**Week 8 – Interacting with Databases and Client-Server Programming**

**Learning Outcomes**

By the end this lesson, you will:

* Design, build, and query a simple database using SQL
* Write a class to query a MySQL database in Python.
* Encapsulate the data returned by an SQL query in lists of objects so that your program can process them.
* Program the interaction of a client and server at the socket level.
* Describe what a thread is and why it can be useful to distribute tasks among multiple threads
* Write a multi-threaded application in Python.
* Explain why it is important to synchronize threads that need to share access to particular data sources.

You will demonstrate attainment of these skills by writing simple Python programs that connect to a database, coordinate the interaction of a client and server, and separate tasks among multiple threads.

**Accessing a Database**

So far, we’ve looked at getting data from a comma-separated text file and from an xml file.

One of the most popular data sources, however, is a database. We’ll now consider how to grab data from a database. Specifically, we’ll work with mysql.

A database is hosted on a database server. A server may host multiple databases. A database, in turn, can consist of multiple tables.

MySql is one of the most popular databases, and it drives a lot of database-driven websites.

You can install mysql on your own machine. Go here:

http://dev.mysql.com/downloads/

You can download the MySQL Community Server and install it on your machine. Or, if you are on Windows and want more bells-and-whistles, you can click on the MySQL on Windows link and access even more tools.

Alternatively, if you want easy setup of not only mysql but also apache and php, presumably so that you can set up a database-driven website, then you might wish to download WAMPServer or MAMPServer. Just go to http://www.wampserver.com/en/ for Windows or https://www.mamp.info/en/ for mac. These products are one-install ways to set up a database-driven website.

In the Lewis computer labs, mysql is set up on each laptop with particular user name and password. To access it, issue the following command:

mysql –h localhost –u root –p

and press Enter.

The password you’ll then type on the next line is Flyers.

Of course, for those of you who have installed your own mysql instance, including those of you online and not at the Lewis Computer Science labs, the username and password will be different. The documentation for whatever version you have downloaded should indicate what the default username and password are.

Once connected, you can see a list of databases hosted on the server like so:

show databases;

You can adopt, or “use”, any of these databases as your target one.

use name\_of\_database;

Once you have identified the database you want to work with using use, you can then create a table in it. A table is nothing more than a set of rows and columns. Your job is to define the columns. For example, let’s create a table of Rectangle objects:

create table rectangle (

name varchar(20) not null unique,

xpos integer,

ypos integer,

width integer,

height integer,

primary key(name)

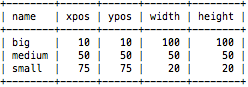
);

Once you’ve created a table, you can then fill it with values a row at a time using the insert command.

insert into rectangle values (“big”,10,10,100,100);

insert into rectangle values (“medium”,50,50,50,50);

insert into rectangle values (“small”,75,75,20,20);



You can then select records from the database using the select command:

select \* from rectangle;

will select all the information for all the records

select \* from rectangle where width < 50;

will select all the information for rectangles whose width is less than 50.

select name,xpos,ypos from rectangle where width < 50;

will select just the name, xpos, and ypos for rectangles whose width is less than 50.

So far, we’ve done all this from the mysql command line tool, but it turns out that you can issue the same kinds of queries from a program you write. There are two ways to do this, depending on the version of python you have installed on your system.

**If you have a version of python that is 3.4 or earlier:**

MySql provides utitlities called *connectors* that enable you to link mysql with a wide variety of programming languages. Python is one of them. You can download the python connector for mysql and begin writing python code that accesses a mysql database.

The Python Connector can be found here:

https://dev.mysql.com/downloads/connector/python/

Here’s some sample code. This particular example connects to a database called test on a server called “host” that contains a table called Junk. Each row of Junk has two columns, one that specifies the id of something, and one that stores its description.

import mysql.connector

conn = mysql.connector.connect(host="host",user="root",password="pass",database="test")

cursor = conn.cursor()

cursor.execute("select \* from Junk")

for (id, desc) in cursor:

print(id,desc)

To summarize:

1. You connect to a database using mysql.connector.connect. This returns a Connection object.
2. That Connection object has a cursor() function, which enables you to launch queries against the connected database and then process its rows.
3. You use the cursor to execute queries.
4. You use a for loop to march through the data returned by the query.

**If you have a version of python that is 3.5 or later:**

At the time of this writing, there is no connector for Python 3.5. So, instead, you should use pymysql, a library for interacting with mysql.

You first need to install pymsql. Doing that depends on which version of Python 3.5 or later you have.

