University of Washington

iSchool Info 330

# Module 09 – Multi-Tier Applications

In this module, we will look at how applications are divided into multiple layers (Presentation, Processing, and Data) to work with databases**.**

## Outline

Here is a general outline of what we will be doing this module:

|  |
| --- |
| **Module09: Multi-Tier Applications** |
| Session01 Lectures and Labs < 110 mins |
| Multi-Tier Applications |
| Lab 1: Working with Modules  SQLite |
| Lab 2: Installing and Testing SQLite |
| SQLite Databases |
| Lab 3: Creating a SQLite Database |
| The SQLite Studio GUI Editor |
| SQLite Tables |
| Lab 4: Using SQLite Studio |
| PySQLite |
| Session02 - Lab |
| Final Project - Milestone03 |
| Session03 Lectures and Labs < 110 mins |
| Connecting to SQLite with PySQLite |
| Executing SQL Code from Python |
| Lab 5: Working with PySQLite |
| Creating Multiple Tiers using Modules |
| Lab 6: Creating a Multi-Tiered Application |

**Note**: Times are only an estimate and may change without notice!

# Session01 < 110 mins

In this session, we start exploring **how applications are created using multiple components (or Tiers)**. To do this, we will create an **example using a popular set of database and application development software called Python and SQLite**. This software **runs on Mac, PC, Linux, and even Phones**! Plus, it is relatively easy to work with, especially with your experience in this course!

## Multi-Tier Applications

The **most multi-tiered applications include 3-tiers**. A three-tiered application divides an application into **three layers**. The first layer is an application's presentation, the second is its processing, and the third is its data. So, you need to think about what code represents these layers when you program.

You create an application with a set of classes or functions to support data, processing, or presentation tasks. This division of tasks is called the "Separation of Concerns."

“**Three-tier architecture:**

**Presentation tier**

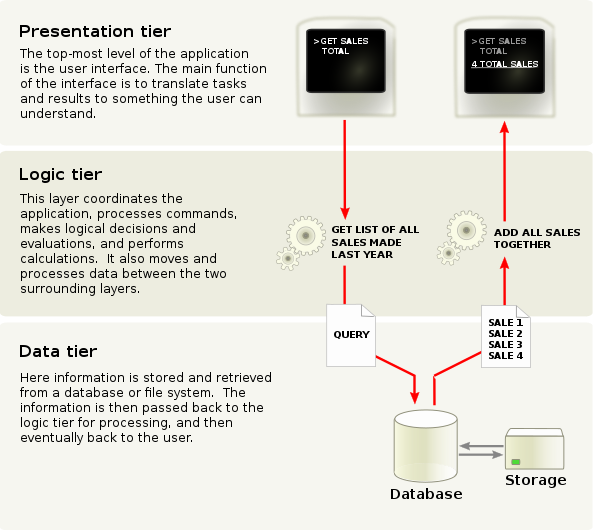
This is the topmost level of the application. The presentation tier displays information related to such services as browsing merchandise, purchasing and shopping cart contents. It communicates with other tiers by which it puts out the results to the browser/client tier and all other tiers in the network. In simple terms, it **is a layer which users can access directly (such as a web page, or an operating system's GUI**).

**Application tier** (business logic, logic tier, or middle tier) **(Edit: Processing Layer)**

The logical tier is pulled out from the presentation tier and, as its own layer, it controls an application’s functionality by **performing detailed processing**.

**Data tier**

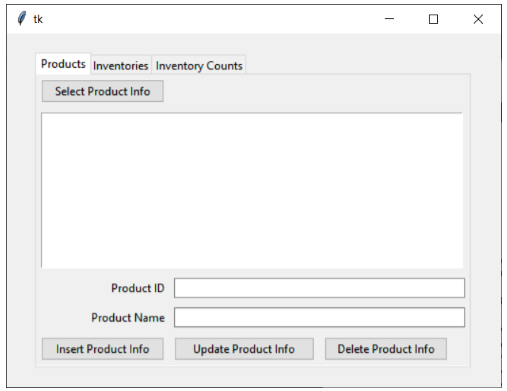
The data tier includes the **data persistence mechanisms** (**database** servers, **file** shares, etc.) and the data access layer that encapsulates the persistence mechanisms and exposes the data...” (<https://en.wikipedia.org/wiki/Multitier_architecture>, 2017)



Source: <https://en.wikipedia.org/wiki/File:Overview_of_a_three-tier_application_vectorVersion.svg>

### Creating a UI Layer (Presentation)

The User Interface (UI) layer **allows humans to use the application**. Sometimes the UI is a simple terminal application, but more often, it is a windowed or web application.



### Creating a Logic Processing Layer

The processing layer is where code processes data to and from the user to and from the data storage. **Processing code does not include ways for the user to interact with it directly**. Instead, all data submitted by the user is passed to the processing code from the presentation layer.

Here is an example of presentation and processing layers working together.

# ---------------------------------------------------------- #  
# Title: Multi-layer Windowed app example  
# Description: Demonstrate using multi-line textboxes  
# ChangeLog (Who,When,What):  
# RRoot,1.1.2030,Created started script  
# ---------------------------------------------------------- #  
**import** tkinter **as** tk  
**from** tkinter **import** ttk  
  
**class** MathProcessor(object):  
 @staticmethod  
 **def** add(n1, n2):  
 **return** n1 + n2  
  
 @staticmethod  
 **def** subtract(n1, n2):  
 **return** n1 - n2  
  
 @staticmethod  
 **def** multiply(n1, n2):  
 **return** n1 \* n2  
  
 @staticmethod  
 **def** divide(n1, n2):  
 **return** n1 / n2  
  
  
**class** MathIO(object):  
  
 @staticmethod  
 **def** clear\_textbox(textbox):  
 textbox[**'state'**] = **'normal'** textbox.delete(1.0, tk.END)  
 textbox[**'state'**] = **'disabled'** @staticmethod  
 **def** write\_sum\_to\_textbox(n1, n2, textbox):  
 textbox[**'state'**] = **'normal'** text = str.format(**'The Sum of {n1} and {n2} is {result}\n'**,  
 n1=n1, n2=n2, result=MathProcessor.add(n1, n2))  
 textbox.insert(tk.END, text)  
 textbox[**'state'**] = **'disabled'** @staticmethod  
 **def** write\_difference\_to\_textbox(n1, n2, textbox):  
 textbox[**'state'**] = **'normal'** text = str.format(**'The difference of {n1} and {n2} is {result}\n'**,  
 n1=n1, n2=n2, result=MathProcessor.subtract(n1, n2))  
 textbox.insert(tk.END, text)  
 textbox[**'state'**] = **'disabled'** @staticmethod  
 **def** write\_product\_to\_textbox(n1, n2, textbox):  
 textbox[**'state'**] = **'normal'** text = str.format(**'The product of {n1} and {n2} is {result}\n'**,  
 n1=n1, n2=n2, result=MathProcessor.multiply(n1, n2))  
 textbox.insert(tk.END, text)  
 textbox[**'state'**] = **'disabled'** @staticmethod  
 **def** write\_quotient\_to\_textbox(n1, n2, textbox):  
 textbox[**'state'**] = **'normal'** text = str.format(**'The quotient of {n1} and {n2} is {result}\n'**,  
 n1=n1, n2=n2, result=MathProcessor.divide(n1, n2))  
 textbox.insert(tk.END, text)  
 textbox[**'state'**] = **'disabled'**# End class  
  
