under the socketsintro sub-directory in the Folders pane. A tab for it opens in the editor pane. The following code appears in line 1: import socket.* [Video description ends]

[Video description begins] *On line 3, the following code appears: s = socket.socket(socket.AF_INET, socket.SOCK_STREAM).* [Video description ends]

We can create a socket object by calling socket.socket and passing to it two arguments. The first one which is socket.AF_INET represents an address family. AF_INET, which is the default value, suggests that we will associate this socket with a host and port combination. The host can be represented by either an IPv4 address or a hostname.

We will soon see how that works. But then moving along to the second argument here, which is SOCK_STREAM, which conveys that this is a TCP socket. This is as opposed to a UDP socket which we will take a look at later on in this path.

So just to summarize, we are initializing a socket object here and this is going to be a TCP object which will be attached to a host and port combination, and how exactly do we attach this socket to the host and port? This is where we will call the function s.bind.

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

This is the function used to bind our socket to a particular address, the address here being represented by a hostname and a port number. In order to attach this socket to our own machines, we can make use of the function socket.gethostname which we have taken a look at a little earlier in this course. As for the port number, this can take up pretty much any value.

But in order to avoid conflicts with some of the system ports, make sure that it is a high value and in my case, I have chosen the value of 4571, which I know does not conflict with any of the existing applications on my system. Do keep in mind though that when you run this server.py a little later, it is possible that you will get the message that the address is already in use, in which case you may need to change this port number at that point.

So until now we have initialized a socket and then we have bound it to a particular hostname and port. This of course represents just one end of the socket. We follow this up by invoking the socket's listen function. This will enable a server to accept connections and by specifying a value of 5 as the argument, we are saying that this server can accept up to five different connect requests before it refuses any further connections.

Starting from version 3.5 of Python, this is an optional value. So at this point our server process will be listening for any incoming connections and in fact it will print out to the console that the server is up and running and it's listening for incoming connections from client applications.

[Video description begins] *On line 8, the following code appears: print('Server is up. Listening for connections...').* [Video description ends]

And we follow this up by making a call to the sockets accept function.

[Video description begins] *Four lines of code appear on lines 10-13. He points at code line 10. It reads: client, address = s.accept().* [Video description ends]

This will make sure that the server process blocks until an incoming connection is made to the socket. Do keep in mind that in order to invoke this accept function, we need to bind a socket to a particular address and then have the socket listen for incoming connections. However, once that connection request has been made, this accept function will be invoked and this will return two different values.

The first of these represents the client's end of the socket and we capture this in the variable client. We will need to use this in order to communicate any data over to the client. The second value which is returned by the call to accept is an address representing the client.

For IPv4 addresses, this will be a combination of a host and a port. We will, in fact, visualize the client object as well as the address by printing them out to the console in the print statements which follow, and then, once we have established the connection with the client, we will be sending them some data in the form of text.

[Video description begins] *He highlights code lines 11 and 12. Code line 11 is: print('Connection to', address, 'established\n'). Code line 12 is: print('Client object:', client, '\n').* [Video description ends]

[Video description begins] *He highlights code line 13. It reads: client.send(bytes('Hello! Welcome to socket programming.','utf-8')).* [Video description ends]

So note here that we are making use of the client object's send function. The data which we send over must be in the form of bytes, for which we make use of the bytes function and the arguments to this include the data which we wish to send over, which in our case is the string, hello!

Welcome to socket programming, and the second argument, is the type of encoding which we wish to perform. So we will encode the string in the UTF-8 format. Once we have sent over this data, we don't really need the socket anymore. So we can then close the socket by invoking the closed method.

[Video description begins] *On line 15, the following code appears: s.close()*. [Video description ends]

So this concludes our code for the server.py file. So let us just quickly summarize what we have done. We have initialized a TCP socket and then we bound it to an address. The address being represented by a combination of a host name and a port number.

We then set up the socket to listen for incoming connections and then following that, we got it to wait for incoming connections by invoking the socket's accept method and once a client does connect to the socket, we will send it some text as data.

So this concludes the definition of our server. Let us now move on and define the client application. So we can create a new source file for this and this is going to be within the same sockets introdirectory.

[Video description begins] *A pop-up appears. It has input fields for Save As and Tags. Cancel and Save buttons are present at the bottom*. [Video description ends]

I'm going to call this client.py. So let us begin coding up the client.

[Video description begins] *In the input field for Save As, he types client.py. He then clicks the Save button. A file named client.py appears under the socketsintro directory in the Folders pane. A tab for it opens in the center pane*. [Video description ends]

This will also need to make use of the socket module, so we import that in to the source and then we initialize a socket at this end as well.

[Video description begins] *The following code appears on line 1: import socket. Two lines of code appear on lines 3 and 4. Code line 3 is: s = socket.socket(socket.AF_INET, socket.SOCK_STREAM). Code line 4 is: s.connect((socket.gethostname(),4571))*. [Video description ends]

This is in fact going to be similar to the socket which we set up at the server end. So, this is a socket, which is going to be attached to a host name and a port combination, which is represented by the first argument, socket.AF_INET and this is also going to be a TCP socket represented by the second argument, SOCK_STREAM.

So this variable s represents the client's end of a socket. We can now use this in order to connect to an already established remote socket and for that we make use of the connect method of the socket object. Since we have coded up the server already, we know that it's going to be listening on our own machines at port 4571 and we now need to get the client to connect to that same host and port.

Since this client is going to be running on the same host as the server, we can specify the host by calling socket.gethostname. However, if the server happens to be remote, we will need to specify the remote address of the server in that case. So with these few lines of code, we have initialized the socket at the client end as well and we have got it to connect to the server. Once a connection has been established, we will now need to configure the client to receive any data which the server has to send.

[Video description begins] *Two lines of code appear on lines 6 and 7. Code line 6 is: msg = s.recv(1024). Code line 7 is: print('Message from server:', msg.decode('utf-8'))*. [Video description ends]

The data can be received by invoking the receive function. The value of 1024, which we specify, represents the buffer size. Since the server will be sending us data in the form of bytes, we are setting up the client to receive a 1024 bytes at a time and once we have received up to a 1024 bytes, we will print out the received message onto the console.

