Week 4
Video Transcripts

Video 1 (06:29): Introduction to MySQL

All right! So today, we're going to look at structured query language, SQL, and this is the way that we talk to relational databases. So, what is structured query language? It's the language for relational databases, and that's the way we query it, we change it, we make modifications to the database itself, It's a declarative language. What that means is that we roughly, or in a manner of speaking, we tell...tell the database what we want, and the database figures out how to deliver it to us. So, we describe the result, and we get the response. It's a very different thing from Python because Python is a procedural language. You tell—you list all the steps that are required and you get your result. Here, you tell it what you want, and the steps are taken care of by the language itself. SQL acts on tables. If you recall in our relational database, we are dealing with tables. And, each table has a bunch of columns and a bunch of rows.

The rows define—the rows are the data and the columns define the different fields or different things that are inside our database. So, SQL essentially acts on tables. You've given a query and what you get back is another table that you can then use as...as if it were an existing table, and...and query it and pull out the rows and get your data and all that kind of stuff. So, that's the basics of SQL. The elements that we want to, again, to sort of, rehash what we said before is that we're dealing with tables, which are logically connected sets of data organized as a set of columns and rows. We've done all this under relational databases. A record is a row of information about a single item in the table. A field is an individual data item in a record. So, if you had, for example, a table that contains employees of a company, then you could have the employee name, employee number, maybe their designation, their rank or whatever, maybe salaries, you know, all that kind of stuff. Each of these things like, name, designations, employee number, salary is a field, and each row contains data about a single employee, and the entire table of employees is the table that we work with as a...as a...as a SQL, the relational table.

The–in SQL, on any database, when you are actually dealing with a database itself, you need to precisely tell the database what format a field has. So, if for example, you want to– do you want to tell it whether your format is a number or a string, a date etcetera, and you also need to tell it, how big each field can be so that the database can figure out how much space to keep for it. It's—since, we're dealing with physical, actual physical storage, you have to give it as much information as possible so that the computer or the server can designate storage and store it appropriately and you know, do all that kind of good stuff that it's supposed to do. And finally, we want—in any table, we want something called a key field. It's not a requirement but it's a good idea. The key field is a field whose value uniquely identifies a record in the table. Why it a good idea?

Well, it's a good idea because the database then can use the key to create a separate index, and then that index can serve as a very fast mechanism for accessing data from the database. If you don't have a key and you're searching for something in the database, the–even the database server will...will have to

go through every row before it finds the row that you're looking for. And, since the data table can be very big, and you can have a...a single row that could be hundreds of bytes long that would be very inefficient. It's much easier to take a single field out or maybe a couple of fields out, create a key for that and then, create a separate index table and then, the database can search through that index to find your–the data that you're looking for. So, it's a good idea to put a key field, and as a general practice you should do it, okay! Always create a key field, we will do that here. So, we're dealing with our SQL Servers. Servers are the programs or—that manage the database, and they usually sit on a separate machine somewhere.

They provide database services to clients. Clients are you, me, maybe other programs. It could be people. It could be programs. It could be anything, right! And, they're...they're modeled as a client server system. You must have heard this term before. A client server system is the server runs independently, and you can have multiple clients, and each client queries or acts upon the server and gets data back. The server somehow manages to make sure that there's no conflict between what the clients are doing and that there are no inconsistencies that arise because of multiple clients accessing the system, you know, changing stuff, while other people are looking at it, that kind of thing.

So, there...there are many—so, databases typically run on servers, and there are many different choices that you can have—that we can have for that. The most popular ones are listed out here. There's Microsoft SQL Server, there's Oracle, there's DB2, there's PostGreSQL servers, and there's MySQL. We're going to look at MySQL because MySQL is a...a open source server that's, you know, is very easy to get. It's free, so, why not! And, we can just use that. And, it also has a very nice server APIs for connecting to Python, other servers do too, but MySQL is probably the easiest to deal with. So, what's MySQL? It's an open source relational DBMS. An open source is like—Python is an open source language, MySQL is also open source. There's lots of resources available for that. The server, MySQL server is where the database resides.

So, we are going to set up on your local machines–we can set up a server– it's not a remote server, it's a local server, but you can have multiple clients even on your local server. Clients are the programs or the interface or the–whatever that talks to the server. We'll use one client called, MySQL Workbench, and we'll also use Python, our Python program as another client. And, a server can have many clients so that's the...the server that we're going to be using. You should have downloaded MySQL, following the instructions on our course site, and should be ready to go with that but, if you're not, it's a good time to stop and deal with that.

Video 2 (10:10): Introducing MySQL Workbench

All right! So now, if you've got your server running, MySQL Server running, and everything's good to go, let's take a look at what SQL queries look like. So generally, you know, kind of a very rough way, we can think of queries, as being of three types. They are data definitional queries, which are used to define the table to change the table in some way, maybe change a field definition and that kind of stuff, the data manipulation queries, where you insert, delete, or update actual data. So, data definitional queries are the queries that are defining or changing the schema. The schema is structure of the database in some way. Data manipulation is the queries that are changing the data itself. And, the third element is data

querying. Data querying is the process of getting data from the database. This is where we will typically do maximum work—is this—because usually you are querying a database, trying to find information and that's where the power of the...the...the querying language, the one that is descriptive, comes into play. So, let's take a look at some of these queries. So, we start with our basic data definition queries, and we can see that there are a bunch of them. So, let's take a look at here– our—the...the basic queries.

Well, the first thing you want to do, of course, is see what databases exist in your server, right! So, there's a simple command called 'SHOW', and what 'SHOW' does is, it will show you the— 'SHOW databases' will show you the list of databases. 'SHOW' shows you something from the database. We'll see that. It can show tables, it can show databases, etcetera, right! So, this is our SHOW database command. You can create a database and that uses the command called 'CREATE', which says create a database. So, it creates that and we give it a clause here that says 'IF NOT EXISTS', which means that if the database actually exists, and you try to create it, you're going to get an error saying 'The database already exists'. So, 'IF NOT EXISTS' is sort of protection, and says only if it doesn't exist, then you should create it. Once you've got that and you want to actually use a particular database, so inside a database server, you can have multiple databases. So, you tag which one you want to use and use a command called 'USE', use this particular database. And finally, if you want to drop the database itself, you use a command called 'DROP' and tell it what you're dropping is a database and you give the name of the database.

So, we have a sample database called 'SchoolDB', which I will take a look at in a minute, and that should do that thing. So, let's now move on and see how this works on our...on our MySQL Workbench. MySQL Workbench is a GUI client for MySQL. So, let's take a look at it. You should be to run it on your machines, if you downloaded it following the instructions on the—on our class site. And, when you open up MySQL Workbench, you are going to get a screen that looks something like this. So, here we've got our screen, which says we've got MySQL connections. So, connections are the connection that you as a client, or in this case, the workbench as a client makes to the server. So, from the same MySQL Workbench client software, we can open multiple connections, but we're not really interested in that.