If you installed the Anaconda python suite for data analysis, you should open up a command line and type this:

conda install pymysql

conda is the built-in tool the Anaconda version of Python uses to install new libraries.

If you installed the standard default python version, then you should type this:

pip install pymysql

pip is the built-in tool the standard version of Python uses to install new libraries.

Once you do that, the only changes you’ll have to make to the code we wrote above to interact with a database are:

1. import pymsql
2. conn = pymsql.connect( … )

So, here is the updated source code using pymysql instead of mysql.connector:

import pymysql

conn = pymsql.connect(host="host",user="root",password="pass",database="test")

cursor = conn.cursor()

cursor.execute("select \* from Junk")

for (id, desc) in cursor:

print(id,desc)

In the code below, we’ll use mysql.connector. Just know that all you’ll need to do to make a pymsql-based solution work is replace any occurrence of mysql.connector, including in the import statement, with pymysql.

Let’s see now if we can encapsulate this into a class.

Let’s have a MySql\_Database class that enables us to connect to the database. It should have the server, database, username, and password stored in it. It should have the ability to connect to the database, as well as the ability to return records that match what a query is asking for.

# Ray Klump

# define a class that models a database connection to

# a mysql database server

import mysql.connector

class Mysql\_Database:

def \_\_init\_\_(self,server,database,user,pword):

self.server = server

self.database = database

self.user = user

self.pword = pword

self.conn = None

def connect(self):

try:

self.conn = mysql.connector.connect( \

host = self.server, \

database = self.database, \

user = self.user, \

password = self.pword)

except:

self.conn = None

def execute(self, query):

if self.conn != None:

cursor = self.conn.cursor()

cursor.execute(query)

return cursor

else:

return None

Here’s code that uses it:

dbase = Mysql\_Database("localhost","shapes","username","password")

dbase.connect()

results = dbase.execute("select \* from rectangle")

if results != None:

for (name,xpos,ypos,width,length) in results:

print(name,xpos,ypos,width,length)

else:

print("no results")

dbase.conn.close()

If you want to centralize the unpacking of the data directly as some kind of object, you can write a class that does that. The following class goes to the database and asks for rectangle data. It can then create and return a list of rectangles.

class MySql\_Rectangle\_Data\_Fetcher:

def \_\_init\_\_(self,server="",db="",user="",password=""):

try:

self.dbase = MySql\_Database(server,db,user,password)

self.dbase.connect()

except:

self.dbase = None

def is\_connected(self):

return (self.dbase != None)

def get\_data(self,query):

cursor = self.dbase.execute(query)

rectangles = []

for (name,xpos,ypos,width,length) in cursor:

rec = Rectangle(name,xpos,ypos,width,length)

rectangles.append(rec)

return rectangles

Here is a complete application that retrieves Rectangle data from a database:

# Ray Klump

# define a class that models a database connection to

# a mysql database server

import mysql.connector

class Rectangle:

def \_\_init\_\_(self,name,x,y,w,h):

self.name = name

self.x = x

self.y = y

self.width = w

self.height = h

def to\_string(self):

return "%s %d %d %d %d" % (self.name,self.x,self.y,self.width,self.height)

class MySql\_Database:

def \_\_init\_\_(self,server,database,user,pword):

self.server = server

self.database = database

self.user = user

self.pword = pword

self.conn = None

def connect(self):

try:

self.conn = mysql.connector.connect( \

host = self.server, \

database = self.database, \

user = self.user, \

password = self.pword)

except:

self.conn = None

def execute(self, query):

if self.conn != None:

cursor = self.conn.cursor()

cursor.execute(query)

return cursor

else:

return None

class MySql\_Rectangle\_Data\_Fetcher:

def \_\_init\_\_(self,server="",db="",user="",password=""):

try:

self.dbase = MySql\_Database(server,db,user,password)

self.dbase.connect()

except:

self.dbase = None

def is\_connected(self):

return (self.dbase != None)

def get\_data(self,query):

cursor = self.dbase.execute(query)

rectangles = []

for (name,xpos,ypos,width,length) in cursor:

rec = Rectangle(name,xpos,ypos,width,length)

rectangles.append(rec)

return rectangles

def main():

fetcher = MySql\_Rectangle\_Data\_Fetcher("localhost","shapes","root","Flyers")

if fetcher.is\_connected():

rectangles = fetcher.get\_data("select \* from rectangle")

if rectangles != None:

for r in rectangles:

print(r.to\_string())

else:

print("no rectangles retrieved")

else:

print("Could not connect.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Network Programming**

While this is not specifically an object-oriented concept, it is related to databases, since databases are network-aware applications. Also, one of the important purposes of this course is to expose you to programming applications you might not have seen in other courses. We’ll do that right now with network programming and threads. First, we’ll talk about network programming.