Given that the message object which you receive is going to be in the form of bytes, we will need to call the decode method in order to retrieve the original string. Since the server had made use of utf-8 encoding, we will need to specify the same value in order to get the original data. So with that done, we will have received the data from the server, in which case we can close down the socket.

[Video description begins] *In line 9, the following code appears: s.close()*. [Video description ends]

In the next video, we will run both our server, as well as a client applications and see whether the data gets transmitted from the server to the client.

Using socket Objects in a with Block

[Video description begins] *Topic title: Using socket Objects in a with Block. Your host for this session is Kishan Iyer.* [Video description ends]

In the previous video, we had coded up a server as well as a client application in order to communicate some text via a socket.

[Video description begins] *The Sublime IDE opens. The client.py tab is currently open in the editor pane. It has some code lines*. [Video description ends]

Both the client and server applications will be running on our own machines and the server end of the socket will be accessible on port 4571. It will then be listening for incoming connections. We then coded up the client to initialize its own end of the socket and then connect to the server socket.

[Video description begins] *He highlights code lines 3 and 4. Code line 3 is: s = socket.socket(socket.AF_INET, socket.SOCK_STREAM). Code line 4 is: s.connect((socket.gethostname(),4571))*. [Video description ends]

Once the connection has been established, the server will send over some text data which has been encoded as bytes, and the client will receive that data, decode it, and then print it out to the console.

[Video description begins] *He highlights code lines 6 and 7. Code line 6 is: msg = s.recv(1024). Code line 7 is: print('Message from server:', msg.decode('utf-8'))*. [Video description ends]

It is now time for us to put our applications to the test. In order to do that, I'm going to bring up two terminal windows.

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

From the window on the left, I will start up the server application and then we'll switch over to the window on the right, in order to start up the client. So, we'll now go ahead and start up the server.

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

Once we do that, we get the message that the server is up and listening for connections. Note that at this point, you may get a message that the address is already in use, which means that the port which you specified in your server's source file may be used by another application. In which case, you could either wait for that port to get free or change the port number in your source.

However, once your server is up and listening, it is effectively blocked, since it's waiting for an incoming connection and we will provide exactly that by switching over to the right terminal window and then running the client, and when we do this, we see data in the consoles for both the server as well as the client and both the applications have in fact run successfully and have terminated.

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

So let us begin with the console output at the server end. Right after the listening for connections message, we see that a connection was established to this tuple starting with the hostname of 127.0.0.1 and the port of 50367. So this represents the client's address. So while the server's end of the socket was listening on port 4571, the client's end will pick up any free port which is available on your machine and in my case this was 50367.

You will most likely see a different value in your output. Following that, the entire client object was printed out to the server's console and there are a few things of note here. First of all, you will see that the socket type for the client was SOCK_STREAM and that the socket address family was AF_INET, just as we had set up the client socket and then there are also two additional fields here, laddr, which represents the local address of the server and this is host 127.0.0.1 and port 4571, and then our address which represents the remote address of the client.

This is also host 127.0.0.1 and in my case, the port is 50367. So this confirms that the socket object contains two different endpoints. There is a local address and a remote address. You'll observe that even though we initialized both the server as well as the client using the hostname when we called socket.get hostname. What we observe in the output here is that the hostname has been translated to its IP address of 127.0.0.1.

[Video description begins] *A screenshot of the client.py file from the Sublime IDE appears*. [Video description ends]

So while this covers the mechanics of the socket initialization, what we are really concerned with is whether the data has been transmitted from the server to the client and when we move our attention to the console at the client end, we see that the data has been transmitted and received.

[Video description begins] *In the right terminal window, he highlights the following output: Message from server: Hello! Welcome to socket programming*. [Video description ends]

So we have now successfully created our first socket in Python and we have used it in order to transmit some text from a server to a client application. At this point though, let us tweak our code just a little bit in order to adopt a best practice when it comes to socket programming.

[Video description begins] *He shifts back to the Sublime IDE. The server.py tab is currently open in the editor pane*. [Video description ends]

So I have now switched over to the text editor and I have brought up the server.py file. So you will observe that we initialized the socket and then we explicitly close it at the end.

[Video description begins] *He highlights code lines 3 and 15. Code line 3 is: s = socket.socket(socket.AF_INET, socket.SOCK_STREAM). Code line 15 is: s.close()*. [Video description ends]

If you're familiar with the "with" keyword in Python, you will know that it can be used with objects such as files, which need to be worked on, and then explicitly closed once you're done with it. In fact, the same also applies to sockets. So we will now move all of our relevant code into a with block.

So the statement here where we initialize the socket can be modified now. Where we make use of the with keyword and then we create a new instance of the socket.socket object using the same parameters as before and once again, this is going to be referenced by the variable s.

[Video description begins] *He modifies code line 3. It now reads: with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:.* [Video description ends]

Following this, all of the code which we have below this can be moved into the with block. So I'm just going to select all of this and then introduce a tab, so that they're now all part of the with block.

[Video description begins] *He highlights code lines 5-15. Code line 5 is: s.bind((socket.gethostname(),4571)). Code line 6 is: s.listen(5). Code line 8 is: print('Server is up. Listening for connections...'). 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'). Code line 13 is: client.send(bytes('Hello! Welcome to socket programming.','utf-8')).* [Video description ends]

When using the with block though, the object we are working with will be automatically closed at the end of the block. So we don't really need to explicitly close the socket here, that will be taken care of by the with block. So I'll just go ahead and get rid of that line and at this point, the server application should work in exactly the same way as it did before.

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

We can now switch over to the client and pretty much introduce the same set of changes.

[Video description begins] *He shifts to the client.py tab.* [Video description ends]

So I'm going to replace this socket initialization line by using the with block and then all of the lines which follow that can be moved inside the with block.

[Video description begins] *He modifies code line 3. It now reads: with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:.* [Video description ends]

So I'll just select them and then introduce a tab and once again, the call to the close function can be removed.

