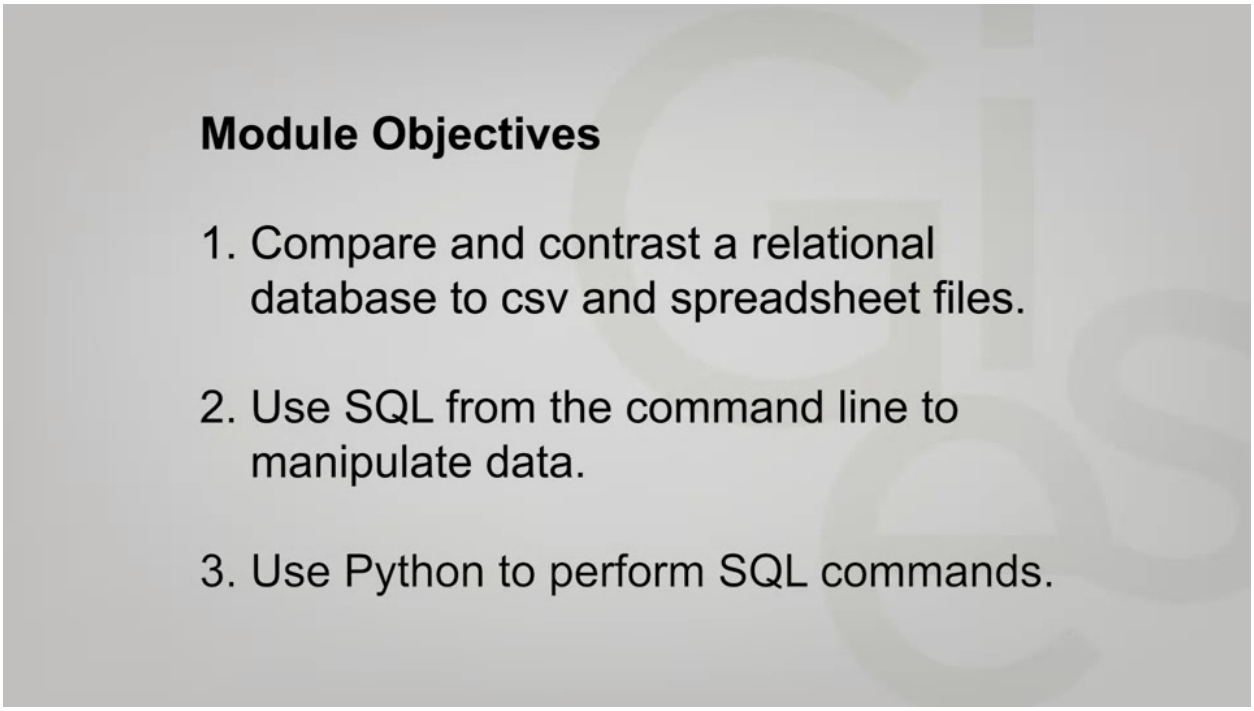
Module 8: Introduction to Database in Python

Table of Contents

Module 8 Information

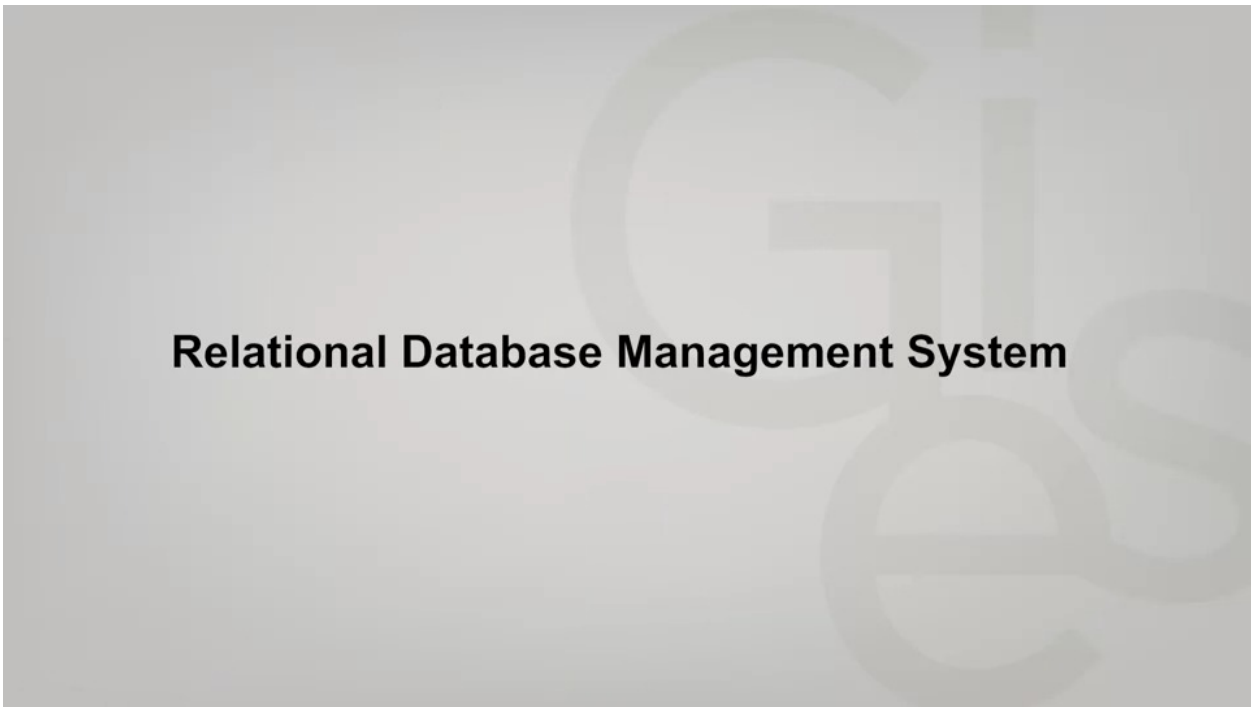
Module 8 Introduction



Lu: This module is about relational databases and the language that's used to interact with them which is called structured query language or SQL. Relational databases are designed to store data in a secure efficient way. They consist of one or more tables that can be joined to each other. In this module, you will learn about the benefits of a relational database relative to a CSV or spreadsheet file. You will also learn some of the SQL commands for interacting with a SQL database directly and then python code for interacting with a database within python. Well it may at first seem daunting to learn an additional language. I think you will find that learning a few basics will go a long way and it will make your code much more efficient.
Guymon: Absolutely.

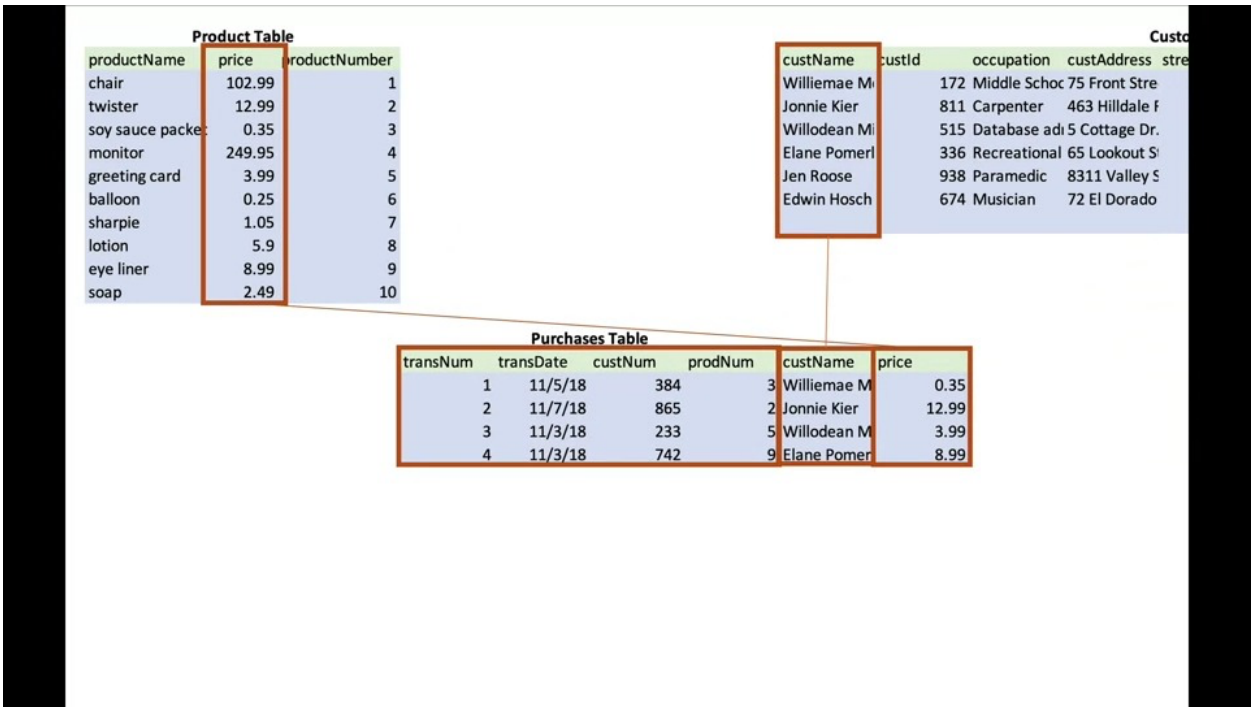
Guymon: Speaking of efficiency, I remember a story called Cheaper by the Dozen. This story is about a family with 12 children. I don't recall much from that story. But I do recall that the father was an efficiency expert. I figured that he probably had to become one to effectively raise that many children. Similarly efficiently working with data is really important when you deal with large amounts of it. From a business perspective data has a lot in common with inventory. Storing inventory requires that you have an efficient system for keeping track of where the inventory is located and then retrieving that inventory. Because you have to warehouse the inventory somewhere and ensure it in case it's lost in a fire, it's costly to store inventory. And this is why in manufacturing settings, there's a focus on Lean production processes and the optimal order quantity of inventory. So that unneeded costs are avoided from storing excess inventory.

Lu: Yeah, similarly when storing data is important to have an efficient way to keep track of where the data is stored so that you can retrieve it quickly. Storing more data is costly as well because it physically needs to be stored somewhere. In fact, it's often stored in multiple places in case one of the repositories is destroyed.

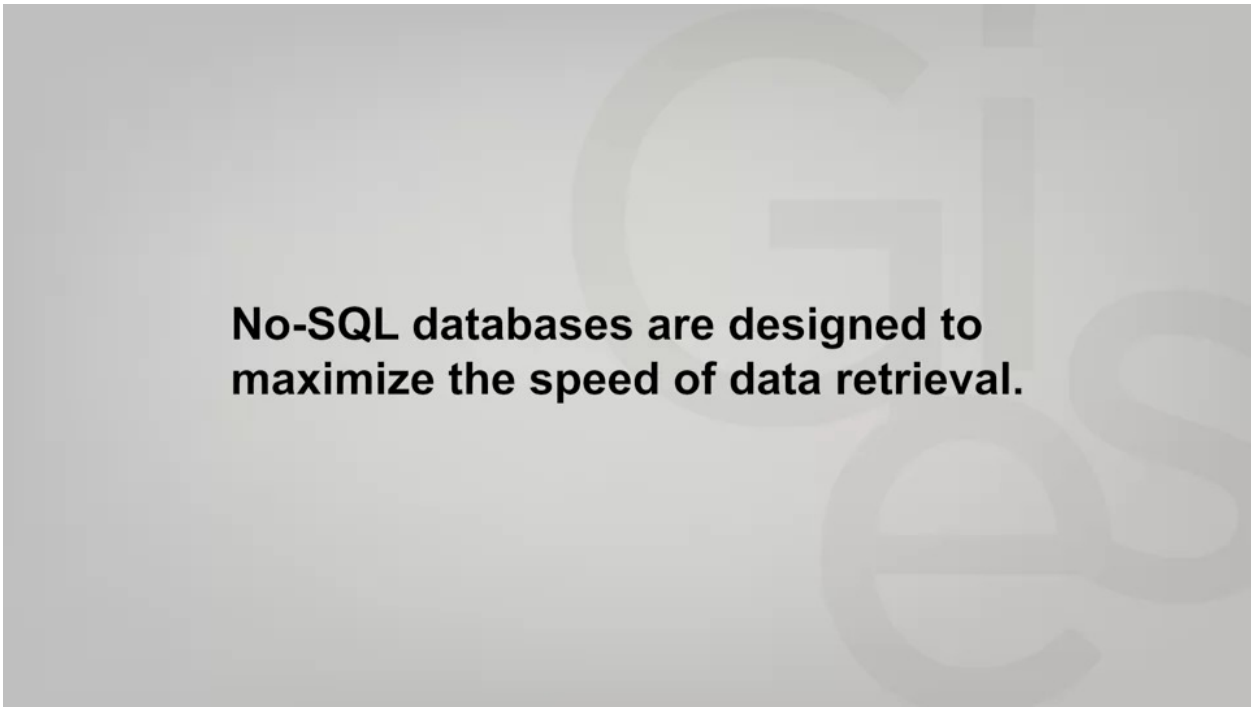


So in this module you're going to learn about one of the most common ways for storing data, which is in a relational database management system

known as RDBMS. Even though this is older technology is still widely used and very effective.



A key feature of this type of database system is that it's made up of many tables as a way to reduce the amount of data that's stored. This means that when you want data, you will likely have to merge data from different tables and then summarize it somehow.

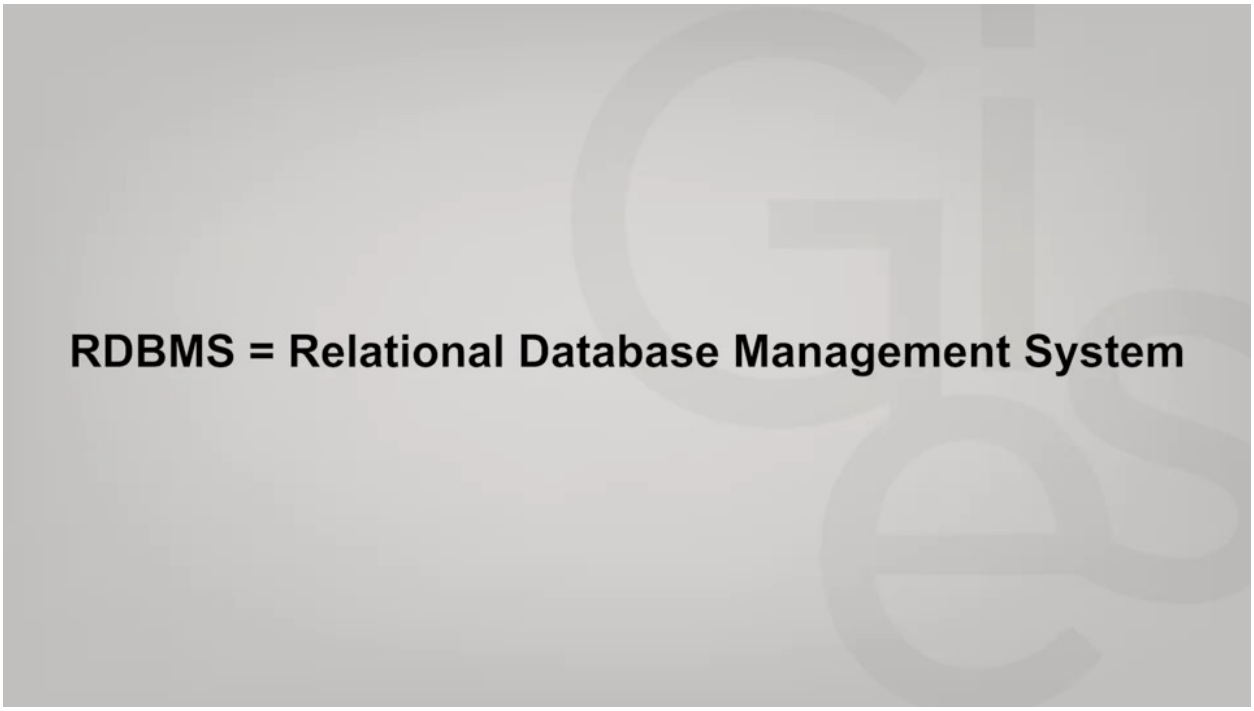


Guymon: Because of the advanced processing power and the reduction in storage costs, there are a growing number of non SQL databases known as No-SQL databases. Rather than focusing on reducing the amount of storage space, No-SQL databases are designed to maximize the speed of data retrieval or a certain type of advanced analysis. No-SQL database systems can be more

complex though and are less common. So it makes sense for you to first learn about SQL databases at this point.

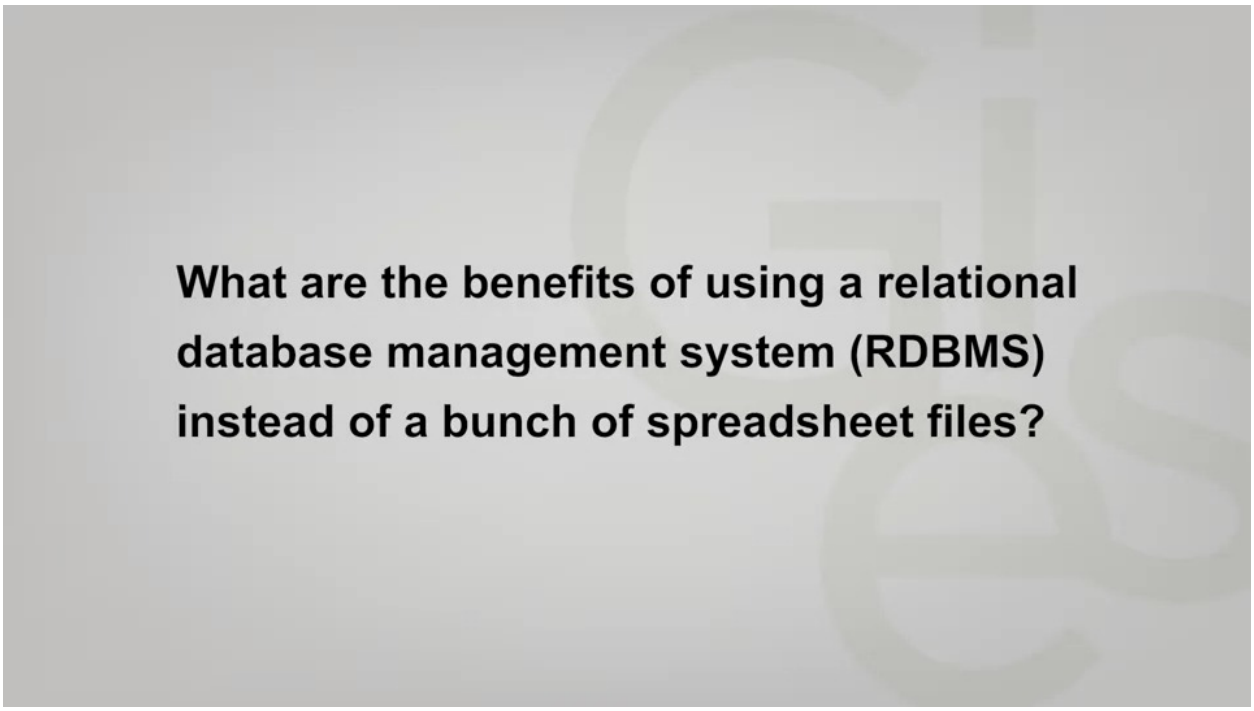
Lesson 8.1 Introduction to Data Persistence

Lesson 8.1 Introduction to Data Persistence

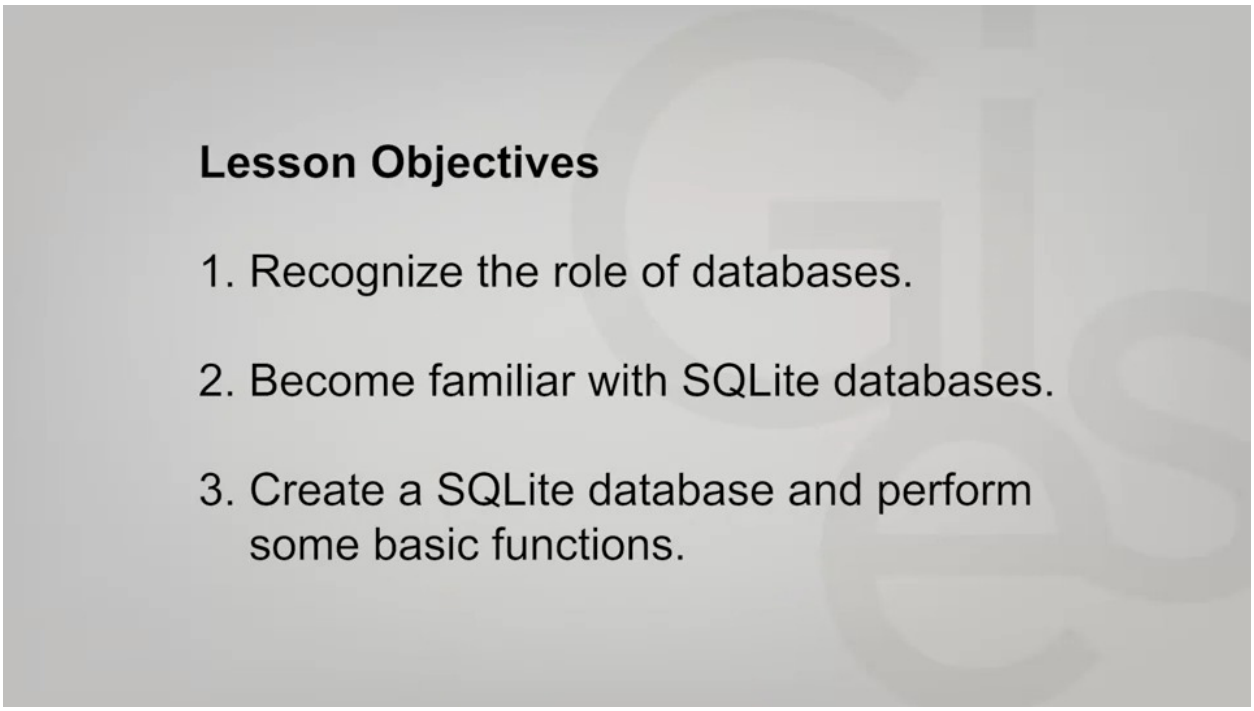


Guymon: In this lesson, you'll learn about some important details of saving data. The technical term that's used for saving data is called persistence and it includes a number of important details associated with saving data. One of the most popular ways to save data is in a relational database management software system, or RDBMS, which is the focus of this whole module.

Tyler: My name is Tyler, I work at Qualtrics. We are a software company owned by SAP and I am a data scientist there. So I work with our sales team to make sure that we do our Predictive Analytics correctly.