Python supports two different approaches to accessing a network from a program you write. One is low-level, and it involves using sockets. The other approach is higher-level and builds upon sockets to implement specific network protocols, such as HTTP. (We actually already used this high-level approach earlier when we learned how to download an xml file from a website.)

So, now we’ll focus on sockets.

A socket is a bidirectional link between processes. The processes could be running on the same machine, or they could be running on different machines.

To program with sockets, you need to import socket at the top of our program.

What you do next depends on whether you are programming a client or a server. Let’s work with a server first.

Create the socket. In this case, we’ll call the socket “server” since this is going to function as a server.

server = socket.socket()

That creates a socket called server. You can then associate that socket with a particular address using bind:

server.bind(("localhost",8092))

You then tell the socket to switch to listening mode so that it can hear when a process wants to interface with it:

server.listen(5) # server socket; max five connections

You can then make the server sit and wait until a connection comes in. To do that, use the accept command:

[connection, address] = server.accept()

Then, process the ongoing communications using a while loop. In this while loop, we are assuming that the client will eventually type q to end the communications. Otherwise, we’ll just report what the client wrote to the screen:

keep\_going = True

while keep\_going:

buf = connection.recv(64)

if len(buf) > 0:

msg = buf.decode()  
 print(msg)  
 msg = buf.decode()  
 if msg == "q":

print("Thank you for connecting. Good-bye")

connection.close()

keep\_going = False

server.close()

Basically, we take 64 bytes at a time and print what arrived. If they entered a q, we print a parting message, leave the loop, and close the server socket.

the str.decode(buf) line is needed because the socket receives a stream of bytes. If we want to work with the bytes as character strings, we have to decode them. We do that using str.decode, where str represents the string data type in python.

The client is similar. To set up the client, we need to create a socket. The socket could be named anything, but we’ll call it client here:

client = socket.socket()

Now connect to the address and port where the server was bound:

client.connect(("localhost",8092))

Now enter a loop that allows the user to keep sending the server messages until they enter q:

msg = input("Enter message: ").lower().strip()

while msg != "q":

client.send(str.encode(msg))

msg = input("Enter message: ").lower().strip()

client.send(str.encode(msg))

client.close()

Once they enter a q, we send the server one more message, and then we close the client socket.

The str.encode is needed to convert the strings the user enters to a byte stream, because sockets require byte streams and can’t work directly with character strings.

It’s a good idea to wrap attempts to send data to the server in a try…except block, in case communications are interrupted.

Here’s a revised version of the code for the server:

import socket

server = socket.socket()

server.bind(("localhost",8092))

server.listen(5) # server socket; max five connections

[connection, address] = server.accept()

print("connected to %s" % address)

keep\_going = True

while keep\_going:

buf = connection.recv(64)

if len(buf) > 0:

buf = buf.decode()

print(buf)

if buf == "q":

print("Thank you for connecting. Good-bye")

connection.close()

keep\_going = False

server.close()

Here’s the revised version of the code for the client, including a try…except:

import socket

client = socket.socket()

client.connect(("localhost",8092))

msg = input("Enter message: ").lower().strip()

while msg != "q":

try:

client.send(str.encode(msg))

msg = input("Enter message: ").lower().strip()

except:

print(“An error happened”)

client.send(str.encode(msg))

client.close()

Let’s modify this slightly now to have the server send a confirmation message back to the client. Here’s the new server with the additional line highlighted. This is file simple\_server.py.

import socket

server = socket.socket()

server.bind(("localhost",8092))

server.listen(5) # server socket; max five connections

[connection, address] = server.accept()

print("Connected to %s" % address)

keep\_going = True

while keep\_going:

buf = connection.recv(64)

if len(buf) > 0:

connection.send(str.encode("Message received by server."))

msg = buf.decode()

print(msg)

if msg == "q":

print("Connection closed.")

connection.close()

keep\_going = False

server.close()

Here’s the new client that can receive this message from the server. The additional lines are highlighted. This is file simple\_client.py.

import socket

client = socket.socket()

client.connect(("localhost",8092))

msg = input("Enter message: ").lower().strip()

while msg != "q":

try:

client.send(str.encode(msg))

data = client.recv(64)

print(data.decode())

msg = input("Enter message: ").lower().strip()

except:

print("There was a problem.")

client.send(str.encode(msg))

client.close()

The downside of this solution is that it supports only one connection at a time. To support multiple connections, we need to use threads. Each thread is responsible for managing the back-and-forth of the server with one connected client. We’ll learn about that in a little bit.