[Video description begins] *He highlights code lines 4-9. Code line 4 is: s.connect((socket.gethostname(),4571)). Code line 6 is: msg = s.recv(1024). Code line 7 is: print('Message from server:', msg.decode('utf-8')). Code line 9 is: s.close().* [Video description ends]

[Video description begins] *He removes code line 9.* [Video description ends]

So now, the client application should also run in exactly the same manner. Of course, we will need to run these applications to ensure that their behavior hasn't really been altered. So we head back to the shell and then from the left terminal, I'm going to bring up the server.

[Video description begins] *He shifts back to the two terminal windows.* [Video description ends]

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

So it is now up and listening for connections, and now we will establish a connection to the server by bringing up the client.

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

And the outputs in each of the consoles confirms that the behavior of these applications has not been modified by the introduction of the with blocks. What you will observe in the output though, is that the port on which the client was running is different from the previous run or at least is very likely to be different.

In my case, the client was previously running on port 50367 whereas it's now on 50426. The reason for this is that we are not binding the client to a specific port, which we are in fact doing at the server end. So the client will just pick up any free port which is available on its machine.

Setting Timeouts for Python Sockets

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

So far we have successfully set up a socket between a server and a client application. And we used that in order to transmit some data from the server to the client. The way we have coded up the server application though, it will be listening for incoming connections and in fact, it will listen indefinitely until a connection has been made from some client.

[Video description begins] *The Sublime IDE opens. The server.py tab is open in the editor pane. It has some code lines.* [Video description ends]

This is because the call to the accept method of the socket object will block the server process indefinitely until an incoming connection has been made.

[Video description begins] *He highlights code lines 5-8. Code line 5 is: s.bind((socket.gethostname(),4571)). Code line 6 is: s.listen(5). Code line 8 is: print('Server is up. Listening for connections...\n').* [Video description ends]

[Video description begins] *He highlights code lines 10-14. 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'). Code line 14 is: client.send(bytes('Hello! Welcome to socket programming.',"utf-8")).* [Video description ends]

However, when communicating between different applications, such a setup may not exactly be viable. Since the server is pretty much stuck until there is some client which connects to it. In order to prevent that, we will now explicitly set a timeout for the server, so that it waits for a specified amount of time for a client to connect to it and if that time has expired, it will go on with this code execution.

So we now just need to make a few small changes to the source. So once we have bound the server to a specific address represented by the host and the port number, we will make a call to the sockets.settimeout method and the argument to this is the number of seconds which the server needs to wait until a connection has been established.

[Video description begins] *He highlights code line 5.* [Video description ends]

[Video description begins] *He creates space on lines 6-8. The following code appears on line 7: s.settimeout(10). The succeeding code lines shift three lines down.* [Video description ends]

So here, we are setting the server to wait for incoming connections for up to ten seconds. If no connection is established, then the server will continue with this code execution. What this translates to is that, the call to the socket's accept method will not be blocking indefinitely but will only be blocking for ten seconds beyond which the call will be made and a socket timeout error will be thrown. So with this change made, we can now save down our server.py and switching over to the client. We don't really need to make any changes to this.

[Video description begins] *He switches to the client.py tab. It contains some code lines. Code line 1 is: import socket. Code line 3 is: with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:. Code line 5 is: s.connect((socket.gethostname(),4571)). Code line 7 is: msg = s.recv(1024). Code line 8 is: print('Message from server:', msg.decode('utf-8')).* [Video description ends]

So we can now head over to the terminal and then from the left window I'm now going to bring up the server.

[Video description begins] *Two terminal windows open side by side. In both the windows, the command prompt is ~/workspace/pythonsockets/socketsintro>. In the left terminal window, the following command appears: python server.py.* [Video description ends]

So the server is now up and listening for connections. But this time it will only do so for ten seconds, so I'm waiting and waiting, and since the ten seconds have expired, a socket.timeout has been thrown and the server process has terminated.

So we have now ensured that the server does not go on forever waiting for an incoming connection. We do however, need to make sure that if a client does establish a connection within the ten seconds, the message is transmitted just as it did before. So now I'm going to bring up the server and then within ten seconds we'll head over to the other terminal and then run the client app.

[Video description begins] *The following command appears: python server.py.* [Video description ends]

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

And this time the connection has been established and the data has been sent over from the server to the client. So while we have configured the server to not wait indefinitely for incoming connections, it doesn't make sense for the server to throw an error and then terminate as it has right now.

It needs to be able to go on with its work and in order for that to happen, we need to catch this exception and handle it. We will now do exactly that by making use of a try and accept block at the server end. So switching back to the text editor, following the setting of the timeout for the socket we will now introduce the try block.

[Video description begins] *He shifts to the server.py tab in the Sublime IDE.* [Video description ends]

[Video description begins] *In code line 9, he types: try:. The succeeding code lines shift one line down.* [Video description ends]

So we add in the try key word here and then all of the code following that can move within the try block. So I'll just select all of this and then introduce a tab character and this try block will need to be followed by an except block.

[Video description begins] *Two lines of code appear on lines 20 and 21. Code line 20 is: except socket.timeout:. Code line 21 is: print('The timeout has been exceeded. Closing the connection...').* [Video description ends]

The specific type of exception which we need to handle here is available through the property socket.timeout. So when handling this exception, we simply print out the message that we have exceeded the socket timeout, and that the connection is being closed.

Of course, if you do wish your server application to continue with its work, you will have some additional code after this with block. However, in our case, this server is only meant to send data over to the client which is why we don't introduce any additional code.

So we can now save down this server and then heading over to the client, once again, no changes are required here. So it's time now to head back to the shell and then from the left window, I'm now going to bring up the server and again, I'll just wait ten seconds for the timeout to expire.

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

And as soon as that happens, you'll observe that we have now failed gracefully. The socket timeout exception has been raised. But we have coded a handler for it where we simply print out the message that the timeout has been exceeded.