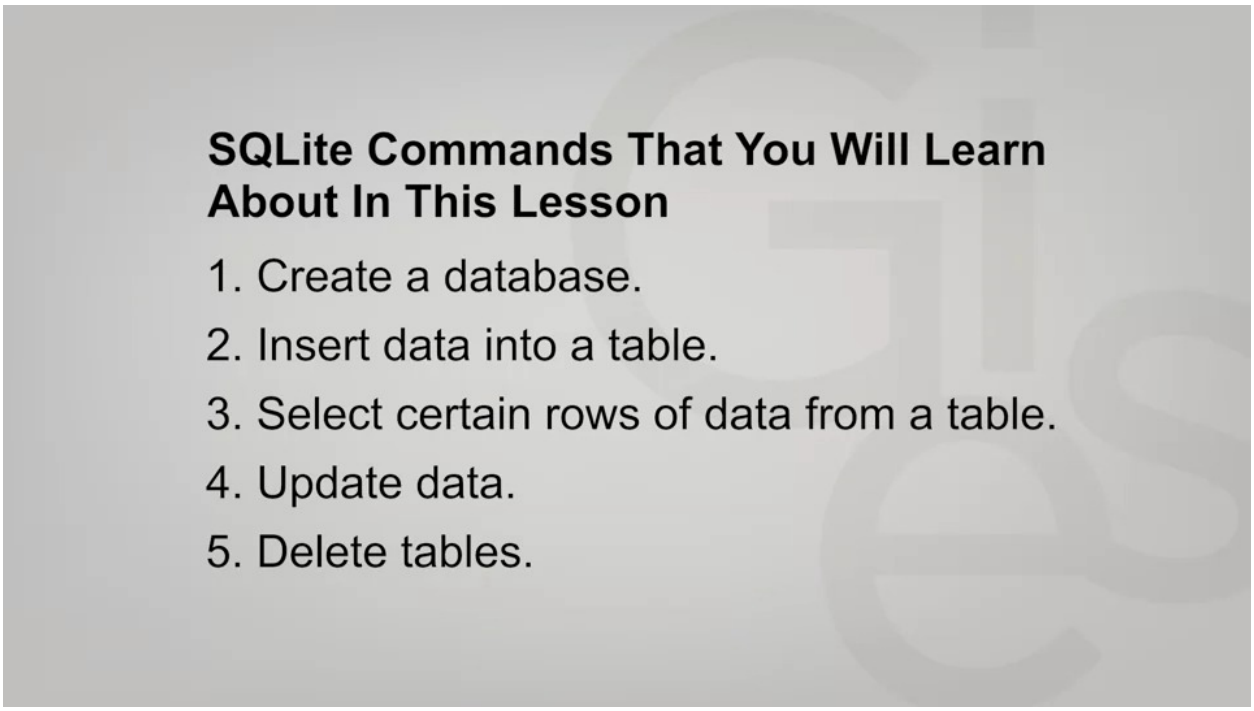
Yeah, that's a great question. I think using SQL is preferred to just using Excel spreadsheets in many various ways. I actually use SQL every single day in my day-to-day. And the reason I do that instead of using Excel spreadsheets is it allows me to look at the data from different viewpoints very quickly. Typically, I want to see the data grouped in a certain way at one moment. And then the next one, I want to break it back out and then group it at a different way, or I want to join it to another table, and if I were to do that all in spreadsheets it would just take a lot more time. So in a spreadsheet, I would need to do maybe a VLOOKUP to match the data, then I would need to summarize the data, maybe do a pivot table just to get one of those findings, but with SQL I can write a simple query. Say group by my variable, and it's a group result to me as opposed to doing that all. And then I could simply eliminate that group by statement, rerun it and I have my new result instead of having to do the whole process over again.



Guymon: The specific objectives of this lesson are to, one, recognize the role of databases, two, become familiar with SQLite databases, and three, create a SQLite database and perform some basic functions. Learning how to interact with relational databases is critical for dealing with business data. CSV files, pickle files and spreadsheets are great tools for working with small groups of people and small data sets. But they are woefully inadequate for production level data storage and interaction, which is why relational databases were created.

If you've ever worked with a group of people on an Excel file that saved in Dropbox or Box or that's emailed around. Then you may know some of the frustration that can occur when more than one person wants to modify the document at the same time. One of the advantages that relational databases have over CSV and spreadsheet files is that many people, perhaps hundreds of thousands can access and modify the data. Moreover, there are restrictions to prevent some people from viewing or modifying the data. There're also protocols in place that prevent a database for malfunctioning if a modification, for some reason, was cut off halfway through the process.

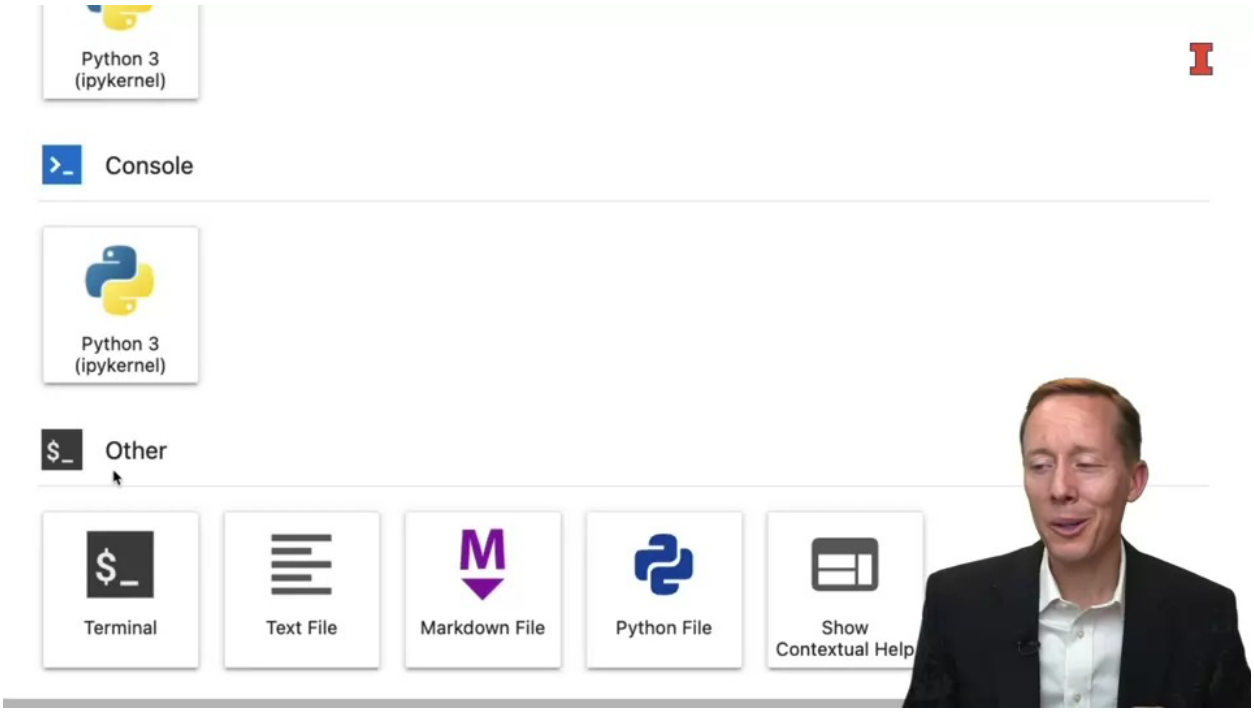
There are a variety of relational database implementations that have their own strengths and weaknesses, but they all have some common features that you'll learn about. Because there are a variety of relational databases, it could be very burdensome if each version had its own language to create, modify and query the data. Fortunately, there's a pretty standard structured query language known as SQL or S.Q.L that is frequently used with only minor variations. So in this lesson, you'll start learning some of the main SQL commands. In this lesson, you'll learn how to do some SQL programming using SQLite.



Specifically, you'll learn how to create a database structure, insert data, select certain rows of data, update data and delete data. Why are we choosing SQLite instead of MySQL for instance? Well SQLite is easy to setup, it's included with Python and it's very popular. It's also used in many modern software applications. It's worth noting that while relational databases based on SQL are really popular, there are other database systems, which are known generally as No-SQL. You'll learn the names of some of these, but learning more details about them is beyond the scope of this lesson.

Introduction to Terminal

In this video, I am going to introduce you to the terminal. Now, you can think of the terminal as the most basic integrated development environment that there is on a machine. Every machine that I know of has something like the terminal built into it, whether it's called the terminal or the command prompt. It's basically a simple way to interact with the machine using only text, I'm using a Macbook. And so this apple operating system has an application called terminal built into it, and I can open it just like any other application. And it looks very basic like this, I said, it's just text, now, Jupiter is pretty awesome and it has integrated the ability to interact with the terminal.

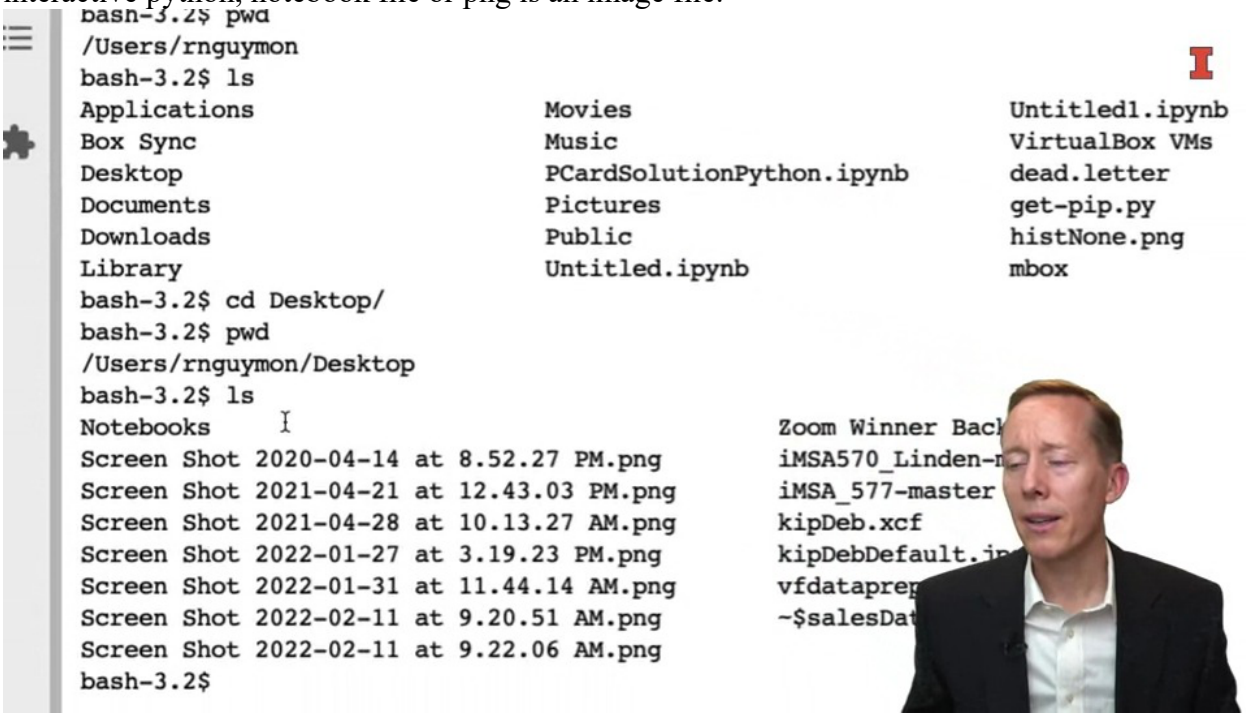


I've got Jupiter lab opened right here, and instead of opening a new interactive python notebook file, if I scroll down a little bit further, there are some other things that we can do. And one of those is to open a terminal window, so let's go ahead and do that. And you can see that this looks very similar to what I have here using the actual terminal application. From the terminal there are a variety of things that I can do, such as interact with python from here, where I can perform some sequel commands from here. What I want to illustrate in this video are five or six different commands that you can use for navigating around your machine, and creating new folders and new files.

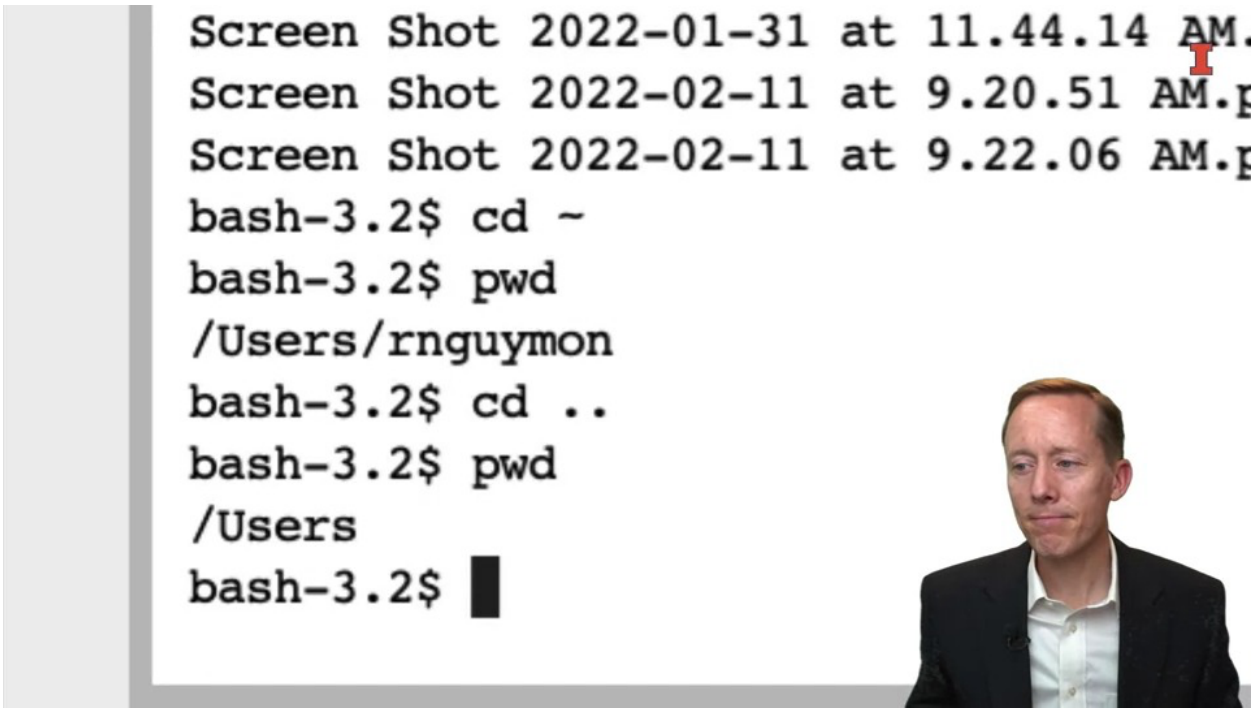
So the first command that I want to introduce to you is pwd, and that stands for print working directory. So if you type out pwd hit Enter, you can see where you're located, and if I were to open my finder on this machine, a finder window here. The directory refers to a folder that I'm in, so if I go to this, can I folder right here, this is where I'm at, okay. So I went into users rnguymon, and that's where I am right here, and you can see that I've got a variety of folders and documents in here as well. All right, so pwd is great to just know where you're at on your machine.

From here if you want to see what's in a folder or a directory, then we would use the LS

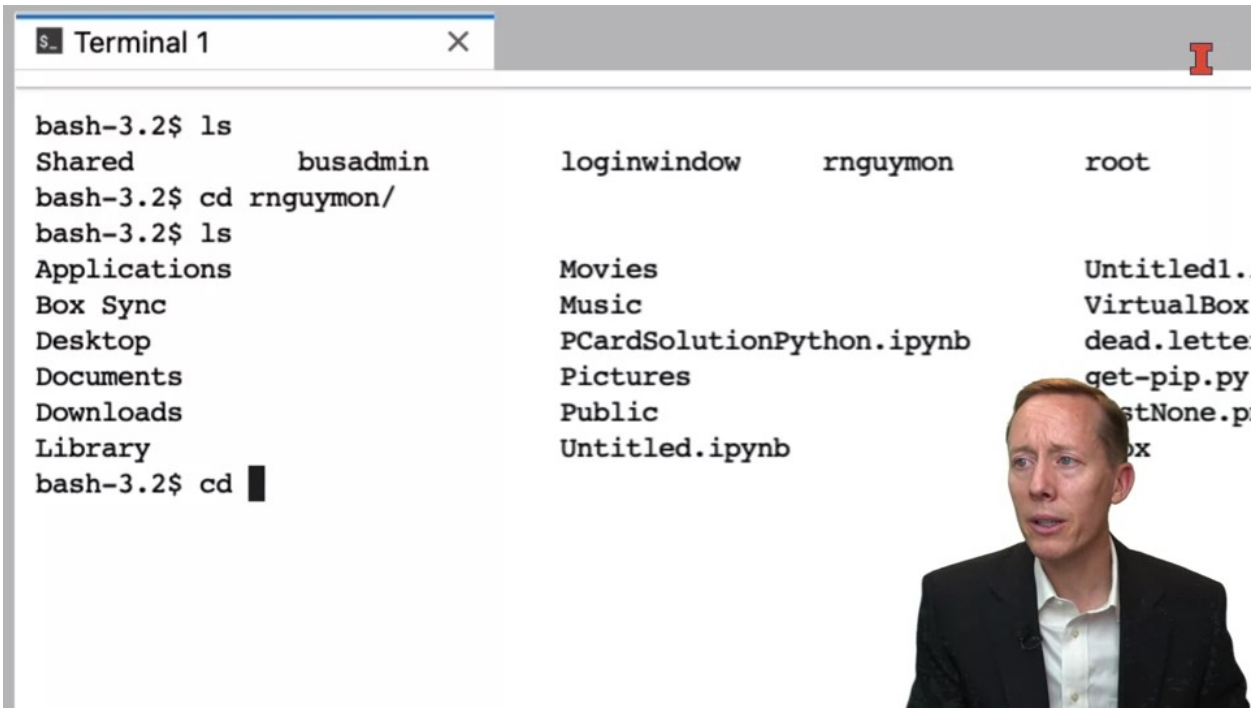
command, and that will list out all the different folders and files. And if you have a name that doesn't end in dot something, then that means it's a folder. Otherwise if it ends in .ipynb, that's an interactive python, notebook file or png is an image file.



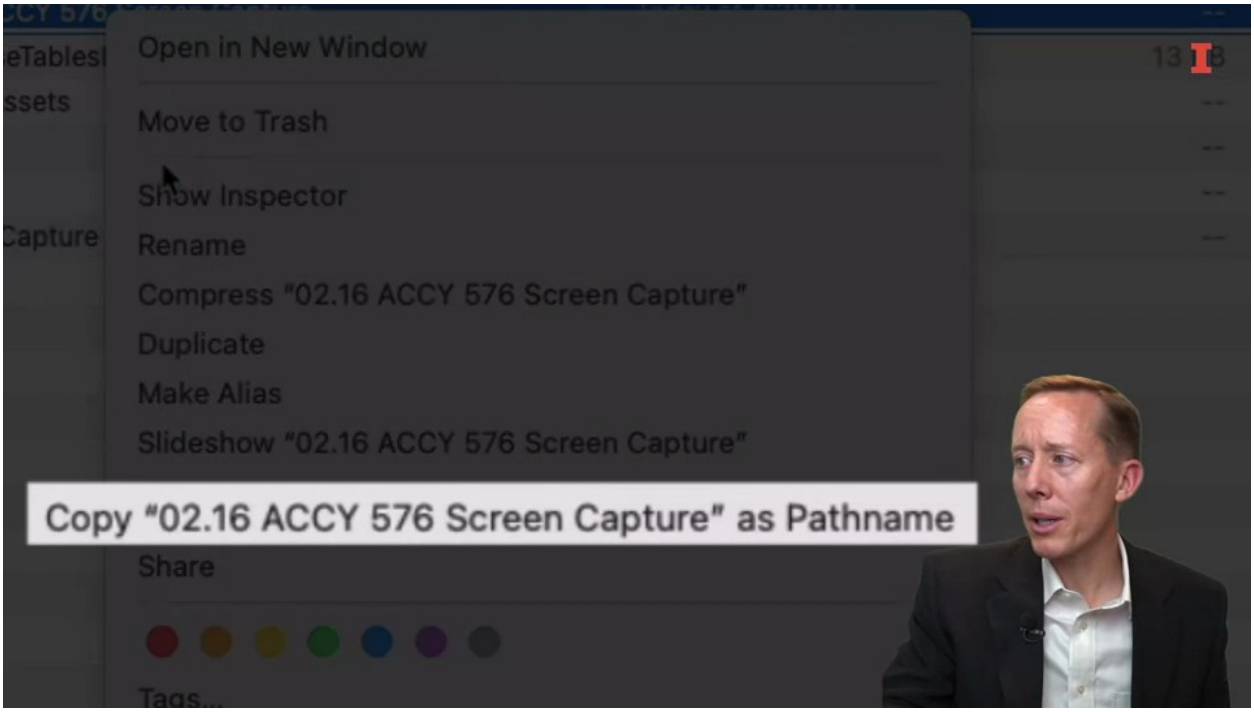
Let's talk about the cd command, now, cd stands for change directory, so I can navigate through my machine by typing cd. And then the name of a folder or file within my current working directory. So if I want to go to the desktop, I can start typing it out desk, and there's actually a tab complete that you can do. So if there's no other file or folder that starts with the desktop, and I hit tab it'll auto complete for me. Hit enter and you can't really see that anything has happened, but if I use pwd, now you can see that I have gone into the Desktop. And I can use LS to see what is in here, and I've got a bunch of files and some folders as well.



If I want to navigate back to this home directory right here, I can use the cd tilde command and then pwd. And you can see I'm back to my home directory, if I want to move up just one directory. So let's say I'm in the rnguymon directory here, if I want to go up to the user's directory, all I have to do is type cd and then two dots. And now I'm in the user's directory. All right, so that's how you use cd to navigate around, and Ls to see what is in a folder. Hopefully it's making some sense about how to use the terminal. Let me introduce you to another command to help clean things up in the terminal window. And that is the clear command, C-L-E-A-R if you type that, then it removes all the history of what you've done, and it looks nice and fresh again. My main objective in this particular video is to create a folder where I can store a database and some related files. So what I need to do is navigate to the folder where I want to create the new folder.



One way I could do that is to list the files that I'm in, and then keep changing directories until I get to the one I want. So for instance here I want to go into the rnguymon folder, and I'll list the folders here. And from here I want to go into the box sync folder, and then list out the files again. So I could navigate this way and keep typing cd, and then the name of the folder until I get to where I want.



Another way to do this, if you can use the finder is to navigate to that folder on your machine, and then copy the path name for that folder. So that's what I'll do here, in my finder I have identified the folder where I want to create a new folder. So I'll go ahead and right click on that,

and hold down the option button and then copy that as a path name. I can then go in to the terminal, let me clear this out, and I can change the directory cd. And then in quotation marks, I will paste that path name and end it with quotation marks. And that's really important, because if you have spaces and your folder name, you have to escape those out with escape characters, and that can get cumbersome. So it's easiest to just put quotation marks around it and then hit enter, and now if I type pwd you can see that I have navigated to this particular folder.

```
bash-3.2$ ls
Connect to Remote MySQL Database.ipynb  Recording Notes.ipynb  people.csv
Create Database on AWS.ipynb            cars.csv
bash-3.2$ mkdir testDatabase
bash-3.2$ ls
Connect to Remote MySQL Database.ipynb  Recording Notes.ipynb  people.csv
Create Database on AWS.ipynb            cars.csv               testDatabase
bash-3.2$
```

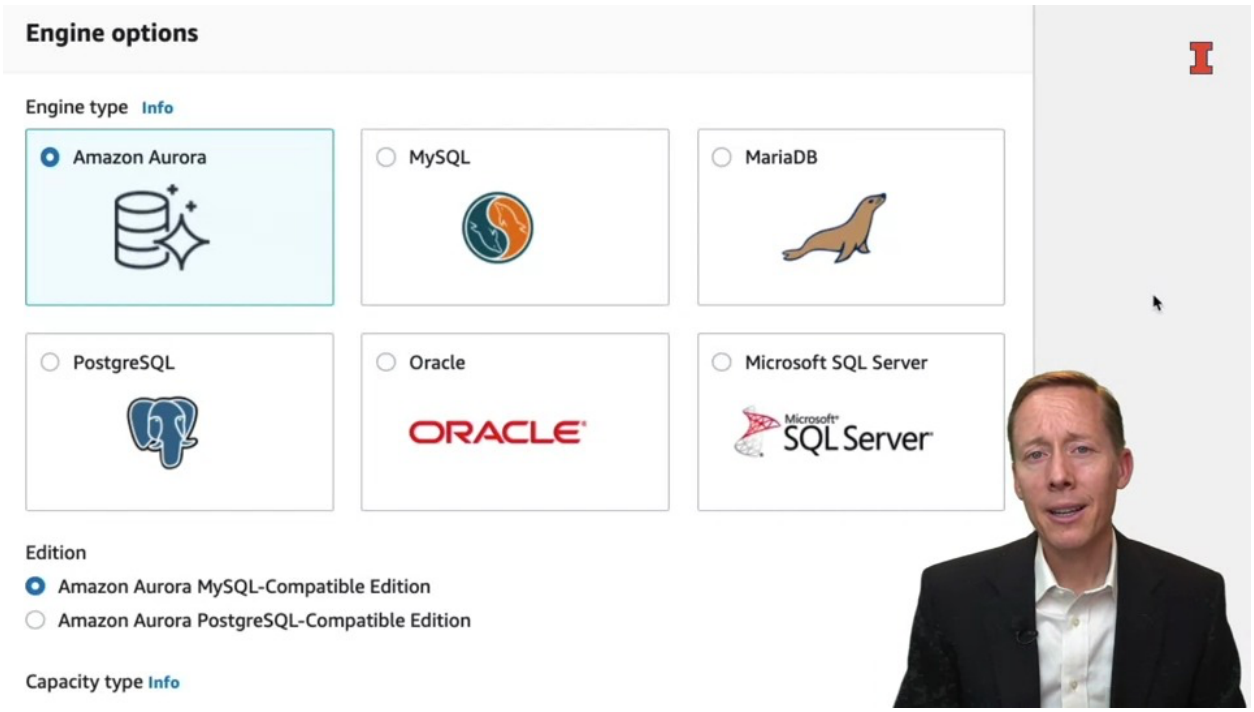
And let's see what's in this folder here. All right, I've got one, two, three, four, five files, from here I want to create a new folder where I can store a database. To create a new folder, we use the mkdir, and that stands for make directory, and then we give it a name. So I'll give this folder the name of test database, and then hit Enter. And now if I type Ls within this working directory, I've got this test database folder.

```
bash-3.2$ cd testDatabase/
bash-3.2$ ls
bash-3.2$ touch temporaryFile.txt
bash-3.2$ ls
temporaryFile.txt
bash-3.2$ rm temporaryFile.txt
```

The last two commands that I want to show you are how you can create a simple text file as well as how you can delete files and folders. So to create a simple text file, once you're in the folder that you want to be and let me go ahead and change my directory to the test database folder. I don't have anything in here, I can use the touch command and then type out the name of a file. So let me go ahead and just say temporaryFile.txt, run that. And now if I listed out, I've got that text file, now I can actually edit that from within the terminal, but we won't get into that right here. I do want to show you how you can remove files though, and that's pretty simple, you can delete or remove files by using the rm command. So if I type rm and then start typing out the name, hit enter, I can now list it out and see that it is gone. Let me go up to the parent directory here, so cd.., so now you can see I'm in this parent folder. And if I list out the files here, actually let me clear that out and list the files up here.

