122COM: Talking to databases

David Croft

January 18, 2016

Abstract

This week we are going to be looking at how we can get our code to talk to databases. This week will be taught in Python only.

1 Introduction

The source code files for this lab and the lecture are available at:

```
https://github.com/covcom/122COM_sql.git
```

```
git@github.com:covcom/122COM_sql.git
```

2 Databases

When you want to store data from a program you generally have two choices, flat files or databases. Databases have a lot of advantages over flat files.

- Safety - Data is safe if your software crashes.
- Remote access - Can be accessed over a network.
- Integrity - Aids to ensure that your data remains meaningful.
- Concurrent access - Can be accessed and modified by multiple people at the same time.
- Fault tolerance - Being disconnected half way through a transaction etc.
- Scalability - Can handle large/small amounts of data in a uniform manner. They can handling indexing tasks for you.
- Offload processing - Itty bitty computer can easily access a database containing vast amounts of data.
- Complex queries - Easily write queries that perform complex tasks.

They do, however, add an additional layer of complexity.

Row/record →

id	forename	surname	job
0	Malcolm	Reynolds	Captain
4	Zoe	Washburne	Co-captain
11	Hoban	Washburne	Pilot
23	Kaywinnet	Frye	Mechanic

↑ Column/attribute

Table 1: Example database table "staff".

2.1 Relational databases

Emerged in the 1970s and are made up of tables. Each table is made up of multiple rows and columns. Each row represents a single record in a table. Each column represents a single attribute of the records in a table. All the information in a column will be of the same type (i.e. number, date, string) and format. See table 1 for a visualisation.

So far so good, a database can be imagined as being like a collection of spreadsheets where each table is a different sheet. When you want to interact with the database you will be using Structured Query Language (SQL). SQL is a special purpose language for interacting with relational databases. The use of SQL means that how the information is stored on disk and how you access it are kept separate[1]. The benefit of this is that you can easily use the information stored in a database without having to know the full details of how it is being stored. There are other database query languages but SQL is by far the most common.

Using SQL you can retreive, change, remove and add to the information held in a database. You can join multiple tables together to enhance the information held in one table with the information in another. You can even change the structure of the database, adding and removing tables or changing how the tables function. The full capabilities of SQL are beyond the scope of this module but you will need a basic understanding of the `SELECT` and `INSERT` statements for the rest of this weeks work.

2.2 SQL queries

2.2.1 `SELECT`

The `SELECT` statement is used to select data from a database. If for example we wanted to get all the information held in the table shown in table 1 we would use the following query.

```
SELECT * FROM staff;
```

The `SELECT *` says that we want every attribute `FROM` the table `staff`. This query will return the whole table shown in table 1.

#	id	forename	surname	job
1	0	Malcolm	Reynolds	Captain
2	4	Zoe	Washburne	Co-captain
3	11	Hoban	Washburne	Pilot
4	23	Kaywinnet	Frye	Mechanic

[1] Data independence principal.

Of course we're not always going to be interested in every attribute, we can select only those attribute we actually care about by replacing the * with the names of the attributes.

```
SELECT id, job FROM staff;
```

#	id	job
1	0	Captain
2	4	Co-captain
3	11	Pilot
4	23	Mechanic

2.2.2 COUNT

If we want to know how many records we have then we can use the query:

```
SELECT COUNT(*) FROM staff;
```

#	COUNT(*)
1	4

2.2.3 WHERE

We can also use SQL to only select records that match certain criteria using the WHERE clause and a conditional.

```
SELECT * FROM staff WHERE job = 'Captain';
```

#	id	forename	surname	job
1	0	Malcolm	Reynolds	Captain

The WHERE cause conditionals follow the same rules as Python. Multiple causes can be combined using the AND, OR, =, != etc are all usable.

```
SELECT * FROM staff WHERE job = 'Captain' OR job = 'Pilot';
```

#	id	forename	surname	job
1	0	Malcolm	Reynolds	Captain
2	11	Hoban	Washburne	Pilot

2.2.4 INSERT

If we want to add more records to a table we need to use an INSERT INTO statement. There are two main formats for this type of statement.

The first is to supply values for all the attributes in the value in the order that they appear in the table.

```
INSERT INTO staff
VALUES (42, 'Simon', 'Tam', 'Doctor');
```

The second way is to specify which attributes you want to supply values for and in what order. This approach has the advantage that you only need to deal with the attributes that you care about.

```
INSERT INTO staff (forename, id, surname)
VALUES ('River', 43, 'Tam');
```

If we run a `SELECT` on the staff table now we would get an updated table.

#	id	forename	surname	job
1	0	Malcolm	Reynolds	Captain
2	4	Zoe	Washburne	Co-captain
3	11	Hoban	Washburne	Pilot
4	23	Kaywinnet	Frye	Mechanic
5	42	Simon	Tam	Doctor
6	43	River	Tam	

3 SQLite

SQLite is an Application Programming Interface (API) masquerading as a relational database. Unlike most database systems when you have a database server and multiple programs can connect to it as clients, SQLite is fully embedded inside your software. This has the advantage of not needed a separate database process and not needing any configuration. It has the disadvantage that it's badly suited to having multiple clients working with the same data, limited performance and having a smaller feature set than most 'full' databases.