So we now know, not only how to set a timeout for a particular socket connection, but also how to handle a timeout exception and then move on with the code execution. We should still do a regression test though by running the server and then getting our client app to connect to it within 10 seconds. So I'll just bring up the server from the left-window and then run the client from the right and the data transmission has taken place just as it did previously.

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

So we have now covered some of the basics of socket programming in Python. Next, we will take a look at how we can transmit not just simple text data from a server to a client, but how we can transfer Python objects from one application to another via a socket.

Transferring Python Objects Over Sockets - Part 1

[Video description begins] *Topic title: Transferring Python Objects Over Sockets - Part 1 . Your host for this session is Kishan Iyer.* [Video description ends]

So far in this learning path, we have covered the initialization of a simple socket and we have used it in order to transfer some text data from one application to another.

[Video description begins] *A sublime IDE appears. A tab titled server.py is open in the editor pane. It has twenty one code lines.* [Video description ends]

It is now time for us to see how sockets can be used in order to transfer some more complex objects. We will create some additional source files for this purpose. So for that, we can head back to the terminal where I'm currently in the socketsintro directory.

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

I'm just going to go up one directory level and then from here, create a new folder called pythonobjects.

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

So once this has been created, we can cd into it and then we can switch back to the text editor where this pythonobjects directory is now visible.

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

[Video description begins] *In the Folders pane, the pythonobjects folder appears.* [Video description ends]

So we will now create some new files within this, specifically we'll create another client and another server application and then set up a new socket in order to transmit python objects from the server to the client application. So before we get into that, I'll just close down these server and client files. So closing down the server and then the client as well. And then within the pythonobjects folder, I will create a new file.

[Video description begins] *In the Folders pane, he right clicks on pythonobjects folder. A context menu appears. He selects New File. A file titled untitled opens in the editor.* [Video description ends]

So this file is going to contain the class definition for a custom object, which we will be transmitting via a socket. So this is going to represent a product. So I'm going to call this product.py. So this is going to be a source which we import into a client and server applications.

[Video description begins] *A pop-up appears. It has input fields for Save As and Tags. Cancel and Save buttons are present at the bottom.* [Video description ends]

And will contain the definition of a class called product.

[Video description begins] *In code line 1, the following code appears: class Product:.* [Video description ends]

Since our goal is to show the transport of instances of this product class via sockets, we don't really need to make this definition too complicated. So this is just going to have a single init function defined within.

[Video description begins] *In code line 3, the following code appears: def__init__(self, pid, pname, pprice):.* [Video description ends]

So when we create a new instance of a product, all we need to supply are a product id, a product name, and the product price. And we will use these arguments to the constructor to initialize three different fields within this product instance.

[Video description begins] *In line 4, the following code appears: self.pid = pid.* [Video description ends]

The ID, the name, and the price.

[Video description begins] *In line 5, the following code appears: self.pname = pname. In line 6, the following code appears: self.pprice = pprice.* [Video description ends]

So this is pretty much all we have within this product.py file. So we can just save this down and then create a new file to represent our server application.

[Video description begins] *A pop-up appears. It has input field for Save As. He types server.py. Cancel and Save buttons are present at the bottom. In the background, a tab titled untitled is open.* [Video description ends]

So this is also called server.py. And when we begin writing the code for this, we will once again import a socket module. But along with that, we will also import the product class from our product.py source.

[Video description begins] *Two code lines appear in lines 1 and 2. Line 1 is: import socket. Line 2 is from product import Product.* [Video description ends]

Following that we make use of the with block in order to initialize a socket and this is identical to the sockets we have been working with so far.

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

This is a socket which will be attached to a host and port combination represented by AF_INET and it's going to be a TCP socket, represented by SOCK_STREAM. So we can initialize this socket by binding it to our own machines and to port 4571.

[Video description begins] *In line 6, the following code appears: s.bind((socket.gethostname(),4571)).* [Video description ends]

And following that, let us create the objects which you wish to transport over the socket.

[Video description begins] *In line 8, the following code appears: python_dictionary = {'a' : 1, 'b' : 2}.* [Video description ends]

So the first instance I'm going to create is of Python dictionary and this again need not be too complex. It just has two key and value pairs. And then following that, we can create an instance of our custom product object. So this

is represented by the variable custom_object and let's just say that this product is a torch with a product ID P024, which has a price of $13.

[Video description begins] *In line 10, the following code appears: custom_object = Product('P024', 'Torch', 13).* [Video description ends]

So these are the two objects which you will try to transport over a socket connection. And we can now get the server to listen on port 4571 of the current host, and then we print out the message that the server is up and listening.

[Video description begins] *In line 12, the following code appears: s.listen(5).* [Video description ends]

And then we get it to wait for a client connection to be made and once that is done, it will accept the connection.

[Video description begins] *In line 14, the following code appears: print('Server is up. Listening for connections...\n').* [Video description ends]

[Video description begins] *In line 16, the following code appears: client, address = s.accept().* [Video description ends]

Just as we did previously, we print out the client's address and also the client object.

[Video description begins] *Code appear in lines 17 and 18. Line 17 is: print('Connection to', address, 'established\n'). Line 18 is: print('Client object:', client, '\n').* [Video description ends]

Following that, we invoke the client.send function twice. The first of these is to send over our dictionary instance.

[Video description begins] *The following code appears in line 20: client.send(bytes(str(python_dictionary), 'utf-8')).* [Video description ends]

So note here that we are making use of the str function to convert this python dictionary object into a string and then we cast that value into bytes which are encoded using the utf-8 format. We do exactly the same with our custom product object as well, where we first convert it to a string and then to bytes.

[Video description begins] *The following code appears in Line 21: client.send(bytes(str(custom_object), 'utf-8')).* [Video description ends]

So note here that once again we are effectively transmitting strings over the connection which are cast as bytes. This is going to be our starting point and we will soon see what steps are necessary in order for us to be able to transmit these objects in a form where the original objects can be recovered. So with this, we have coded up the server end of our socket. So we can just save down this file and then create a new one. So this is going to be our client application.

[Video description begins] *A pop-up appears. It has input field for Save As. He types client.py. Cancel and Save buttons are present at the bottom. In the background, a tab titled untitled is open.* [Video description ends]

Once we begin coding, we will once again import the socket module and then also import our product source.