**class** MainWindow(object):  
 """ Description: Creates the following UI objects:  
 window\_root (tk.TK)  
 lbl\_math\_info (ttk.label)  
 txt\_first\_number (ttk.entry)  
 txt\_second\_number (ttk.entry)  
 mtx\_results (ttk.textbox)  
 btn\_add (ttk.button)  
 btn\_subtract (ttk.button)  
 btn\_multiply (ttk.button)  
 btn\_divide (ttk.button)  
 """  
 @staticmethod  
 **def** create\_main\_window():  
 """ Create and configure a root node Window object"""  
 application\_window = tk.Tk()  
 application\_window.geometry(**"425x250"**)  
 application\_window.title(**"Simple Math"**)  
  
 lbl\_math\_results = ttk.Label(application\_window, text=**"Math Results"**)  
 lbl\_math\_results.grid(row=1, column=1, sticky=tk.NW, padx=10, pady=5)  
  
 lbl\_first\_number = ttk.Label(  
 application\_window,  
 text=**"First Number "**,  
 width=20,  
 anchor=tk.E  
 )  
 lbl\_first\_number.grid(row=2, column=1, sticky=tk.E)  
 txt\_first\_number = ttk.Entry(application\_window, width=40)  
 txt\_first\_number.grid(row=2, column=2, columnspan=3)  
 txt\_first\_number.insert(0, **"0.00"**)  
  
 lbl\_second\_number = ttk.Label(  
 application\_window,  
 text=**"Second Number "**,  
 width=20,  
 anchor=tk.E  
 )  
 lbl\_second\_number.grid(row=3, column=1, sticky=tk.E)  
 txt\_second\_number = ttk.Entry(application\_window, width=40)  
 txt\_second\_number.grid(row=3, column=2, columnspan=3)  
 txt\_second\_number.insert(0, **"0.00"**)  
  
 # Adding a multi-line textbox  
 mtx\_results = tk.Text(width=50, height=5)  
 mtx\_results.grid(row=4,  
 column=1,  
 sticky=tk.N,  
 columnspan=4,  
 padx = 10,  
 pady =10  
 )  
  
 btn\_add = ttk.Button(application\_window, text=**"Add"**, width=10)  
 btn\_add.grid(row=5, column=1, sticky=tk.E, padx=15, pady=5)  
 btn\_add[**'command'**] = **lambda**: MathIO.write\_sum\_to\_textbox(  
 float(txt\_first\_number.get()),  
 float(txt\_second\_number.get()),  
 mtx\_results)  
  
 btn\_subtract = ttk.Button(application\_window, text=**"Subtract"**, width=10)  
 btn\_subtract.grid(row=5, column=2, sticky=tk.W, padx=5, pady=5)  
 btn\_subtract[**'command'**] = **lambda**: MathIO.write\_difference\_to\_textbox(  
 float(txt\_first\_number.get()),  
 float(txt\_second\_number.get()),  
 mtx\_results)  
  
  
 btn\_multiply = ttk.Button(application\_window, text=**"Multiply"**, width=10)  
 btn\_multiply.grid(row=5, column=3, sticky=tk.W, padx=5, pady=5)  
 btn\_multiply[**'command'**] = **lambda**: MathIO.write\_product\_to\_textbox(  
 float(txt\_first\_number.get()),  
 float(txt\_second\_number.get()),  
 mtx\_results)  
  
  
 btn\_divide = ttk.Button(application\_window, text=**"Divide"**, width=10)  
 btn\_divide.grid(row=5, column=4, sticky=tk.W, padx=5, pady=5)  
 btn\_divide[**'command'**] = **lambda**: MathIO.write\_quotient\_to\_textbox(  
 float(txt\_first\_number.get()),  
 float(txt\_second\_number.get()),  
 mtx\_results)  
  
  
 btn\_divide = ttk.Button(application\_window, text=**"Clear Results"**, width=55)  
 btn\_divide.grid(row=6, column=1, padx=15, pady=5, columnspan=4)  
 btn\_divide[**'command'**] = **lambda**: MathIO.clear\_textbox(mtx\_results)  
 **return** application\_window  
# End class  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 mw = MainWindow.create\_main\_window()  
 mw.mainloop()

### Creating the Data Layer

The data layer is **where you store data to and retrieve data**. The data layer may be in memory or a file stored on a drive. When stored on a drive, the files will be **either a standard text file or a binary file**.

**Note:** We have already seen how to create a text file in the last module, so this time **we will use a database file, after a brief lab on modules**.

### Python Modules

**The different layers are often organized into code modules. Modules are a way to group functions into a reusable script file**. For example, let's say I want to use some simple math functions in many of my Python applications. If that were the case, I could create a simple python module file with those functions, as shown in this example.

**def** AddValues(v1, v2):  
 **try**:  
 sum = v1 + v2  
 **except**:  
 **raise  
 return** sum  
  
**def** SubtractValues(v1, v2):  
 **try**:  
 dif = v1 - v2  
 **except**:  
 **raise  
 return** dif  
  
**def** MultiplyValues(v1, v2):  
 **try**:  
 prod = v1 \* v2  
 **except**:  
 **raise  
 return** prod  
  
**def** DivideValues(v1, v2):  
 **try**:  
 quot = v1 / v2  
 **except**:  
 **raise  
 return** quot

Note that **none of these functions have any code that will present data** to the user of the functions (such as a print() statement). That is because the module is not made as part of a presentation layer of the application. Instead, it is part of the Processing (business or logic) layer.

The functions in the module can then be linked to a Presentation layer script using the import command.

**import** MathProcessor **as** mp  
  
v1 = 5  
v2 = 0  
**try**:  
 print(mp.AddValues(v1, v2))  
 print(**"{},{}:Addition Passed"**.format(v1, v2))  
**except** Exception **as** e:  
 print(**"{},{}:Addition Failed > {}"**.format(v1, v2, e))  
  
**try**:  
 print(mp.DivideValues(v1, v2))  
 print(**"Division Passed"**)  
**except** Exception **as** e:  
 print(**"Division Failed - Error Msg: "**, e)

## LAB 1: Working with Modules

In this lab, you will create the MathProcessor module and test its class methods from the Main module.

1. Create a python file called MathProcessor.py using the provided code in the previous demos.

2. Create the main module called MathPresention.py using the provided code in the previous demos.

## SQLite

For our data layer**, we will create a new database using SQLite instead of MS SQL Server**. As you learn about SQLite database software, **note** **how** **your existing knowledge and experience can be applied to something new!**

SQLite is a simple SQL relational database used to implement **small, fast, self-contained, high-reliability, full-featured, SQL databases**. SQLite is **the most used database engine in the world**.

"SQLite is **built into all mobile phones** **and** most **computers** and comes bundled inside countless other applications that people use every day." (<https://www.sqlite.org/index.html>, 2019)

### Installing SQLite

"SQLite is famous for its great feature zero-configuration, which means **no complex setup or administration is needed**. This chapter will take you through the process of setting up SQLite on Windows, Linux and Mac OS X." <https://www.tutorialspoint.com/sqlite/sqlite_installation.htm>

### Download SQLite

The first step in the installation is to download the software. You can get it at: <https://www.sqlite.org/download.html>

Be careful to download the correct version for your computer's OS.

***NOTE:*** *"Nowadays, Most of the* ***Mac OS X*** *(edit: and* ***Linux****) distributions, if not all, are being* ***shipped with Sqlite****. So* ***before installing*** *Sqlite manually, it's not a bad idea to* ***check if the SQLite is installed******on your Mac OS X*** *system or not."*

…

**So, open your terminal and type sqlite3** in it. If you the following result on the terminal, then SQLite is already present on your **Mac OS X** operating system.

