Programming with Databases

Objectives

- Write short programs that execute SQL queries.
- Trace the execution of a program that contains an SQL query.
- Explain why most database applications are written in a general-purpose language rather than in SQL.

To close, let's have a look at how to access a database from a general-purpose programming language like Python. Other languages use almost exactly the same model: library and function names may differ, but the concepts are the same.

Here's a short Python program that selects latitudes and longitudes from an SQLite database stored in a file called `survey.db`:

```
import sqlite3
connection = sqlite3.connect("survey.db")
cursor = connection.cursor()
cursor.execute("select site.lat, site.long from site;")
results = cursor.fetchall()
for r in results:
    print r
cursor.close()
connection.close()
```

```
(-49.85, -128.57)
(-47.15, -126.72)
(-48.87, -123.4)
```

The program starts by importing the `sqlite3` library. If we were connecting to MySQL, DB2, or some other database, we would import a different library, but all of them provide the same functions, so that the rest of our program does not have to change (at least, not much) if we switch from one database to another.

Line 2 establishes a connection to the database. Since we're using SQLite, all we need to specify is the name of the database file. Other systems may require us to provide a username and password as well. Line 3 then uses this connection to create a cursor (../../gloss.html#cursor); just like the cursor in an editor, its role is to keep track of where we are in the database.

On line 4, we use that cursor to ask the database to execute a query for us. The query is written in SQL, and passed to `cursor.execute` as a string. It's our job to make sure that SQL is properly formatted; if it isn't, or if something goes wrong when it is being executed, the database

will report an error.

The database returns the results of the query to us in response to the `cursor.fetchall` call on line 5. This result is a list with one entry for each record in the result set; if we loop over that list (line 6) and print those list entries (line 7), we can see that each one is a tuple with one element for each field we asked for.

Finally, lines 8 and 9 close our cursor and our connection, since the database can only keep a limited number of these open at one time. Since establishing a connection takes time, though, we shouldn't open a connection, do one operation, then close the connection, only to reopen it a few microseconds later to do another operation. Instead, it's normal to create one connection that stays open for the lifetime of the program.

Queries in real applications will often depend on values provided by users. For example, this function takes a user's ID as a parameter and returns their name:

```
def get_name(database_file, person_ident):
    query = "select personal || ' ' || family from Person where ident
='" + person_ident + "';"

    connection = sqlite3.connect(database_file)
    cursor = connection.cursor()
    cursor.execute(query)
    results = cursor.fetchall()
    cursor.close()
    connection.close()

    return results[0][0]

print "full name for dyer:", get_name('survey.db', 'dyer')
```

```
full name for dyer: William Dyer
```

We use string concatenation on the first line of this function to construct a query containing the user ID we have been given. This seems simple enough, but what happens if someone gives us this string as input?

```
dyer'; drop table Survey; select '
```

It looks like there's garbage after the name of the project, but it is very carefully chosen garbage. If we insert this string into our query, the result is:

```
select personal || ' ' || family from Person where ident='dyer'; drop tale Surve
y; select '';
```

If we execute this, it will erase one of the tables in our database.

This is called an SQL injection attack (../../gloss.html#sql-injection-attack), and it has been used to attack thousands of programs over the years. In particular, many web sites that take data from users insert values directly into queries without checking them carefully first.

Since a villain might try to smuggle commands into our queries in many different ways, the

safest way to deal with this threat is to replace characters like quotes with their escaped equivalents, so that we can safely put whatever the user gives us inside a string. We can do this by using a prepared statement (../../gloss.html#prepared-statement) instead of formatting our statements as strings. Here's what our example program looks like if we do this:

```
def get_name(database_file, person_ident):
    query = "select personal || ' ' || family from Person where ident
=?;"

    connection = sqlite3.connect(database_file)
    cursor = connection.cursor()
    cursor.execute(query, [person_ident])
    results = cursor.fetchall()
    cursor.close()
    connection.close()

    return results[0][0]

print "full name for dyer:", get_name('survey.db', 'dyer')
```

```
full name for dyer: William Dyer
```

The key changes are in the query string and the `execute` call. Instead of formatting the query ourselves, we put question marks in the query template where we want to insert values. When we call `execute`, we provide a list that contains as many values as there are question marks in the query. The library matches values to question marks in order, and translates any special characters in the values into their escaped equivalents so that they are safe to use.

Challenges

1. Write a Python program that creates a new database in a file called `original.db` containing a single table called `Pressure`, with a single field called `reading`, and inserts 100,000 random numbers between 10.0 and 25.0. How long does it take this program to run? How long does it take to run a program that simply writes those random numbers to a file?
2. Write a Python program that creates a new database called `backup.db` with the same structure as `original.db` and copies all the values greater than 20.0 from `original.db` to `backup.db`. Which is faster: filtering values in the query, or reading everything into memory and filtering in Python?

Key Points

- We usually write database applications in a general-purpose language, and embed SQL queries in it.
- To connect to a database, a program must use a library specific to that database manager.
- A program may open one or more connections to a single database, and have one or more cursors active in each.
- Programs can read query results in batches or all at once.