[Video description begins] *Two code lines appear in line 1 and 2. Line 1 is: import socket. Line 2 is: from product import Product.* [Video description ends]

We initialize the socket at this end as well, which is going to be a TCP socket, and then we invoke the connect function, so that it connects to our server socket which is listening on port 4571.

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

And this of course, is running on the same host as a client.

[Video description begins] *In line 6, the following code appears: s.connect((socket.gethostname(),4571))*. [Video description ends]

So this initialization of the client socket is the same as what we have taken a look at previously. However, this time, we know that the server is going to send two messages over to the client. The first containing a Python Dictionary and the second containing the product object.

[Video description begins] *A screenshot of code lines 8 and 10 of server.py file appears*. [Video description ends]

So we need to make sure that we receive all of that data, which is why we will now make use of a while loop.

[Video description begins] *The screenshot disappears. In line 8, the following code appears: while True:*. [Video description ends]

So by setting this to while True we will have this loop execute indefinitely until the break keyword is encountered. So we will code that in just a little bit, but first, we will receive any data which has been transmitted from the server over to this client. So we will receive the first 1024 bytes of data into the variable message.

[Video description begins] *In code line 9, the following code appears: msg = s.recv(1024)*. [Video description ends]

This works for us for the moment because the objects which we will be sending over will be small enough in size to fit within this 1024 byte limit and then we evaluate for the condition under which we need to break out of this while loop.

[Video description begins] *In line 11, the following code appears: if not msg:*. [Video description ends]

We will do this when no bytes of data are received from the server. In fact, this will also occur when the server explicitly closes down the socket connection. So once the server is done transmitting all of the data, it'll close down its own end of the socket and we will get into this if block, where we will print out the fact that there are no more messages coming in from the server, and we'll break out of the while loop.

[Video description begins] *Code appear in lines 12 and 13. Line 12 is: print('No messages from the server. Closing the connection...'). Line 13 is: break*. [Video description ends]

However, if the received message is greater than 0 bytes, then we will decode the message and then print out its contents into the console. And then we will also print out the type of the received message.

[Video description begins] *Code appear in lines 15 and 16. Line 15 is : print('Message from server:', msg.decode('utf-8')). Line 16 is: print('Type of received message:', type(msg))*. [Video description ends]

So note here, that we are casting the dictionary as well as the product instance at the server end as strings, before sending them as bytes over to the client. So with this done, we can save down this client application as well. And now we head over to the terminal in order to execute our server and client apps.

[Video description begins] *Two terminal windows appear side by side*. [Video description ends]

Let us start up the server, all right.

[Video description begins] *The following command appears in the terminal on the left: python server.py*. [Video description ends]

So this is now listening on port 4571 and then head over to the client in order to get this to receive the two instances which are being transmitted and this transmission has been a success.

[Video description begins] *The following command appears in the terminal on the right: python client.py*. [Video description ends]

So at this point, the console output at the server end is not as interesting. Because we have already seen this before. However, let us switch our attention to the console output at the end of the client. So in the first iteration of the while loop at the client end, we received the dictionary and you can see that this is in fact, a string representation of the dictionary, which was transmitted from the server.

Following that, we printed out the type of the received message which you can clearly see is in fact, bytes. Following that, we're headed into the second iteration of the while loop and this is where the product instance was transported over. So from the output you can clearly see that once again this is a string representation of that product.

And the type of the raw message was once again bytes. And then we went into the third iteration of the while loop, where 0 bytes were received, which is why the break statement was encountered. And then we exited from the while loop and also closed down the client connection. It is now time for us to begin making some changes to our server and client applications, so that they can be used in order to transmit Python objects.

Transferring Python Objects Over Sockets - Part 2

[Video description begins] *Topic title: Transferring Python Objects Over Sockets - Part 2 . Your host for this session is Kishan Iyer.* [Video description ends]

In the previous video, we created server and client applications where a Python dictionary as well as a custom object was being transmitted over a socket connection from the server to the client. While we were able to perform a transmission successfully, what we were in fact transferring, were the string representation of those instances cast as byte.

[Video description begins] *A sublime IDE appears. A tab titled client.py is open in the editor pane. It has sixteen code lines*. [Video description ends]

We can now head over to the product.py source file and we can look at what exactly we need to be able to do when we receive an instance of this product class at the end of the client.

[Video description begins] *In the Folders pane, he clicks on product.py. A tab for it opens in the editor pane. It has six code lines*. [Video description ends]

[Video description begins] *He highlights code line 1. It reads: class Product:*. [Video description ends]

We should be able to access each of the fields within the product instance, the product id, the product name, as well as the price of the product.

[Video description begins] *He highlights code lines 3-6. Code line 3 is: def__init__(self, pid, pname, pprice):. Code line 4 is: self.pid = pid. Code line 5 is: self.pname = pname. Code line 6 is: self.pprice = pprice*. [Video description ends]

So how exactly can this be done? We head back now to the server application and you will observe that we are casting each of the objects as strings and then casting them in turn as bytes, and clearly, this is not something which will allow us to recover the original objects at the end of the server.

[Video description begins] *In the Folders pane, he clicks on server.py. A tab for the same opens in the editor pane. It has twenty one code lines*. [Video description ends]

[Video description begins] *He highlights code lines 20 and 21. Code line 20 is: client.send(bytes(str(python_dictionary), 'utf-8')). Code line 21 is: client.send(bytes(str(custom_object), 'utf-8'))*. [Video description ends]

But what will happen if you were to transmit the objects directly without performing this casting operation? Let us try and take a look. So when we invoke the client.send function, we will send the dictionary as is. So I'm going to clear out these contents and then just send the Python dictionary as is.

[Video description begins] *He edits code line 20. It now reads: client.send(python_dictionary)*. [Video description ends]

And then we'll do the same with the second invocation of client.send. So instead of casting our custom object as a string and then bytes, we will just try to send it as is and see what is recovered at the side of the client.