$sqlite3

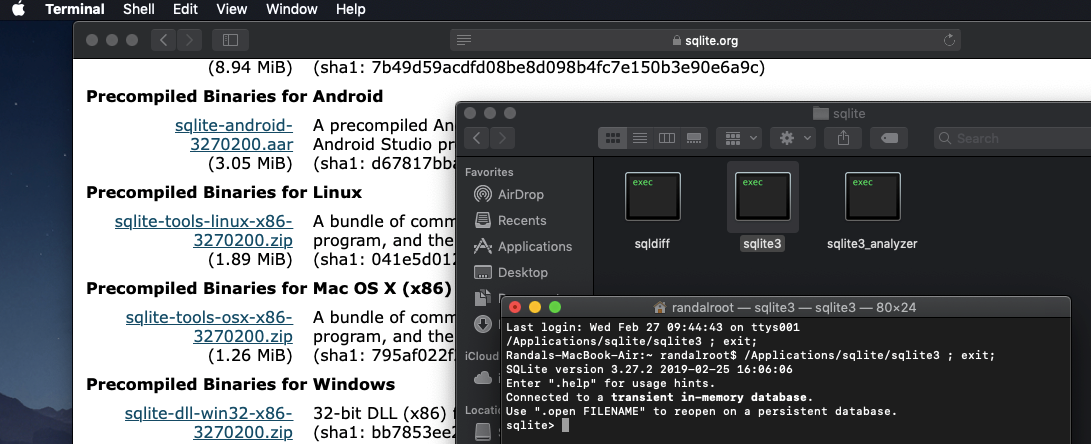
SQLite version 3.13.0 2016-05-18 10:57:30

…

### Installing on Mac

(<http://www.codebind.com/sqlite/how-to-install-sqlite-on/>, 2019)

If you do not find SQLite on your Mac, just download the latest version then unzip it in you Applications folder:



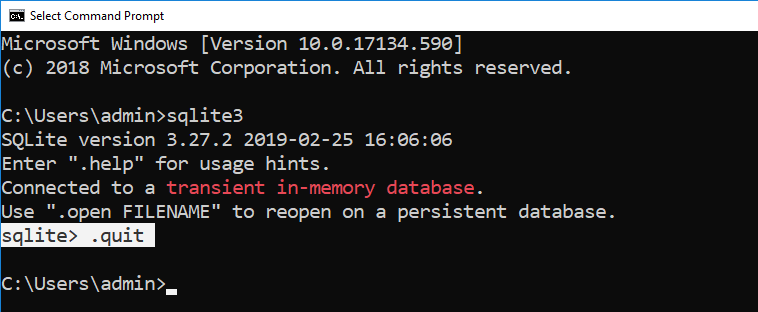
Once it is installed, you can **use it by navigating to its folder** (or modifying the OS's path) and **launching** the applications using "**SQLite3.exe**."

Using "**SQLite3.exe**." opens a **SQLite command** prompt where you can use standard SQL commands to create, store, and manipulate data.

**Tip**: You can quit the command prompt using the "**.quit**" command.

### Installing on Windows

Installing on Windows is pretty much the same, but you must place it in a different folder of your choosing. Make sure to put the unzipped files somewhere that is easy to navigate to as shown in this example:



## Lab 2: Installing and Testing SQLite

In this lab, you will install SQLite on your computer.

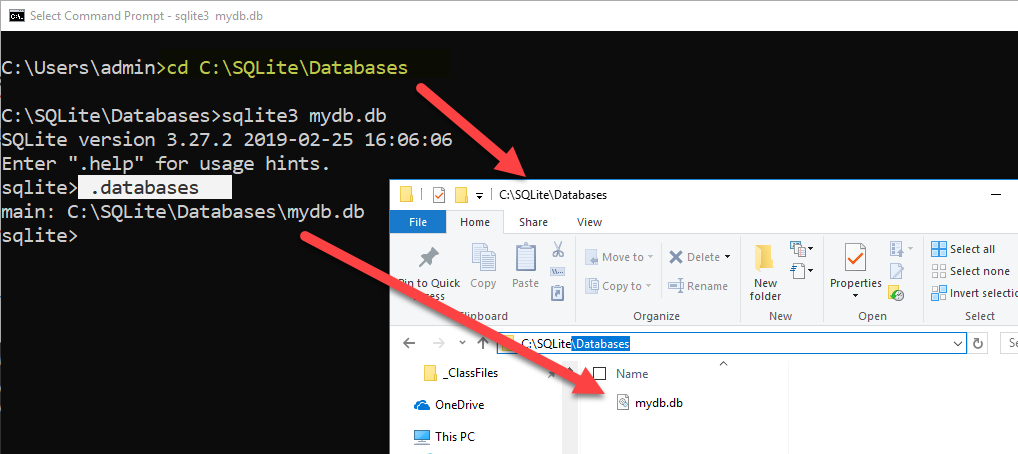
1) Navigate to the SQLite website, download, and install the software.

2) Use a command shell to verify that it is installed and working!

## SQLite Databases

**Once you have SQLite installed, you need to make a database to work with it. To create a database** in SQLite, you open it using this command, "**sqlite3.exe <Name of database>.db**!

**NOTE:** At first, this will only **create an in-memory version of the database, but not an actual file** on your computer. You can **list the currently open databases** using the "**.databases**" command and doing so **will materialize the database onto your computer**.



## Lab 3: Create a SQLite Database

In this lab, you will create a SQLite database on your computer.

1) Use the command shell to create a new database called MyLabsDB.db.

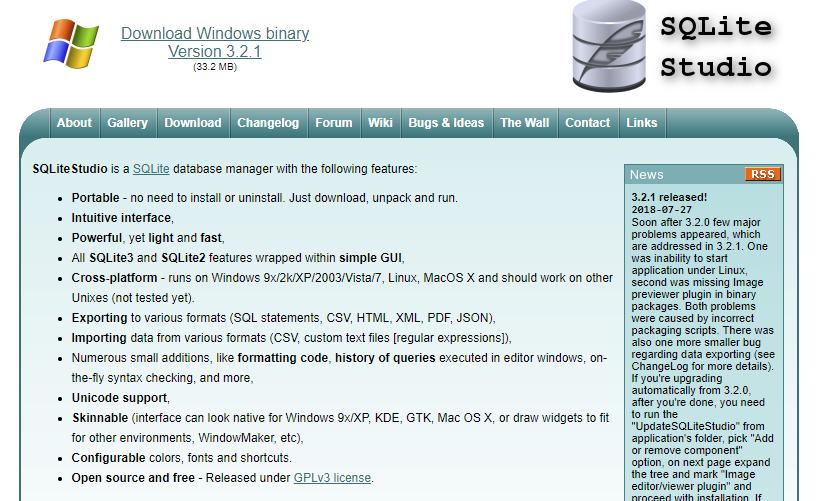
2) Run the ***.databases*** command in SQLite to materialize the database file.

3) Locate the file on your hard drive.