We'll start with a single connection, and it says over here that the connection we're going to open is the Local instance 3306, which is— 3306 is the port on which it's operating and it says that the connection remains open with the user name root, and that's okay! Ideally, when you have a client, the client should be opening the connection not as a root because root has, you know, complete privileges to destroy stuff, if they feel like it, which is not a great idea. But, we are running a very–a local installation, and we are not really worried about messing stuff up, so let's do that. So, you click on this and it should open up this screen here and here you can give a password to open it.

My password that I've done is NONE, which is our Python none. And, as a general rule, you can just save that, so you don't have to keep entering it every time. And, then say okay, and that brings up the...the interface—to the...the gray interface to your MySQL server. So, once you've got this—in mine I already have something else, so I'm just to cut that out. So, this comes up like this and it– what it does is, it's got a whole bunch of stuff but you don't worry too much about it. The main thing here you want to look at is the thing called 'schemas'. So, we see there are two schemas, there's a 'test' schema in mine, which you may or may not have, and there's a 'sys' schema, which is the system stuff. We're not really going to mess with the system schema but—or the test schema for that matters.

We'll create our own stuff along the way, So, the...so the main thing that we're interested in is this big window in the middle that is our query window. This is where we enter the queries and get our results. So, let's try a couple of things. We have two databases here already, let's create a third one. So, I say 'CREATE database', and note how it very nicely helps us by— the...the workbench are of helps us by prompting us with stuff, and we notice that it's got a little red X next to that which means the query is not yet complete or it's an incorrect query because it's not complete. So, we can do CREATE database let's call it 'Schooldb'. That's what I called it there and once you've done this, we can execute this thing by hitting the little lightning thing over there that executes a query, and it says 'CREATE database Schooldb', then it does this in the bottom of the screen over here it says, 'one row affected' and there's a little green checkmark next to the line that tells us that yes, it's worked fine. So, we've done this.
Now, what we want to do is we can— next we can try creating a table inside—or using the database, of course, we need to use the database first. So, I say 'USE Schooldb' and I can execute this. So, what happens with this MySQL workbench is that you can either execute the entire thing by hitting the little lightning stuff or execute only one line and that's what we're going to do here is just one line. So, let's do that. It's a little bit clumsy. So, we've got 'USE Schooldb' that does this, yes, we're using 'Schooldb' now and then I create a table 'create table', let's call it 'student', and I give it a list. So, this is the table that I'm creating. I need to tell it what different elements I'm going to have in that. So I could say here, 'name VARCHAR (20) comma' and let's give it— what else should we say? 'city VARCHAR(20)' and again, just to make sure, I don't run the whole thing— if, I run the whole thing, what's going to happen is that, oh, yikes. Sorry about that.
Let me run this again, run this and I get a student table over here and now, you know, what I want to do, I want to just delete all this stuff so that I don't have to keep highlighting it. And, now I can do 'show tables' and it'll show me the tables, it tells me one table called 'student' and I can see what's inside it, and [inaudible] or I can describe the table, so see the contents of the table itself, 'describe student' and tells me I have two elements: a name and a city, so, this is a very simple way of using the...the...the client that we have here, the 'workbench client'. And, what we will do here is we will now go on and take a look at what else we can do with this.
So, let's clean this up till we come back to our data definition queries in a second. Let's clean this thing out. So, what I will do is, I'm going to 'drop the database SchoolDB'. So now, we don't have that, database is gone, and we are now—if I look at my all the databases, I shouldn't see that anymore. 'show databases' Right! There's no SchoolDB now, right! So, the other databases that are there in the system, test—there's a test once, this is the—you know, and the rest are actually we don't really want to worry with them. They're really essentially system-level databases. So, this is our—the...the workbench client.

Video 3 (09:58): Database Example (Part 1)

So, we will...we will use the simple example database to look at queries in our lecture today. The database contains three tables, a table first called 'Student' that contains student information, a table called 'Course' that contains information about courses, and a table called 'Enrolls-in' which as you can see is a relationship table between these two entities. So, 'Student' and 'Course' correspond to entities in our Entity-relationship model and 'Enrolls-in' is a relationship and we can see that the relationship as

defined by a...a key from the student table and a key from the class table. So, what we have in a database is three tables, the 'Student' table which has a key social security number, a 'Course' table that has a key course number, and the 'Enrolls-in' table that has a composite key that contains social security number and course number right! Assuming of course that no student can enroll in more than once in a class, right! That's an assumption there. So, we've got this stuff here, so this is our...our basic structure that we're going to work with. So, the first thing you want to do is to add this data into your database, right!

We'll take the database schema itself and create that in our database and then we're going to work through that to add...add the various table data as well and use that in our examples. So, let's do that and walk through these steps a little bit over here. So, the first thing you want to do is to download these two files, 'school_schema dot sql', and 'school_data dot sql' from our course website. Once you've downloaded them and you are set to go, we're going to run these two files. So what we can do in SQL is you can take a bunch of queries and put them into a single file and then run them as a group in your– from your client, whatever client it is. So, these two files, 'school_schema dot sql' contains the Schema definition, which is the definitions of the tables, the fields, and the keys and all that kind of stuff. So it's...it's called the 'CREATE' commands, and the 'school_data dot sql' contains the actual data, it's going to go into the tables itself. So, once we've done that, you want to open MySQL workbench which we've done before, then you want to open 'school_schema dot sql', and let me walk you through those steps, and execute the script. So, let's walk through that. So, what I'm to do is, I'm kill this window here and if you look at this little tab over here which looks like a folder under...under query '1', I click on that and now it will...this allows me to open up a...a file that contains the...the query that I want to execute, the bunch of query that I want to execute. So, my query—my files are sitting in this directory called Hardeep Johar, and if I go down here, I get 'school_schema dot sql' and I want to open that, I want to open that—this the shows up over here.

So, let's walk through, some of these steps to understand how SQL works. So, here the first line that I have in this thing is 'DROP DATABASE IF EXISTS SchoolDB'. So, what this does is it makes sure that if I already have 'SchoolDB' defined for some reason or other in my database...my database server, then I want to drop it, I don't want it to be there anymore. So, that's because it's a test sample, definitely you don't want to do this as a general rule because you might be dropping a database that is kind of useful, right! and it'll just drop it. So, you want to be careful especially in it's your root. Then I create the 'SchoolDB' and this is— we've already seen this command, it says 'CREATE DATABASE IF NOT EXISTS SchoolDB'. We know it won't exist because I just dropped it but it's there and then I use 'SchoolDB'. Now I want to create the three tables that we have in our— that we want to define for our database. The three tables are 'Student', that's this one, 'Course', that's this one, and 'Enrolls_in', that's this one. So, each table I have a 'CREATE TABLE' command and what...what I say is if the table doesn't exist then create it, I don't want to override an existing table. For each table, I define the fields.

So, for the student table, I have six fields ssn, f_name, l_name, phone, city, and zip. And, for each field, I have to explain, what data type that field is. So, in this case, they're all VARCHAR. VARCHAR means a character field of variable length. So, some people's name is John, and other people's name is Matthew, or Gellibrand, or whatever, so everyone has different name lengths, and so VARCHAR essentially says, you can have differing name lengths and they don't have to be exactly, you know, X characters or

something like that. So, generally text fields, it could be, for example I could have used CHAR11, for social security number because social security numbers are always 11. So, if you use CHAR, you have to have the same exact amount, if you use VARCHAR, then you can have a variable amount. So generally, you know, it's safer to use VARCHAR but I could have actually used for both CHAR—for...for zip as well as social security number, I could have just used CHAR. The next thing I use, I say is the social security number, for example is NOT NULL that means that I cannot have a student in my student table who does not have a social security number, right!