You can see that I have this test database folder, that's just a folder, it's not a database yet. If I want to remove a folder, I have to use this r command for recursive to remove everything in it. And then I can type out the name and hit enter and it will remove it, and if I use ls is no longer there. Let me go ahead and create it again mkdir. Great, so now I have a folder where I can create a database. In this video I wanted to introduce you to those five commands, and help you get a feel for how you can navigate around a machine using the terminal.

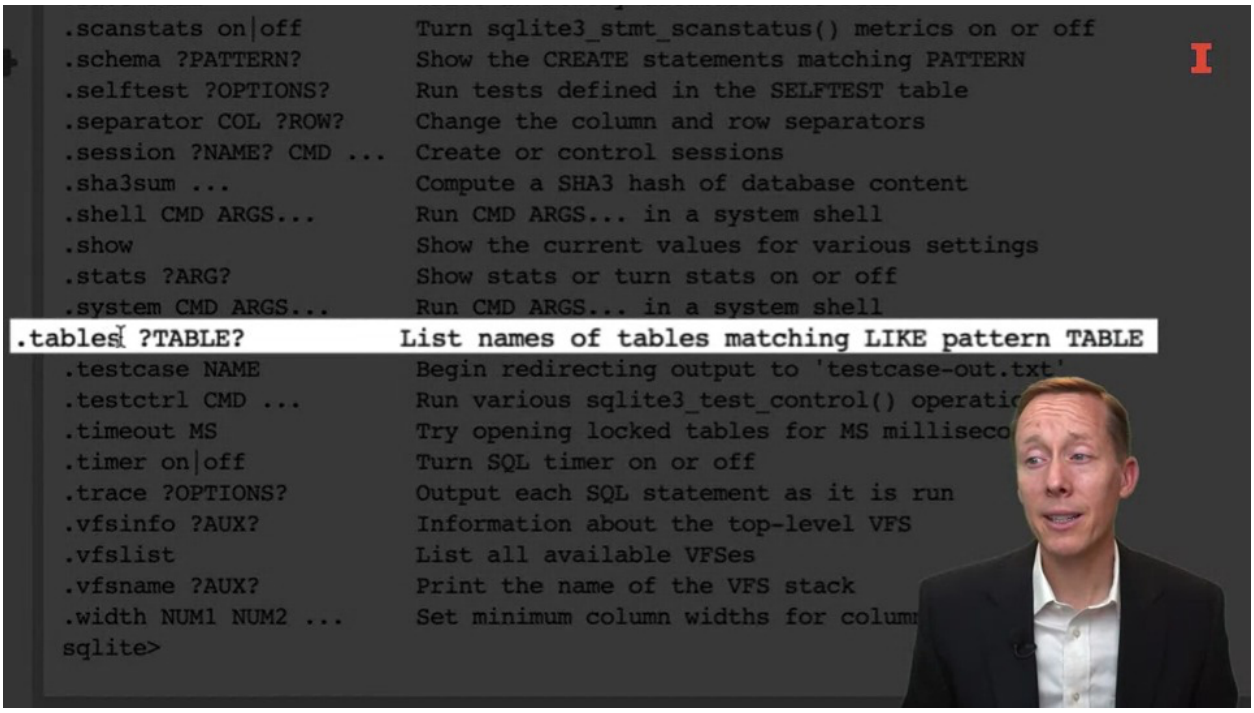
Creating a SQLite Database From Terminal



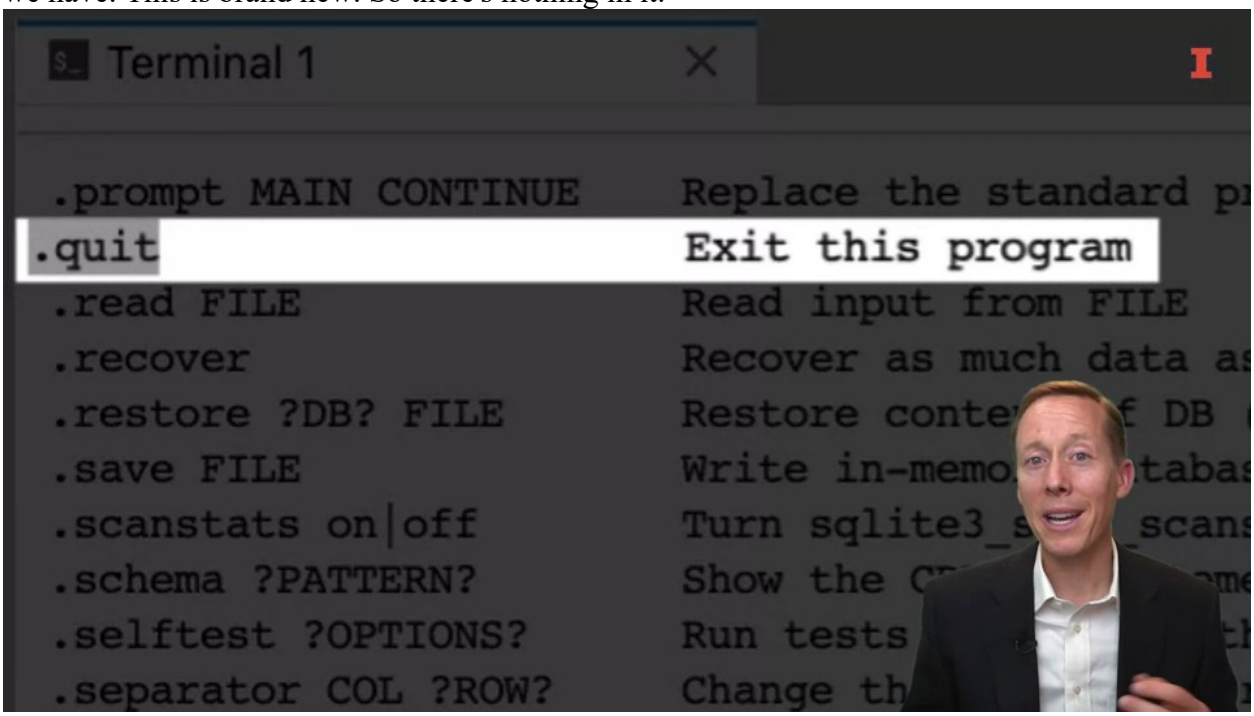
In this video I'm going to walk through how you can establish a relational database on your own machine. For most practical purposes, you will probably be connecting to a remote database server that is located either on a cloud server or some other location rather than your own machine. And the purpose of having a dedicated server for a relational database is because there are probably lots of people interacting with it and also to keep it very secure. There are a variety of flavors of relational databases. If you wanted to set up a cloud database on amazon web services, you would get at least these six options here at least. This is what I get when I log in. You can see there's amazon aurora, my sequel Maria dB each of these presumably have strengths and weaknesses but importantly for us, the way you interact with each of these is quite similar. So what we want to do is set up a sequel light to database. You don't see sequel light in here. But sequel light comes with python.

```
bash-3.2$ sqlite3 rons_test_db
SQLite version 3.36.0 2021-06-18 18:58:49
Enter ".help" for usage hints.
sqlite> .help
```

So I am going to use the terminal to show you how you can create a database and then add files to that database. We're going to create a Sequel Light database because it comes with python. It's really simple and you don't need a dedicated machine to hold that database. So from terminal, let's make sure that you're in the right working directory here for me, I can see that there's nothing in this test database folder, which is fine. But you do want to make sure that you're in the right folder. All right so from there, what we will do is we will say we want to use sequel light by typing out sequel light three and then we type out the name of the database. So I'll call this Ron's test dB for Ron's test database and if it doesn't already exist then it will create that database and take you into it. And so after I hit enter, you can see that it tells me what version of sequel that I'm using. And when that version was created and then it gives me this really important function here that I'm really big on, which is the help, learning how to use the built in help. So if I type dot help, then I get all these usage hints in here and there's a bunch of them. Let me illustrate a couple of them real quickly.



First of all, one of the things that you use most often is trying to identify the tables that live within a database because databases typically are made up of a bunch of different tables that you can then query and join together. So if we type out dot tables, then it will tell us the tables that we have. This is brand new. So there's nothing in it.



Another important thing is how to exit and get back into just the bash script here or back outside of sequel light so we can do that by typing dot quit. And if you do that now I'm back out to the bass here and I can type clear, clears everything out remember the the working director I'm in and you can see that I have this Ron's test database in here. So now if I want to get back into this

test database, I can type sequel light three and then start typing out the name and then hit tab and we'll complete it for me. Hit after nine back in it.

```
bash-3.2$ sqlite3 rons_test_db
SQLite version 3.36.0 2021-06-18 18:58:49
Enter ".help" for usage hints.
sqlite> CREATE TABLE cars (mpg REAL, name TEXT, year INT);
sqlite> .tables
cars
sqlite>
```

I will now show you how you can create a table within the sequel light database. Now when you create a table, the only thing you have to do is give it a name and then identify the column names and the data type for the columns. I'm going to create a very simple table here and I'll just explain it to you as I type it out. So we'll type out create table and I've used all caps there. You don't have to use all caps but that helps distinguish between what is a sequel command and what is a variable name. So I'm going to call this table cars and then I need to indicate the column names. So I want to have three columns in here. I have MPG and that will be a real data type which is synonymous with a float or it allows for decimal values to be in here. I want to have the name of the car and that will be a string and in sequel, that data type is known as text and I'll need to make sure and put a comma between these column names. And then finally I want to indicate that I want to column to keep track of the year and that will be an integer value for that column. All right, so then at the end I type a semicolon indicating that that's the end of my sequel command. And if I hit enter, I have just created a new table and now if I type dot tables, I can see that I have this car's table but there's nothing in it yet.

So now I need to add observations to this table to add observations to a table that already exists. We can use the insert into commands and then we indicate the name of the table and then we type out values and then we basically provide, it's kind of like a tuple but it's not a tuple but we enclose each observation within parentheses. So I'll just go ahead and type in three different observations. All right, so I've got three different cars here, a 2007 Corolla that gets 36.4 miles per Gallon, 2020 wrapped that gets 20.8 MPG. I need to make sure and put a closing parentheses there and then in 2020 Tesla that gets 50.4 MPG, we'll go ahead and hit enter and you don't see anything that's good. If there were an error then it would tell us that there was an error. So the next step is I want to see if that data actually exists and see what it looks like.

```
sqlite> SELECT * FROM cars;
36.4|Corolla|2007
20.8|Raptor|2020
50.4|Tesla|2020
sqlite>
```

So to be able to see every observation in the table, we'll use this select command and star means every column and then from cars will hit the semicolon there. And now you can see the three observations and it's divided up into columns for that table. Now that we have a table, there's some additional information that I want to show you.

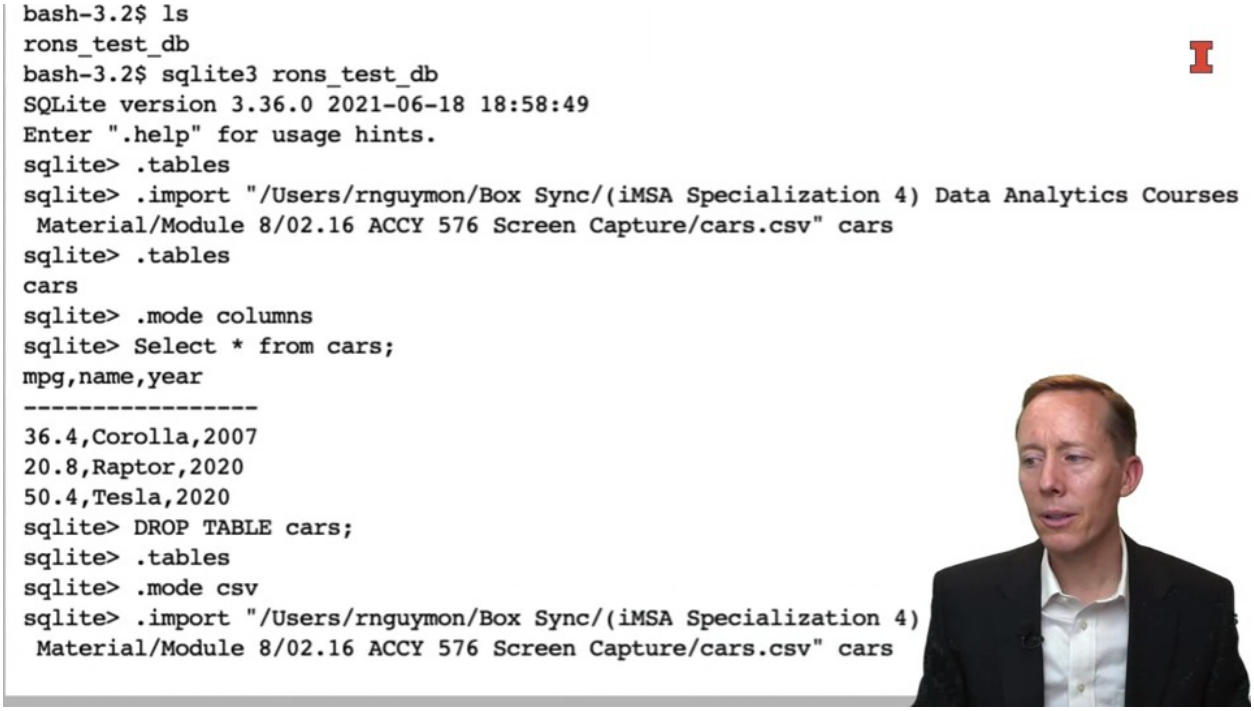
```
.indexes ?TABLE?          Show names of indexe
.limit ?LIMIT? ?VAL?      Display or change th
.lint OPTIONS             Report potential sch
.log FILE|off             Turn logging on or o
.mode MODE ?TABLE?        Set output mode
.nullvalue STRING         Use STRING in place
.once ?OPTIONS? ?FILE?    Output for the next
.open ?OPTIONS? ?FILE?    Close exist  datab
.output ?FILE?            Send output  FILE
.parameter CMD ...        Manage SQL  meter
.print STRING...          Print l
.progress N               Invoke
```

Let's read the help here and one of these usage hints is the mode and we can change the way that the data is viewed by changing the mode.

```
sqlite> .mode csv
sqlite> SELECT * FROM cars;
36.4,Corolla,2007
20.8,Raptor,2020
50.4,Tesla,2020
sqlite> .mode columns
sqlite> SELECT * FROM cars;
mpg   name     year
----  -------  ----
36.4  Corolla  2007
20.8  Raptor   2020
50.4  Tesla    2020
sqlite> .schema cars
CREATE TABLE cars (mpg REAL, name
sqlite> DROP TABLE cars;
sqlite> .tables
sqlite>
```

So if we use dot mode CSV and hit enter and then I'll type the up arrow a couple of times and it goes back and automatically inserts recent commands that I've done. So I'll use a select star from cars and hit enter and you can see that now it's a comma separated value type format. So that CSV format and then the mode columns format I find to be very useful. So we'll go to mode columns and then again select every observation from the cars table and I like this column mode because it has the column name and a fixed width for each column. It just looks really nice here so I can see all three of those observations. So that's how you can create a table and populate it with different values similar to pandas data frames. We often want to know something about the structure of the data frame. You can get something like that with sequel light tables. So if we type dot schema and then the name of this table here cars, we can see the different column names and the data type for each of those. So that is something that can be really helpful. The last thing I want to show you is how you can delete a table. So once we have a table in here, if we say drop table and then the name of the table and semicolon and then use the dot tables. We no longer have that table in there. So there's a quick introduction to how you can create a database and populate it with tables all from the terminal on your machine.

Creating a SQLite Table From a CSV File



In this video, I will demonstrate how you can create tables in a SQLite database by importing CSV files. I will show you how you can do this from a terminal window. If you want to follow along, make sure that you have opened the terminal in JupyterLab or from the Terminal application and then navigate to the folder where you have the files that you want to import. I have a couple of CSV files here. Let's just start with this one; cars.csv. I want to take that and import it into a database within this test database folder. I'm going to need the full path name for where this CSV file is located. I will copy and paste this right here and then I will change directories to the test database folder. Then here you can see I have this rons_test_db. I will enter that by using the SQLite3 command and if I use the dot tables command, you can see I don't have anything in here. You can read a little bit of help about this, but we're going to use the import function. The way we use this is we need to type out.Import and then in quotation marks the full path name to the CSV file. I need to add in here a cars.csv after pasting that path name and then I need to give a name to this table that I want to create and I'll just call it cars. I'll go ahead and run that.

Now if I type out.tables, you can see I have a cars table in here. That's a whole lot faster than manually typing in every observation. Let's go ahead and explore this table a little bit and I'm going to change the mode to columns and then I will select every column and every observation from the cars table. If I do that, you can see that it doesn't look quite right. It looks like I have a single column that is named mpg, name, year, and three observations for that single column. That's not quite right. I need to improve this a little bit. Here's what I need to do. First, I will drop the cars table. Drop table cars. We can see that I no longer have that table and I need to change the mode to CSV first. Now I can import that CSV file. I'm just hitting the up arrow.

```
sqlite> .mode csv
sqlite> .import "/Users/rnguymon/Box Sync/(iMSA Specialization 4) Data Analytics Courses (U of I Box
 Material/Module 8/02.16 ACCY 576 Screen Capture/cars.csv" cars
sqlite> .mode columns
sqlite> Select * from cars;
mpg   name     year
----  -------  ----
36.4  Corolla  2007
20.8  Raptor   2020
50.4  Tesla    2020
sqlite> .schema cars
CREATE TABLE IF NOT EXISTS "cars"(
  "mpg" TEXT,
  "name" TEXT,
  "year" TEXT
);
sqlite> DROP TABLE cars;
sqlite> CREATE TABLE cars (mpg REAL, name TEXT, year INT);
sqlite> .tables
cars
sqlite> .schema cars
CREATE TABLE cars (mpg REAL, name TEXT, year INT);
sqlite> .mode csv
sqlite> .import "/Users/rnguymon/Box Sync/(iMSA Specialization 4) Data Ana
 Material/Module 8/02.16 ACCY 576 Screen Capture/cars.csv" cars
```

Now when I run this, it will know that it should interpret the data as a CSV file. Now if I hit Enter and then I'll go ahead and change the mode back to columns, and then I will select every column and observation from that cars table and display it. This looks much better. So I've got the three observations, a column for each of those. The one thing that is not quite right is if I use the schema to look at the datatype for each of these columns, all of them are texts and that's not what I want. Now, SQL lite database tables are pretty rigid. You can't change the datatype after you've created your table. You can add new columns, but you can't change the type of an existing column. I'm going to go ahead and drop this table again and I will first create the shell for the table and then import the CSV values. Great. Now I have this shell here, and if I look at the schema, I've got the right datatype. Now I can go ahead and change the mode to CSV and then import that CSV file. I'll refer to that table that already exists and I'll change the mode back to columns and then display every column and every row from cars.

```
sqlite> .tables
cars
sqlite> .schema cars
CREATE TABLE cars (mpg REAL, name TEXT, year INT);
sqlite> .mode csv
sqlite> .import "/Users/rnguymon/Box Sync/(iMSA Specialization 4) Data Ar
 Material/Module 8/02.16 ACCY 576 Screen Capture/cars.csv" cars
sqlite> .mode columns
sqlite> Select * FROM cars;
mpg   name     year
----  -------  ----
mpg   name     year
36.4  Corolla  2007
20.8  Raptor   2020
50.4  Tesla    2020
sqlite> .schema cars
CREATE TABLE cars (mpg REAL, name TEXT, year INT);
sqlite>
```

Now it looks about the same but if I look at the schema, I can see that I have the right data type for each of those columns and that's important if we want to make calculations and import the data correctly. Anyway, there's how you can create a table in a database by importing a CSV file. Later on we'll show you how you can create a table from Python using Pandas, DataFrames, which is even better. But really at this point, I'm just trying to illustrate the basics of how a SQL lite database works and how you can import CSV files into a database.

Using Dump and Reading in Files to Create Tables

```
bash-3.2$ ls
Connect to Remote MySQL Database.ipynb
Create Database on AWS.ipynb
bash-3.2$ cd testDatabase/
bash-3.2$ ls
rons_test_db
bash-3.2$
```

In this video, I will introduce you to the dump command in a database. I will also show you how you can perform some SQL commands by reading in a text file into a database. First, you should know that I want to create a new table from this people.csv file. I'm going to copy the path to where this file is located and then navigate to my database. Going to the test database folder and I've got that rons test database in there.

```
Enter ".help" for usage hints.
sqlite> .tables
cars
sqlite> .mode csv
sqlite> .import "/Users/rnguymon/Box Sync/(iMSA Specialization 4) Data A
 Material/Module 8/02.16 ACCY 576 Screen Capture/people.csv" people
sqlite> .mode columns
sqlite> Select * from people;
firstName  lastName        carOwned  age  homeCountry
---------  --------------  --------  ---  -----------
Phineas    Flynn           BMW       35   USA
Ferb       Fletcher        Tesla     33   England
Heinz      Doofenshmirtz   Corolla   60   Germany
Buford     Van Stomm       Corolla   34   Holland
Baljeet    Khatri          Tesla     30   India
Isabella   Garcia-Shapiro  Corolla   36   Mexico
Coltrane   Green           BMW       42   Somalia
sqlite>
```

I'll enter it using sqlite3. You can see right now I just have the cars table. I'm going to change the mode to csv and then I will use import and type out the full path name to the people.csv file, and I will call this new table People. Then I will change the mode to columns and select everything from the people table. You can see I've got seven different rows. I've got first name, last name, car owned, age, and home country.

```
  "carOwned" TEXT,
  "age" TEXT,
  "homeCountry" TEXT
);
sqlite> .dump
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE cars (mpg REAL, name TEXT, year INT);
INSERT INTO cars VALUES('mpg','name','year');
INSERT INTO cars VALUES(36.399999999999996802,'Corolla',2007);
INSERT INTO cars VALUES(20.80000000000000071,'Raptor',2020);
INSERT INTO cars VALUES(50.400000000000000355,'Tesla',2020);
CREATE TABLE IF NOT EXISTS "people"(
  "firstName" TEXT,
  "lastName" TEXT,
  "carOwned" TEXT,
  "age" TEXT,
  "homeCountry" TEXT
);
INSERT INTO people VALUES('Phineas','Flynn','BMW','35','USA');
INSERT INTO people VALUES('Ferb','Fletcher','Tesla','35','England');
INSERT INTO people VALUES('Heinz','Doofenshmirtz','Corolla','60','Germany');
INSERT INTO people VALUES('Buford','Van Stomm','Corolla','34','Holland');
INSERT INTO people VALUES('Baljeet','Khatri','Tesla','30','India');
INSERT INTO people VALUES('Isabella','Garcia-Shapiro','Corolla','36','Mexico');
INSERT INTO people VALUES('Coltrane','Green','BMW','42','Somalia');
COMMIT;
sqlite>
```

Now if I look at the schema, all of these columns are text columns. I would like for this age column to be an integer or a float value. Well, I could type out every column name, but since this table already exists, there's a helpful command which is the dump command. If I type.dump, then it tells me everything that I've done with this database. If I look at this, I can see how I created the cars table and then how I created this people table. I want to go ahead and copy this and rather than type everything out manually, I want to paste it and then rerun it. If I were to try and do that within SQL here, I can't really navigate up to prior columns here. I can't really go up here and just change this age value to a real datatype. I'll make sure this doesn't run by not having a semicolon there and maybe adding some other things and semicolon, then there's an error. What I want to do now is create a text file and then paste these commands into there and I'll just read that text file in rather than having to type everything out. I could create a text file all from the terminal. If I exit out of SQL, I clear this out.