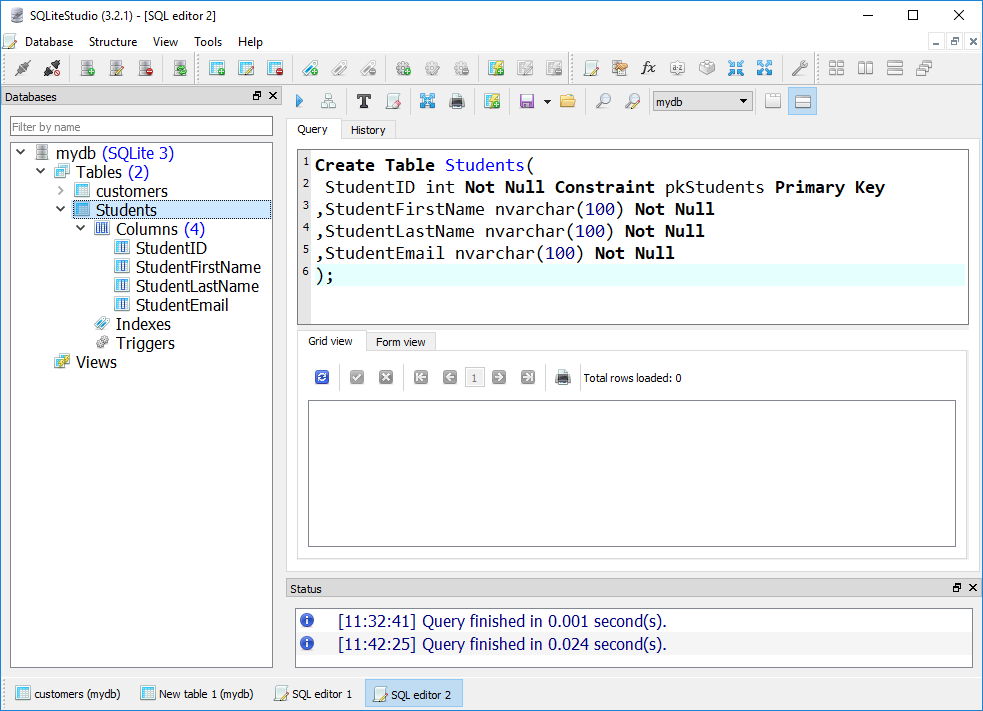
## The SQLite Studio GUI Editor

If you want **a UI for working with SQLite**, download the free **SQLiteStudio** for **Mac, Linux, and Windows** ( <https://sqlitestudio.pl>).

**Note:** For Mac users read this article which includes changing a security option: <https://www.dev2qa.com/how-to-install-sqlite3-on-mac/>



Once you have **downloaded and installed SQLite Studio**, you need to launch and see how it works! Since you have some experience with using a SQL editor, you should be able to figure out quite a bit of this new software's functionality on your own.



## SQLite Tables

Creating a table will materialize the new database file if it has not already done so. **To create a table**, you use the standard SQL code like this:

**create** **table** customers**(**CustomerID int primary key, CustomerName char(100)**);**

### Column datatypes

Different from other database systems, **SQLite uses a *dynamic type system***. For example, the value stored in a column determines its data type, not the column’s data type.

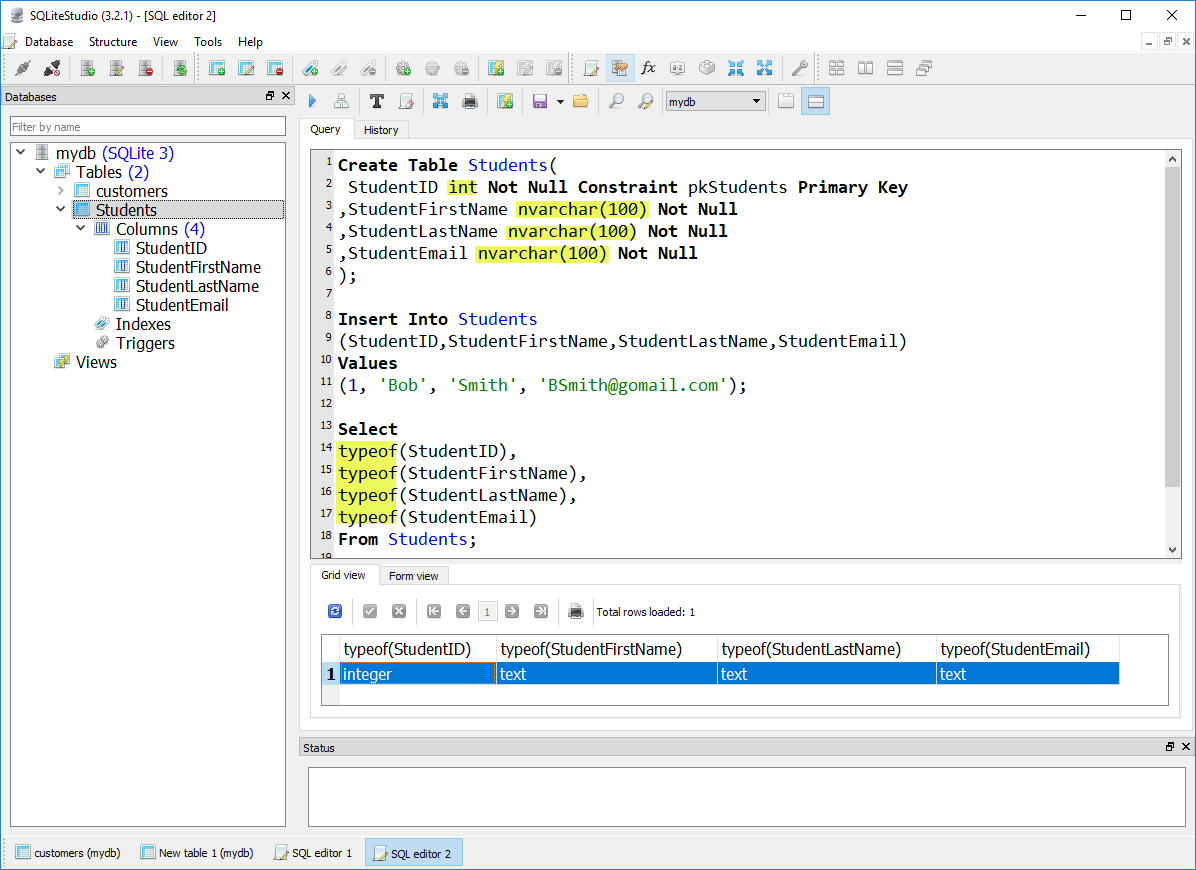
Besides, you don’t have to declare a specific data type for a column when you create a table. In case you declare a column with the integer data type, you can store any kind of data types such as text and BLOB, SQLite will not complain about this.

That said, there are some **basic categories of data** that are used. These are **known as "Storage Classes."**

"SQLite provides **five primitive data types** which referred to as storage classes.

The concept of **storage classes** describes the format SQLite uses to store data on disk. A storage class is **more general than a data type** e.g., INTEGER storage class includes 6 different types of integers. In most cases, you can use storage classes and data type interchangeably." (<http://www.sqlitetutorial.net/sqlite-data-types/>, 2019)

**Like Python, the data type of** **data is chosen based on its value**. So, if you insert data into the table, a data class will be selected for the data **automatically**. It is up to you to perform validation!



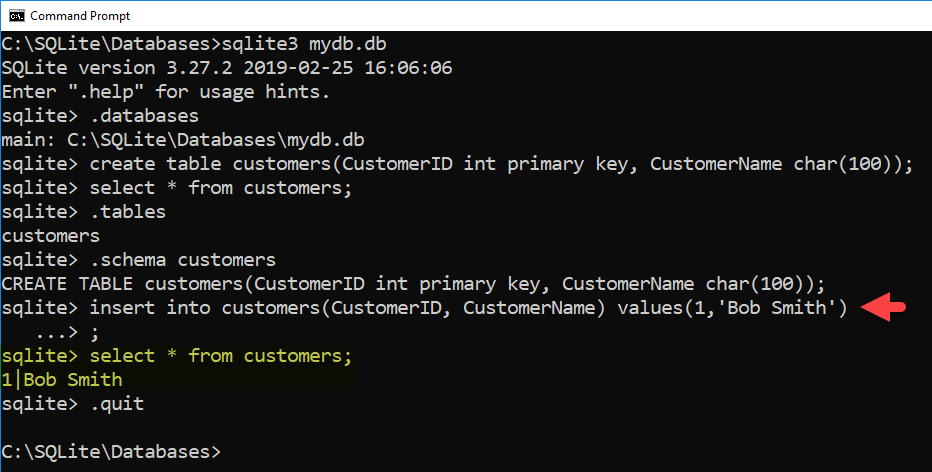
**Notes:**

* SQLite **automatically begins and commits the transaction**. You are not required to use "begin," "commit," or "rollback" transaction statements.
* Since SQLite is serverless, made to run on a single device, it **doesn't have stored procedures**! Validation logic must be placed either in triggers or in your application code!