Everyone has to have a social security number. So, NOT NULL says that this particular field cannot be null. So in general, all my fields are NOT NULL except for the phone field, so, a student perhaps doesn't have a telephone, kind of odd but you never know. And, the score field in the 'Enrolls_in' because obviously before the semester started, they don't have a final score, so that could be null–they don't have– there have been no, let's say, evaluation mechanisms that have been started so far, initiated so far. So, that's the main thing here, so NOT NULL essentially says, that this field cannot be null. If you add data into the table, then you want to make sure that...that data is there, otherwise it won't work. Then finally, the last thing I have here is this word called PRIMARY KEY and what PRIMARY KEY says is that social security number is the PRIMARY KEY for the table. What the database will do is, it'll ensure that I cannot have more than one instance of the social security number field in my table. So, that's one of the advantages of PRIMARY KEYs. It makes sure that you can't duplicate them. So, only one rule for every student.

In the case of 'Course', one row for each course number because course number is the PRIMARY KEY and in the case of the 'Enrolls_in', my key is a composite key because I have both the social security number as well as the class and the key. So what I do is, I add a 'CONSTRAINT' over here which says that I'm going to create a 'PRIMARY KEY' called 'pk_enroll', that's like a creation on the fly and that is a...a composite of social security number and class. In this case, the PRIMARY KEY is going to be used mainly for indexing the table. That's the...the goal really with this thing over there. So, that's our basic structure and this is now—is going to—once we run this is actually going to create these three tables. And, to run this all, we need to do is to go to this little lightning thing over there and click on it. And, I get this stuff here and I want to make sure that it's all run properly. It says one warning over here, because I'd already deleted SchoolDB, so I couldn't actually create it again, so that's gone. And then, everything else seems to have worked fine, So, I am going to check to see that all these— every query has a green checkmark next to it to make sure everything worked properly. So, that's our database defined and we can go back and check this stuff to see whether it actually worked or not.

So, if I just delete all this, I can do 'SHOW tables', and run that, and I get 'Course', 'Enrolls_in' and 'Student' right! My three tables are defined in this thing here. I can see if they've been defined correctly by looking at the describe. So, let's say 'describe enrolls_in'. A couple of things about queries, every SQL query has to end with a semicolon(;) because that allows us, of course, to use multiple lines, if we want to. And, as a matter of convention, the keywords in the query are, you know, the...the actual SQL commands are often in uppercase. So, I usually have to do something like this right! That's just convention, it's not required. And, the third thing is that even though the table is called 'Enrolls_in', it's not case-sensitive. So, this is not really case— SQL is not case-sensitive. It doesn't care whether you use, whether you define it as uppercase. It has to be uppercase. it's very unlike Python variables in that

respect. So yes, we have all these three things here and it says ssn, class, score, not that we have Key PRIMARY KEY primary here because this was a...a this was a required thing there and that's it really, right! So, and this says NO, NO for Null, there can't be null and score can be null, okay, that's what we define it as. So, that's our—the first step here, we record that in.

Video 4 (04:29): Database Example (Part 2)

The next thing we should do is to load up our data itself. So, again, we go click on that folder icon and select 'school_data dot SQL'. Now, open that and—we don't really care, so, I will say don't save. If you want to change it, you can save it. So, we get this here and this shows us our various commands that we're going to use to add the data to it. The main command for adding data into a database—into a table is the 'INSERT' command, and you can say 'INSERT INTO', and give it the name of the table. So, the...the word 'IGNORE' over here essentially says that if the social security number already exists in the Student table, then don't do this insertion. The reason is that since social security number is the primary key, if I try to insert a new row with an existing social security number that... that, if someone already has that social security number, then I'm going to get an error, because it will say duplicates aren't allowed. So, what 'IGNORE' does, it says try to insert this in but if it doesn't work, then if...if the key is already there, then just don't do anything, forget it.
There is also an...an 'INSERT UPDATE', that what it will do that, if...if the key already exists, then it will change all the data, so that it now reflects the new data that we are inserting into it. So, there is 'IGNORE and UPDATE', but that's a finer point, we can ignore that for now. So, just for the sake of completeness, I've– and to make sure the...the query doesn't get messed up, the query script doesn't get messed up, I put 'IGNORE' over there. So, 'INSERT IGNORE INTO' and I give it the table name over here, 'Student', and here I'm just saying 'VALUES', and these are the values. These values now need to be in the same sequence as the table structure. So, the table structure sequence has social security number, first name, last name, phone number, city, and zip. So, they have to be in the same order because this is going to get inserted into that order.
I could also, if I like, specify the fields that I want to enter. So, for example, in the Student table, the phone field was not a required field, it could be null. So, if I don't have a phone, I don't have to do that and we have a problem here. I have to put a null where this is, right! So instead, what I can do is, in the case of a student number two Mary Arias, I don't have a phone, so now, I'm saying 'INSERT INTO STUDENT', and instead of going straight to values and giving the fields that I want to insert inside parentheses. So, the fields I want to insert are 'SSN', 'f_name', 'l_name', 'city', and 'zip', and now the values need to be in the same order, as the fields that I've given here. So, these don't have to be in the order that's in the table, I can use any order over here but the values should match that order. By default, if you don't specify the fields, it's going to assume that you're entering all of the data and they're going to be in the sequence that is there in the table definition itself. So, that is 'INSERT'. So, here I 'INSERT' all this stuff and 'INSERT' all this stuff and so it's pretty clear, I think and we can run this. Again, by clicking on that and check and make sure, that it all ran nicely. Yep, everything seems to work well. So, starting with this one here, inserting new into student values, and we've got our table in. So, that's the...the basic thing. So now, once we've done this, we have all the data that we want. So, again

let me get rid of this and I can take a look at my stuff, so I can say 'SHOW'. Well, the simple way to check this is to use something called 'SELECT' command, which I'll talk about in a second but let's say I want to see if the data is inside here, so I can say 'SELECT * FROM' and give it a table name. So, let's take 'Enrolls_in' and run that and it tells us that we have these rows and—are—and—rows in table that seemed to work well, so I think we're in good shape over here. So next, we're going to start looking at the queries itself.