Terminal 1 | createPeople.txt

```
bash-3.2$ sqlite3 rons_test_db
SQLite version 3.36.0 2021-06-18 18:58:49
Enter ".help" for usage hints.
sqlite> .tables
cars    people
sqlite> DROP TABLE people;
sqlite> .tables
cars
sqlite> .read createPeople.txt
sqlite> .tables
cars    people
sqlite>
```

I could go ahead and do that, but I want to show you how you can do that from within JupyterLab. It makes it easier because you don't have to learn another type of text editor. It should be standard for most people. I'm going to go over to the file explorer and make sure that I go into the test database. We don't have to, but I'll go ahead and do that. From here, I will go to the launcher and I want to create this text file. I can just paste in here the text that I want. It's not a real nifty editor, but I can highlight things with my mouse and then I'll go ahead and just change the datatype for the age column to real. Then I will save this. I'll save it as create people.txt. Now I've got that file in here. From here, I'll go back into terminal and I will enter my database. We still have that people table in there, so I want to get rid of that. Now it's not there. I'll go ahead and read in that file by using the read command. I need to indicate the name of the file, and since it's in the same folder as this database, I don't have to type out the full path name. I'll just type out the filename, create people.txt and that's it. Now if I type out.tables, I have a new people table in there, and if I look at the schema for that table, I can see that age is a real datatype.

```
   "lastName" TEXT,
   "carOwned" TEXT,
   "age" REAL,
   "homeCountry" TEXT
);
sqlite> .mode csv
sqlite> .quit
bash-3.2$
```

From here, I can go ahead and insert values into this by reading in the CSV file. I will change the mode to csv. I guess I'll actually need to quit, I'll clear this.

```
SQLite version 3.36.0 2021-06-18 18:58:49
Enter ".help" for usage hints.
sqlite> .mode csv
sqlite> .import "/Users/rnguymon/Box Sync/(iMSA Specialization 4) Da
 Material/Module 8/02.16 ACCY 576 Screen Capture/people.csv" people
sqlite> .schema people
CREATE TABLE IF NOT EXISTS "people"(
  "firstName" TEXT,
  "lastName" TEXT,
  "carOwned" TEXT,
  "age" REAL,
  "homeCountry" TEXT
);
sqlite>
```

I will print my working directory and I need this part of it right here and then go back into the database, change the mode to csv and we'll import people.csv file and we'll insert those values into the people table. Now let's go ahead and look at the schema, make sure it's still same. Age is a real number and let's go ahead and select all the observations from that table.

```
Ferb,Fletcher,Tesla,35.0,England
Heinz,Doofenshmirtz,Corolla,60.0,Germany
Buford,"Van Stomm",Corolla,34.0,Holland
Baljeet,Khatri,Tesla,30.0,India
Isabella,Garcia-Shapiro,Corolla,36.0,Mexico
Coltrane,Green,BMW,42.0,Somalia
sqlite> .mode columns
sqlite> Select * From people;
firstName  lastName        carOwned  age   homeCountry
---------  --------------  --------  ----  -----------
firstName  lastName        carOwned  age   homeCountry
Phineas    Flynn           BMW       35.0  USA
Ferb       Fletcher        Tesla     35.0  England
Heinz      Doofenshmirtz   Corolla   60.0  Germany
Buford     Van Stomm       Corolla   34.0  Holland
Baljeet    Khatri          Tesla     30.0  India
Isabella   Garcia-Shapiro  Corolla   36.0  Mexico
Coltrane   Green           BMW       42.0  Somalia
sqlite>
```

It's still a csv format. Let's change the mode to columns and then look at it. This looks very good. Now I can see the name for each column and all the rows in here. The one problem is that I have this first row which is the column headers and that's not quite right. We'll take care of that in another video where I will show you how you can alter some of the information in a SQLite database. Before I conclude, I want to show you a more efficient and effective approach for recreating a table using the dump command.

Untitled.ipynb | Terminal 1 | createPeople.txt

```
1  CREATE TABLE IF NOT EXISTS "cars"(
2    "mpg" REAL,
3    "name" TEXT,
4    "year" INT
5  );
6  INSERT INTO cars VALUES('36.4','Corolla','2007');
7  INSERT INTO cars VALUES('20.8','Raptor','2020');
8  INSERT INTO cars VALUES('50.4','Tesla','2020');
9  CREATE TABLE IF NOT EXISTS "people"(
10   "firstName" TEXT,
11   "lastName" TEXT,
12   "carOwned" TEXT,
13   "age" REAL,
14   "homeCountry" TEXT
15 );
16 INSERT INTO people VALUES('Phineas','Flynn','BMW','35','USA');
17 INSERT INTO people VALUES('Ferb','Fletcher','Tesla','35','England');
18 INSERT INTO people VALUES('Heinz','Doofenshmirtz','Corolla','60','Germany');
19 INSERT INTO people VALUES('Buford','Van Stomm','Corolla','34','Holland');
20 INSERT INTO people VALUES('Baljeet','Khatri','Tesla','30','India');
21 INSERT INTO people VALUES('Isabella','Garcia-Shapiro','Corolla','36','Mexico');
22 INSERT INTO people VALUES('Coltrane','Green','BMW','42','Somalia');
```

The main purpose of the dump command is to print out all the instructions for recreating a database. If I copy everything for creating the people table as well as the cars table. I'll go ahead

and copy that. Then if I go back into this create people text file, I'll paste the instructions for creating cars and I'll go ahead and change miles per gallon to real and year to int and importantly the age to real. Now I'll save this. Perfect.

```
INSERT INTO people VALUES('Buford','Van Stomm',
INSERT INTO people VALUES('Baljeet','Khatri','
INSERT INTO people VALUES('Isabella','Garcia-Sh
INSERT INTO people VALUES('Coltrane','Green','I
COMMIT;
sqlite> drop table cars;
sqlite> drop table people;
sqlite> .tables
sqlite> .read createPeople.txt
sqlite> .tables
cars    people
sqlite> .
```

Now I'll go back into the terminal. Let me go ahead and drop the tables that are in here. I will drop the cars table and drop the people table. Make sure those are not there. Now I can use the read command with that create people.txt file. Let's make sure it works. I've got now the two tables, cars and people, and if I look at the schema for people, I can see that age is real.

```
  "homeCountry" TEXT
);
sqlite> .mode columns
sqlite> Select * from people;
firstName  lastName        carOwned  age   homeCountry
---------  -------------  --------  ----  -----------
Phineas    Flynn          BMW       35.0  USA
Ferb       Fletcher       Tesla     35.0  England
Heinz      Doofenshmirtz  Corolla   60.0  Germany
Buford     Van Stomm      Corolla   34.0  Holland
Baljeet    Khatri         Tesla     30.0  India
Isabella   Garcia-Shapiro Corolla   36.0  Mexico
Coltrane   Green          BMW       42.0  Somalia
sqlite>
```

If I print out the data for the people table, let's go ahead and change the mode to columns and

then select everything from people. Perfect. Now I've got all those observations. I don't have the header column in there and the age column is a real datatype. This is really what the dump function is for. It allows you to recreate a whole database without having to recreate just the table container and then import data from a csv file.

Altering Existing SQLite Tables

```
sqlite> .tables
cars    people
sqlite> ALTER TABLE people RENAME TO peeps;
sqlite> .tables
cars   peeps
sqlite>
```

In this video, I'm going to demonstrate some ways in which you can alter existing tables within a SQLite database. Specifically, I will demonstrate that you can change the names of tables and you can add and remove rows from a table. You can also add columns to a table but you won't be able to remove columns from a table or change the datatype of those columns. Let's go ahead and get into it. I'm working from this runs test DB database and in here I've got two tables, cars and people. I will demonstrate first how you can change the name of one of these tables. Specifically, I will change the name of the people table. It's fairly simple, but you do have to remember the wording but you use alter table, then the name of the table, and then rename it to the new name. Now if we look at the tables, we've got cars and peeps in here.

```
Buford|Van Stomm|Corolla|34.0|Holland
Baljeet|Khatri|Tesla|30.0|India
Isabella|Garcia-Shapiro|Corolla|36.0|Mexico
Coltrane|Green|BMW|42.0|Somalia
sqlite> .mode columns
sqlite> Select * from peeps;
firstName   lastName        carOwned   age   homeCountry
---------   --------------  --------   ----  -----------
firstName   lastName        carOwned   age   homeCountry
Phineas     Flynn           BMW        35.0  USA
Ferb        Fletcher        Tesla      35.0  England
Heinz       Doofenshmirtz   Corolla    60.0  Germany
Buford      Van Stomm       Corolla    34.0  Holland
Baljeet     Khatri          Tesla      30.0  India
Isabella    Garcia-Shapiro  Corolla    36.0  Mexico
Coltrane    Green           BMW        42.0  Somalia
sqlite>
```

Let's go ahead and look at all the observation in the peeps table. Let me go ahead and change the mode to columns , here we go. We can see that we've got lots of observations that look good. But when I imported this table, it read it in from a CSV file and it used the column headers as an observation and I don't want that. Here's how you can get rid of rows systematically from a table.

```
firstName   lastName        carOwned   age   homeCountry
Phineas     Flynn           BMW        35.0  USA
Ferb        Fletcher        Tesla      35.0  England
Heinz       Doofenshmirtz   Corolla    60.0  Germany
Buford      Van Stomm       Corolla    34.0  Holland
Baljeet     Khatri          Tesla      30.0  India
Isabella    Garcia-Shapiro  Corolla    36.0  Mexico
Coltrane    Green           BMW        42.0  Somalia
sqlite> DELETE FROM peeps WHERE firstName == 'firstName';
sqlite> Select * from peeps;
firstName   lastName        carOwned   age   homeCountry
---------   --------------  --------   ----  -----------
Phineas     Flynn           BMW        35.0  USA
Ferb        Fletcher        Tesla      35.0  England
Heinz       Doofenshmirtz   Corolla    60.0  Germany
Buford      Van Stomm       Corolla    34.0  Holland
Baljeet     Khatri          Tesla      30.0  India
Isabella    Garcia-Shapiro  Corolla    36.0  Mexico
Coltrane    Green           BMW        42.0  Somalia
sqlite>
```

We use DELETE FROM peeps and then we will indicate a condition that needs to be met. In this case, I'll look in here and I can see that I can just use this firstName column, where the firstName values equal to and then in quotation marks firstName, and then run that. Then if I select all the rows again, you can see that is no longer there. You can obviously use this to remove more than

one row at a time. You could remove rows for which the age is less than a certain criteria or something like that. Now I've already shown you this before, but I want to remind you that you can insert new values into an existing table.

```
Isabella    Garcia-Shapiro  Corolla    36.0  Mexico
Coltrane    Green           BMW        42.0  Somalia
sqlite> sqlite> INSERT INTO pe
   ...> ;
Error: near "VALU": syntax error
sqlite> Insert into peeps values ('Francis', 'Monogram', 'Corolla', 65, '
sqlite> Select * from peeps;
firstName  lastName        carOwned  age   homeCountry
---------  --------------  --------  ----  -----------
Phineas    Flynn           BMW       35.0  USA
Ferb       Fletcher        Tesla     35.0  England
Heinz      Doofenshmirtz   Corolla   60.0  Germany
Buford     Van Stomm       Corolla   34.0  Holland
Baljeet    Khatri          Tesla     30.0  India
Isabella   Garcia-Shapiro  Corolla   36.0  Mexico
Coltrane   Green           BMW       42.0  Somalia
Francis    Monogram        Corolla   65.0  USA
sqlite>
```

It's fairly straight forward. We say insert into and then the name of the table and then the values that we want to insert. Within parentheses we list the value for each of the columns, and then make sure we have that semicolon at the end. Now you can see the very bottom. It has appended this new observation for Francis monogram into the table. Here's how you can add a new column to an existing table.

```
Phineas    Flynn            BMW        35.0  USA
Ferb       Fletcher         Tesla      35.0  England
Heinz      Doofenshmirtz    Corolla    60.0  Germany
Buford     Van Stomm        Corolla    34.0  Holland
Baljeet    Khatri           Tesla      30.0  India
Isabella   Garcia-Shapiro   Corolla    36.0  Mexico
Coltrane   Green            BMW        42.0  Somalia
Francis    Monogram         Corolla    65.0  USA
sqlite> ALTER TABLE peeps ADD ageplus REAL;
sqlite> .schema peeps
CREATE TABLE IF NOT EXISTS "peeps"(
  "firstName" TEXT,
  "lastName" TEXT,
  "carOwned" TEXT,
  "age" REAL,
  "homeCountry" TEXT
, ageplus REAL);
sqlite>
```

I just added a new column to this table, and I can see that if I look at the schema and say I've got this ageplus column, but nothing is in it yet. I want to actually make this column a calculation of an existing column. So I'm going to go ahead and add 15 years to the existing age.

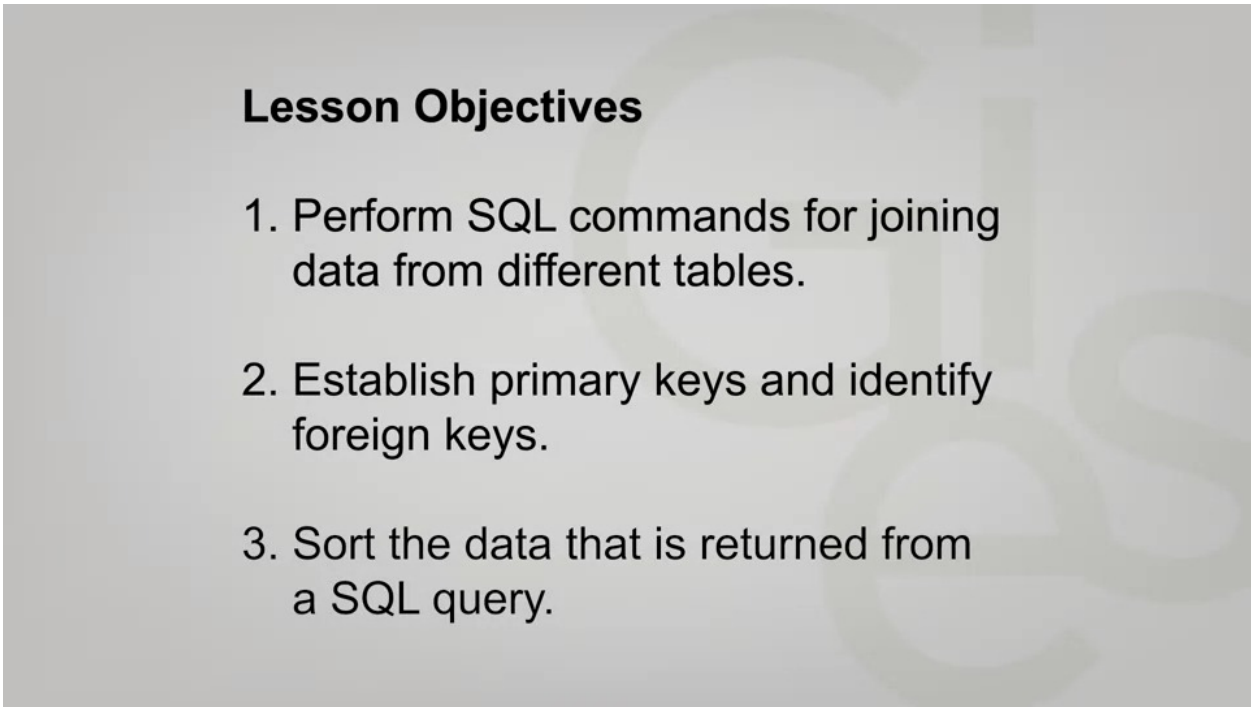
```
sqlite> ALTER TABLE peeps ADD ageplus REAL;
sqlite> .schema peeps
CREATE TABLE IF NOT EXISTS "peeps"(
  "firstName" TEXT,
  "lastName" TEXT,
  "carOwned" TEXT,
  "age" REAL,
  "homeCountry" TEXT
, ageplus REAL);
sqlite> UPDATE peeps SE ageplus = age + 15
sqlite> Select * from peeps;
firstName  lastName        carOwned  age   homeCountry  ageplus
---------  -------------  --------  ----  -----------  -------
Phineas    Flynn          BMW       35.0  USA          50.0
Ferb       Fletcher       Tesla     35.0  England      50.0
Heinz      Doofenshmirtz  Corolla   60.0  Germany      75.0
Buford     Van Stomm      Corolla   34.0  Holland      49.0
Baljeet    Khatri         Tesla     30.0  India        45.0
Isabella   Garcia-Shapiro Corolla   36.0  Mexico       51.0
Coltrane   Green          BMW       42.0  Somalia      57.0
Francis    Monogram       Corolla   65.0  USA          80.0
sqlite>
```

Let's go ahead and look at all the observations. Now you can see that this new age plus column is the existing age plus 15. There's an example of some things that you can do to alter existing tables in a SQLite data frame. As I said before, there's no way to get rid of existing columns or to change the datatype. In order to do that, you might want to use the dump function to find all the commands that you have used for the table and then just modify some of those. You might

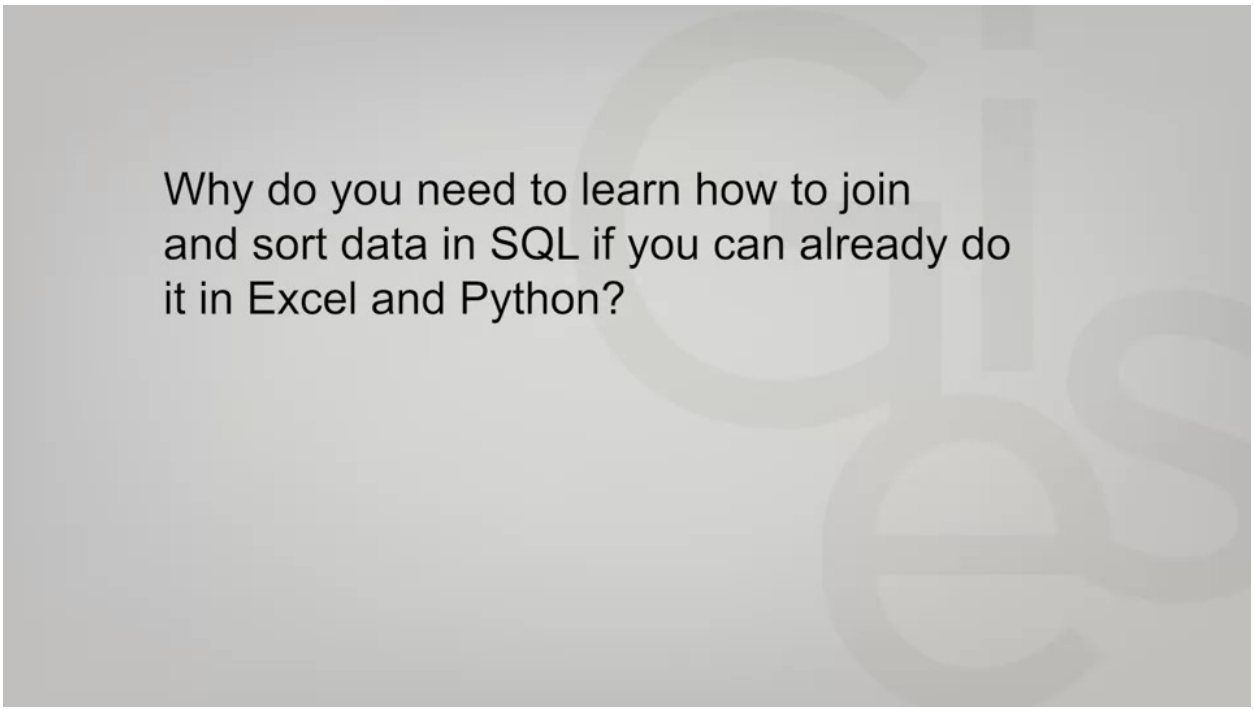
wonder, why can we not alter tables as easily or extensively as we can pandas, dataframes. I'm not sure exactly why, but I think it's somewhat due to security issues. There would be a lot of lost data that would occur if people used the wrong command and they unintentionally deleted a column or something like that. But anyway, you can make some alterations to database tables from the command line, and that's how you do it.

Lesson 8.2 SQL: Advanced Concepts Information

Lesson 8.2 Advanced Concepts

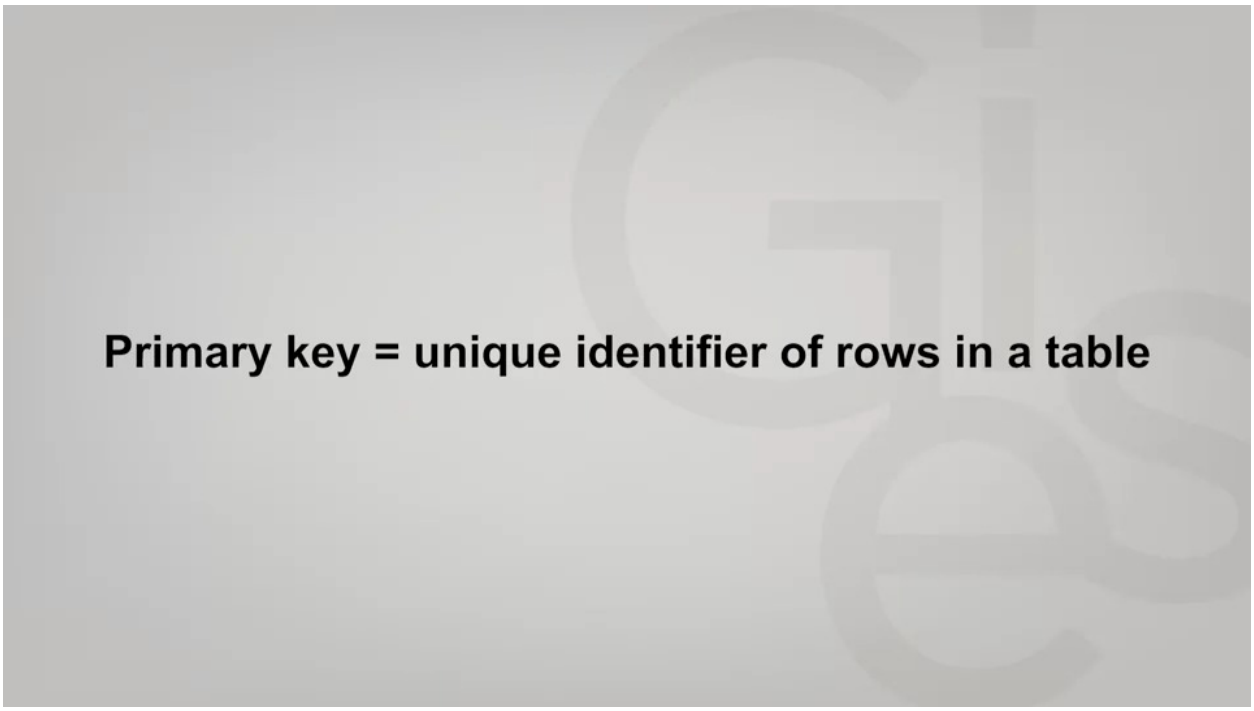


At this point you should already know how to create a SQL database, add data, update data, select certain rows of data, and delete data. Since SQL databases are made up of many tables, in this lesson you'll learn how to perform SQL commands for joining data together from different tables. To better understand the joins, you'll also learn about primary keys and foreign Keys. You'll also learn to order data from SQL query. Just like Excel and python can pivot and combine data, SQL can do so as well.