### Inserting Data

If the table is empty, you will not see any results from your select statement. To add data to a table, you use a SQL insert command, like this:

**Note**: I am showing the command shell version this time for variety!



**Note**: If you forget the **semi-colon,** the code **will not run until one is added**!

## Lab 4: Using SQLite Studio

In this lab, you will create a new SQLite database and table using SQLite Studio (or the command shell if you have any problems with SQLite Studio).

1) Download and **installed SQLite Studio**.

2) Create a **new Database called Enrollments**.

**Important:** Place the database file in a folder called "DataFiles"!

3) Create a **new table called Students** using the following code:

CREATE TABLE Students (

StudentID INT NOT NULL CONSTRAINT pkStudents PRIMARY KEY,

StudentFirstName NVARCHAR (100) NOT NULL,

StudentLastName NVARCHAR (100) NOT NULL,

StudentEmail NVARCHAR (100) NOT NULL);

3) Add some data to the table, then select that data to prove that the insert worked.

## PySQLite

Once we have the database software installed and tested, **we need to set up and test the application software**. Python includes an **easy way to connect to your SQLite database from Python** **using the PySQLite** **module**.

"pysqlite is an interface to the SQLite 3.x embedded relational database engine." ( <https://pypi.org/project/pysqlite/> ,2019)

**Note**: PySQLite is part of the Python standard library and **should be installed** with Python. **If not**, it **can be installed or updated** using pip ( *pip install pysqlite* ).

To test the installation, you can create the following code:

import sqlite3

con = sqlite3.connect('C:/sqlite/databases/Enrollments.db')

Important: Make sure to change the path to the location on your computer! Mac users can use this page to find the path. <https://osxdaily.com/2015/11/05/copy-file-path-name-text-mac-os-x-finder/>

Note: We will work with the PySQLite Module in Session 3

# Session02 Lectures and Labs < 50 mins

In this session, you will continue to work on the final. You will work on your own for the whole 50 minutes of this lab but may ask questions whenever you would like help.

# Session03 < 110 mins

In this session will **create the presentation and middle-tier components using Python and a data component using SQLite.**

## Connecting to SQLite with PySQLite

With SQLite installed, you can **create new or connect to existing SQLite databases by using PySQLite in a Python Script. You do so using the connection() method as shown here:**

**import** sqlite3 # this imports code from the SQLite module of PySQLite!  
  
# Functions -----------------------------------------------------  
**def** create\_connection(db\_file):  
 **try**:  
 con = sqlite3.connect(db\_file) # This opens OR creates the database  
 print(**'Connected! - SQLite Version is: '**, sqlite3.version)  
 **except** Exception **as** e:  
 print(e.\_\_str\_\_())  
 **return** con  
  
# Main body of the script------------------------------------------  
db\_con = create\_connection(**'C:/DataFiles/test.db'**)   
db\_con.close() # Always close the connection when your done

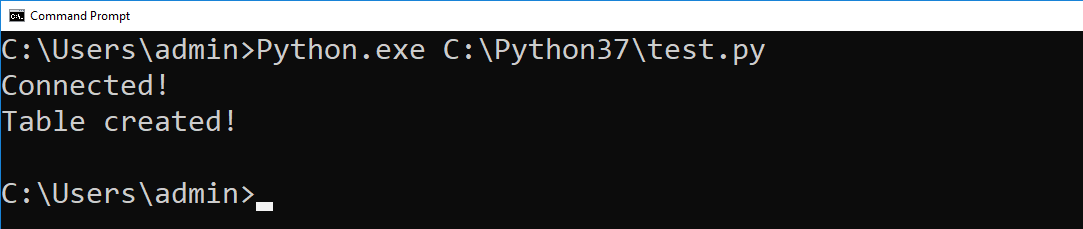
**Notes**:

* If the database file does not exist, **it will create it**!
* You can **specify a path** to the database file like **'C:/DataFiles/test.db**' on Windows and **'~/Desktop/Datafiles/test.db'** on Mac.
* If you **do not specify a path**, the database will be **created in the same folder you are running Python.exe from**!

## Executing SQL Code from Python

Now that we have created a database, we need to use it! You use the **execute**() function of a **sqlite3.cursor object** to submit SQL code to the database. Here is an example:

**import** sqlite3  
  
*# Functions ----------------------------------------------------***def create\_connection**(db\_file):  
 **try**:  
 con = sqlite3.connect(db\_file) *# This opens OR creates the database* **except** Exception **as** e:  
 **raise** e  
 **return** con  
  
  
**def create\_demo\_table**(con):  
 **try**:  
 csr = con.cursor() *# A cursor object allows you to submit commands* csr.execute(**"Create Table Demo (ID [integer], Name [text]);"**) csr.close() *# Always close the cursor when your done* **except** Exception **as** e:  
 **raise** e  
  
  
*# Main body of the script ------------------------------------*db\_con = **None  
  
try**: *# Connecting* db\_con = **create\_connection**(**'test.db'**)  
 print(**"Connected!"**)  
**except** Exception **as** e:  
 print(e)  
  
**try**: *# Creating* **create\_demo\_table**(db\_con)  
 print(**"Table created!"**)  
**except** Exception **as** e:  
 print(e)

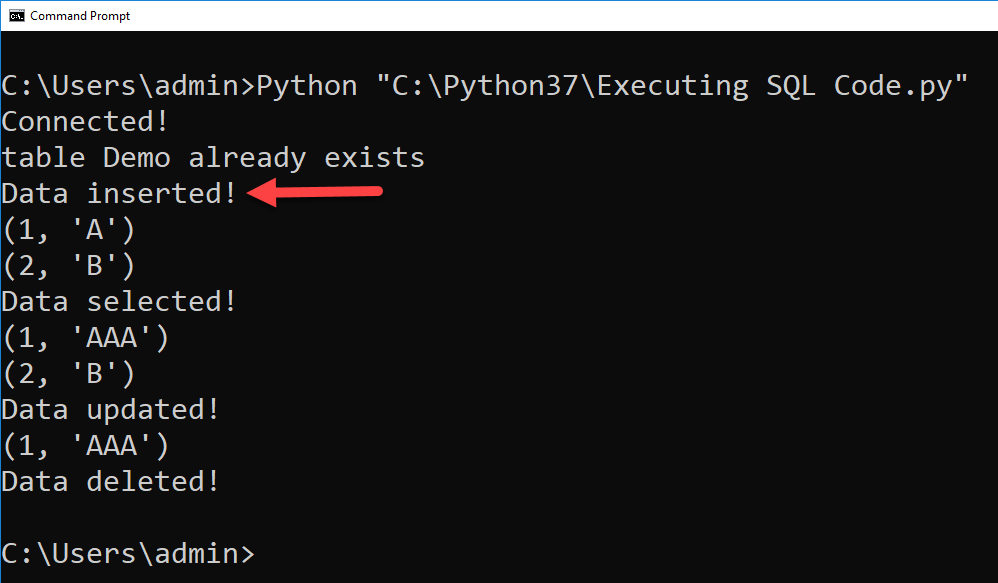


### Executing Transaction Statements

Unlike using SQLite directly, when **using PySQLite any transaction statement** (insert, update, or delete) **must include a commit** statement.