Video 5 (10:37): Select Statement (Part 1)

So, now let's see how we can query our database. So, the main command that we use to actually create the database is the 'SELECT' statement and this is a very powerful statement with lots of variations that we can add onto it, clauses and conditions, and all kinds of stuff. So let's, you know, focus a little bit on that. So what does 'SELECT' do? Well, the idea is very simple. What 'SELECT' does is it, given a command—it what it does is it says, give me a set of records from one or more tables. So, it could be one table or many tables. So essentially, what you're doing is you have a database that contains a lot of information and you want some information out of it. The data is scattered. It's sitting in multiple tables, in 'student', and in 'Enrolls-in', and 'course' or maybe, you know, hundreds of tables and you want to be able to look across these tables and pull out information or the...the information that you need. So, it might need to go through multiple tables to actually get the information that you need and that's the idea here.
So, when you give a 'SELECT' command, what you're doing is you're saying, go to my database, and get me the information that I want. And this is—to do that, you have to, of course, know the structure of your database, the scheme of your database and you tell it which tables to access and what data to return to you finally, okay! That's the idea here. And, what you're doing is, you're actually just describing the result set. The goal is that the server will take care of the processing or actual procedure. A computer, by definition, is going to go through its process step by step and very micro step by step but we don't have to tell it how to get the information because you tell it what we want. We want data that combines, information from this table, that table and that table and bring it all in and that's the...the idea here with this thing. That's what 'SELECT' does. So, let's take a look at the 'SELECT' statement. The...the basic clauses that we have are— the basic structure is...is we...we saw an example earlier is that you start with the word 'SELECT' and what you're saying is, this is the selection of information that you want and you tell it where the data is coming from. So, this is pretty much the basic structure. You're only going to have 'SELECT' and we tell it, what you want selected. So, the star here is what we want and the 'FROM' is for where is it coming. So, 'SELECT' is the verb that we are saying to get the data and 'FROM' tells the database server, which tables to access the data from. You can, you know, the same information in multiple tables, so you want be...be very clear, which tables you want it from. So, the simple statement, of course, is the 'SELECT star FROM Student', which is going to return all the data, every row in the student. Well, of course, you don't always want everything from a table. You want to choose something about a particular student or some information about that is restricted. So, the next thing that we have in our 'SELECT' statement is, the clause called 'WHERE' and what the 'WHERE' clause does is, it's like the 'if' in case of in...in Python, what the 'WHERE' clause does is it sets a condition for the

selection. So, here in this second query, what we are saying is, 'SELECT star FROM Student WHERE City equals "New York".'
So, we are saying get me all information about every student, who has a city value that maybe that's where they live, that's New York, right! And notice that this equal sign over here, is the ordinary equal sign, it's not the double equals that we normally see in procedural languages. So, this is our description. It's saying, "Hey database! give me the list of students, all information of students who live in New York." That's what we are saying. So, database figures out how to do it and delivers it for us. We can also choose specific fields. So, when we say 'star', what we're doing is, we are saying give me all the information but we may not want them all. Maybe, we want only the first name and last name. So, now you can say, "Hey! the only things I want are f_name and l_name." So, these field names should correspond to the names of the columns in the database and, you know, everything else is pretty much the same. So now, what we're getting is the first name and the last name of the student from New York. We can also sort them.
So, this we want to select Social Security number from the student table and order them by first name and we can see ascending. The default is ascending but you don't have to say ascending. But, you can say ORDER BY, and ORDER BY is saying now that the data that we get should be ordered by the first name of the students, sorted by the first name of the students. ASC is ascending and we can also see it by descending, and that will take care of that for us. And finally, we can get— we can also say something like 'SELECT DISTINCT', which says get me the unique cities from the Student table, if you want to know, for example, which cities our student population comes from. So, let's take a look at some of these things here. So, let's say we go here and so I say 'SELECT star FROM student' and that gives us the entire student table, all right! I could say 'SELECT star FROM student WHERE city equals New York' and run that. I thought, I put it, what the heck. I forgot the double quotes, so, it gave me an error, and I can run that. And, that tells us that the students that we have are from New York are John Childs and Mary Arias, I could say, 'SELECT distinct city FROM student' and that will work.
City equals New York for that and I run this and it tells me the distinct cities are New York, Seattle, and San Francisco. And, I could actually add, if I wanted to do it here. We can see, that these are not sorted, because Seattle is—should come after San Francisco, so I could say ORDER BY city and run that and now they're sorted. I could order by city descending and we should see them in reverse order, and roughly, that's what we get here, right! So...so that's the basic structure for this stuff here, I've got them all.
Yeah, I can— Oh, yeah. So that's, the basics of the 'SELECT' command. The next thing we can do is, add some more information to that. So, what we're doing is, we've chosen in our initial set-up, we've chosen to extract actual columns but we can apply functions to the columns as well. So, for example, if I look at the 'Enrolls_in' table, there's a grade column or a score column in that table and I can find the average. So, average, obviously you can only find for numerical columns, the integers of floats or whatever. And so, I can find an average for the score from that thing here.
So, what this says here is, that you want to select the...the average score from 'Enrolls_in'. So, it's going to find, the average from that table there, we can do that selectively. We can say only for the class 'c1', so that would find it only for class 'c1' and we can also GROUP BY. We can, for example, let's take a look at this actually. So, I could say ‘SELECT’. So, let's take a look at, our table here. So, let's start from— so looking at the 'Enrolls_in' table, it has three columns, social security number, class, and score and what I

want to find is the average, so I replace the 'star' by the average of the score and that tells me the average is '71', okay! Then, I can select, and redo this. I could say, 'WHERE' class equals "c1". So what that is doing is, it's telling...telling us that we want to choose only for where the class is 'c1' and I click on this and it tells me that for 'c1', it's '72' is the average, okay! And then, I can replace the 'WHERE' clause by— what I want to do is, I want to say here, if I do 'SELECT star, from Enrolls_in', I have three different classes; 'c1', 'c2', and 'c3', right!

So what I want to do is, I want to find the average by the class. So, rather than having to do where class equals 'c1', class equals 'c2', class equals 'c3', I can say do this for the entire set-up, by using a GROUP BY clause and saying 'GROUP BY class', and give me the 'average score'. So, what this is going to do now is, it's going to take each class and group them together. So, in a sense, what's happening is, that we get these records here. So, 'c1', 'c1', and 'c1' grouped together, and you find an average for this group and then 'c2' which is only '1', we find the average of that group. Then 'c3' which is only '1', we'd find the average of that group, then we run this and we get the average score 'c2', '95'— 'c1', '72'; 'c2', '95'; 'c3', '44'. We can see that we can also add over here the class, the thing we're grouping by, and we get the information, that's organized like that. So, it's a very handy way of getting information or groups of data that we get from our database itself.

Video 6 (07:07): Select Statement (Part 2)

We can also do some other stuff. For example, we can use what's called the 'HAVING' clause. What the 'HAVING' clause does is it says that we are doing the same thing here, 'SELECT class, average(score) FROM Enrolls in GROUP BY class' Looking number four here, and to this, we add another clause, says 'HAVING count(score) greater than 1' So, 'count' is another function we can apply directly to our sum. You can add them up, average, count the many possibilities. But here, what we can do is, we could say, count the number of elements in the—number of students in the class. So, we're only interested in finding the average score for classes that have more than one student, for example. Right! So—in this case. So, let's take a look at that, and we can 'ORDER BY'–we can 'GROUP BY class' let's see, 'having count'—whatever, it doesn't matter, '(score)'—no, 'count class', sorry! greater than one, and I run this, and we only get 'c1'.