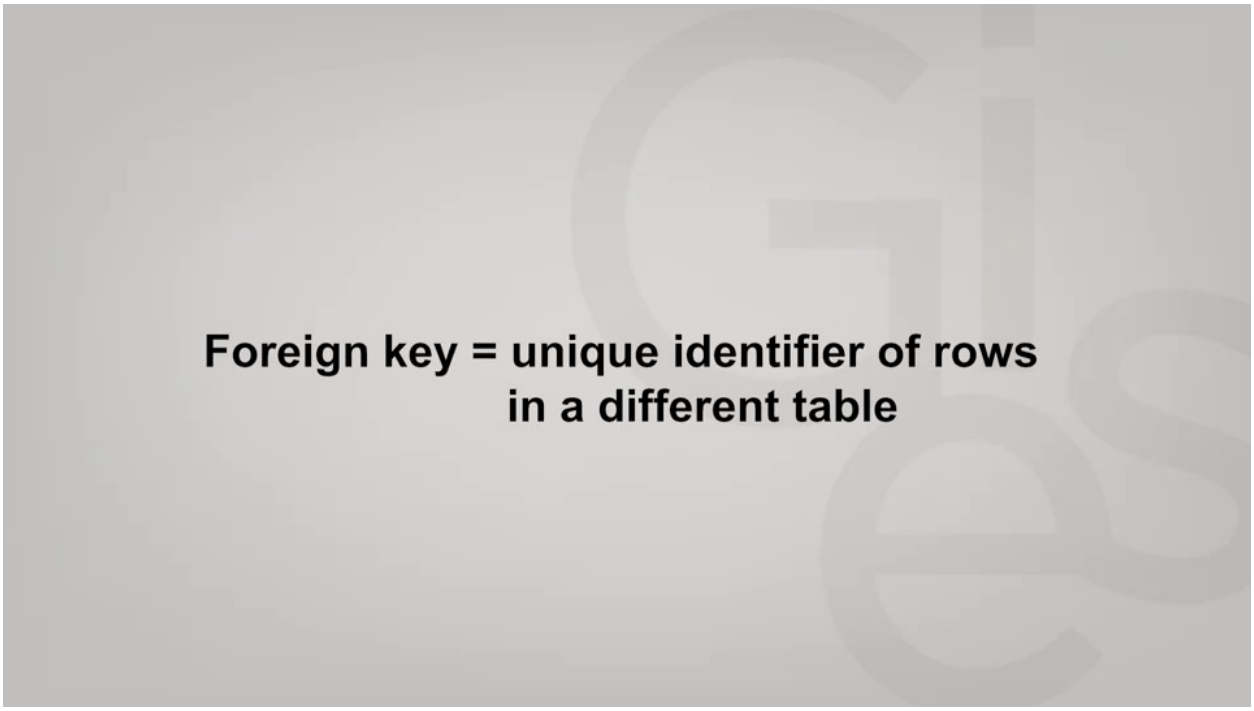


You might wonder why it's important to learn how to do data joins and pivots and SQL if you already know how to perform them in Python and Excel. One good reason is because databases are often made up of so much data that they're too large to completely export into a desktop application. Even if you did have a data analysis application that could handle all of the data it would be very inefficient to essentially replicate the database.

This inefficiency problem is magnified by the fact that SQL databases could consist of hundreds or thousands of tables. So it's not just one large table that you'll have to worry about managing but hundreds or thousands. So learning how to join tables in different ways using SQL will greatly reduce inefficiencies. For this reason, it's important to understand the different types of joins and how to perform them in SQL so that you only have to export the relevant pieces of data. Joins are made possible by keys or the ability to uniquely identify a row of data. In a relational database every row in a table has what's known as a primary key. This can be set to be the combination of values in one or more columns of data or set to simply be row number.



Primary keys are essentially a unique way to identify each row of a database



and foreign keys indicate the unique row in a different table. Finally. You'll also learn about the order by clause for sorting the way that data from a query is returned. In short, this lesson will teach you a few SQL commands that will allow you to perform many of the pivot joint and sort functions so that you retrieve only relevant information for your analyses.

Querying Tables with SQL

```
sqlite> .schema cars
CREATE TABLE cars (mpg REAL, name TEXT, year INT);
sqlite> Select * from cars;
mpg|name|year
36.4|Corolla|2007
20.8|Raptor|2020
50.4|Tesla|2020
sqlite>
```

In this video, I want to demonstrate how you can query a table in a database to extract only certain rows and certain columns and this is probably one of the most important things you can learn. It's really easy to learn, but you will use this over and over again. In contrast, creating tables and changing names of tables, you probably won't do that as often as you were getting the data from the table. So I'm going to go ahead and demonstrate this using a table, cars that's within this rons_test database and let's just go ahead and look at the schema. That's something that's very important if you want to be able to query a table. So it's important to recognize that the mpg column is a numeric type column. And so is the year column while the name is a text column. Let's go ahead and use the select function to just select every column, and every row from this table. All right, so that star in there is a wild card. That means I want every column.

```
sqlite> .mode columns
sqlite> Select * from cars;
mpg   name      year
----  -------   ----
mpg   name      year
36.4  Corolla   2007
20.8  Raptor    2020
50.4  Tesla     2020
sqlite>
```

And actually, let's go ahead and change the mode two columns, so that it looks prettier here. All right, so we can see that I've got a header row in there that will get rid of in just a minute and then I've got three observations for different cars in there. So first of all, let's say that hey, I just want observations from the state of frame for which the miles for gallon is greater than 25. All right, so we would exclude the raptor from this one.

```
20.8  Raptor   2020
50.4  Tesla    2020
sqlite> Select * from cars WHERE mpg > 25;
mpg   name     year
----  -------  ----
mpg   name     year
36.4  Corolla  2007
50.4  Tesla    2020
sqlite> Delete from cars where mpg == 'mpg';
sqlite> Select * from cars;
mpg   name     year
----  -------  ----
36.4  Corolla  2007
20.8  Raptor   2020
50.4  Tesla    2020
sqlite> Select * from cars WHERE mpg > 25 and year > 2015;
mpg   name   year
----  -----  ----
50.4  Tesla  2020
sqlite>
```

So it's really easy to do and we'll say, Select *. We want all the columns from cars. We start out the same, but then we use this WHERE clause and I'll capitalize that to highlight that. That is a SQL command and then it's pretty straightforward. We'll say WHERE mpg is greater than 25

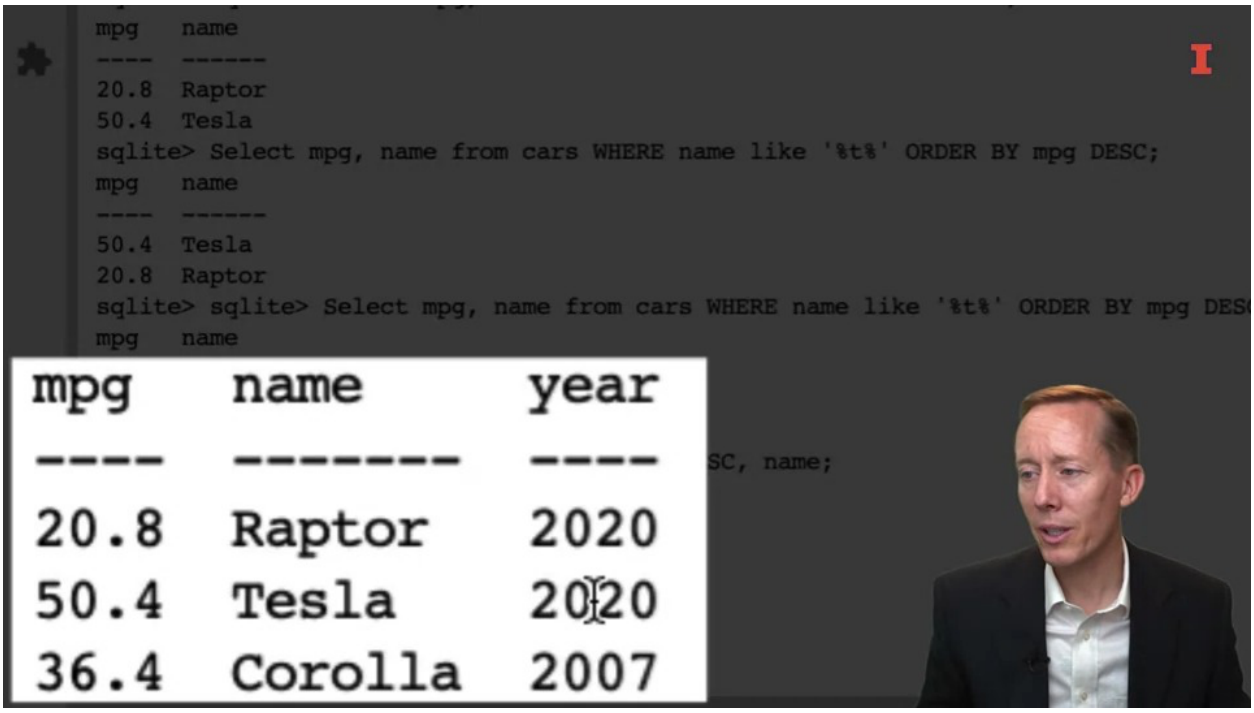
and makes here with a semicolon. And now you can see that we only have Corolla and Tesla, and we still have this other observation here. That's really just the column headers. I'm not sure how mpg is converted to a numeric value. But interestingly, you can have text values that are in a column. That is a numeric value. All right, now you've already seen me in another video. Delete rows from a data frame, but I'll demonstrate that here as well. So let's go ahead and use delete from cars and then we'll use that where operator again where mpg is equal to 'mpg' and run that. And then if we select everything from cars, we no longer have that row. All right, so that where operator is pretty handy. We can use it for querying data, as well as for removing rows from data. Now, what if we want to query the data based on more than one column? There are multiple conditions in here. I'll use the up arrow and I want to select from cars where the miles per gallon is greater than 25, and let's also say where the year is greater than 2015. So I simply type out the word and, and then year is greater than 2015. And now I just get Tesla, so it's really simple to use. It's not too hard to understand at all. It's just making sure you remember a few of these keywords but it's very intuitive.

```
sqlite> Select * from cars WHERE mpg > 25 and year > 2015;
mpg   name   year
----  -----  ----
50.4  Tesla  2020
sqlite> sqlite> Select * from cars WHERE name == 'Corolla';
mpg   name     year
----  -------  ----
36.4  Corolla  2007
sqlite> Select * from cars WHERE name like '%t%';
mpg   name    year
----  ------  ----
20.8  Raptor  2020
50.4  Tesla   2020
sqlite>
```

All right, what if we want to query observations from a table based on a condition from a text column or a string column? That's also pretty easy to do. So I'll start out the same as before. Select * from cars WHERE and then I'm just going to identify those rows where the name is equal to Corolla. So notice I'm using two equal signs there and in quotation marks, Corolla. And if I run that I get just that one row, because that's the only one that has Corolla in it. Now, I can also filter the data or return observations based on a substring. So I'll start with this again where I'll say, hey, I want to select every column from cars. But instead of where the name equals exactly Corolla, we can use like.. And then within the quotation marks, we can enter the substrings surrounded by percentages. All right, so if we say we want everything that has a t in it, then I will run it like this and you'll notice that I get both rapture and Tesla. So this is important to recognize that SQLite is not case sensitive. Whether it's an uppercase T or lowercase t, it will extract those rows.

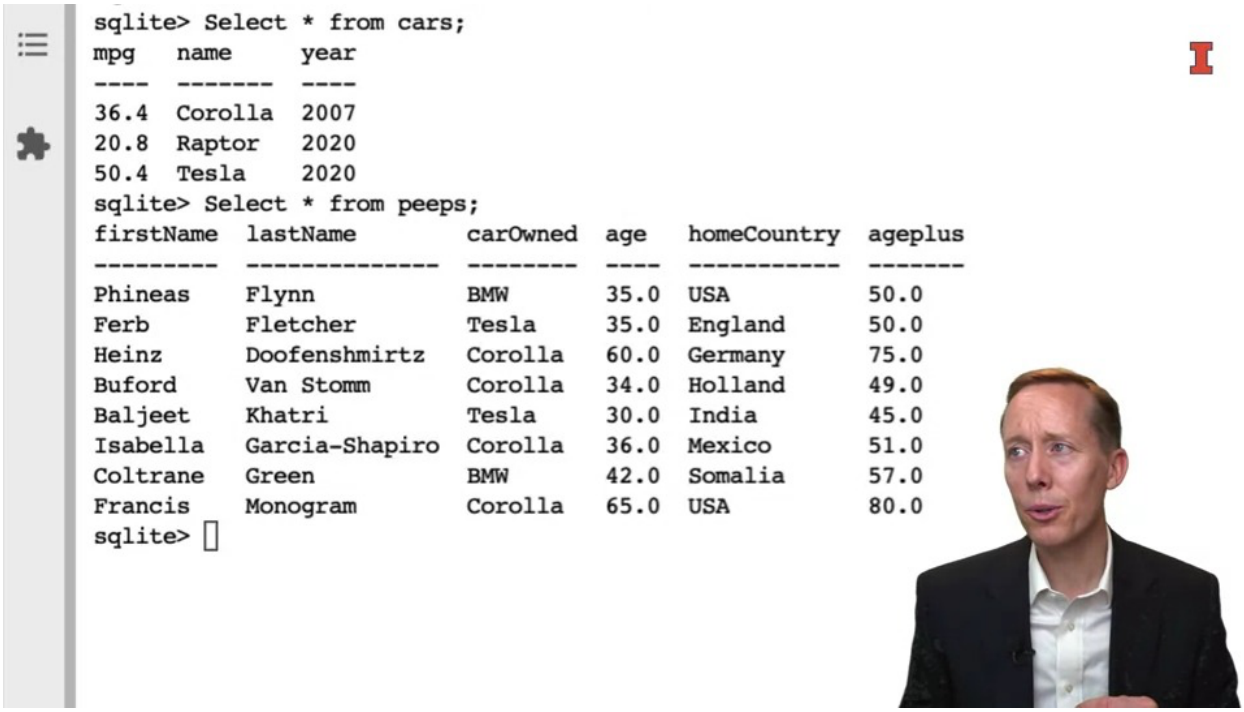
```
sqlite> Select * from cars WHERE name like '%t%';
mpg   name    year
----  ------  ----
20.8  Raptor  2020
50.4  Tesla   2020
sqlite> sqlite> Select mpg, name from cars WHERE name like '%t%';
mpg   name
----  ------
20.8  Raptor
50.4  Tesla
sqlite> Select mpg, name from cars WHERE name like '%t%' ORDER BY mpg DESC;
mpg   name
----  ------
50.4  Tesla
20.8  Raptor
sqlite> sqlite> Select mpg, name from cars WHERE name like '%t%' ORDER  DESC;
mpg   name
----  ------
50.4  Tesla
20.8  Raptor
sqlite> Select * from cars ORDER BY year DESC, name;
```

What if you only want to select certain columns from a table that you're querying? Well, rather than using star, we can just type out the names of the columns separated by commas. So in this case here, I indicated that I just want miles per gallon and name and only those rows where the name contains a t. Now, another thing you can do is you can order the data and that's pretty easy. And the way we would do that is we would add in this ORDER BY and then indicate the column and I will start with mpg. And by default, it is ascending. So if we want it to be descending, then we'll need to indicate it by typing in DESC. And then if we run that, you can see that Tesla is on the top and then wrapped her. And if you want to sort the data by more than one column, that's also pretty straightforward. And in this case, actually, let me just start a new query. We'll select everything, every column from cars, but we will ORDER BY by. Now, let's go ahead and order first by year. And by default, that will be ascending. So let's change that to descending and then we'll use a comma to indicate the next column, and that will be named. And again, I don't need to indicate ascending,cuz that's the default.



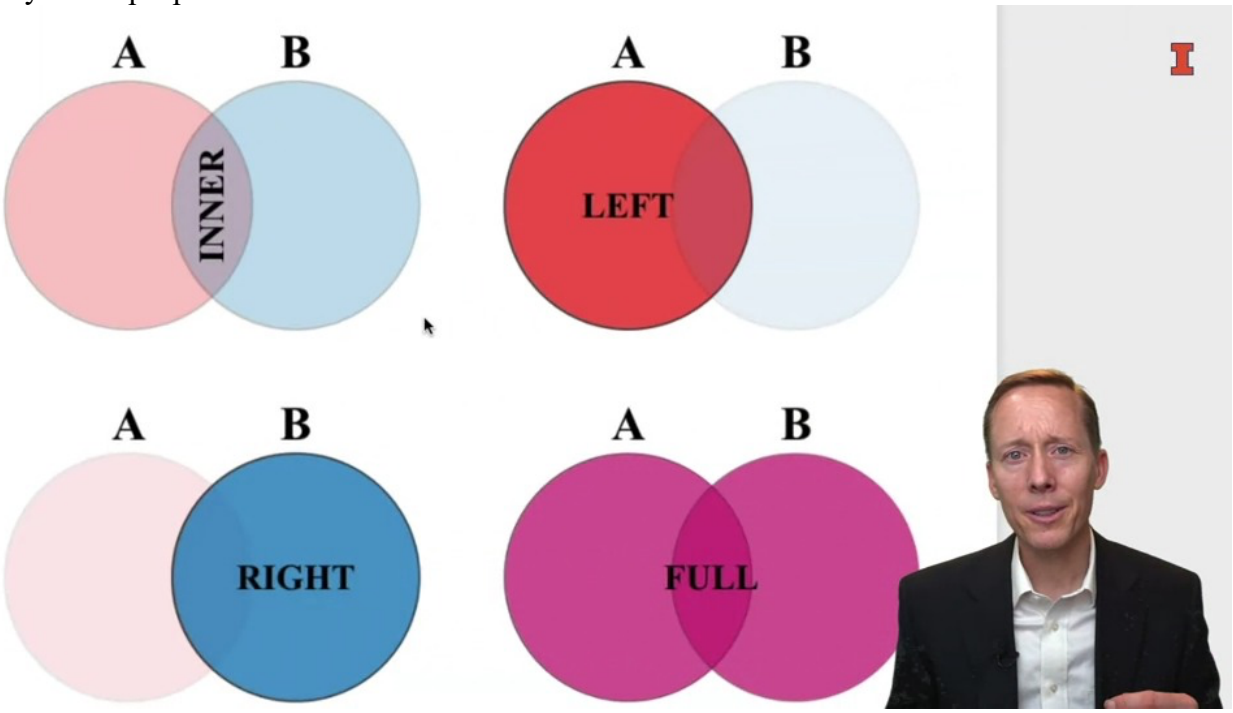
And if I do that, you can see that it first sorts it by year 2020 before 2007. And then by name, Raptor and then Tesla. So these few basics of querying tables in a database are going to be almost identical, regardless of whether you're using a SQLite database or MySQL database or PostgreS or anything else. And these are things that you will use all the time and just keeping track of a few keywords can get you a long ways.

SQL Join Queries

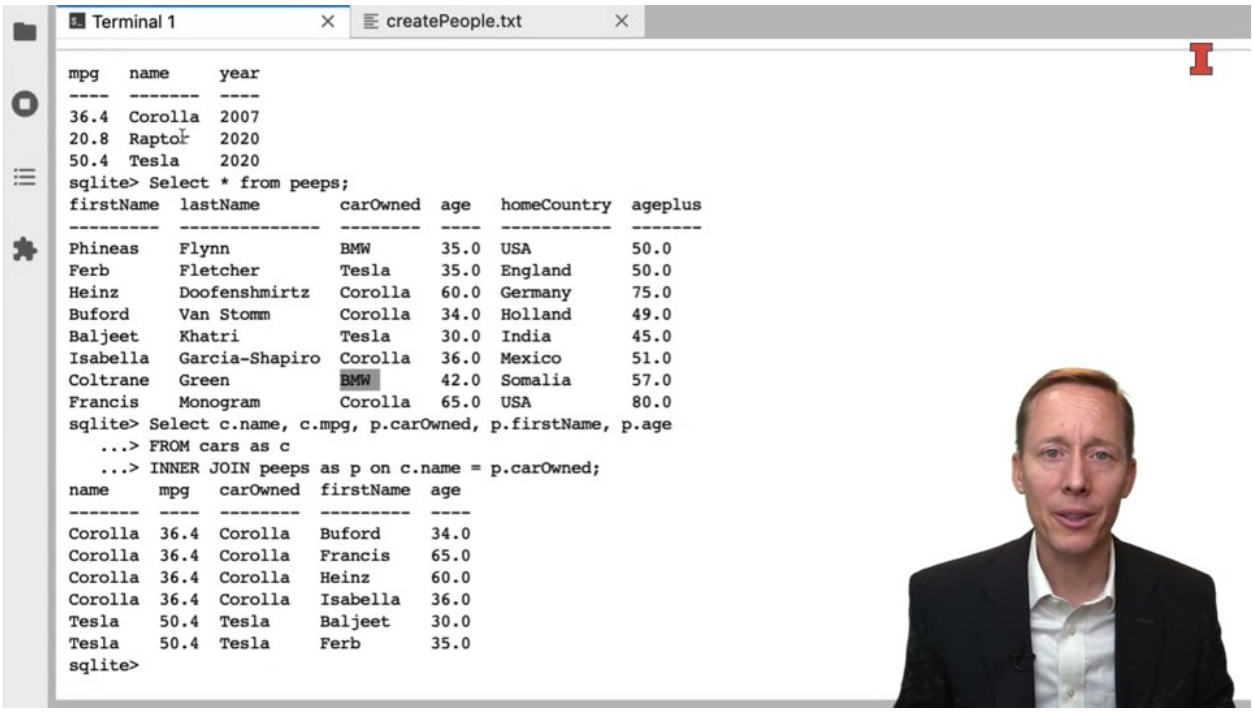


In this video, I will demonstrate how you can perform queries on two tables in a SQL database

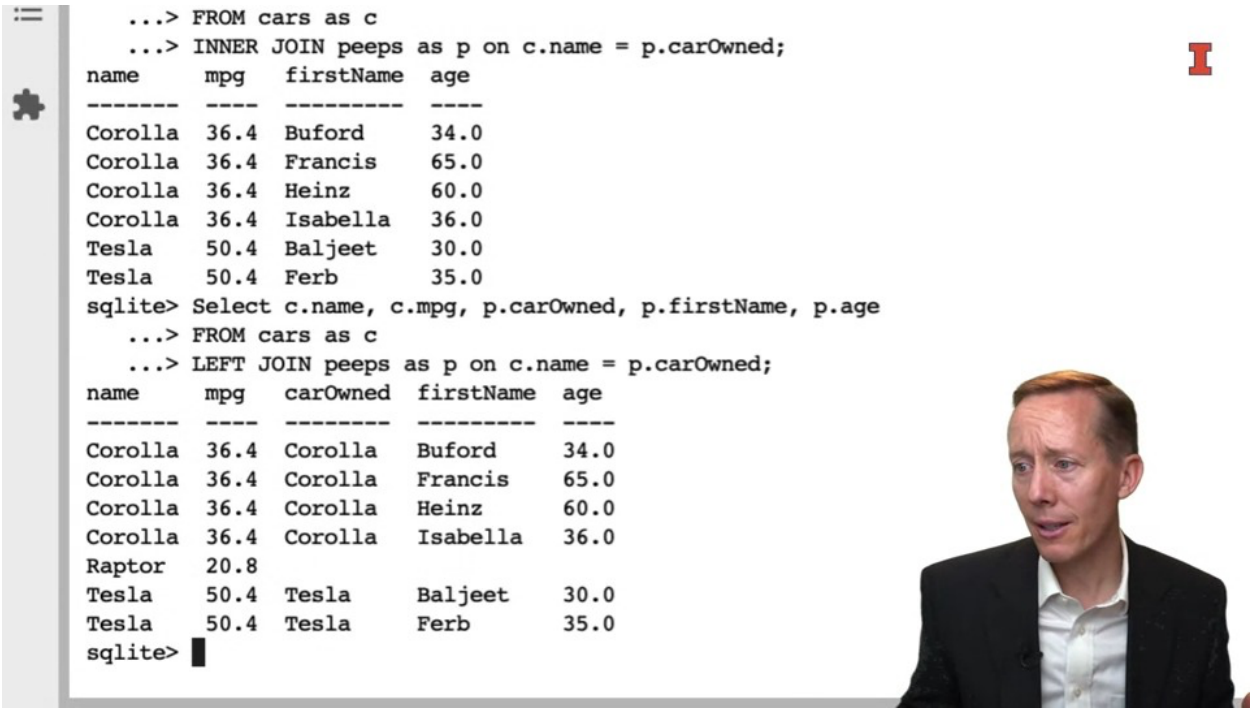
that combines information from both tables and then returns that combined table. As an example, I've got two tables in this test database here, I've got the cars table, and in the name column I have three unique names, Corolla, Raptor and Tesla. I also have this peeps table, and in the carOwned column I have three different unique values of cars. I've got BMW, Tesla and Corolla. So Corolla and Tesla are in both of these tables but Raptor is only in the cars table and BMW is only in the peeps table.