**def insert\_demo\_data**(con):  
 **try**:  
 csr = con.cursor() *# A cursor object allows you to submit commands* csr.execute(**"Insert Into Demo (ID, Name) values (1, 'A'), (2, 'B');"**)

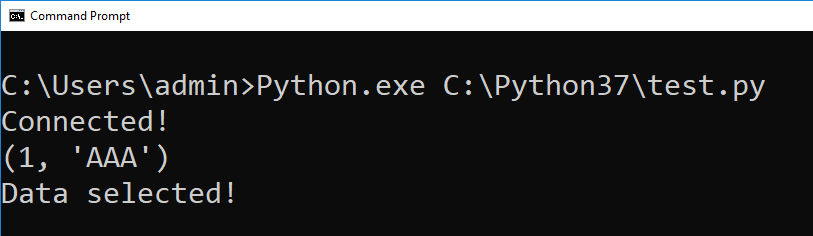
csr.execute(**"commit;"**) *# You need to add this when using PySQLite!* csr.close() *# Always close the cursor when your done* **except** Exception **as** e:  
 **raise** e  
  
**def update\_demo\_data**(con):  
 **try**:  
 csr = con.cursor() *# A cursor object allows you to submit commands* csr.execute(**"Update Demo Set Name = 'AAA' Where ID = 1;"**) *# Single quotes* csr.execute(**"commit;"**) *# You need to add this when using PySQLite!* csr.close() *# Always close the cursor when your done* **except** Exception **as** e:  
 **raise** e  
  
**def delete\_demo\_data**(con):  
 **try**:  
 csr = con.cursor() *# A cursor object allows you to submit commands* csr.execute(**"Delete From Demo Where ID = 2;"**) *# Single quotes for strings!* csr.execute(**"commit;"**) *# You need to add this when using PySQLite!* csr.close() *# Always close the cursor when your done* **except** Exception **as** e:  
 **raise** e



### Executing Select Statements

If the SQL statement is a **SELECT**, you can **use** the **fetchall**() method of a sqlite3.cursor object to retrieve the results.

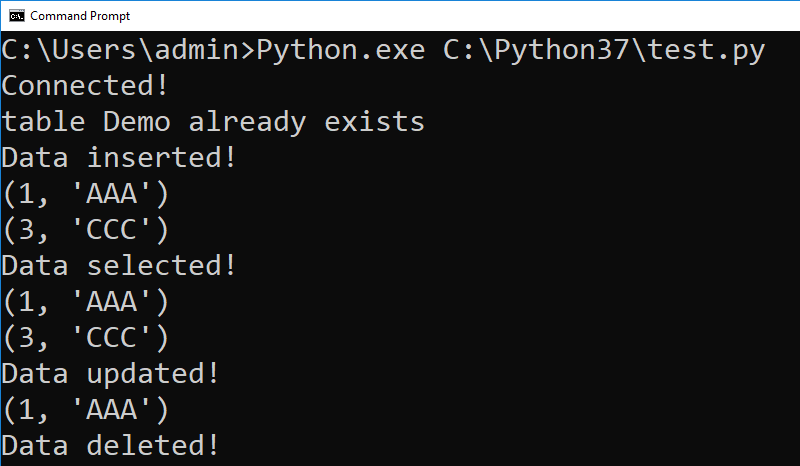
**def** select\_demo\_data(con):  
 **try**:  
 csr = con.cursor() *# A cursor object allows you to submit commands* csr.execute(**"Select ID, Name From Demo;"**)  
 rows = csr.fetchall() *# fetchall puts all rows from the result into a list* csr.close() *# Always close the cursor when your done* **return** rows  
 **except** Exception **as** e:  
 **raise** e  
  
*# Main body of the script ------------------------------------*db\_con = **None  
  
try**: *# Connecting* db\_con = create\_connection(**'test.db'**)  
 print(**"Connected!"**)  
**except** Exception **as** e:  
 print(e)  
  
**try**: *# Selecting* rows = select\_demo\_data(db\_con)  
 **for** row **in** rows:  
 print(row)  
 print(**"Data selected!"**)  
**except** Exception **as** e:  
 print(e)



Using Parameterized Commands

You can use **parameters to create dynamic SQL statements**. PySQLite, makes this easy by allowing you to **pass in a list of arguments** to configure a SQL string with **question marks for parameters**.

**def** insert\_demo\_data(con, ID, Name):  
 **try**:  
 csr = con.cursor()  
 csr.execute(**"INSERT INTO Demo (ID, Name) values (?,?);"**, [ID, Name])  
 csr.execute(**"commit;"**)  
 csr.close()  
 **except** Exception **as** e:  
 **raise** e  
  
**def** update\_demo\_data(con, ID, Name):  
 **try**:  
 csr = con.cursor()ds  
 csr.execute(**"Update Demo Set Name = ? Where ID = ?;"**, [ID, Name])  
 csr.execute(**"commit;"**)  
 csr.close()  
 **except** Exception **as** e:  
 **raise** e  
  
**def** delete\_demo\_data(con, ID):  
 **try**:  
 csr = con.cursor()  
 csr.execute(**"Delete From Demo Where ID = ?;"**, [ID])  
 csr.execute(**"commit;"**)  
 csr.close()  
 **except** Exception **as** e:  
 **raise** e



Here is the **completed version of the code**:

**import** sqlite3  
  
*# Functions ----------------------------------------------------***def** create\_connection(db\_file):  
 **try**:  
 con = sqlite3.connect(db\_file) *# This opens OR creates the database* **except** Exception **as** e:  
 **raise** e  
 **return** con  
  
**def** create\_demo\_table(con):  
 **try**:  
 csr = con.cursor() *# A cursor object allows you to submit commands* csr.execute(**"Create Table Demo (ID [integer] Primary Key, Name [text]);"**)  
 csr.close() *# Always close the cursor when your done* **except** Exception **as** e:  
 **raise** e  
  
**def** insert\_demo\_data(con, ID, Name):  
 **try**:  
 csr = con.cursor()  
 csr.execute(**"INSERT INTO Demo (ID, Name) values (?,?);"**, [ID, Name])  
 csr.execute(**"commit;"**)  
 csr.close()  
 **except** Exception **as** e:  
 **raise** e  
  
**def** update\_demo\_data(con, ID, Name):  
 **try**:  
 csr = con.cursor()  
 csr.execute(**"Update Demo Set Name = ? Where ID = ?;"**, [Name, ID])  
 csr.execute(**"commit;"**)  
 csr.close()  
 **except** Exception **as** e:  
 **raise** e  
  
**def** delete\_demo\_data(con, ID):  
 **try**:  
 csr = con.cursor()  
 csr.execute(**"Delete From Demo Where ID = ?;"**, [ID])  
 csr.execute(**"commit;"**)  
 csr.close()  
 **except** Exception **as** e:  
 **raise** e  
  
**def** select\_demo\_data(con):  
 **try**:  
 csr = con.cursor() *# A cursor object allows you to submit commands* csr.execute(**"Select ID, Name From Demo;"**)  
 rows = csr.fetchall() *# fetchall puts all rows from the result into a list* csr.close() *# Always close the cursor when your done* **return** rows  
 **except** Exception **as** e:  
 **raise** e  
  