[Video description begins] *He edits code line 21. It now reads: client.send(custom_object)*. [Video description ends]

So we save down the server now and in fact, we won't make any changes to the client application.

[Video description begins] *In the Folders pane, he clicks the client.py file. A tab for the same opens in the editor pane. He highlights code lines 1-16. Line 1 is: import socket. Line 2 is: from product import Product. Line 4 is: with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:. Line 6 is: s.connect((socket.gethostname(),4571)). Line 8 is: while True:. Line 9 is: msg = s.recv(1024). Line 11 is: if not msg:. Line 12 is: print('No messages from the server. Closing the connection...'). Line 13 is: break. Line 15 is: print('Message from server:', msg.decode('utf-8')). Line 16 is: print('Type of received message:', type(msg))*. [Video description ends]

Now we head to the terminal and then start up the server.

[Video description begins] *Two terminal windows appear side by side. In both the terminals, the command prompt is: ~/workspace/pythonsockets/pythonobjects>. In the left terminal, the following command appears: python server.py*. [Video description ends]

And you'll observe that I'm within the new Python objects directory which we have just created. So now that the server is up and running, we can head over to the other window and then bring up the client application.

[Video description begins] *The following command appears in the right terminal: python client.py*. [Video description ends]

So as soon as the connection is established, well, what we observe is that at the end of the server, a type error has been encountered. This is because, when we invoke the clients send function, what we need to send is a bytes like object and not a dictionary or our own custom object.

So we clearly need some way in which we can transmit these Python objects over a socket in the form of bytes and these need to be packaged in such a manner that the original objects can be recovered by the client. To do that, we can now head back to the text editor and we will need to make some small changes to the client, but we will get to that a little later.

[Video description begins] *He shifts to the Sublime IDE*. [Video description ends]

Heading over to the product, this is going to remain exactly the same but we will begin making our changes at the server end.

[Video description begins] *The server.py file opens in the editor pane. He highlights code lines 1-21. Line 1 is: import socket. Line 2 is: from product import Product. Line 4 is: with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:. Line 6 is: s.bind((socket.gethostname(),4571)). Line 8 is: python_dictionary = {'a'*

: 1, 'b' : 2}. Line 10 is: custom_object = Product('P024', 'Torch', 13). Line 12 is: s.listen(5). Line 14 is: print('Server is up. Listening for connections...\n'). Line 16 is: client, address = s.accept(). Line 17 is: print('Connection to', address, 'established\n'). Line 18 is: print('Client object:', client, '\n'). Line 20 is: client.send(python_dictionary). Line 21 is: client.send(custom_object). [Video description ends]

Specifically in order to send over our objects in serialized form, we will make use of the pickle library.

[Video description begins] *He creates a two line space after code line 2. The succeeding code lines shifts two lines down. In line 3, the following code appears: import pickle.* [Video description ends]

So the pickle library can be used for the serialization and deserialization of Python objects. Pickling or serializing is the process where a Python object is converted into a byte stream and the inverse operation of that is unpickling or deserializing. In the previous video, what we transported over our socket connection was a string representation of our objects which does not include all of the information required to rebuild the object.

This however, is not the case when we make use of the pickle library where all of the required information will be included when we convert our objects into a byte stream. To demonstrate how this works, after the definition of our Python dictionary, we will create a serialized or pickle version of that dictionary by invoking the function pickle.dumps.

[Video description begins] *He highlights code line 10.* [Video description ends]

[Video description begins] *In code line 11, the following code appears: pickled_dictionary = pickle.dumps(python_dictionary). Succeeding code lines shift one line down.* [Video description ends]

So we pass along our Python dictionary and this will return a pickle dictionary. We do the same with our custom object as well once again by invoking the dumps function and then we will print out the types of both of these serialized objects.

[Video description begins] *In code line 14, the following code appears: pickled_object = pickle.dumps(custom_object). Succeeding code lines shift one line down.* [Video description ends]

[Video description begins] *He creates two lines of space after code line 14. The succeeding code lines shifts below. Two code lines appear in line 16 and 17. Line 16 is: print('Serialized dictionary type:', type(pickled_dictionary)). Line 17 is: print('Serialized object type:', type(pickled_object)). Succeeding code lines shift two lines down.* [Video description ends]

Following that, we'll need to make changes to our invocation of the send functions. So rather than sending over the Python dictionary, we will now send over its pickled version.

[Video description begins] *He edits code line 27. It now reads: client.send(pickled_dictionary).* [Video description ends]

We will do the same with our custom object as well. So instead of sending it raw, we will send over its serialized version and these are the changes which we will now make for the server file.

[Video description begins] *He edits code line 28. It now reads: client.send(pickled_object).* [Video description ends]

We can now save these down and then head over to the client.

[Video description begins] *The client.py file opens in the editor pane.* [Video description ends]

In its current form, the client will receive the pickled version of the object sent over from the server and it will then invoke the decode function.

[Video description begins] *He highlights code line 16. It reads: print('Message from server:', msg.decode('utf-8'))*. [Video description ends]

This will not work for us anymore, what we need to do in order to recover the original objects at the end of the client, is to once again make use of the pickle module.

[Video description begins] *The following code appears in line 3: import pickle. The succeeding code lines shift one line down*. [Video description ends]

So we import that into the source file and following that, we will get rid of this first print statement right here.

[Video description begins] *He highlights code line 17*. [Video description ends]

So we first print out the type of the message which was received directly from the server and then we print out the contents of that message as well.

[Video description begins] *He highlights code line 18. It reads: print('Type of received message:', type(msg))*. [Video description ends]

[Video description begins] *He moves code line 18 to code line 17. The following code appears in line 18 now: print('Message data:', msg)*. [Video description ends]

Following that, we will make use of the pickle module's loads function.

[Video description begins] *The following code appears in line 20: unpickled_msg=pickle.loads(msg)*. [Video description ends]

So this is what is going to be used in order to deserialize or unpickle the object. When we do this, we should be able to recover the original object. We will in fact confirm that by first printing out the type of this unpickled object and then printing out the contents of that object itself.