And this brings us to the idea of inner joins, left joins and outer joins. So, to first talk about these conceptually, let's look at these Venn diagrams, and inter join indicates that we only want rose from both tables where there is a value that matches in both tables. A left join means we want all observations from the left table, and only those that can match from the right table in this case the B table. Right joins are very analogous to left joins, except that we're referring to the table on the right, which is kind of arbitrary really, but the one on the right will keep all the values for and only the one on the left. We will include if they match something from the right table, and then a full join just combines every observation from both tables together. So let's demonstrate these joins and actually in SQLite, you can only perform inner and left joins, right joins are basically the same as left joins, but you can't perform a full outer join.



So let's go ahead and just start by demonstrating an inner join. I will jump back over to the terminal where I'm interacting with these databases, and rather than type this out, I'll just go ahead and copy and paste the SQL query here, and now let's talk about it. So the first line, I have the columns that I want to select, and you can see that I'm indicating that I want columns from the cars database, only some of the columns. So the name and the mpg column, and then from the peeps database, I want the car owned first name and the age, and then I say from cars as C. And that's how we know that these column names are from the cars database. And then the next line I'm indicating I want an inner join and peeps will be as P otherwise I'd have to type out the full name of the table there. And I will join these observations based on the name column in the cars table, and then the car owned column in the people table, let's go ahead and run that. What do we get? We've got only observations from cars that have Corolla and Tesla because there's nothing in peeps that has Raptor. And then from the peeps table, these last 3 columns, carOwned, firstName and age. We again have the name of the car, and then we have the first name on the age and notice that we don't have BMW in there because there's no match in the cars table. So this is an inner join. Now it's worth recognizing that you don't have to include the name of the column that you are matching on, because that's kind of redundant here. So we could retype it as follows. All right, now when we run that, we only have the name of the car in there once, and that's probably more ideal.



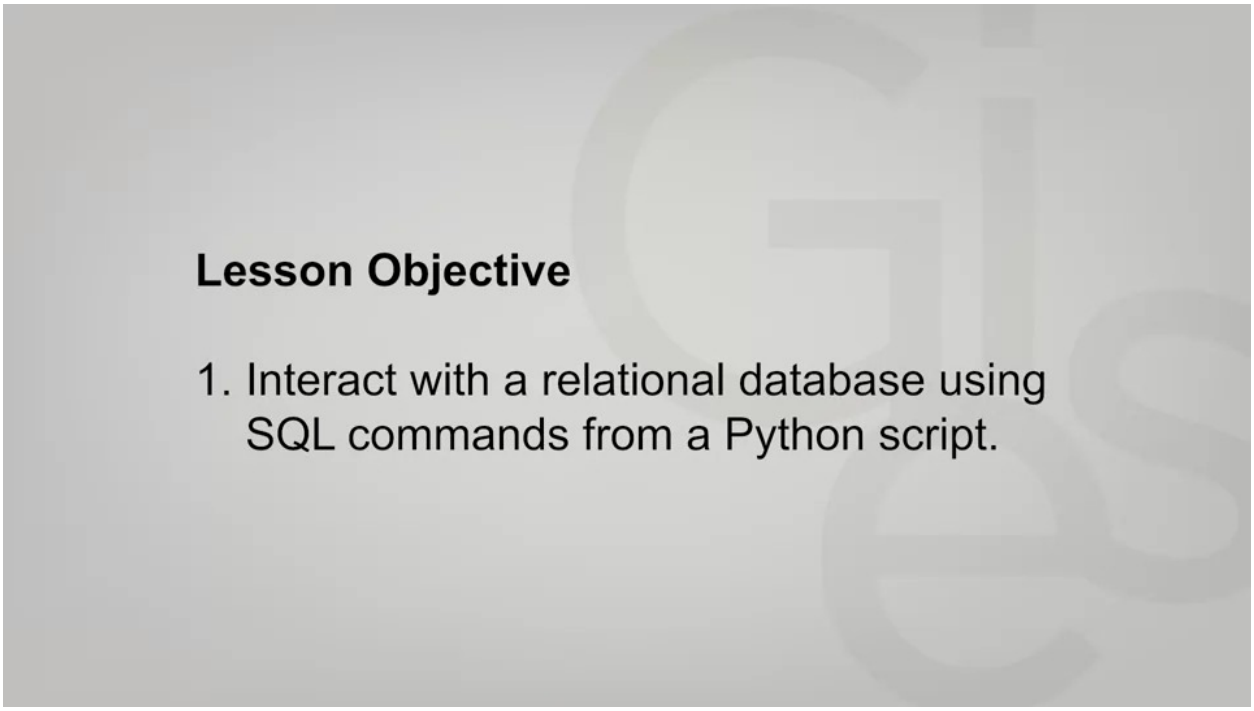
Let's compare that inner join to a left to join, you'll notice that the only thing that I've changed from the original inner join, is that I changed that inner to a left. So now when I run this, you'll see that I have every observation from the cars table, including that row for Raptor, that doesn't have a match in the peeps table. And then the rest of these observations are the same. So, again, a left join means since we're starting with cars here, we're going to include every observation from the cars table, and then only those from the peeps table if they match on the car. So I don't have these observations like Phineas Flynn or Coltrane Green that own a BMW. Now a right join is very similar to a left join because what you put on right and left is arbitrary. So you could really perform a right to join in the following way.

```
sqlite> Select c.name, c.mpg, p.carOwned, p.firstName, p.age
   ...> FROM peeps as p
   ...> LEFT JOIN cars as c on c.name = p.carOwned;
name      mpg   carOwned  firstName  age
--------  ----  --------  ---------  ----
                BMW       Phineas    35.0
Tesla     50.4  Tesla     Ferb       35.0
Corolla   36.4  Corolla   Heinz      60.0
Corolla   36.4  Corolla   Buford     34.0
Tesla     50.4  Tesla     Baljeet    30.0
Corolla   36.4  Corolla   Isabella   36.0
                BMW       Coltrane   42.0
Corolla   36.4  Corolla   Francis    65.0
sqlite>
```

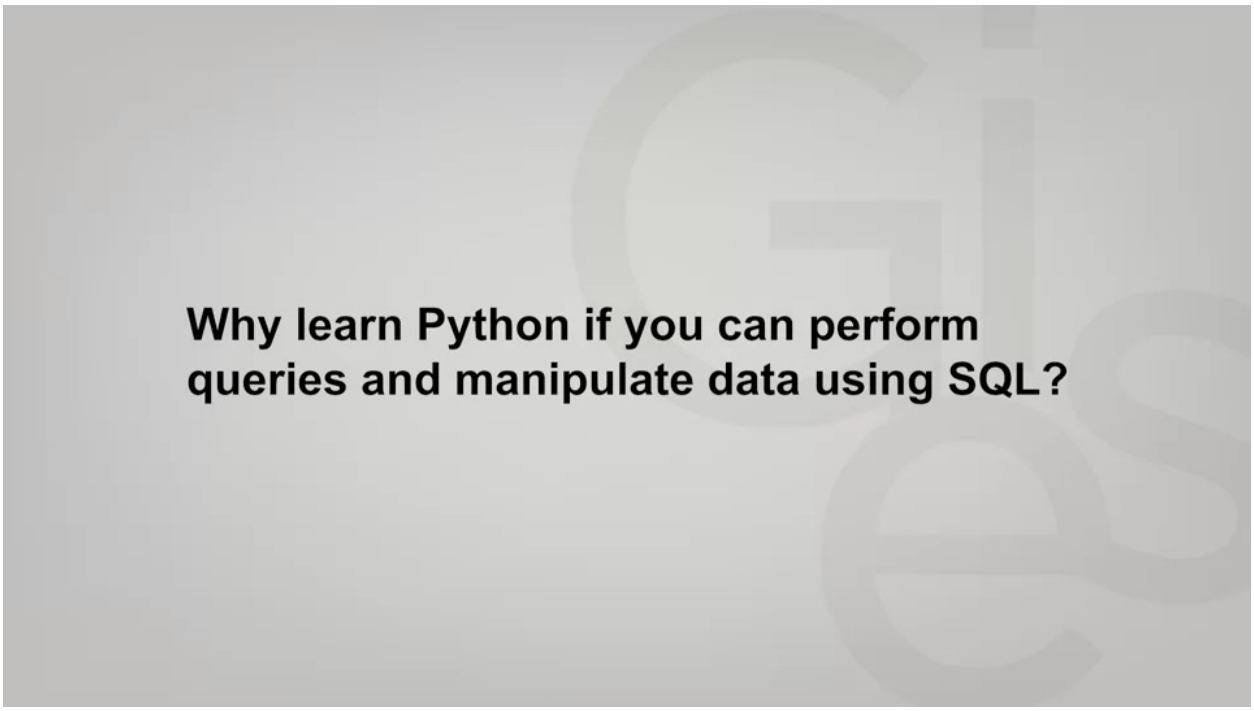
The only thing I'm doing is I'm switching where peeps and cars are. So up before I had cars on the front line, peeps on the left join line, now I have peeps on the front line, cars on the left join line. If I run that you'll see that, now I have BMW for Coltrane and for Phineas. All right, those are the two observations that don't have matches in the cars table, but I don't have the observation from the cars table where the car is Raptor. So anyway, that's a right join but like I said, it just depends on where you put the name of the tables, and outer or full join is not supported by SQLite. You could emulate it with a really complex query but it's really not worth our effort here, in my experience most of the time a left join is what we would use, and then second most would be an inner join and sometimes an outer join. Other versions of SQL allow you to perform outer joins and it would be very similar to a left join in SQLite.

Lesson 8.3 Python Database Programming

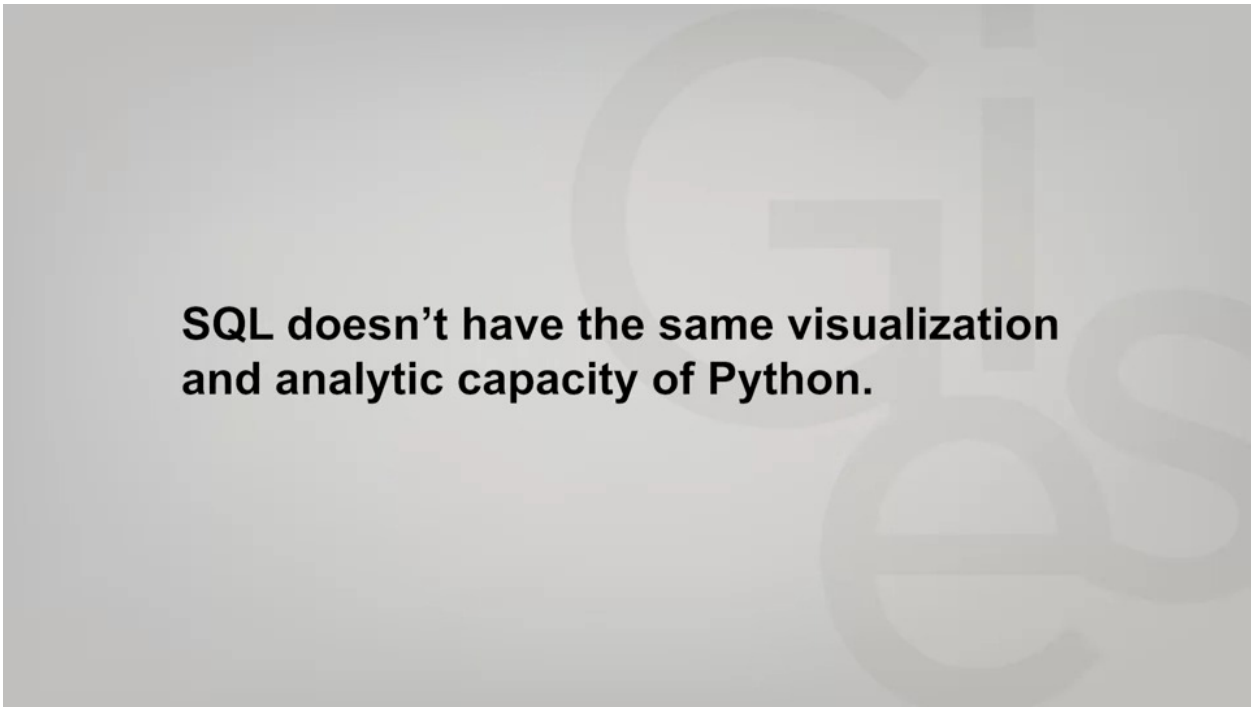
Lesson 8.3 Python Database Programming



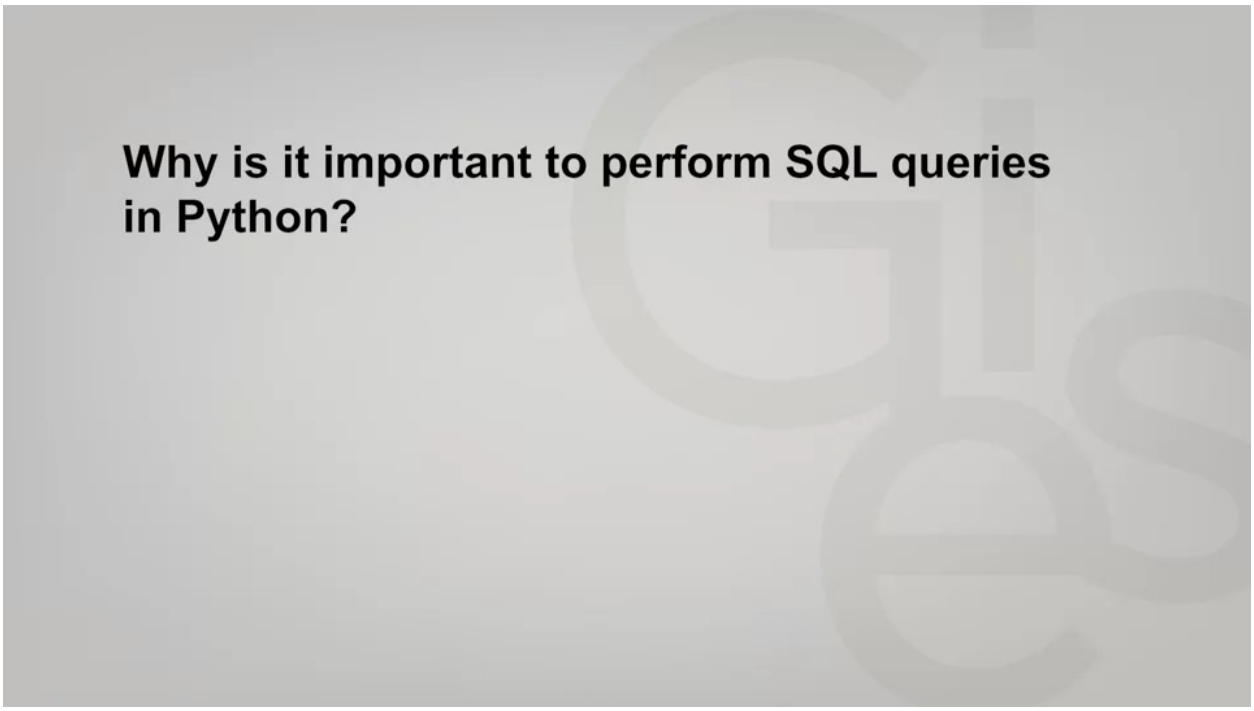
Guymon: Now that you know how to directly interact with SQL databases, it's time that you learn how to interact with SQL databases indirectly through python. This will be really useful for you so that you don't have to have two different windows open. It might be useful to walk through a couple of questions related to the overall workflow.



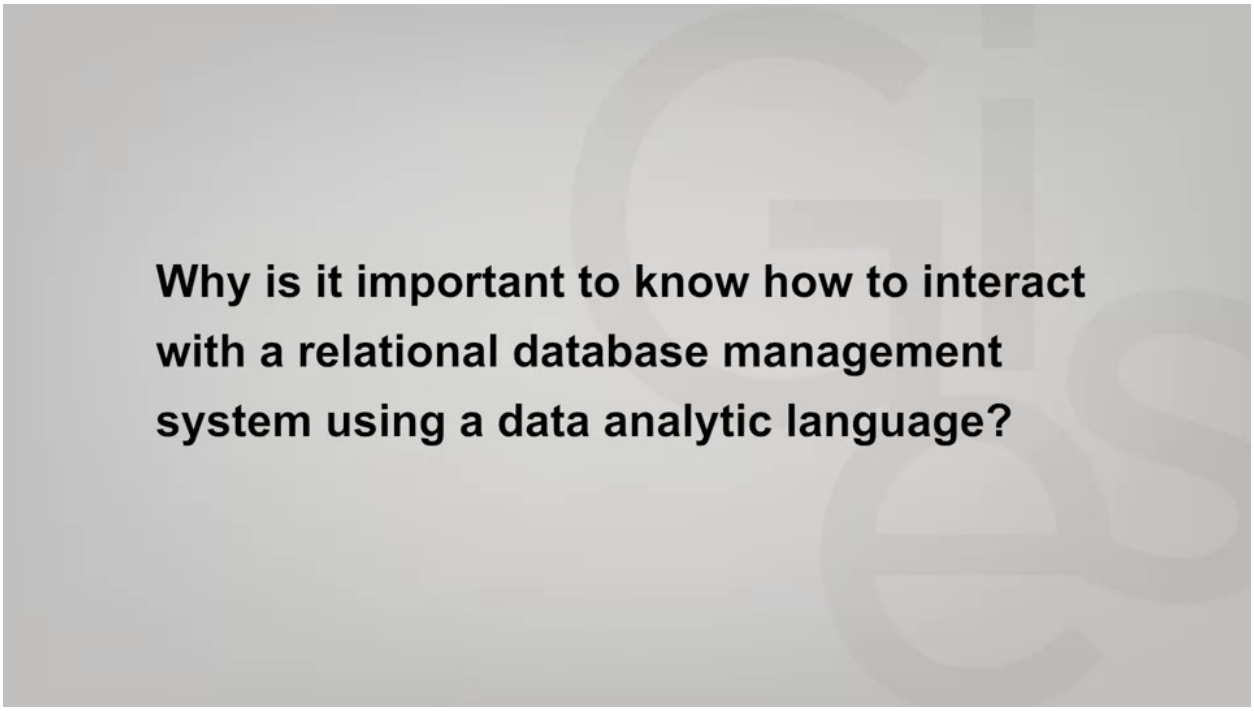
So first, why is it important to Learn Python if most businesses will store their data in a SQL database? I mean couldn't we just learn SQL and do everything with that language?



Well, there's a lot that you can do with SQL but to my knowledge, it doesn't have the same capacity to visualize and analyze data. We need python to perform those visualizations and advanced analytics. We've already established the importance of databases relative to CSV files and spreadsheets in another lesson. So essentially we need both tools.

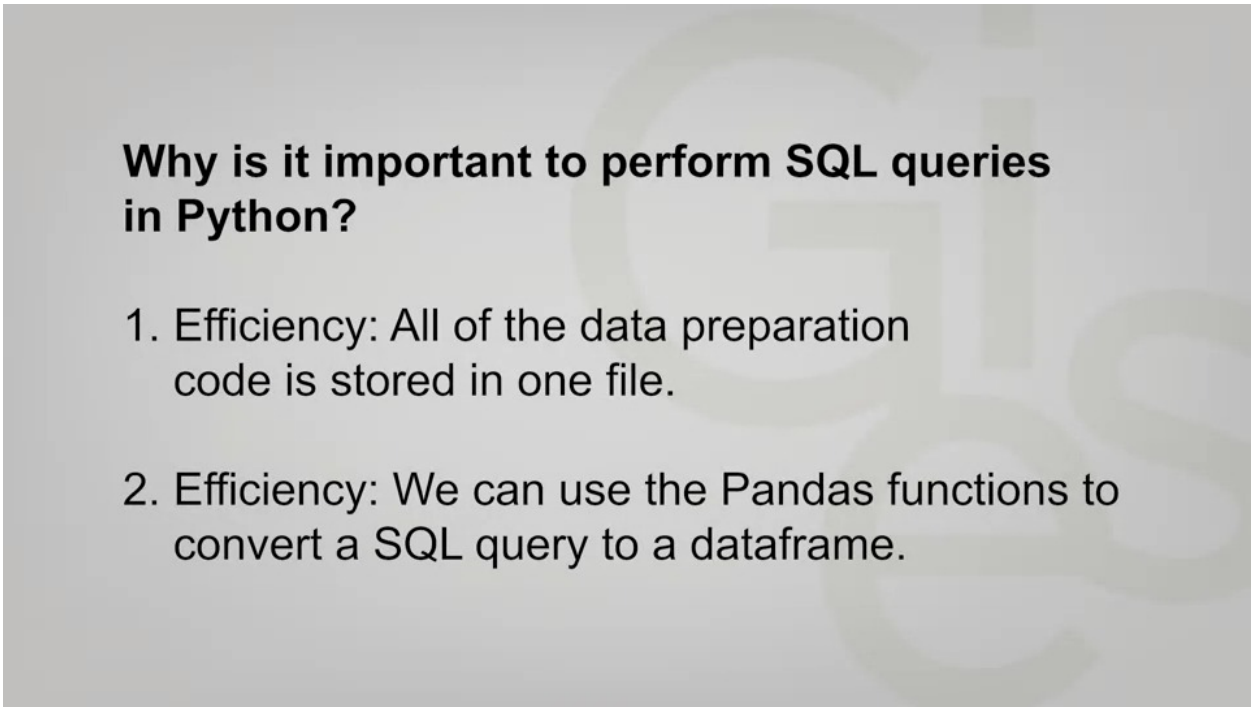


My next question is why do we need to learn how to interact with the relational database using Python? We've been interacting with SQLite and Python from the Jupyter Notebook file and that seems to work fine, right?

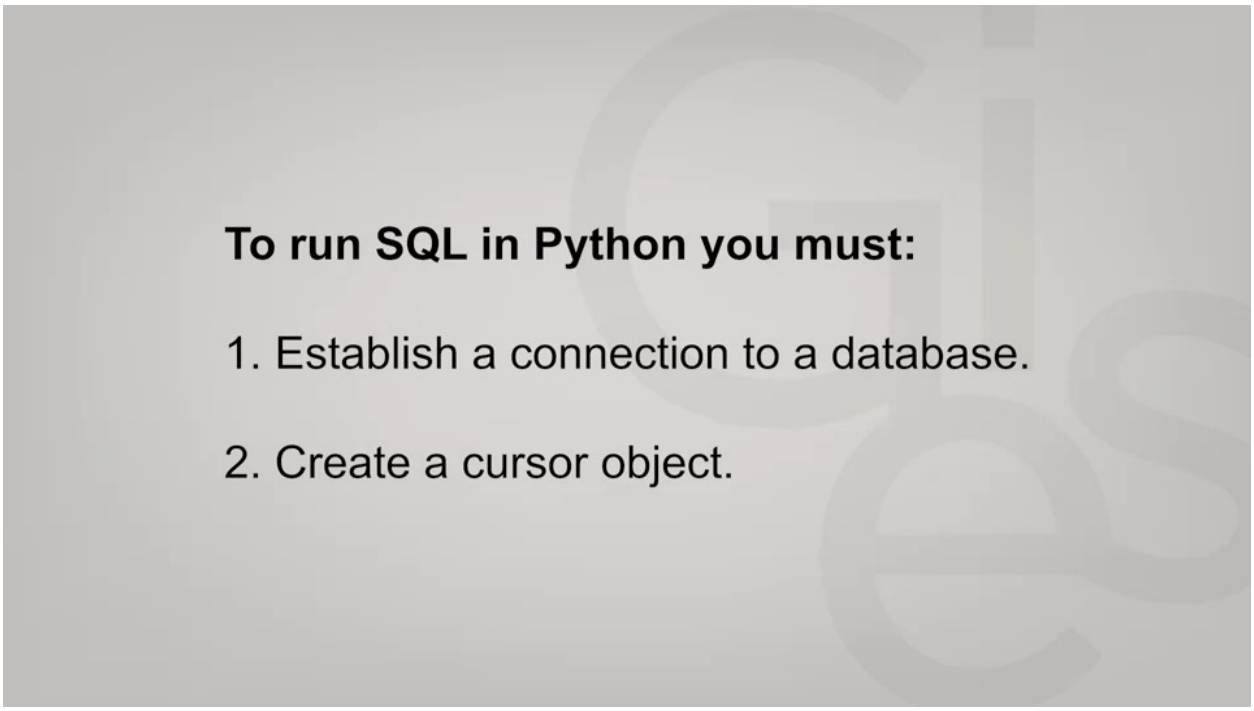


Tyler: I mainly use R to access our database as opposed to using the terminal it allows me to be more clear with my code. And then it also allows me to already have the data because a lot of times I pull the data and it's not just getting the data and then you're done. We're going to do a lot of analysis on the data itself. So I'll use R sometimes python but typically mainly are to pull the

data and then I can use packages in R to then manipulate the data to then summarize the data to then run tables run graphs charts, you name it, on the data even put it into predictive models a lot of times. And so that is a lot easier to do in a platform like R or Python as opposed to just running in the terminal whereas in the terminal you just get a data export and then you'd have to end up uploading that into R or into Excel if you really wanted to into some sort of data Vis software and that wouldn't be as as easy.