If I did 'class' equals '1', then I would get 'c2' and 'c3'. So, that's another convenient thing to do with this. So, 'GROUP...GROUP BY' is a very powerful command, and you probably use it lot often because typically when you are creating stuff you want to group them and by some kind of categorical data and...and get the information from there. You can also do simple things like renaming columns. So, for example, in our— list over here, we have a 'class' and an 'average(score)'. But I may not want to—when I'm presenting this table to someone, and giving a result, I may not want to put it like this. I might want to say something as—more descriptive. So I could say 'AS'—whoops. I call it 'average', and run this, and now the column name gets changed. So, it's kind of a convenient way especially for reporting that is useful to do. Then, I could also save the data in a temporary table. For example, if I were running this query here and I don't have this clause in the end, the 'HAVING' clause, so now I have this thing here, I run this thing getting this table.

I might want to do some more research on this table. For example, I get this table and then I might want to, you know, look for classes that are high average classes versus low averages classes or do some other grading stuff on it. So, what I can do then is, I can use this command here that says 'CREATE TEMPORARY TABLE' to create a table that is temporary, and what temporary means is now is that it has a short-term existence and I can then use it for some further analysis, and then we drop it. We don't want to store it in our database. That's the idea of a temporary table. So, here what I'm saying is, 'CREATE this TEMPORARY TABLE' called...called 'averages'. All right! The table called 'averages' and, what it does is it does our rate 'score' as 'average' from class and 'average(score)' is 'average', from 'Enrolls_in' 'GROUP BY' class, all that kind of stuff. So, I can just go back here, and save this in my thing by adding this in the front, 'create temporary table averages' and run this, and I'll go back and then I can say 'select star from averages'. Just run that and it tells me what the data over there is.

So now, I have a table that–I can then use for further analysis. So I could, for example, 'select star from averages where average is less than 90'. So, these are, let's say our problem classes. Right! So, I can go here and run that, and I get just these two tables. From something like that. You know, you can–you could do more averages, counts, or sums or whatever, right! So, that's the thing with that and we can of course, you know, complicate this as much as we like. So, for example, here I take the same query that I have and I can 'ORDER' them 'BY' 'count' or, you know, 'ORDER' them 'BY' 'class' name, or whatever, right! So, this is exactly the same as that. So now, here what I'm doing is I'm saying 'SELECT' the class, the 'average(score) AS average, get the 'count' of the number of students in the class 'AS count FROM Enrolls_in' 'GROUP BY class' 'ORDER BY count' all that stuff. So, let's go ahead and do that here. Let's 'create temporary table', and we call that 'summary', 'SELECT' class comma 'avg(score) AS average' comma 'count(score) as count from Enrolls_in', and 'GROUP BY class', and 'ORDER BY count', and we'll do it in 'DESCENDING' order, and here I'm just going to 'select star from summary'. Let's take a look at this.

So here now, what I get is, I get three columns because that's what I've said over here. We have a column called 'class', a column called 'average' and a column called 'count'. And note that I'm using this 'count' as my 'ORDER BY' clause over here. So, I've renamed the 'count(score) as count', and I can use that term over here, count, in this. And I'm saying, "Hey! Give me all the three columns and order them in descending order by the number of students." So, it's 'c1' has '3' students, 'c2' has '1' student, and 'c3' has '1' student, and these are the average scores over there. That's the idea of it. So, that's the basics of the 'SELECT' command. You may want to practice a few queries on your own and then we're going to come back because typically when you work in the database, you have many-many tables. So, you want to be able to get data not just from one table, but across multiple tables.

Video 7 (10:04): Working Across Multiple Tables (Part 1)

Now we're going to see how we can use the–what's called a JOIN feature in a database in SQL to get data from multiple tables. So, first before we worry about that, let's just take a look at what we really want to do. So, let's say we want to get a list of student names and classes. So, we want to know from every—we have— in our...in our database, we've got a bunch of students, and we have a bunch of classes that they're enrolled in, and we want to get the...the names of the students and their classes

that they're enrolled in, like maybe c1, c2 is the class numbers. So, to do...to do this, we need to go across our database, and look at more than one table because if we look at our database structure here again, what we have is we've got 'Students' over here and 'Enrolls-in' over here, and we want to figure out–we know that they are linked together by social security number. But the names are in this table over here. Right! So, the name of 'John' 'Childs' or 'Mary' 'Arias' or whatever is in this table over here, and the classes 'c1', 'c2', 'c3' are in this table. So, we need to go across these two tables, and somehow get the data to take it out. We can see that the two tables are linked by social security number, right! So this, and this are, this '111-222-333' is John's social security number, and that's the linkage between the students, and the Enrolls_in tables. So, what we want to do is we want to say, "hey, I want to get", so, in the SELECT statement, the first thing you want to tell it is what data do you want to see. You know remember what we're getting from a 'SELECT' command is another table. So, what we want to tell it is what is the structure of that table, and the structure of the table is our SELECT something, right! So, the structure of our table that we want is the 'f name', the 'l name', first name, last name, and the 'class'. So, that becomes the start of our query. So, we can go here, and say 'SELECT', 'f_name comma 'l_name' comma 'class', right! That's what we want to select, right! Where do you want to select this from? So, our SELECT structure says SELECT the columns that we want, and from the tables that we are getting the data from. So, the tables that we want to get our data from in this case is the 'Student' table, and the 'Enrolls_in' table, so we can just list those from 'student' comma 'enrolls_in'. Okay! And let's run this and see what we get. I mean this is not going to be our...our answer, but let's see what we get.

So, we get is error code unknown column 'f_name', and field, and all that kind of stuff, right! So, the– well it's six sorry– forgot the word 'FROM'. So, here now what I get is a sort of composite, which has taken all our data, and said 'f_name', 'l_name', 'c1', so we get 'John Childs' 'c1', 'John Childs' 'c2', 'John Childs' 'c1', 'John Childs'—so it's sort of like a mess because it's taken all the possible combinations of every row in our 'Student' table with every row in our 'Enrolls-in' table. So, we get 'John Childs' for every possible combination here, 'c1', 'c2', 'c3', 'c1', 'c2', 'c3', everything, right! So, we don't really want that, what we want is we want a restricted thing, we want to say we want this to be contained to the condition where the social security number in this table here– so, we want it constrained so that we only want to choose rows where this social security number matches this social security number. Make sense? So, what we want to say is we want to say, "hey, look— take this row here, and look for a corresponding row over here, and include only that, don't include every row that shows up in that table there."