First, though, let’s consider wrapping up our client and server in a class.

**Server and Client Classes and Subclasses**

In this unit, we encapsulate our server and client code in a class.

First, here are the superclasses – one for Server, and another for client. This would be stored in a file called socket\_classes.py, perhaps.

import socket

class Server:

def \_\_init\_\_(self,addr,port,conn\_lim):

self.setup()  
 self.establish\_connection(addr,port,conn\_lim)

def establish\_connection(self,addr,port,conn\_lim):

self.socket = socket.socket()

self.socket.bind((addr,port))

self.socket.listen(conn\_lim)

[self.connection, self.address] = self.socket.accept()

def send(self,msg):

self.connection.send(str.encode(msg))

def recv(self,size):

buf = self.connection.recv(size)

return buf.decode()

def serve(self):

pass

def setup(self):

pass

class Client:

def \_\_init\_\_(self,addr,port):

self.establish\_connection(addr,port)

def establish\_connection(self,addr,port):

self.socket = socket.socket()

self.socket.connect((addr,port))

def perform(self):

pass

def send(self,msg):

self.socket.send(str.encode(msg))

def recv(self,size):

buf = self.socket.recv(size)

return buf.decode()

For Server, the serve function is where custom work specific to a subclass will be done. For Client, the perform function is where custom work specific to a Client subclass would be done.

Notice the send and recv functions in each. They make it easier to send and receive data because they have encoding and decoding built into them.

Now that we have the socket classes built, we can build specific clients and servers by extending them.

Here’s a particular example. In this one, we build a client-server chat tool. We have a Chat\_Server class and a Chat\_Client class. The session begins when the client connects to localhost on port 8092. It ends when the client sends a message of “q”.

The server might be expressed in a file called chat\_server.py, which appears below.

from socket\_classes import Server

class Chat\_Server(Server):

def serve(self):

keep\_going = True

while keep\_going == True:

msg = self.recv(64).strip()

if msg != "":

self.send("Message received.")

print(msg)

if msg == "q":

print("Connection terminated.")

self.connection.close()

keep\_going = False

self.socket.close()

chatsrv = Chat\_Server("localhost",8092,1)

chatsrv.serve()

The client might be expressed in a file called chat\_client.py, which appears below.

from socket\_classes import Client

class Chat\_Client(Client):

def perform(self):

msg = input("Enter message: ")

msg = msg.strip()

while msg != "q":

try:

self.send(msg)

data = self.recv(64)

print(data)

msg = input("Enter message: ")

except:

print("There was an error")

self.send(msg)

self.socket.close()

chatcli = Chat\_Client("localhost",8092)

chatcli.perform()

**Another example: Client-Server Flash Cards**

Here’s another example. In this one, the server presents the client a series of flash cards.

Here’s the server. It asks some questions on launch to set up the flash card environment. Then it asks the client multiplication problems.

This is file fcserver.py.

from socket\_classes import Server

class Flashcard\_Server(Server):

def setup(self):

which = input("Enter which times tables you know (separated by spaces): ")

parts = which.split(" ")

self.tables = []

for part in parts:

self.tables.append(int(part))

self.how\_many = int(input("How many problems do you want? "))

self.correct = 0

def serve(self):

self.send(str(self.how\_many))

for i in range(self.how\_many):

term1 = random.choice(self.tables)

term2 = random.randint(0,12)

ans = term1 \* term2

msg = "%d \* %d = ? " % (term1,term2)

self.send(msg)

user\_ans = int(self.recv(64))

if user\_ans != ans:

self.send("Wrong. %d \* %d = %d" % (term1,term2,ans))

else:

self.send("Correct!")

self.correct = self.correct + 1

msg = "You got %d out of %d correct." % (self.correct, self.how\_many)

self.send(msg)

self.connection.close()

self.socket.close()

fcsrv = Flashcard\_Server("localhost",8092,1)

fcsrv.serve()

Here’s the client, expressed in file fcclient.py. Notice that the first thing it receives from the server is the number of questions to answer.

from socket\_classes import Client

class Flashcard\_Client(Client):

def perform(self):

how\_many = int(self.recv(64))

for i in range(how\_many):

msg = self.recv(1024).strip()

print(msg)

ans = input()

self.send(str(ans))

data = self.recv(1024)

print(data)

data = self.recv(1024)

print(data)

self.socket.close()

fccli = Flashcard\_Client("localhost",8092)

fccli.perform()

**Multithreading in Python**

As we learned in Java, a thread is an operating system mechanism that enables multiple processes to take place seemingly simultaneously. Rather than wait for a process to be completely done, another process that doesn’t depend on the first process to be completed can start and run by the original process side-by-side.