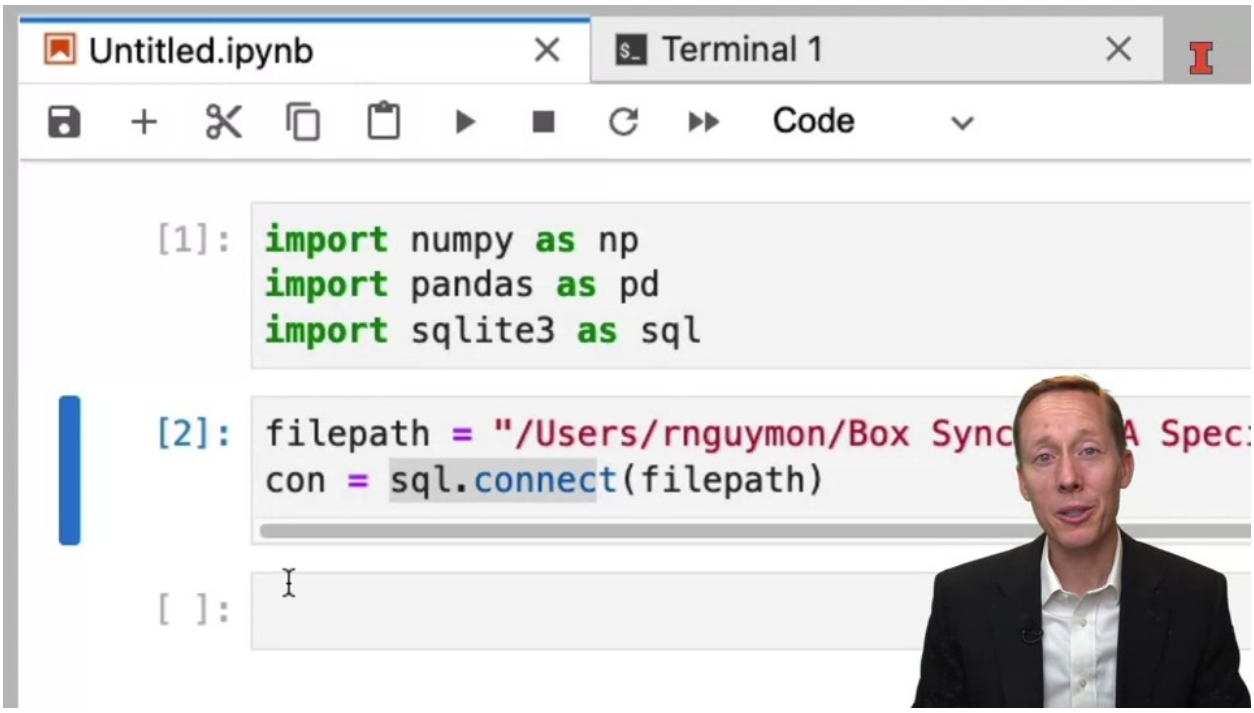


Guymon: One other efficiency of running SQL in Python is that pandas has some built-in functionality to quickly create a data frame from the results of a SQL query. That's actually really handy and may not seem like a big deal when you're first learning about it, but it makes life so much easier when you can contain all the code in a single Python file.



There are a few complexities to running SQL indirectly from python. For instance, you have to establish a connection to a database and then create a cursor object that we will use to execute the SQL commands. These complexities are still much easier to live with compared to writing files and then finding those files and running them. In short learning how to perform SQL commands from python takes just a little bit more effort, but overall you'll be saving a lot of time.

Querying Relational Database with Python and SQL



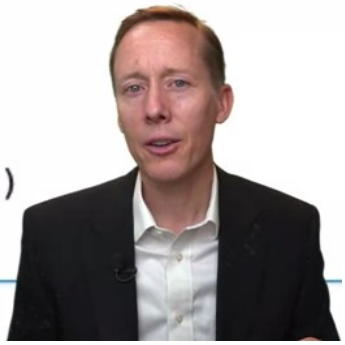
The ability to interact with SQL databases from Python is powerful for at least two reasons. First of all, it allows you to programmatically perform queries and read in data, and secondly, once you bring the data into a Python environment, you now have all the power of Python for performing advanced algorithms and creating visualizations which you can then share with others. In this video, I want to introduce you to a SQL module that you can use to help connect to a database. What I demonstrate in this video, will be with a database that's on my machine. Realistically, you would probably be interacting with a database on a different machine that you connect with through the Internet. Let's go ahead and get started. The database that I'm going to use on my machine is located in the following location. This is the path and the path isn't as important, but just recognize it. It's on my machine and the name of the database is rons_test_db. I'm going to go ahead and copy this path here, and then I will open the Launcher within Jupiter lab and create a new interactive Python Notebook file. From here, I will import a few modules. I'm importing NumPy and Pandas. Then the new module that I will use for the database on my machine is SQLite 3. If you were using a remote database, that was in MySQL database, you would need to import a different module to connect with a MySQL database. But most of the rest of this is pretty similar. Right now, to connect with the database, you need to create a connection object, and here's how you would do that using SQLite 3. We're simply going to use the connect function within the SQLite 3 module, and that's really the main function that we use in this module. But we need to indicate the file path rather than type out that long file path there, I will create a new object that has the full file path. Let me make sure I've named it correctly, rons_test_db. Perfect. Once I have that, I can just take this file path variable and replace it in here, and then run this. You don't see anything happening, but it's nice to know that there's not an error. That means it probably worked. This is the key thing that will separate how you connect to a database that's on your machine versus a remote machine. It may look the same, but if you're using a remote database, you'll have to enter information about the host address, the username, the password, and then the database name. Once you get that connection, the rest is going to be

really easy because of Pandas. Remember that in this database, I've got a couple of tables, and I want to show you how I can query a table right within Python using Pandas and then read in the results of that query and convert it to a DataFrame. It's really simple.

```
[3]: cars = pd.read_sql('Select * from cars;', con)
     cars.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   mpg     3 non-null      float64
 1   name    3 non-null      object
 2   year    3 non-null      int64
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes

[ ]:
```



You can see how simple it is. I basically use the read SQL function from Pandas, then the first argument is the query, and then the next argument is the connection object that I've created, and that's all I have to do. It will perform that query, read in the data, and convert it into DataFrame. Let's go ahead and try it out. Awesome. All right, you can see I've got three entries. I've got three columns here, miles per gallon, name, year. Notice that it took the database types and converted it to a datatype used in Pandas. Instead of a real datatype, it converted it to a float 64 for miles per gallon. Instead of text, it's object for string and instead of integer, it's int 64.

```
[4]: cars
```

[4]:

	mpg	name	year
0	36.4	Corolla	2007
1	20.8	Raptor	2020
2	50.4	Tesla	2020

```
[5]: con.close()
```

```
[ ]:
```

Let's go ahead and just print out what this table looks like to make sure It's what we expect it to be. Fantastic. It looks good and you can see it automatically assigned index values to each of those rows in that table. Now a key thing to remember is that when you connect to a database, you have to close that connection afterwards. It's really easy to do that. We'll just take the connect object and we'll use the close function on it, and that's it. Now it's closed. Now to help us so that we don't forget that, the best practice is to use a context manager. It's very similar to the contexts manager that we used for reading and writing data. In other words, we will say with and then we will just copy this connection code here.

With that SQL connect command, and then we'll give it a variable name as con. You can use any variable name there. Then from here, we can just copy the query that we would perform. We can go ahead and print it out. That makes it easier because once it's finished with all the processes in here, it will automatically close the connection. It works. This is typically the way that we would want to interact with databases again, just to ensure that the connection is closed when we're done. I want to highlight how simple it is to query a database with Python. Really the hard part is just establishing the connection, and then once you do that, reading in the data and converting it to a DataFrame is pretty trivial. Let me show you that you can perform even more advanced queries, and it's not that much more difficult. I'm just going to just go ahead and paste in this new code cell the query that will perform a left join on two tables in this database and save it as a query object. I'm putting a backslash at the end of each line here so that Python knows this code continues on to the next line.

```
[8]: with sql.connect(filepath) as con:
         leftJoin = pd.read_sql(query, con)
     leftJoin
```

[8]:

	name	mpg	carOwned	firstName	age
0	None	NaN	BMW	Phineas	35.0
1	Tesla	50.4	Tesla	Ferb	35.0
2	Corolla	36.4	Corolla	Heinz	60.0
3	Corolla	36.4	Corolla	Buford	34.0
4	Tesla	50.4	Tesla	Baljeet	30.0
5	Corolla	36.4	Corolla	Isabella	36.0
6	None	NaN	BMW	Coltrane	42.0
7	Corolla	36.4	Corolla	Francis	65.0

I'll go ahead and create that query object, and then I'll just copy and paste this code here. I'll create a new DataFrame object. I'll call it leftJoin, and rather than use this query here, I will use this query object that I just created that has the query in it, and then we'll go ahead and print out leftJoin after performing that query. Excellent. This looks just like the query that we performed from the terminal interface. It's a DataFrame object. Boom, it's so easy. As you can see with the help of the Pandas module, as well as the SQLite 3 module or some other module to connect to a database. It's really pretty easy to read data in from a database into a Python data analytic environment and then perform some additional analyses on the data.

Exploring Databases and Adding Rows to Tables with Python

```
import numpy as np
import pandas as pd
import sqlite3 as sql
filepath = "/Users/rnguymon/Box Sync/(iMSA Specialization 4) Data Analy
```

```
with sql.connect(filepath) as con:
    cur = con.cursor()
    cur.execute("SELECT name FROM sqlite_master WHERE type='table';")
    myTables = cur.fetchall()
myTables
```

```
[('cars',), ('peeps',)]
```

In this video, I will demonstrate how you can do a little bit of exploration of a SQL database within Python, as well as how you can modify tables from a SQL database. I'm going to use a database on my machine, and it's located in this particular folder. The name of the database is rons_test_db. To access it from Python, I'm going to use the numpy, pandas, and sqlite3 modules. Then I also have this file path that I am saving as an object. That whole string is just this variable file pass. I'll go ahead and run that cell. Now, I need to establish a connection. I will do that with a context manager. I'm indicating that I'm creating this variable or this object, con, that is a connection to the database. This is the thing that will differ if you're connecting to a remote database. You'll have to enter the database location, the name, and username, and password. Now to explore the database programmatically, we need to create a cursor object. I'll call that cursor object cur, and it's pretty simple to do. We'll take the connection object and use the cursor function on it. This may vary a little bit if you're connecting to a different type of database like a MySQL database. Once we have that, we can perform some commands that will allow us to look at tables in the database. Now, it's not necessarily straightforward how to do this, you may have to look up how to do this or do an Internet search, but let's go ahead and type a command that will allow us to find all the tables of a database. I will use this execute function, and this function allows me to perform a query. This query, again, is not straightforward, but I'm going to select the name of the tables from the database that I'm connected to. Once I do that, I will need to fetch all the results from that query. I'll go ahead and run that. You can see that I have the cars and the peeps table. Now, this may be something that you'll want to jot down or be willing to look up. But anyway, that's how you can find the names of tables in a SQLite database. It would probably be something like that in a MySQL database as well.

```
myTables
```

[2]:
```
[('cars',), ('peeps',)]
```

[3]:
```
# List of items to insert into cars
newCars = [(14.3, 'Transit', 2018)
          , (24.3, 'Q5', 2022)]
```

[4]:
```
with sql.connect(filepath) as con:
    cur = con.cursor()
    cur.executemany('INSERT INTO cars VALUES(?, ?, ?)', newCars)
    cars = pd.read_sql('Select * from cars;', con)
cars
```

[4]:

	mpg	name	year
0	36.4	Corolla	2007
1	20.8	Raptor	2020
2	50.4	Tesla	2020
3	14.3	Transit	2018
4	24.3	Q5	2022

[ ]:

Now, I will demonstrate how you can add more rows of data to an existing table in the database. Specifically, I'm going to add new rows to this cars table, and I will just copy and paste a list of new observations. I have one observation that is for a transit that is year 2018 and gets 14.3 miles per gallon, as well as a 2022 Q5 that gets 24.3 miles per gallon. Now, I will do something very similar to what I did up here. I'll create a context manager and type out the command that I need to use in Python to add those values to the SQL table. Great. I have to add this cursor object in here again that allows me to programmatically go through the tables in the database. Now, I'm going to use this execute many function. If I were inserting these values using the terminal, I would have to use this type of query right here, except that I would have to replace these parentheses with two separate sets of parentheses. What's nice about doing this in Python is that I can indicate that there are three columns here with the question marks. Then I've got this newCars list that I created that I will insert into this. Python will know that that needs to be inserted into this query here. It will need to do this twice because I've got two observations. After that, I'm going to read all the rows from this cars table and return it and make sure that we have these two new observations in it. Let's go ahead and do that. Fantastic, it works. We've got the transit and the Q5 observations in here. That's one way of doing it. That's analogous if you were using the terminal, but it's even simpler if you use pandas to add rows to a database or to create a new table in a database altogether.

2	50.4	Tesla	2020
3	14.3	Transit	2018
4	24.3	Q5	2022

```
[5]: # However, we can do this even more easily with pandas
     newCars = [(39.1, 'Civic', 2022)
               , (12.8, 'Silverado', 1995)]
     newCarsDf = pd.DataFrame(newCars, columns = ['mpg', 'name', 'year'])
     newCarsDf
```

[5]:

	mpg	name	year
0	39.1	Civic	2022
1	12.8	Silverado	1995

```
[ ]:
```

To illustrate how I can add new rows from a pandas DataFrame to an existing table in a SQL database, I need to create some new rows here. I'm going to just add some new rows in this DataFrame. I've created a list of two new observations, a Civic and a Silverado, and then I'm creating a DataFrame object by taking that list and then indicating the names of the three columns from that list. Sometimes, this is easier, at least in my mind, to create DataFrames instead of typing out a whole dictionary because I can refer to each observation rather than each column. Anyway, I've got this new DataFrame object that you see right here that's got the Civic and the Silverado.

Code Python 3 (ipyker

```
[6]: help(newCarsDf.to_sql)
```

```
Help on method to_sql in module pandas.core.generic:

to_sql(name: 'str', con, schema=None, if_exists: 'str' = 'fail', index: 'bool_t' = True, index_label=None, c
ksize=None, dtype: 'DtypeArg | None' = None, method=None) -> 'None' method of pandas.core.frame.DataFrame in
nce
    Write records stored in a DataFrame to a SQL database.

    Databases supported by SQLAlchemy [1]_ are supported. Tables can be
    newly created, appended to, or overwritten.

    Parameters
    ----------
    name : str
        Name of SQL table.
    con : sqlalchemy.engine.(Engine or Connection) or sqlite3.Connection
        Using SQLAlchemy makes it possible to use any DB supported by that
        library. Legacy support is provided for sqlite3.Connection objects. The user
        is responsible for engine disposal and connection closure for the SQLAlchemy
        connectable See `here                <https://docs.sqlalchemy.org/en/13

    schema : str, optional
        Specify the schema (if database flavor supports this). If None, use
        default schema.
    if_exists : {'fail', 'replace', 'append'}, default 'fail'
```

From here, I will need to connect to the database, and I'll do that with the context manager. Now, I need to use a function that is the to_sql function. This is actually a method associated with pandas DataFrames. Before I use this function, let's go ahead and read the help documentation on it. The way this function works is I need to identify the name of the SQL table, that's the first argument. The second argument is the connection, and that will be easy. The third argument is a schema if we're creating this brand new. The fourth argument is the one that I really want to focus on here, and that is the if_exists. By default, it will be fail, meaning if there's a table that already exists, it won't do anything with this. But if we read a little bit more about this, there are a couple of other options that we can use. We can replace the table if it exists, or we can append new rows to the table, and that's what we will use here.

[8]:
```
with sql.connect(filepath) as con:
    newCarsDf.to_sql('cars'
                     , con
                     , if_exists = 'append'
                     , index = False)
    cars = pd.read_sql('Select * from cars;', con)
cars
```

[8]:

	mpg	name	year
0	36.4	Corolla	2007
1	20.8	Raptor	2020
2	50.4	Tesla	2020
3	14.3	Transit	2018
4	24.3	Q5	2022
5	39.1	Civic	2022
6	12.8	Silverado	1995

With that in mind, let's go ahead and finish out this query. We want to connect to the cars table. We'll use our connection, and if_exists, we'll set equal to append. Now, while we are connected to this database, let's go ahead and read in this new cars and see if those lines were added to it. Now, let's go ahead and run this. Looks like I have an error. It says, "Table cars has no column named index." This is a good reminder. By default, when you add values from a pandas DataFrame, it will take the index and try to create a new column in the SQL table. There's another argument, index, and by default, it's true. That means it will use that index column, so we need to indicate that it's false. Let's try it again. Looks like it worked this time. You can see we've got the Civic and Silverado added on to the existing observations in this cars table in SQL. From here, it's pretty trivial to show you if you want to just create a new table in a database using a pandas DataFrame.

```
[9]: with sql.connect(filepath) as con:
         newCarsDf.to_sql('cars'
                          , con
                          , if_exists = 'replace'
                          , index = False)
         cars = pd.read_sql('Select * from cars;', con)
cars
```

[9]:

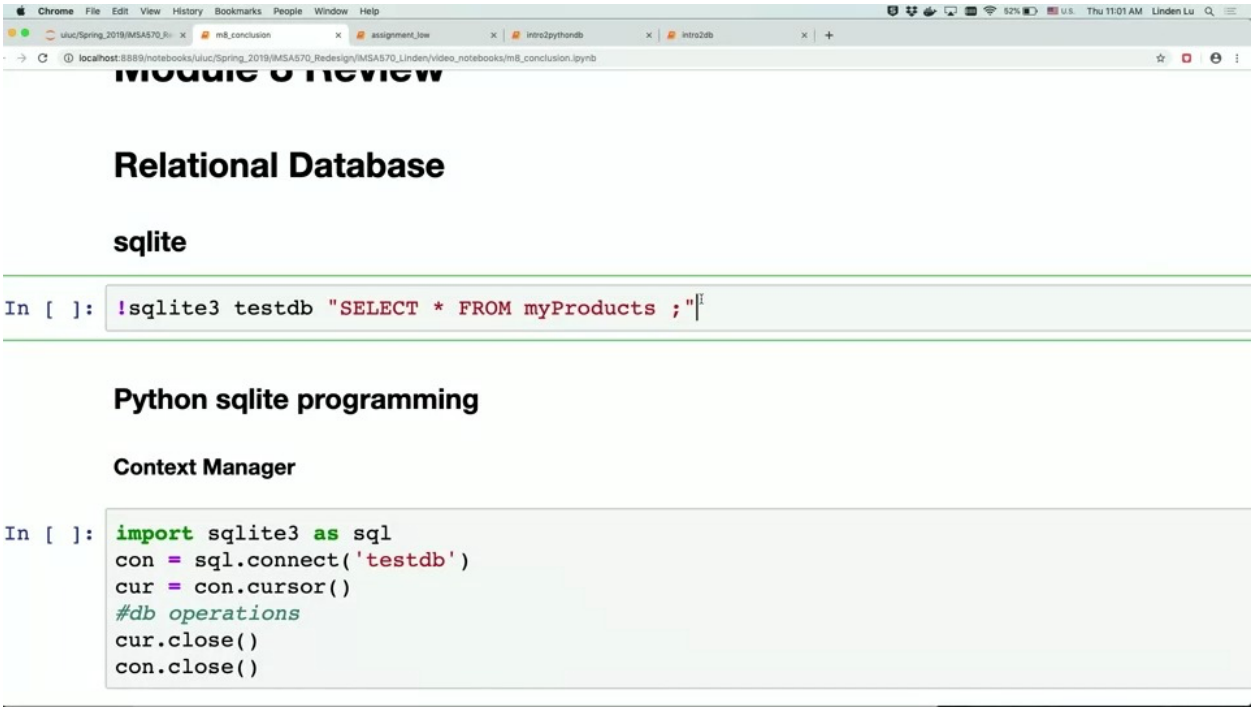
	mpg	name	year
0	39.1	Civic	2022
1	12.8	Silverado	1995

```
[ ]:
```

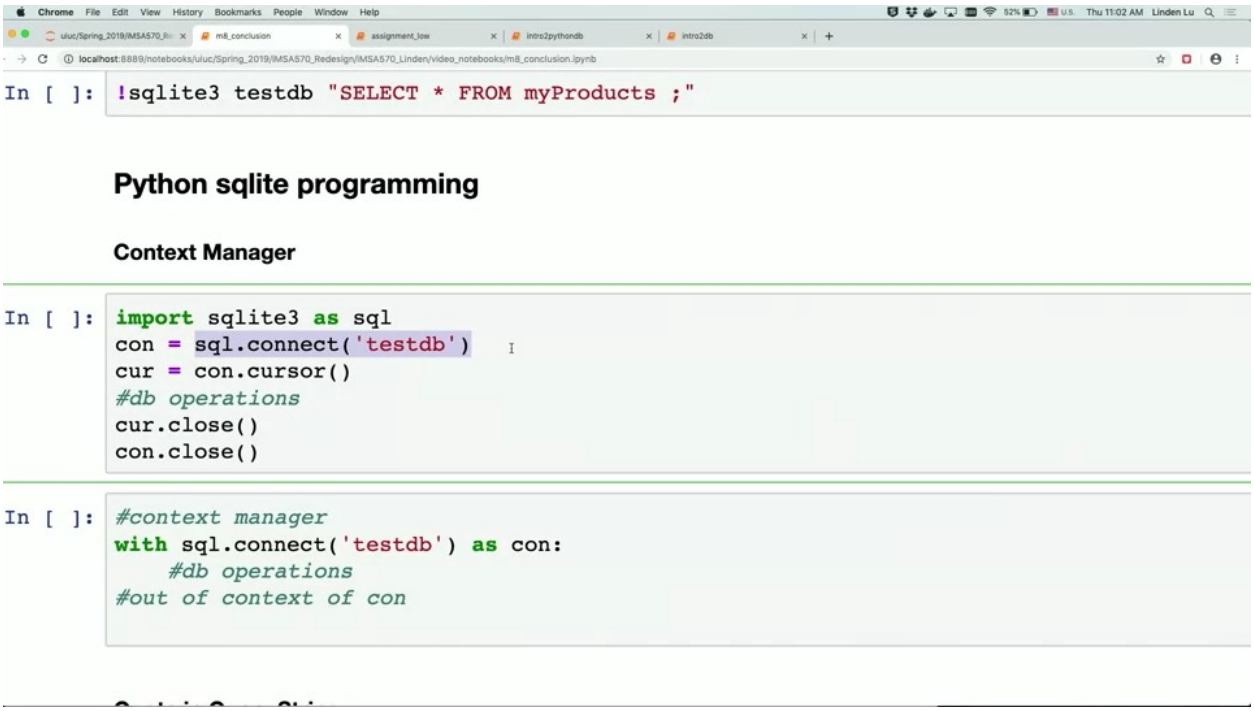
You just need to use this if_exists and change it to replace. If you did that and ran this code again, then it would replace the DataFrame with just those two rows of data. Let's go ahead and try it out. That cars table and the database now only has these two observations in it. This is obviously just an introduction to connecting to databases and interacting with them using Python, but it should be a solid foundation and enough for you to get started doing a lot of work with databases and Python together.