**def** print\_selected\_data():  
 **try**: *# Selecting* rows = select\_demo\_data(db\_con)  
 **for** row **in** rows:  
 print(row)  
 print(**"Data selected!"**)  
 **except** Exception **as** e:  
 print(e)  
  
*# Main body of the script ------------------------------------*db\_con = **None  
  
try**: *# Connecting* db\_con = create\_connection(**'test.db'**)  
 print(**"Connected!"**)  
**except** Exception **as** e:  
 print(e)  
  
**try**: *# Creating* create\_demo\_table(db\_con)  
 print(**"Table created!"**)  
**except** Exception **as** e:  
 print(e)  
  
**try**: *# Creating* insert\_demo\_data(db\_con, 1, **'test insert'**)  
 print(**"Insert Committed"**)  
 print\_selected\_data() *# Show the change***except** Exception **as** e:  
 print(e)  
  
**try**: *# Creating* update\_demo\_data(db\_con, 1, **'test update'**)  
 print(**"Update Committed"**)  
 print\_selected\_data() *# Show the change***except** Exception **as** e:  
 print(e)  
  
**try**: *# Creating* delete\_demo\_data(db\_con, 1)  
 print(**"Delete Committed"**)  
 print\_selected\_data() *# Show the change***except** Exception **as** e:  
 print(e)  
  
db\_con.close() *# Always close the connection when your done*

## Lab 5: Working with PySQLite

In this lab, you will **create a Python script that connects to the Enrollments database you made in Lab04**. You need to create code that will **select and perform transaction statements using the Students table**.

1) Open the Python editing tool, **Idle**.

2) Create a **new** Python **script** file called **Lab05.py**.

3) **Create and test code** that will allow you to **connect to the Enrollments database**.

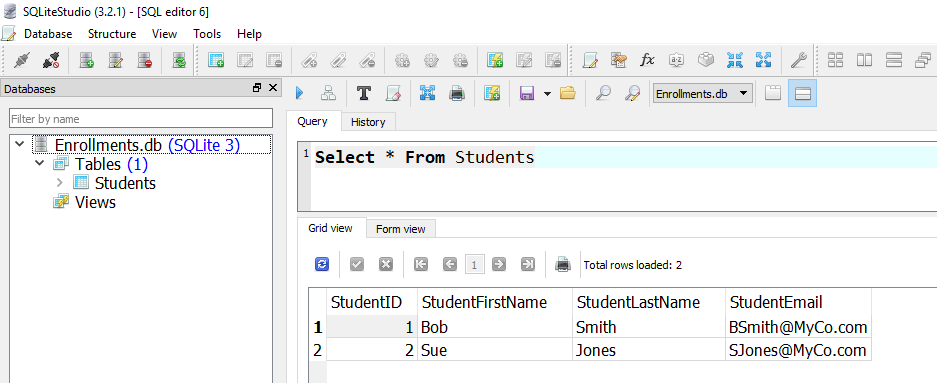
**import** sqlite3  
  
*# Functions ----------------------------------------------------***def** create\_connection(db\_file):  
 **try**:  
 con = sqlite3.connect(db\_file) *# This opens OR creates the database* **except** Exception **as** e:  
 **raise** e  
 **return** con  
  
  
*# Main body of the script ------------------------------------*db\_con = **None  
  
try**: *# Connecting* db\_con = create\_connection(**'C:/DataFiles/Enrollments.db'**) *# Modify as needed!* print(**"Connected!"**)  
**except** Exception **as** e:  
 print(e)

4) **Create and test code** that will allow you to **insert** data into the Students table.

**import** sqlite3  
  
*# Functions ----------------------------------------------------***def** create\_connection(db\_file):  
 **try**:  
 con = sqlite3.connect(db\_file)  
 **except** Exception **as** e:  
 **raise** e  
 **return** con  
  
**def** insert\_students\_data(con, ID, FName, LName, Email):  
 **try**:  
 csr = con.cursor()  
 csr.execute(**"INSERT INTO Students values (?,?,?,?);"**, [ID, FName, LName, Email])  
 csr.execute(**"commit;"**)  
 csr.close()  
 **except** Exception **as** e:  
 **raise** e

*# Main body of the script ------------------------------------*db\_con = **None  
  
try**: *# Connecting* db\_con = create\_connection(**'C:/DataFiles/Enrollments.db'**)  
 print(**"Connected!"**)  
**except** Exception **as** e:  
 print(e)  
  
**try**: *# Inserting* insert\_students\_data(db\_con, 2, **"Sue"**, **"Jones"**, **"SJones@MyCo.com"**)  
 print(**"Data inserted!"**)  
**except** Exception **as** e:  
 print(e)

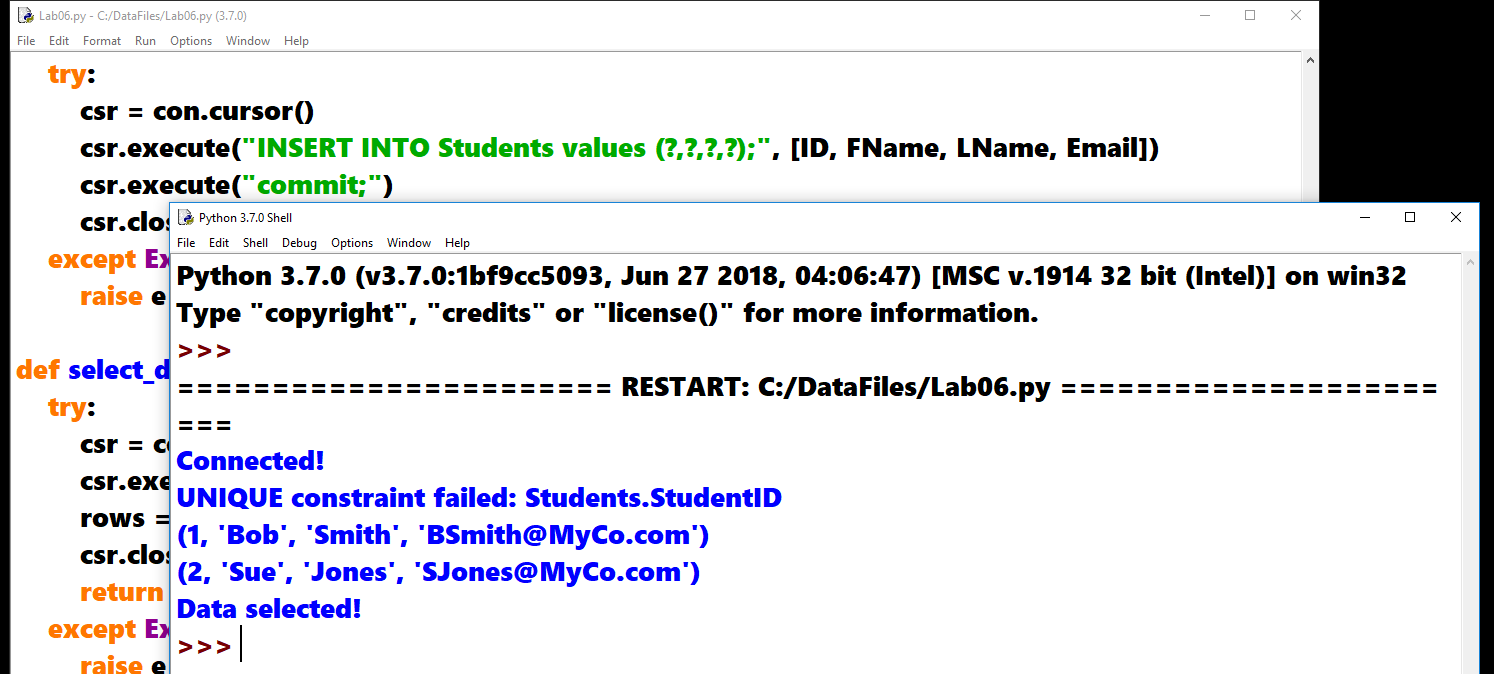
5) **Verify** that the data was inserted **using SQLiteStudio**.