Python enables you to create a thread. Each thread is responsible for doing a particular task.

To access Python’s threading capability, you have to import *threading*.

Once you’ve done that, you have to declare a subclass of threading.Thread:

class myThread (threading.Thread):

In the constructor, you have to call super class’ version of \_\_init\_\_. You can then initialize various other properties of the Thread class. Finally, you need to specify a run function, which is where the work of the thread is done.

You can block the program from proceeding until all the threads you have created have finished. You do that using the join function, just like we did in Java.

In the next example, threading\_example.py, we create a Thread class that will print the current time 5 times, pausing for a delay between each print. The meat of the thread is the run() function. Because it is implemented as a thread, it can run concurrently with other classes that do the same thing.

import threading

import time

class My\_Thread (threading.Thread):

def \_\_init\_\_(self, threadID, name, delay):

super().\_\_init\_\_()

self.threadID = threadID

self.name = name

self.delay = delay

def run(self):

print("Starting " + self.name)

print\_time(self.name, self.delay, 5)

print("Exiting " + self.name)

def print\_time(threadName, delay, counter):

while counter:

time.sleep(delay)

print("%s: %s %d\n" % (threadName, time.ctime(time.time()), counter))

counter -= 1

# Create new threads

thread1 = My\_Thread(1, "Thread-1", 1)

thread2 = My\_Thread(2, "Thread-2", 2)

# Start new Threads

thread1.start()

thread2.start()

thread1.join()

thread2.join()

print("Exiting Main Thread")

**Threads with Locks**

Interestingly, the order in which the threads are processed is unpredictable. To synchronize them, we can use locks. A Lock() object enables a thread to command the undivided attention of the operating system so that it can do something uninterrupted.

You create a thread by calling the thread.Lock() constructor.

You acquire the lock using the Lock class’s acquire() function, and you release it using the Lock class’s release() function. In between calls to acquire() and release(), you perform the task that you want to proceed uninterrupted.

Consider this file, which might be called threading\_with\_locks.py.

import threading

import time

class My\_Thread (threading.Thread):

def \_\_init\_\_(self, threadID, name, delay):

super().\_\_init\_\_()

self.threadID = threadID

self.name = name

self.delay = delay

def run(self):

global threadlock

print("Starting " + self.name)

threadlock.acquire()

print\_time(self.name, self.delay, 5)

threadlock.release()

print("Exiting " + self.name)

def print\_time(threadName, delay, counter):

while counter:

time.sleep(delay)

print("%s: %s %d\n" % (threadName, time.ctime(time.time()), counter))

counter -= 1

threadlock = threading.Lock()

# Create new threads

thread1 = My\_Thread(1, "Thread-1", 1)

thread2 = My\_Thread(2, "Thread-2", 2)

# Start new Threads

thread1.start()

thread2.start()

thread1.join()

thread2.join()

print("Exiting Main Thread")

**Building a multi-threaded server**

Here’s a server that can accept and respond to 5 connections, each run on a different thread:

import threading

import socket

class Server\_Thread(threading.Thread):

def \_\_init\_\_(self,server):

super().\_\_init\_\_()

self.server = server

[self.connection, self.address] = server.accept()

def run(self):

keep\_going = True

while keep\_going:

buf = self.connection.recv(8)

if len(buf) > 0:

buf = buf.decode()

print(buf)

if buf.strip() == "q":

print("Connection closed.")

keep\_going = False

server = socket.socket()

server.bind(("localhost",8092))

server.listen(5) # server socket; max five connections

threads = []

for i in range(5):

threads.append(Server\_Thread(server))

for i in range(5):

threads[i].start()

for i in range(5):

threads[i].join()

And here’s a client that tests it. It sends a random number on one of five randomly chosen connections.

import socket

import random

clients = []

for i in range(5):

client = socket.socket()

client.connect(("localhost",8092))

clients.append(client)

for i in range(100):

index = random.randint(0,4)

clients[index].send(str.encode("%4d%4d" % (index,random.randint(0,100))))

for i in range(5):

clients[index].send(str.encode("%8s" % "q"))