So, that's our condition and we want to add that condition in, and that becomes what's called the JOIN condition really. And we say here 'WHERE' the 'student.ssn' which is the social security number of the student in the student table equals 'enrolls_in dot ssn'. So, now we are...we are restricting our...our combining of these two tables to matching rows, right! Which match our social security number. And then we run this, and now we get our listing of student names and their classes. So, that's the...the way we go about doing this, we can complicate this a little bit, as much as we want actually. And so for example, if you want to see the list of classes that 'John Childs' is enrolled in. So, so far, in our first query, we are just restricting it to social security number matching, and that's it, so we get all the combinations possible. And notice that I have an addition thing here called 'a' and 'b'. So, what I'm saying is, that choose these columns from the table 'Student' but give it an alias 'a'. And the idea here is

that you just want to make typing easier, you don't want to keep saying 'student dot ssn', we might have multiple cases of student, and you know, multiple fields that we're drawing from it.
So, rather than having to keep typing the name of the table I'm going to use the alias 'a' or the alias 'b', and that's what I'm doing over here. I can make this even more specific by saying something like this. I could say 'SELECT a.f_name', 'a.l_name', 'b.class' 'FROM' 'student' 'a', 'enrolls_in b'. Just by–just saying 'a' and 'b' it becomes an alias, and then I'll replace 'student' by 'a dot ssn', equals 'b dot ssn', and execute this. The nice thing about this is that anyone reading the query can know clearly where 'f_name', and 'l_name' are coming from. If I don't—I don't have to say it. The reason I don't have to say it is because 'f name' and 'l name' are unique in my table, right! There's no 'f name' and 'l name' that is duplicated anywhere here, but 'ssn' is not unique. So, if I include 'ssn', I have to specify exactly where I'm getting it from. So, if I did, for example, here 'ssn', and try to run this, I get an error because it tells me 'ssn' column 'ssn' field, 'ssn' field list is ambiguous. So, where there's no ambiguity, you don't have to specify it. But where there is ambiguity, you need to specify it. It doesn't know which to use 'ssn' in table 'A' or table 'B', right! That's the idea there.
So, let's just keep that in mind. But as a general rule, if you use— if you disambiguate, means if you disambiguate every field then you're–'A', you're safer, you're going to run into trouble and 'B', anyone reading your query can see clearly where the data is coming from, which is always a good practice in any programming environment to make sure that everyone knows what the heck's going on, right! So, that's the basic structure. We can, like I said you can make–add more stuff to this. So, here I want to get information from the two tables, but I only want the—to see which courses is...is the student called 'John Childs' doing well in, right! So, by well, I'm defining that you get '95' or higher in the class. So, I can say where this and— and 'AND' is a combining clause over there. 'f_name' equals 'JOHN'.
Note that, once we are in the data itself, we have to be case-compliant, right, because it's...it's literal. And 'l_name' equals 'CHILDS', and score greater than equal to '95'. And again, if I want I'm going to execute this. It's only in 'c2' is John Childs getting more than that. If I want, I can. I don't have to, like I said, but I can always add the 'a dot', 'a dot' and 'b dot'. Remember, I don't have to because they are not duplicated, the same name is not used in multiple tables so, it's not...not going to be a problem. SQL can figure that stuff out, right. So, here what I've done is I'm saying okay take these two tables, combine them at the social security number— equals social security numbers, and then also add...add this condition that from the— in a sense what I'm saying is from the resulting table make sure that the only rows that I get are the ones that contain first name 'John', last name 'Childs', and a score greater than equal to '95'.

Video 8 (12:00): Working Across Multiple Tables (Part 2)

Now, I can go across three tables. So, if I want to get the names of the courses that John Childs is enrolled in, then I need to go and look at not just 'Student' and 'Enrolls-in' from here—from 'Student' and 'Enrolls-in' I'm going to get the class numbers, right! If he's–John Childs is enrolled in 'c1' and 'c2' and then I need to go from there to the course table and pick up the name of the class itself. So, 'Data

analytics' and 'Python' right, and you pick that up from that table here itself. So now, I need to combine data from three different tables, and as you can see, you can make it for four, five, six, whatever, it doesn't matter, right! So, if we look at the query itself that's over here. So, what this says is 'SELECT' the 'name' from 'Student a', 'Enrolls_in b', I'm saying 'name' over here because if you look at our table structure, 'name' is actually a unique column for this thing here.

The 'name' is not the first name, last name, it's not the same as the student name it's the name of the course. So, I'm saying select the 'name' field from— and this is saying now, the 'from' is really saying from the resulting table, right! So, when I say select these three things, I'm saying get me what are the result of all this stuff is right! So, here I'm saying select the 'name' field from three tables, 'Student', 'Enrolls_in' and 'Course' and I'm giving them the aliases 'a', 'b' and 'c'. Then I'm telling them the condition of— the condition is now going to be–I want to join across multiple tables. So I'm going to say 'a.ssn' equals 'b.ssn' and 'b.class' equals 'c.number'. So, here what I'm doing is I'm saying take this table and join it on this, oops wrong one, join it on this. And then, from the result that is what we get after joining these two tables you get one big table, we've seen the result of that, right! We take this table and join it on this.

So, the resulting table, that is a combine…combination of 'Student', and 'Enrolls-in' it will be joined on the class, and the number, right! And that's what this is saying over here. where 'b.class equals c.number'. So, that's our...our query over here, so let's take a look at this. So, we're getting the name of the courses that John Childs is enrolled in and I can say here, 'SELECT c.name FROM student a comma enrolls_in b comma course c WHERE a.ssn equals b.ssn AND b.class equals c.number AND f_name— a.f_name equals JOHN. AND a.l_name equals CHILDS'. I hope that's correct, let's try it. Yes, we get the two names of the two classes that John Childs is enrolled in, and we can add more stuff to this. Or I what could say 'c.name comma b.score'. So, I get not just the name but I also get the score that John Childs has in that— in each of these classes, right! So, it's a...it's a nice reporting feature. And the reason of course is that we have the data, as called, distributed across multiple tables and we need to combine data from all of that to actually get it. So, what we have been doing so far is combining data in these tables and the technical term for this is what's called 'Join'. And the whole idea of the 'Join' is that you take two or more tables and you combine the rows in these two tables. 'Join' is a–there's a formal way of looking at it. We have been looking at very informally because our goal is much simpler.

We are not into really learning SQL. We are more into how to use SQL practically in our applications, right! So, we are not going too deep into it, but the idea as we said before is very straightforward. We have two...two different tables, 'Courses' and 'Enrolls-in', and what we want to do is, we want to combine these rows in some way, so you look for a link between the tables. And typically, the link is going to be in a key or a part of a key, right! So, here the part of the key that we're looking at, so we know this is a relationship right! 'Enrolls-in' is a relationship between 'Student' and 'Course'. So therefore, the key of course is going to figure inside the composite key of 'Enrolls-in', that's how this...this...this works, right! And it's a many, many relationship because one student can be enrolled in one course and one course can have many students in it, right! One student can be enrolled in many courses and one course can have many students who are enrolled in it. Therefore, we have this composite key that really has both these things inside it, social security number, as well as class. So, but the...the linkage between 'Enrolls-in' and 'Course' is going to be through this stuff over here. And the

whole idea in the 'Join' is that rows in 'Enrolls-in' are matched with rows in course where class has the same value as number.

So, what SQL does is it provides us with a special feature called– a special command called 'Join' which will force– which will tell SQL that we are joining the two tables together. Over here, we are putting our 'Join' condition inside the 'WHERE' clause. So, the query doesn't really say there's a 'Join' coming up even though there is a 'Join' coming up, and therefore, in principle, SQL- the server can't optimize this for–this query. Whereas, if you explicitly tell it that we've got our 'Join' coming up, then it can optimize the query. Let's take a look at our first example here. So, we've got—the question here is that we want to get the listing of the course number, the room name, the...the course number, the course name, the room number, the social security number of a student in that class and their score, not the name of the student; we're just doing the social security number.