3.1 Pokémon

Overall SQLite is a good intermediate step between flat files and a full database solution. We are going to be using it here because it's quick and easy to get started with. Listing **??** on page ?? shows a very simple program to query a SQLite 'database'. Python makes database access pretty painless by supplying a standard database interface. The same code can be used to access any supported database e.g. SQLite, MySQL, SQlite, MS-SQL and Oracle.

We are going to be using the Pokedex database created by Alex Munroe[2] during the labs. Hopefully most of you will already be familiar with the subject matter. If you want to look at the structure of this database of the data contained within it then I recommend that you investigate the SQLite Studio tool `http://sqlitestudio.pl/`.

Pre-lab work:

1. Download the files from the github repo.
2. Run the pre_count program.
    - Read through the text below and make sure that you understand what the code is doing.

Having run the code the following sections explain exactly what the code is doing.

Imports the sqlite3 module as gives it the alias of sql. The alias isn't a requirement but it does make things easier if we move to a different database in the future.

```
import sqlite3 as sql
```

[2] `https://github.com/veekun/pokedex`

Opens the connection to the database. Because we are only using SQLite in this example we supply a database file rather than the network location that a 'full' database would use.

```
con = sql.connect(sqliteFile)
```

Creates a cursor on this connection. The cursor is used to execute SQL commands/-queries and to go through the rows of any results that we get back from our queries.

```
cur = con.cursor
```

Executes our SQL query. The first parameter for `execute()` is our SQL query and is just a python string. It is enclosed in triple quotes in this example as this allows it to cross over multiple lines. This line is discussed further in section 3.2.

```
cur.execute('''select count(*)
   from pokemon_species
   where generation_id = ?''', (gen,))
```

Gets the results of the query back from the database. This line does a lot in a single line of code. `cur.fetchone()` gets a single row of results from back from the database. `cur.fetchone()[0]` gets the first column of that row. `int(cur.fetchone()[0])` casts that individual column into an integer.

```
count = int(cur.fetchone()[0])
```

The rest of the code should be self explanatory. The only thing that may be of interest is the `finally` block at the end. If an exception is raised anywhere in our code, the contents of the `finally` block ensures that the database connection will be closed correctly. This is important as otherwise a connection to the database could be left open until it times out, this can cause issues when a large number of programs are trying to use the same database at the same time.

Lab work:

3. Modify the pre_count program so that it returns the names of the Pokémon from each generation rather than just the count.
   - It may be helpful to know that the `pokemon_species` table has an attribute called `identifier`.
   - When you have multiple records being returned you can use `for row in cur:` to iterate over them instead of using `cur.fetchone()`.

3.2 Placeholders

Notice that in the listing **??** program we didn't include the generation number inside our SQL query, instead of a number there is a ?. This is known as a placeholder. When the query runs,

that placeholder will be replaced by the contents of the second parameter, in this case the value of `gen`. This allows us to modify our queries using the values of our Python variables. If you want to pass multiple values into your query then you can have multiple placeholders. Just make sure that the number of variables in the second `execute()` parameter matches the number of placeholders ?s in the query string.

3.2.1 SQL injection

There is a common mistake made when writing software that works with databases by programmers that don't know better. When writing SQL queries that need to use different values, these programmers use the normal string functions to create their SQL queries. An example of this can be seen in the lab_inject file.

Lab work:

4. Run the lab_inject code.
5. What is the program output when you try entering the names of various Pokémon?
   - If you never played Pokémon then try using "Pikachu", "Bulbasaur" and "Bob".
6. Try entering the following. Make sure you enter this exactly, `") or 1=1; --`
   - What is happening here?

What task 6 has just demonstrated is SQL injection. Because our code does not check what the user has written, they are able to enter text that modifies the SQL commands/queries that our program runs. This is a **MAJOR** security flaw, you must never, ever pass user entered text straight into a SQL query. The way to protect against SQL injection is to sanitise our database inputs. This is done for us as long as we use placeholders.

Lab work:

7. Fix the lab_inject code to prevent this attack.

4 Altering data

From section 2.2.4 on page 3 we know that we can add new records to database tables. Bearing in mind what we learnt about SQL injection in section 3.2.1 it should be clear that we need to use placeholders in our `INSERT INTO` queries.

The Python syntax the examples statements we saw in section 2.2.4 would, therefore, look like this:

```
cur.execute('''INSERT INTO staff
                VALUES (?, ?, ?, ?);''', (42, 'Simon', 'Tam', 'Doctor') )

cur.execute('''INSERT INTO staff (forename, id, surname)
                VALUES (?, ?, ?);''', ('River', 43, 'Tam') )
```

But what happens when we want to insert multiple rows? Yes we can iterate over each record and insert them separately but this proves to be very slow when we're inserting lots of records. Fortunately Python has another function called `executemany()`.

Lab work:

8. Write a program that allows you to add new crew members based on user input.
   - These tasks use the firefly database rather than the pokedex one.
   - The lab_insert file has been provided to get you started.
   - You may find it useful to look at the structure and data of the database using SQLite Studio tool mentioned earlier.
   - Make sure that the new crew members are appearing in the database.
9. Write a program that can read in from a file and add multiple individuals to the crew in one go.
   - An example file has been provided in newcrew.csv but feel free to create your own.

Extended work:

- See if you can write a program that can tell you if a Pokémon evolved from another Pokémon and, if so, which one.
- Can you get your code to print out an entire evolution sequence in reverse order?
  - I.e. Venusaur ← Ivysaur ← Bulbasaur ?
  - What parts of your code need to be Python and what parts need to be SQL?