[Video description begins] *Code appears in lines 22 and 23. Line 22 is: print('Type of deserialized message:', type(unpickled_msg)). Line 23 is: print('Deserialized data:', unpickled_msg)*. [Video description ends]

So these are the changes which we need to make to our client application. So we import the pickle library and then first we print out the raw message along with its type and then we unpickle the message and then we print out the unpickled object and its type as well. So when the client has been saved down, it is time for us to test out our changes.

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

So we can start up the server app. So this is now up and it's ready to listen for connections, but we can also see here the types of the serialized objects. So in the case of both the dictionary as well as our custom object, we can see that invoking the pickle library's dumps function has transformed them into bytes.

So these are the serialized objects which will be transmitted over a socket over to the client. That is of course, once we bring up the client, so we can do that and we can see at the client end what the contents of the received data are.

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

In the first iteration of the while loop, you can see that the type of the received message was bytes and we can see the raw contents of that data as well, which of course is not really discernible to us. However, just below that we

can see that the type of the deserialized message was a dictionary. So even though the client received a stream of bytes from the server, it was able to reconstruct a dictionary out of that by calling the pickle.loads function.

You can in fact see from the next print statement that when we print out the deserialized data, the contents of the original dictionary are also accessible here. Moving along now to the next iteration of the while loop at the client. Once again we see that the raw message had a type of bytes and also the contents of that raw message is what seems to us a lot of gibberish.

However, when we unpickle that object, what we were able to recover is of type product and this is something we can clearly see from the type of the deserialized message. And when we print out the deserialized data, we can view the string representation of the product instance. This is of course what you get when you try to print out a custom instance. Our goal, however, is to be able to use this product instance in order to access the individual fields within that. That is exactly what we will do in the next video.

Transferring Python Objects Over Sockets - Part 3

[Video description begins] *Topic title: Transferring Python Objects Over Sockets - Part 3 . Your host for this session is Kishan Iyer.* [Video description ends]

In the previous video, we saw how we could make use of the pickle module in order to serialize and also deserialize Python objects. This is especially useful when we want to transmit such objects over a socket connection and we were able to see that the original objects can be recovered at the end of the client by deserializing the object, which were transmitted from the server.

[Video description begins] *A sublime IDE appears. A tab titled client.py is open in the editor pane.* [Video description ends]

However, the true test of whether we have actually transmitted the Python objects correctly, is when we try to access the individual fields in our custom product instance. That is precisely what we will do in this video.

So I now have my text editor up and then from here, we will first navigate over to the product.py file and what we will aim to do is to access each of the three fields, the product ID, name, and price from the client app.

[Video description begins] *He shifts to the product.py tab.* [Video description ends]

But first, let us head over to server.py where we will begin making some changes.

[Video description begins] *He highlights code lines 3-6. Code line 3 is: def__init__(self, pid, pname, pprice):. Code line 4 is: self.pid = pid. Code line 5 is: self.pname = pname. Code line 6 is: self.pprice = pprice.* [Video description ends]

In this version of our application, we are sending over a dictionary and then a custom Python object.

[Video description begins] *He highlights code lines 9, 10, 12, and 13. Line 9 is: python_dictionary = {'a' : 1, 'b' : 2}. Line 10 is: pickled_dictionary = pickle.dumps(python_dictionary). Line 12 is: custom_object = Product('P024' , 'Torch' , 13). Line 13 is: pickled_object = pickle.dumps(custom_object).* [Video description ends]

In a more realistic setting though, it is likely that you'll only have one kind of object being transmitted over a specific socket connection. This is especially true when we have a setup such as ours where there is a while loop executing at the end of the client.

Where a certain repetitive set of tasks need to be performed, it is hard to have repetitive tasks performed on different types of objects which is why we will now alter the server application to send just an instance of our

custom object. So for that, I'm going to get rid of the references to our dictionary and then once that is done, I will leave the declaration and the pickling of our custom object as is.

[Video description begins] *He deletes code lines 9 and 10. The succeeding code lines shift two lines up.* [Video description ends]

But we'll get rid of the two print statements which follow that.

[Video description begins] *He highlights code lines 12 and 13. Line 12 is: print('Serialized dictionary type:', type(pickled_dictionary)). Line 13 is: print('Serialized object type:', type(pickled_object)). He deletes them. The succeeding code lines shift two lines up.* [Video description ends]

Moving now to the send function invocations, I'm now going to get rid of the first call to send where we send over the dictionary.

[Video description begins] *He highlights code line 20. It reads: client.send(pickled_dictionary). He deletes it. The succeeding code line shifts one line up.* [Video description ends]

So with that done, we now have a server application which will send over one instance of the product object.

[Video description begins] *He highlights code lines 5-20. Line 5 is: with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:. Line 7 is: s.bind((socket.gethostname(),4571)). Line 9 is: custom_object = Product('P024',' Torch', 13). Line 10 is: pickled_object = pickle.dumps(custom_object). Code line 12 is: s.listen(5). Line 14 is: print('Server is up. Listening for connections...\n'). Line 16 is: client, address = s.accept(). Line 17 is: print('Connection to', address, 'established\n'). Line 18 is: print('Client object:', client, '\n'). Line 20 is: client.send(pickled_object).* [Video description ends]

We can now save this down and then head over to the client. From this client application, we want to receive the unpickled product instance and then access each of the three fields within it to make sure that it is exactly the same as what was sent from the server.

[Video description begins] *He shifts to the client.py tab.* [Video description ends]

While we do that, we can also perform a clean up. So I'm now going to get rid of these two print statements, since we already know what type of raw data is received.

[Video description begins] *He highlights code lines 17 and 18. Line 17 is: print('Type of received message:', type(msg)). Line 18 is: print('Message data:', msg). He deletes them. The succeeding code lines shift two lines up.* [Video description ends]

And following that, rather than referencing what we get as an unpickled message, we know that this is going to be a product, so I'm going to call this variable product_object.

[Video description begins] *He edits code line 17. It now reads: product_object = pickle.loads(msg).* [Video description ends]

We can follow that up by getting rid of these two print statements and we will replace them with three additional print statements.