So, in a sense what we are saying is we want to take these two tables, 'Course' and 'Enrolls-in' and in those—from those two tables we want to get these—all the columns, the number, the name, the room, the social security number, the class and the score. We're not interested in the student column at all—table at all. Therefore, our combination is across these two tables and that's what we are trying to do over here, we're going to join them. So, what we can do is we can–rather than saying 'from course' and 'course comma enrolls_in' and then giving a 'WHERE' clause which combines the course number and enrolls_in class number', we can say that we want to get these data items, all of them, all these data items from the 'course' table which is our main entity over there, and we want to take that 'course' table and do something called an 'INNER JOIN'. An 'INNER JOIN' is what we have been doing so far. What an 'INNER JOIN' does is it takes– it'll ignore anything inside the 'Enrolls_in' that doesn't have an entry in 'Class'. Okay. That's the idea here. So 'INNER JOINs' are typically the most common that we work on. There's an 'OUTER JOIN', and a 'LEFT OUTER JOIN', a 'RIGHT OUTER JOIN' but we're not going to worry about all that. So, we say 'INNER JOIN', and we want to inner join the course table with the 'Enrolls_in' table, and we want to join the two on these fields, right! So this is like the 'WHERE' clause, you know the 'WHERE' clause we said— we would have said 'WHERE course.number = enrolls_in.class'. There's absolutely no difference and in many cases, there's actually no performance difference either between using the term 'INNER JOIN' or actually writing the complete thing the way we've been doing so far. However, it's much clearer when you have multiple tables, much clearer to use the term 'INNER JOIN' because then you can see where you're joining the tables together. So, let's just for a second run this thing. What I want to do is I want to say here, 'SELECT name comma number comma room comma ssn comma score FROM course INNER JOIN enrolls_in ON' and ON is telling us what to join on.

The only advantage is that it tells SQL that we are doing a JOIN so it should, maybe hopefully optimize for it, but in practice, it would probably optimize anyway. So 'ON' and then we give it the join–the two columns we want to join 'ON'. So we wanted to 'course dot number equals enrolls_in dot class', I think it was called 'class'. Let's check and see if this works. Yeah, so we joined these two and we get this table here. So, we can see that you know this is pretty much the same thing, but in practice, if you use the 'INNER JOIN', things will become much clearer. So here's an example where joins can be explicit. This is the explicit saying that we're doing 'INNER JOIN' and this is what we would do if we wanted to do it without the 'INNER JOIN', maybe we would say, select the same bunch of fields that we have. And instead of saying 'FROM course' we now give all the tables we're going to join from. And then in our

condition, we give the joining condition, 'course.number equals enrolls_in dot class', okay! That's the idea there. But what 'JOIN' really does then that's really useful for us is that, if we have a case where we have many, many tables that we are joining, then when you write out the clause in this form, you're going to have 'FROM course enrolls_in', etc, and the 'WHERE'.
So, what's happening is that the...the 'Join' conditions that is 'course.number equals enrolls_in dot class' is separated from the 'FROM' conditions, so it's harder to read, whereas when you say you're doing a three-table kind of thing, where we want the names of the courses that John is enrolled in, then what we can do is we could say, 'SELECT course dot name FROM student' and then we can list all the joins one after the other, maybe it could be 10 tables, right! And we're saying 'INNER JOIN enrolls_in ON student dot ssn equals enrolls_in dot ssn', 'INNER JOIN course ON course dot number equals enrolls_in dot class'. So essentially what we are doing is we are saying our tables over here, 'course' and 'enrolls_in' and the join conditions are listed together and it becomes much clearer, thats—it's... it's a much more readable thing.
So, if for no other reason, and like I said, it's—in theory it might help the server optimizer query, but I think in practice it probably won't. But for no other reason than readability it's a good idea to actually do the joins explicitly to say, "hey, we're going to do an 'INNER JOIN', an 'INNER JOIN' so it knows what the heck's going on." So that should be our SQL thing here. There are plenty of resources online that you can look at if you want to study SQL more. There's a very nice book called the Idiots Guide to SQL if you want to read up on SQL more. But this should give us a basic introduction that should be sufficient to– for the rest of this course.

Video 9 (08:26): SQL and Python (Part 1)

Let's now look at how we can use Python as a client to an SQL server, and look at connectivity between Python and– it turns out to be actually fairly simple. And once we've done Python, you know, it's a pretty straightforward process. So, the essential goal here is, irrespective of which particular server you're using, whether it's PostgreSQL or MySQL or some other server, you—we have the same basic structure. So, there's a set of standards that have been set down for a Python database API, and we follow that standard, when you...when you–if you move to a different server, you're gonna have to again look for the equivalent API commands that will sort of connect you to that server itself. So, what...what do we do? Well, the steps are, we import the API module, whatever that is. And remember, in Python, whenever you want to do something, you import a library. So, that's what we want to do here.
Then you acquire a connection with a database server. It's much like we clicked on the local host thing in our MySQL workbench. We will—what that data did was, it created a connection between the workbench, which is a client, and your MySQL server, which is a server, right! So similarly, we also need to do that in Python. We acquire a connection with a database server. So, there's going to be a function that will acquire the connection, then we issue a bunch of commands which are SQL commands essentially, sent as strings to the...to the server, or you call a stored procedure... a stored procedure like our source–our school_schema.sql or school_data.sql, those are stored procedures; sets of queries that

are stored in a single file, and once we've done all that, we can close the connection. That's our...our basic structure here. The library that we work with is called pymysql, and we want to work with this. The structure here is–we do the same steps that we have outlined before. We start by importing the library, which is 'import pymysql'. We acquire a connection with the database server, and there's a 'pymysql' function called 'connect', which you give it the name of the connection itself, and...and the server itself, which could be a URL, but in our case, just localhost; the name of the user, and I'm going to use root, the password; and actually we need one other thing here– I think I forgot to put in, the database itself. We'll see that in example. So, connect is the way of connecting to the database, and then you create what's called a 'cursor' object. What is a 'cursor, object? The 'cursor' object is essentially this: it tells you, "hey, you know, this is where my command is going to be." So, you can see the little blinking line here, that's a 'cursor'. So the 'cursor' object tells it, this is the next command, and you enter commands into your database, and enter all that kind of stuff. So, the key here is connect, and then you create a 'cursor' object, and once you are finished with that particular 'cursor' or the thing you can do, you can close it using the 'close' command. So, let's look at this in practice, and go to our Safari here, and we have Python and MySQL.

So, the first thing I want to do is import 'pymysql'. Now, you may not have it installed. I think I'm not sure if I have it on this machine. No, it's not there, right. So again, just to revisit this in a second, what we can do is, we go here and do '!pip install pymysql'. So, this is going to go and get the pymysql library, and install it, and once we've done it, we can import it, right! So remember, this command, the exclamation point(!), is our command that tells our Jupiter browser to not use Python but to go out to the operating system, and run this command —run the PIP command, which is the Python Installer Program, from the operating system. So it's saying, get this from— and you have to have web connection for that, internet connection for that because this is going out to the internet to get it, and it gets pymysql, downloads it, and installs it. So, that's our step one. So, we've got that. So, now we import this in, and then you create a...a 'cursor' object. So, I'm going to create a ...create a connection, sorry. So I call the connect function from 'pymysql', and I give it the various attributes that I need.