Module 8 Conclusion

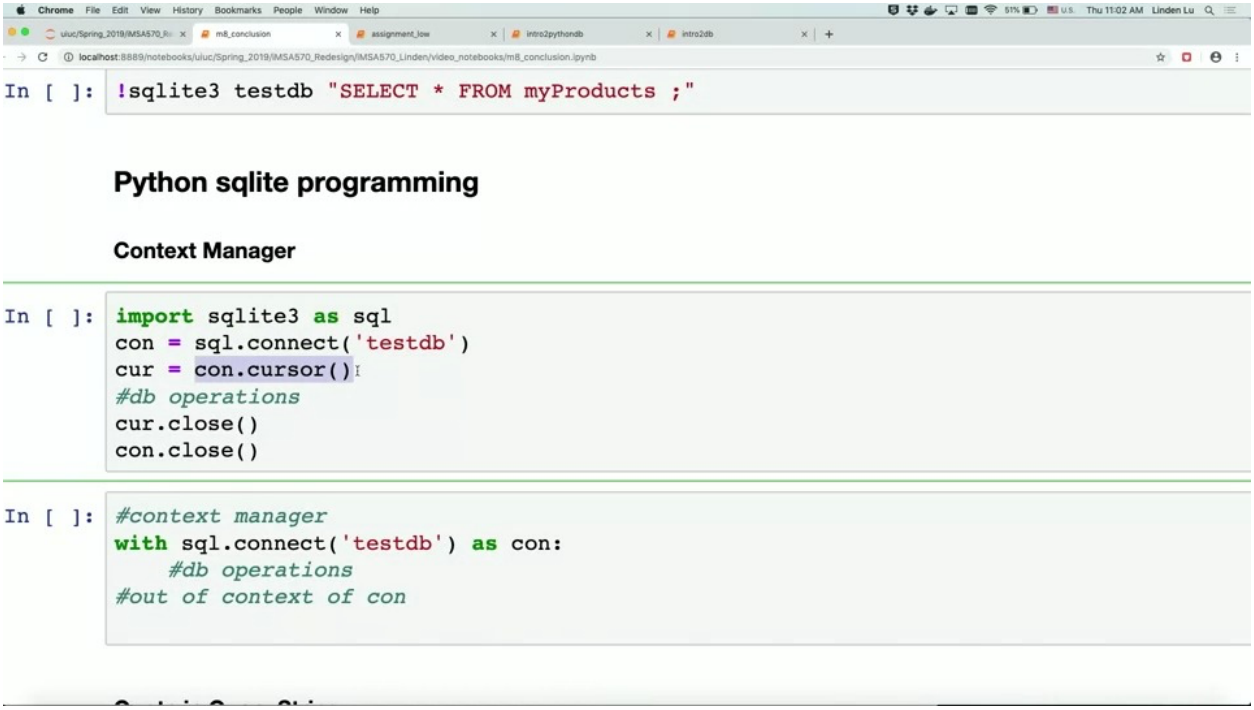
Module 8 Review



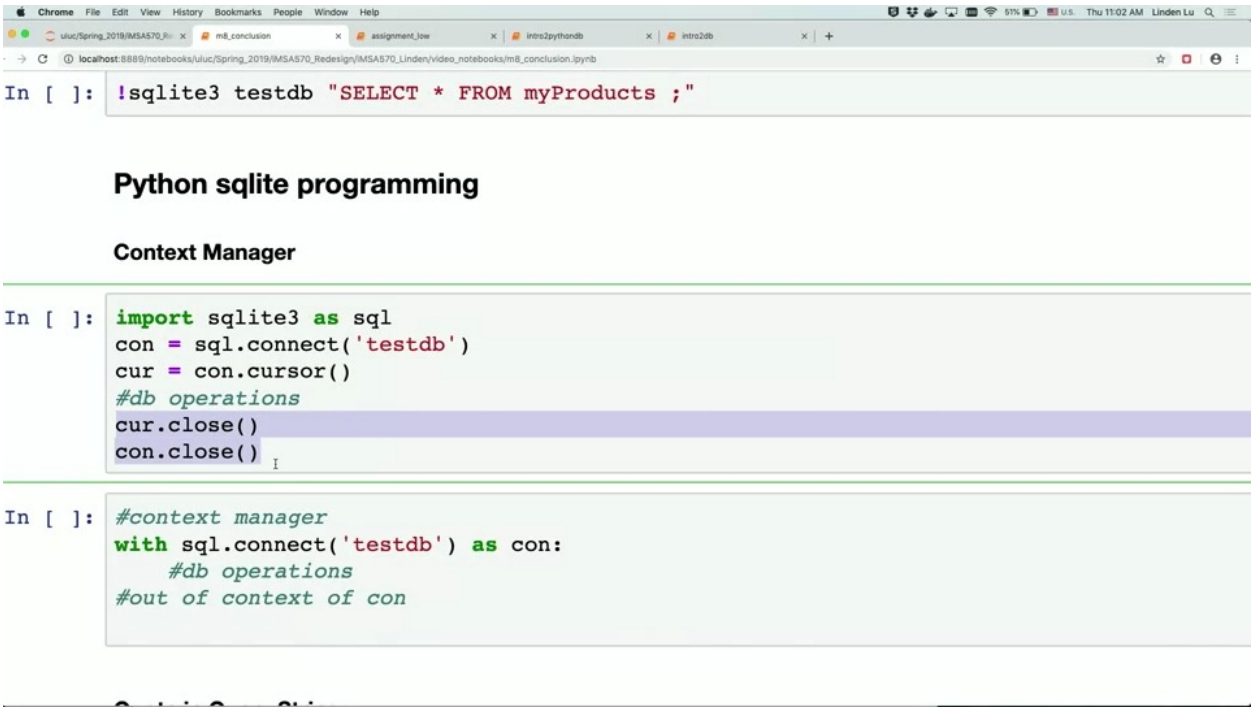
Lu: In this module, we introduced relational database and python database programming. We used SQLite, which is a file based relational database. SQLite is lightweight and built in with Python, which makes it a perfect database to demonstrate Python database programming. The way to work with other relational databases like SQL server, MySQL, and Oracle in Python is very similar to the way we work with SQLite. In lesson two, we demonstrated how to interact with SQLite directly through the SQLite3 command. We can run SQLite3 command with exclamation marks in a notebook like this. But this has nothing to do with Python. It is equivalent to running SQLite3 command without exclamation marks in a terminal.



To connect to a SQLite database in Python, we will need to call connect function in SQLite module.



Once we get the database connection we can get a cursor from the connection and start interact with our database.



```
In [ ]: !sqlite3 testdb "SELECT * FROM myProducts ;"
```

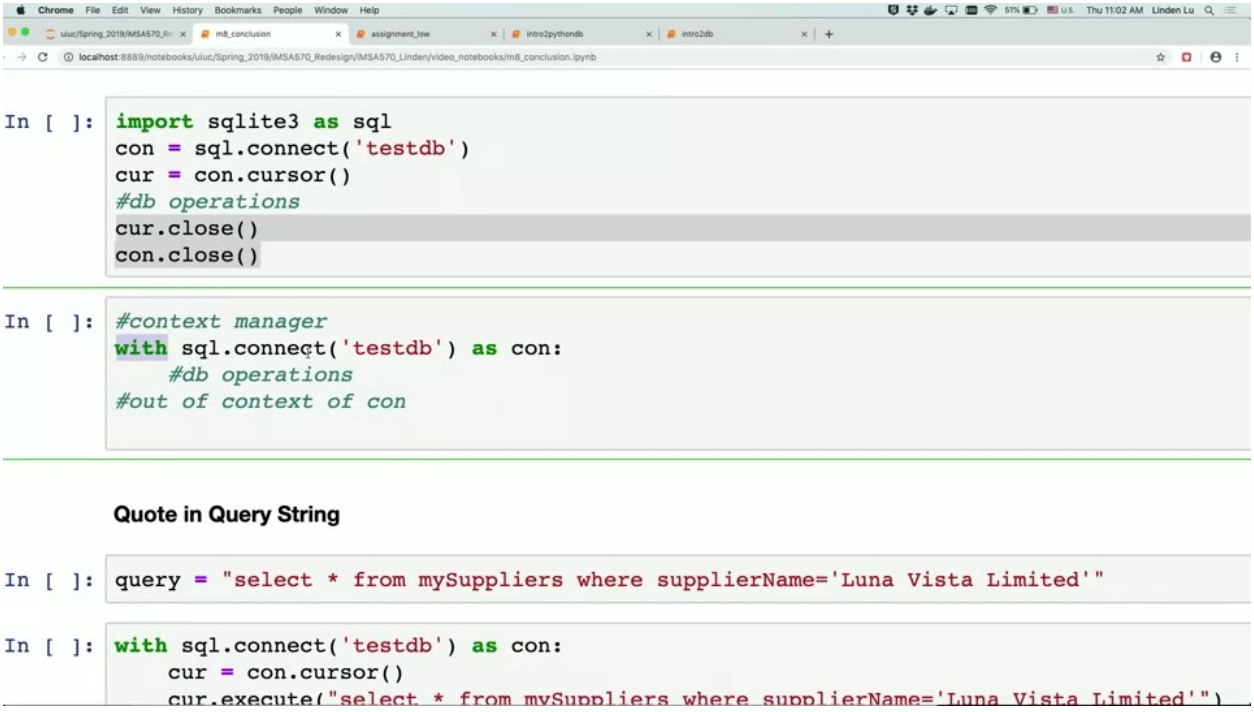
Python sqlite programming

Context Manager

```
In [ ]: import sqlite3 as sql
        con = sql.connect('testdb')
        cur = con.cursor()
        #db operations
        cur.close()
        con.close()
```

```
In [ ]: #context manager
        with sql.connect('testdb') as con:
            #db operations
        #out of context of con
```

To ensure the integrity of the database we will need to close the cursor and connection when all database options are done.



```
In [ ]: import sqlite3 as sql
        con = sql.connect('testdb')
        cur = con.cursor()
        #db operations
        cur.close()
        con.close()
```

```
In [ ]: #context manager
        with sql.connect('testdb') as con:
            #db operations
        #out of context of con
```

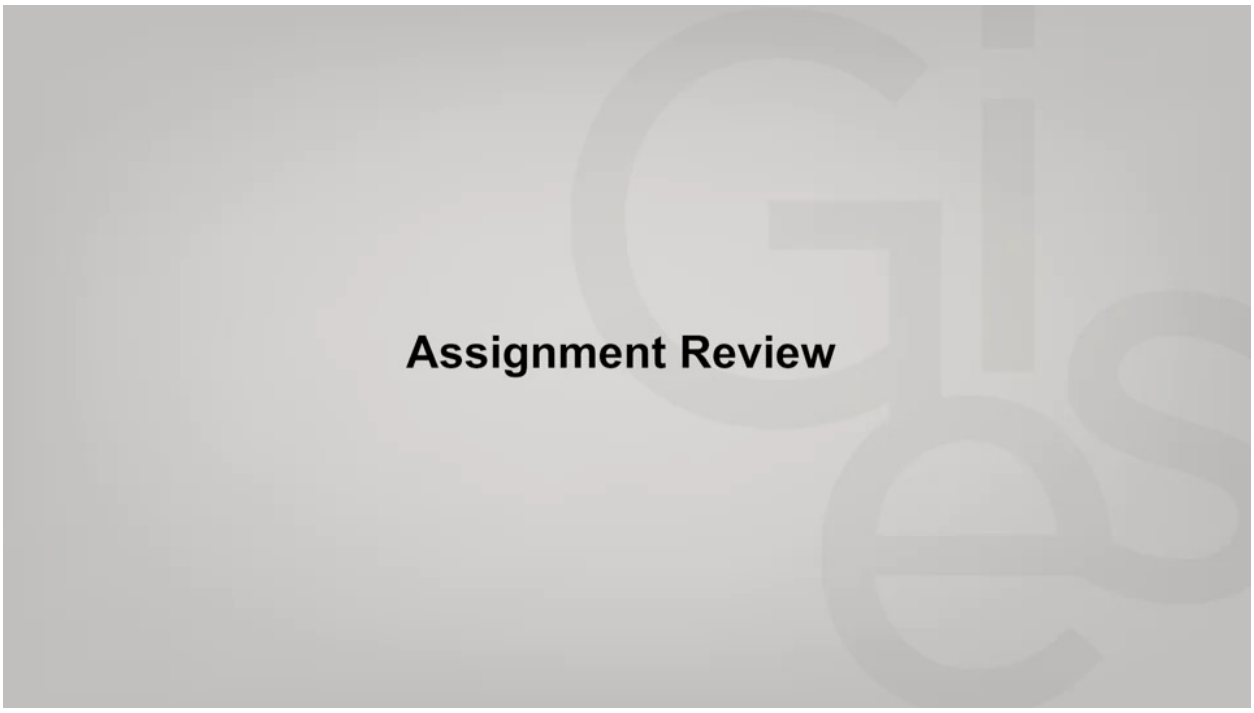
Quote in Query String

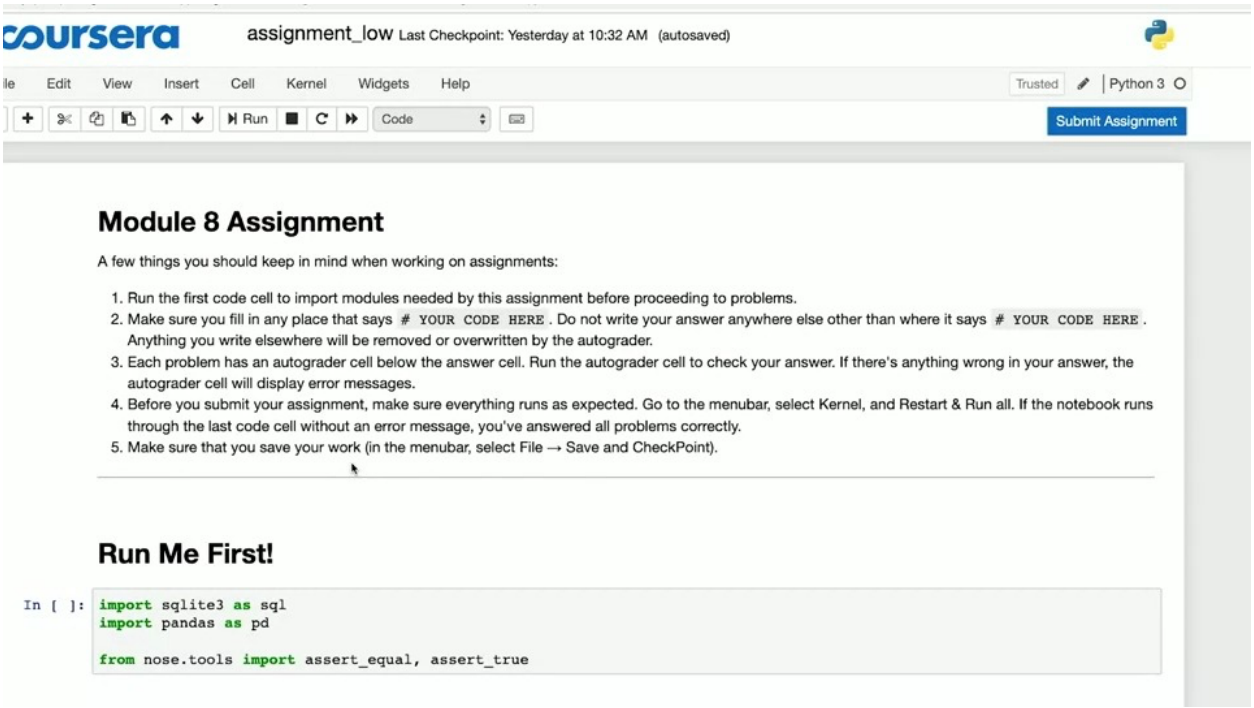
```
In [ ]: query = "select * from mySuppliers where supplierName='Luna Vista Limited'"
```

```
In [ ]: with sql.connect('testdb') as con:
            cur = con.cursor()
            cur.execute("select * from mySuppliers where supplierName='Luna Vista Limited'")
```

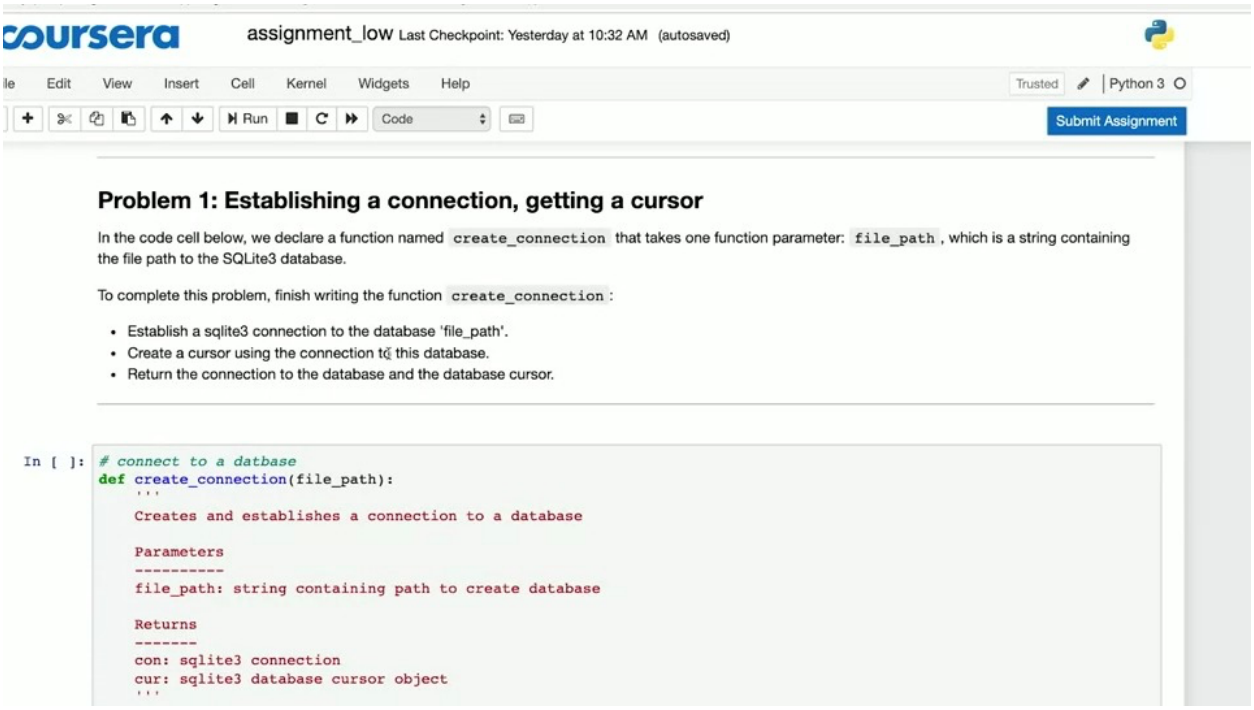
The context manager that uses a with clause makes things simpler. With the context manager, we don't need to worry about closing the connection and the cursor. So the context manager starts with with, then SQL connect test DB which is a database name as con. All indented code under the with clause is considered in the context of connection con. When the first line out of the

context is reached all transactions related to the database connection are probably committed, or wrote back, if there's any problem, and all connections and cursor resources are released.

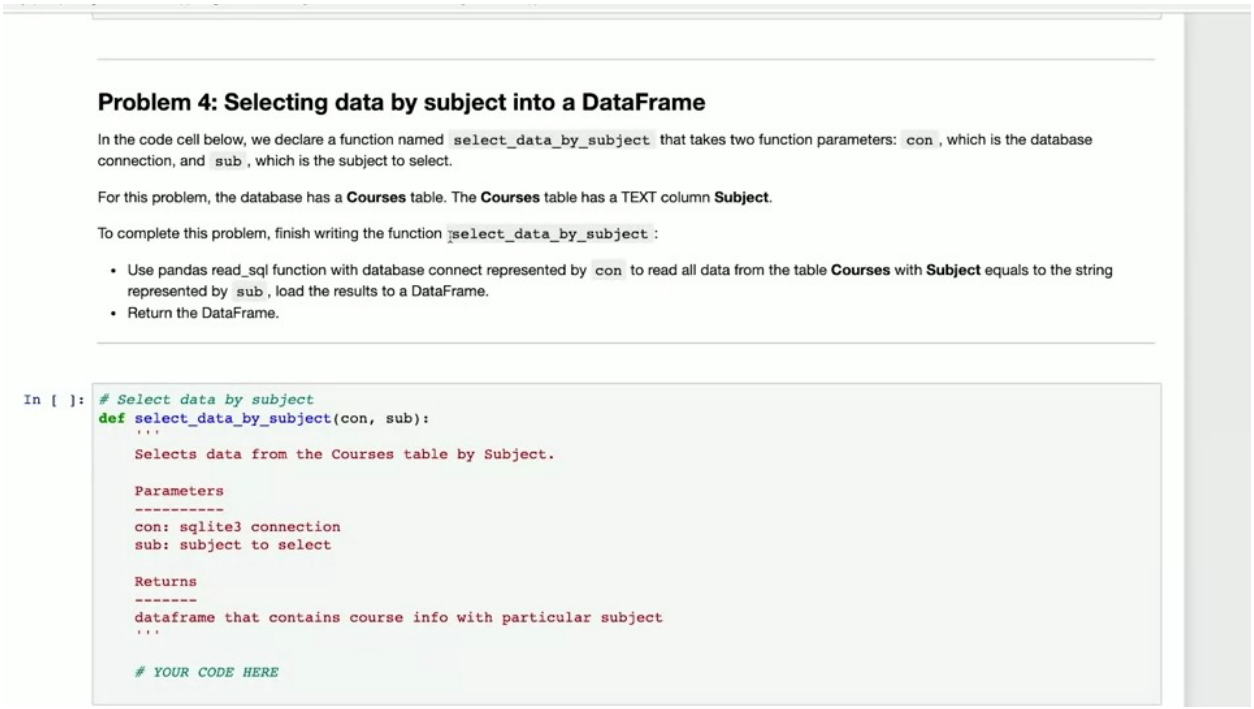




Now I'd like to talk about module eight assignment.



Problem one asks you to establish a database connection and get a cursor. For this problem, please click the database connection directly. Don't use the context manager since you need to return the connection and the cursor from the function. Don't worry about the resource leak. The connection and the cursor will be released at the end of assignment. Problem 2 and problem 3 are very straightforward.



Problem 4 asks you to select data by a particular subject. The subject is a string. It has to be quoted if used in a SQL query. The best way to construct a query with string value is to use the F string.

```
In [5]: # Tuple containing data values to insert into our database
        items = ((10,19.95,104,'2018-03-31','Hooded sweatshirt'),
                 (11,99.99,104, '2018-03-29','Beach umbrella'),
                 (12,0.99,104,'2018-02-28', None))

        # Open a database connection, here we use an in memory DB

        with sql.connect("testdb") as con:

            # Now we obtain our cursor
            cur = con.cursor()

            # Insert some values to myProducts
            cur.executemany("INSERT INTO myProducts VALUES(?, ?, ?, ?, ?)", items)

            print("-"*50)
            print("All products:")
            print("-"*50)
            # Select rows and iterate through them
            for row in cur.execute('SELECT * FROM myProducts'):
                print(f"| {row[0]} | {row[1]} | {row[2]} | {row[3]} | {row[4]} |")

            print("-"*50)
            print("Beach umbrellas:")
            print("-"*50)
            # Select by description
            desc = 'Beach umbrella'
            for row in cur.execute(f'SELECT * FROM myProducts WHERE description="{desc}"'):
                print(f"| {row[0]} | {row[1]} | {row[2]} | {row[3]} | {row[4]} |")

            # Delete some rows
            cur.execute("DELETE FROM myProducts WHERE itemNumber >= 10")

        --------------------------------------------------
        All products:
        --------------------------------------------------
```

In the left three notebook you can find an example that uses F string to construct a SQL query with a stream variable. SQL programming is very strict. An extra comma, or a missing quote, may well prevent your career from working. Please follow the instructions in assignment closely. If you still have trouble passing an assignment problem, come back to this video and you will probably find and the reason. Good luck.