6) **Create and test code** that will **select** data from the Students table.

**import** sqlite3  
  
*# Functions ----------------------------------------------------***def** create\_connection(db\_file):  
 **try**:  
 con = sqlite3.connect(db\_file)  
 **except** Exception **as** e:  
 **raise** e  
 **return** con  
  
**def** insert\_students\_data(con, ID, FName, LName, Email):  
 **try**:  
 csr = con.cursor()  
 csr.execute(**"INSERT INTO Students values (?,?,?,?);"**, [ID, FName, LName, Email])  
 csr.execute(**"commit;"**)  
 csr.close()  
 **except** Exception **as** e:  
 **raise** e

**def** select\_students\_data(con):  
 **try**:  
 csr = con.cursor()  
 csr.execute(**"SELECT StudentID, StudentFirstName, StudentLastName, StudentEmail FROM Students;"**)  
 rows = csr.fetchall()  
 csr.close()  
 **return** rows  
 **except** Exception **as** e:  
 **raise** e  
  
*# Main body of the script ------------------------------------*db\_con = **None  
  
try**: *# Connecting* db\_con = create\_connection(**'C:/DataFiles/Enrollments.db'**)  
 print(**"Connected!"**)  
**except** Exception **as** e:  
 print(e)  
  
**try**: *# Inserting* insert\_students\_data(db\_con, 2, **"Sue"**, **"Jones"**, **"SJones@MyCo.com"**)  
 print(**"Data inserted!"**)  
**except** Exception **as** e:  
 print(e)  
  
**try**: *# Selecting* rows = select\_students\_data(db\_con)  
 **for** row **in** rows:  
 print(row)  
 print(**"Data selected!"**)  
**except** Exception **as** e:  
 print(e)



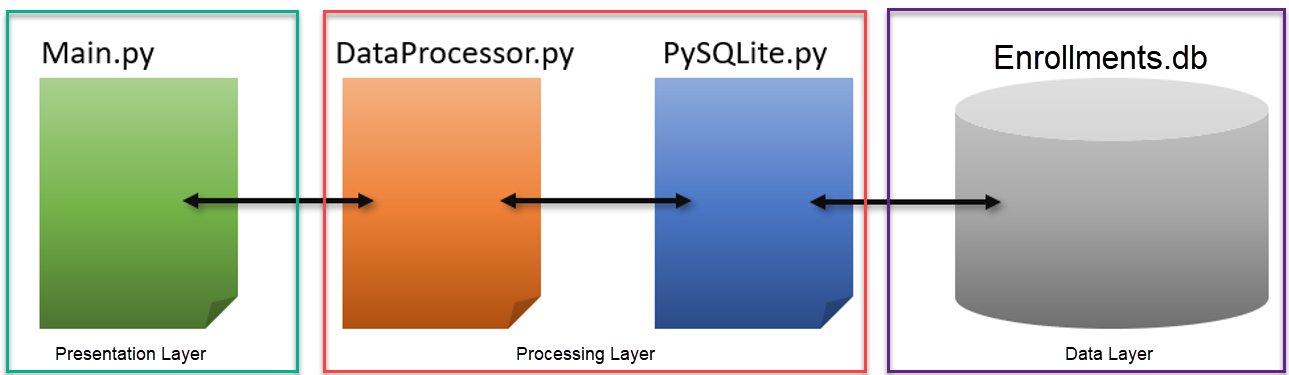
Optional

6) **Create and test code** that will **update** data from the Students table.

7) **Create and test code** that will **delete** data from the Students table.

## Creating Multiple Tiers using Modules

So far, we have all our processing and presentation code in the same file. Since our goal is to demonstrate a simple example of a three-tiered application, **we need to move the processing code into a separate module file.**



### Creating a Processing Module

The **processing module will include the functions we need to perform selects, inserts, updates, and deletes** in the database. Note how our functions use code that was written by the developer who created PySQLite. This code re-usability is a big part of why multi-tier applications are a popular option.

#------------DataProcessor.py -----------------------------------#  
#Desc: Functions that read and write data to a database  
#Dev: RRoot  
#Date: 12/12/2020  
#ChangeLog:(When, Who, What)  
#----------------------------------------------------------------#

**import** sqlite3 *# This references the PySQLite.py module*  
  
*# Functions ----------------------------------------------------***def** create\_connection(db\_file):  
 **try**:  
 con = sqlite3.connect(db\_file)  
 **except** Exception **as** e:  
 **raise** e  
 **return** con  
  
**def** insert\_students\_data(con, ID, FName, LName, Email):  
 **try**:  
 csr = con.cursor()  
 csr.execute(**"INSERT INTO Students values (?,?,?,?);"**, [ID, FName, LName, Email])  
 csr.execute(**"commit;"**)  
 csr.close()  
 **except** Exception **as** e:  
 **raise** e

**Note**: None of the presentation code is copied to the Processing model.

### Creating the Main Module

With the processing module complete, we next need **a presentation module**. This code will be the starting point of our database application and is often considered **the Main interface between a human and the software** being used. As such, we will call the module Main.py.

#------------Main.py -----------------------------------#  
#Desc: Presentation code for working with a database  
#Dev: RRoot  
#Date: 12/12/2020  
#ChangeLog:(When, Who, What)  
#----------------------------------------------------------------#  
  
**import** DataProcessor **as** dp  
  
# Main body of the script ------------------------------------  
db\_con = **None  
  
try**: # Connecting  
 db\_con = dp.create\_connection(**'C:/DataFiles/test.db'**)  
 print(**"Connected!"**)  
**except** Exception **as** e:  
 print(e)  
  
**try**: # Creating  
 dp.create\_demo\_table(db\_con)  
 print(**"Table created!"**)  
**except** Exception **as** e:  
 print(e)  
  
**try**: # Inserting  
 dp.insert\_demo\_data(db\_con, 3, **"CCC"**)  
 print(**"Data inserted!"**)  
**except** Exception **as** e:  
 print(e)  
  
**try**: # Selecting  
 rows = dp.select\_demo\_data(db\_con)  
 **for** row **in** rows:  
 print(row)  
 print(**"Data selected!"**)  
**except** Exception **as** e:  
 print(e)  
  
**try**: # Updating  
 dp.update\_demo\_data(db\_con, 3, **"CC"**)  
 rows = dp.select\_demo\_data(db\_con)  
 **for** row **in** rows:  
 print(row)  
 print(**"Data updated!"**)  
**except** Exception **as** e:  
 print(e)  
  
**try**: # Deleting  
 dp.delete\_demo\_data(db\_con, 3)  
 rows = dp.select\_demo\_data(db\_con)  
 **for** row **in** rows:  
 print(row)  
 print(**"Data deleted!"**)  
**except** Exception **as** e:  
 print(e)  
  
db\_con.close()

## Lab 6: Creating a Multi-Tiered Application

In this lab, you will **separate the** **Python script you made in lab 5 into processing and presentation modules.**

1) Open the Python editing tool, **Idle**.

2) Create a **new** Python **script** file called **StudentDataProcessor.py**.

3) Create a **new** Python **script** file called **StudentMain.py**.

4) **Copy and paste the code you created in lab 5 into either of these two files,** based on the purpose of the code.

**5) Test** that you can still connect to the Enrollments database and interact with the Students table **using the Main module**.