The attributes that I need are the URL of the server, in which case–this case, is localhost, so we just use that, the name of the user, and note that these are all strings, right, these are Python strings. The user I'm using is root, and we is probably in the root two. I set up my password as 'None', because it's just what I set up, and it's—doesn't mean there's no password. It's just a string that has uppercase 'N' followed by one, and I'm telling you to use the database 'schooldb'. So, you have to tell the Python pymysql connector, what database you're going to be using when you call this thing here. And then, once I've got that, so I now have a variable called 'db' that contains my connection, and now, for that connection, I say, set up a 'cursor' object. Why do I need that? Well, you could actually be connecting to multiple databases, you don't, –or multiple servers. Your Python program might be getting data from one server'; like perhaps you're trading application, and you're getting data from your historical price databases.

You're getting data from maybe a fundamentals database somewhere else; you're getting data that's coming in from your trades, and you know, doing all kinds of stuff for portfolio management, mark to market, all kinds of things. So, there could be multiple sources for data. So, you could be having 'db1', 'db2', 'db3', 'db4', whatever, right! So multiple connections. So, for each connection, you create a

connection object, and then this case that's like db, and then in the...in the variable db, and then, for each connection itself, you can create a 'cursor' object. You can create multiple 'cursor' objects because the–you could have multiple clients inside your...inside your program itself, right! So that's a possibility too. So we run this stuff here, and we get these two objects. We can take a look at them.
If I just go 'Insert Cell Above' here, and I say 'db', it tells me, it's a 'pymysql' connection object and if I do 'cursor', it tells me it's a 'cursor' object, right! So now, the next thing you want to do is execute queries. So, the way this works is that when you execute a query, you call this function called 'execute' on a 'cursor'. So take any 'cursor', and call the function 'execute' on it, and that's what gets executed. Right! And what you do is you pass your query as a string to the function. So, this is pure SQL.
There's no–nothing fancy in that, right! So, it says, 'show tables;'. That's my string. Notice, I have my semicolon because every query ends with semicolon. And that's inside a Python string. So, what this is going to do is, it's going to go to the server, and execute this command, and every SQL query returns a table. So, it'll get a table back. A table is what? A table is a bunch of rows, right! We have 'row0', 'row1', 'row2', 'row3'. So, you can think of it as a collection, and what we end up doing is you get this collection back, and you can start–that sort of sits. So, let's say we have a server here, and this...this is your client program. You can think of a buffer over here. So, when you send the query, the query goes to the server and the server puts the result in a buffer, okay, so this is our table. And then, the client can say, get me one item from the buffer. So, it'll take one item and remove it from the buffer and give it to the client. Or it can say, get me all of them. So, the way it does that is it uses these commands 'fetchone', to get one item, or 'fetchall' to get them all.

Video 10 (07:53): SQL and Python (Part 2)

It will say 'cursor dot execute('show tables;')' and then use 'fetchall'. So this is going to run the thing for us and it tells—it fetches all of them and we get back. There are three tables. There's 'Course', 'Enrolls_in', and 'Student'. So, notice the structure of this thing here. I'm doing 'fetchall()' so I'm getting back a collection of rows, and this is each row. Now, each row contains just one item, the name of the table. So, we get a collection of—each row itself is a collection of fields. So, for example, if I did another query, like 'SELECT course.name FROM student', 'INNER JOIN enrolls_in ON student' blah blah, the stuff we looked at before, and execute the query, then I get the fact that a...a student, JOHN is enrolled in data analytics and Python. If I wanted to add more information to this, 'comma enrolls_in dot score', and now run this. I get the score as well.
So, you can see what I'm getting back is a list. So, now I can actually say— do this, let's say 'result equals cursor dot fetchall()', and say 'for thing in result: print(thing)', Oops—I didn't run that— and now, we get it like this, right! Or I could even do it nicer. I could say print... 'print(thing '0'comma thing '1' comma sep equals " backslash t")' as a separator, right! And run this, and I get a nice kind of thing there. I can pop a header on it, I could say, 'print("Course name" comma "Score" comma separator equals "backslash t") and run that, and I get, you know, a nicer looking table. But the goal here is that, we're getting all of them, and we are putting them inside a Python—tuple in this case. And each tuple contains—each—the...the resulting Python tuple contains a bunch of tuples that corresponds to each row. I mean, extract

the data using our normal Python iterator kind of stuff, and we can then do whatever we feel like with it. So that's the roughly a very quick way of looking at the query.
The only other thing that we should— I should point out here which I haven't really done is that, if you add data into the database, right, so if I take—make a 'query equals INSERT INTO'—let's say I get a new student, right—'Student VALUES'— actually, here I got to be little bit careful. I'm going to use a double—single quote over here, rather than a double quote, and I'll explain why in a second. 'VALUES'— and now I want to give each value, so the social security number is, let's say '000 dash 11 dash 2322', something like that, 'ssn comma', say "Bing" comma "Crossby" comma' and let's say, phone number is '2345670000 comma city is "New York" comma ZIP is 10001;'. So, as you can see here, what I've done is— because when I send my data to— oh, I need to put a semi-colon in as well— When I send my data to the SQL server, any string literal has to be in double quotes. So, generally when you send the query, it's a good idea to set it up in single quotes because Python will allow you to do both, right! So, it makes life easier for us. So, now I can say here 'cursor dot execute(query)', and it's executed the query.
So, if I look at this stuff here and I do 'query equals SELECT' matter here 'star from Students;' from 'student cursor dot execute(query)'; 'cursor dot fetchall()'. I run this. So, you notice now I have Bing Crossby added into this. However, if I go to my workbench and...and look at that—there— so, I do 'SELECT star from Student' and I run this, there's no Bing Crossby. The reason is that I'm running this on my client and my client is now— it runs on the client and—my client now sees a view of the database that includes Bing Crossby. However, for the other customers—other clients of the database—that is not actually updated— it's not actually fully updated in the database itself—to make— so, the idea being that, you know, you might be running test kind of stuff... you might not be sure that you to add it in. And it's only when you are sure, that you want to add your—database— the new data permanently or alter or change or something— the new data permanently into the database then, at that point, you want to propagate that change to the rest of the database.
And there's a good reason for this because you may be working with hundreds of lines of data. And perhaps somewhere, like let's say, you were adding a million new records into a database, and somewhere around...around record number 200,000 something happens, you know? You accidentally pull the plug on your computer, for example, right! Or there's a bug somewhere that you hadn't found earlier. And you don't want that the..only part to be only partially updated in the...in the database. So, if something happens in the middle, then it shouldn't be reflected in the entire— the...the change shouldn't be reflected in the entire database because it could be erroneous, you know, for example, maybe there's some relationships that are not fully completed, right! So, you want to make sure that you— only when every possible change or every possible inconsistency is removed from what you're adding, at that point, you want to reflect those changes in the database.
So, in Python what you can do is— or the thing you do is what's called a 'commit'. Say, 'db dot commit()' And what this does is, it now commits the change to the database and now, if I go back to workbench and run the query again, I get Bing Crossby there. That's the idea there. So, I guess that's pretty much it. With...with this, we are in good shape to look at SQL and be prepared to face it, so to speak. So that's it. Thank you.