[Video description begins] *He highlights code lines 19 and 20. Line 19 is: print('Type of deserialized message:', type(unpickled_msg)). Line 20 is: print('Deserialized data:', unpickled_msg). He deletes them.* [Video description ends]

These reference the ID, the name, and the price of the product instance.

[Video description begins] *Three code lines appear in lines 19-21. Line 19 is: print('Product ID:', product_object.pid). Line 20 is: print('Product name:', product_object.pname). Line 21 is: print('Product price:', product_object.pprice, '\n\n')*. [Video description ends]

So with this we will have confirmation of whether the product instance is being transmitted and then unpickled correctly. The data which is received at the client should be exactly the same as what was transmitted from the server. So we can now save down our client application and then in order to test out our changes, we head back to the terminal where we now bring up the server.

[Video description begins] *Two terminal windows appear side by side. In both the terminals, the command prompt is: ~/workspace/pythonsockets/pythonobjects>. In the left terminal, the following command appears: python server.py*. [Video description ends]

So now the server is listening for the client and now we will bring up the client as well and from the output we can see that the product instance has been transmitted and recovered correctly.

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

What we sent from the server was this product with an ID of P024, which was a Torch costing $13 and this is precisely what has been recovered by the client application. So we now know exactly how to transmit Python objects over a socket connection in a form where the original object can be recovered and to do this, we need to make use of the pickle module.

Let us now go ahead and make some additional changes to our application, so that we send not just a single instance of a product, but a sequence of such product instances. So we head back to our text editor and now I'm in the server application and within this, I'm going to make use of the time module.

[Video description begins] *The server.py file opens*. [Video description ends]

The idea here is to create multiple instances of a product and then transmit them.

[Video description begins] *The following code appears in code line 4: import time. The succeeding code lines shift one line down*. [Video description ends]

However, between each transmission, I want there to be a wait for a certain period and this is something which can be done when using the time object. So following that, we will now get rid of this single custom object which we have created and replace this with a list of products.

[Video description begins] *He highlights code lines 10 and 11*. [Video description ends]

So we have not one but five different objects of a custom product.

[Video description begins] *Five lines of code appear in lines 10-14. Line 10 is: custom_objects = [Product('P024', 'Torch', 13),. Line 11 is: Product('P025', 'WaterBottle', 5), . Line 12 is: Product('P026', 'Keyboard', 20),. Line 13 is: Product('P027',' Mouse', 15),. Line 14 is: Product('P028', 'USBCable', 2)]*. [Video description ends]

So how exactly do we transmit all of them? Well, instead of having a single invocation of client.send, we will now make use of a for loop where we iterate over each of the custom objects which we have defined and then following that, we will first pickle each of those products, where we invoke pickle.dumps and then we invoke the send function where we send over this pickled object.

[Video description begins] *He highlights code line 24. It reads: client.send(pickled_object). He deletes it*. [Video description ends]

[Video description begins] *The following code appears in code line 24: for product in custom_objects:.* [Video description ends]

[Video description begins] *Code appears in lines 26 and 27. He highlights line 26. It reads: pickled_product = pickle.dumps(product).* [Video description ends]

[Video description begins] *He highlights code line 27. It reads: client.send(pickled_product).* [Video description ends]

So in each iteration of this for loop, we will serialize this product object and then send it over to the client. Now in order to perform a quick comparison between the product which was sent and the product which was received by the client along with the order in which it took place, we will now print out the ID of the product which was transmitted and in order to prevent the various transmissions from getting mixed up, we will make sure that there is a wait of 2 seconds between each transmission of the product instances.

[Video description begins] *The following code appears in code line 29: print('Sent product:', product.pid).* [Video description ends]

So for that, we invoke time.sleep.

[Video description begins] *The following code appears in code line 31: time.sleep(2).* [Video description ends]

So this concludes our changes to the server application where we now instantiate, serialize, and then transmit five instances of our product and we wait for a total of 2 seconds between each transmission.

[Video description begins] *He highlights code line 16. It reads: s.listen(5).* [Video description ends]

We can now save down server.py and then head over to our client application. You'll observe here, that we are receiving all of the messages from the server, unpickling the message, and then printing out the ID name and price of each of the product instances.

[Video description begins] *He highlights code line 18. It reads: product_object = pickle.loads(msg).* [Video description ends]

So we don't really need to make any changes to our client application.

[Video description begins] *He highlights code lines 20-22.* [Video description ends]

This time though, we should be able to receive five different products. So we can now head straight back to the shell and then run our server application. So the server is now up.

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

So we head over to the client and run this as well and we can see that the first product has been transmitted and received.

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

Product number two has been sent over. Number three has made its way there. Number four, and now all five products have been transmitted from the server over to the client. You'll observe that the IDs of the products which were sent over match the IDs which were received and all of the other data is consistent with what was sent from the server as well.

So we now have a fairly good understanding of how sockets can be used in order to transmit Python objects. Earlier on I had touched upon the fact that sockets are very useful in order to communicate data between two different Python processes or applications. In a realistic setting, there's a very high likelihood that the data which will be transmitted will be in the form of Python objects and now you know that you can make use of the pickle module in order to make that data transfer happen.

The object which we have been transmitting so far, however, have been rather small in size. Things need to be handled a little differently if the objects happened to be much larger. For example, if we're transmitting large files such as images over the connection, we will observe exactly how that can be accomplished using sockets a little later in this learning path.

Course Summary

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

In this course, we covered a number of basic concepts around socket programming in Python and the communication of small amounts of data between Python applications. We started off by looking at the socket module in Python and used it to set up a simple TCP socket in order to transfer some text from one app to another. We then covered some of the properties of Python sockets such as its use with the with context manager and the setting of a timeout for connections.

We finally made use of the pickle library in order to convey Python objects over a socket connection. At this point, you will be familiar with the basics of socket programming in Python and can build apps, which are able to communicate small bits of data ranging from simple text to Python objects. You now have the skills to move on to exploring sockets a little further, in order to see how large files can be transferred between Python apps by breaking them up into smaller chunks.