EMBEDDING SQL

CS-A1153 - Databases (Summer 2020)

LUKAS AHRENBERG

Aalto University
School of Science

INTERFACING A RELATIONAL DATABASE

- Why?
  - Database is only one part of an application
  - The rest of your code is in some language
  - Don't want the user to have to know SQL
  - (basic) SQL is not Turing complete
- Can be done for instance by
  - Embedding SQL statements in code
  - Object-Relational Mapping (ORM)

SQL AS STRINGS

- SQL statements are built up as strings in native code and sent to the DB
- Results are read back to program variables
- Benefits: High level of control
- Drawbacks: Can be dependent on SQL dialect, datatype conversion

EMBEDDED SQL

- Mixing SQL with native code
- A pre-processor translates SQL into native code (library calls) at compile-time
- Benefits: High level of control, mixing of SQL and native code via shared variables
- Drawbacks: Pre-processing, dependency on SQL dialect

OBJECT-RELATIONAL MAPPING (ORM)

Is a programming technique that allows the programmer to write object-oriented programs that take advantage of underlying relational database.

- Tables, rows, and queries are not explicit
- Instead the ORM library/tool maps *object instances* to and from the database by creating queries
- Benefits: abstraction, no need to know SQL or (sometimes) relational concepts
- Drawbacks: efficiency, tuning; good mapping not always possible

(Example: SQLAlchemy for Python)

SQLITE3 AND PYTHON

Interfacing with the same SQLite database files as we have been using in the exercises can be done in Python using the core module sqlite3.

SQLite3 is used by many applications as DB back-end as it is file based, open, and not reliant on a full scale Database Management Server. This means that it is easy to set up and treat like a shred library in application development. (However, remember that for other use-cases, a full DBMS is what you need.)

Examples have been written in Python 3.8. (Try them!)

SQLITE3 AND PYTHON - BASICS

Allows for interfacing with an SQLite database by sending SQL statements as strings.

- Need to
  - **connect** to the database file
  - Get a **cursor**
  - Execute SQL statement by **cursor** object

CONNECTION AND CURSOR

```
import sqlite3
# Create connection to file name, given as string
# Note creates database file if not existing!
# Using ":memory:" as file name will create a
# DB in RAM. Good for testing.
connection = sqlite3.connect("my_books.db")
# Get cursor
cur = connection.cursor()
```

EXECUTING, COMMITTING, CLOSING

```
import sqlite3
connection = sqlite3.connect("my_books.db")
cur = connection.cursor()

# Create a table
cur.execute("CREATE TABLE Books(author, title, shelf);")
# And insert some data
cur.execute("""INSERT INTO Books
                VALUES ('William Gibson',
                   'Distrust That Particular Flavor',
                   'G12');""")
# Without a commit, no changes will be made!
connection.commit()
# When you no longer needs to interface to the DB it is good
# to close the connection
connection.close()
```

EXECUTING MANY

```
import sqlite3
connection = sqlite3.connect("my_books.db")
cur = connection.cursor()

# Lets say I have my data in Python
books = [('Bill Bryson',
          'A Short History of Nearly Everything',
          'B1'),
         ('Terry Pratchett',
          'Equal Rites',
          'P4'),
         ('Neil Gaiman',
          'Neverwhere',
          None)]
# One new insert will be created for every value
cur.executemany("INSERT INTO Books VALUES (?,?,?);", books)

connection.commit()
connection.close()
```

None in Python is mapped to **NULL** in SQL.

GETTING DATA TO PYTHON AFTER SELECT

To get the result of a statement we can use the cursor method `fetchall`:

```
import sqlite3
connection = sqlite3.connect("my_books.db")
cur = connection.cursor()

# This only executes the statement
cur.execute("SELECT * FROM Books;")
# To get ALL the data, use fetchall
# This will give you a list of tuples
tuples = cur.fetchall()
# Print the list of tuples
print(tuples)

connection.close()
```

```
[('William Gibson', 'Distrust That Particular Flavor', 'G12'), ('Bill Bryson', 'A Short History of Nearly Everything', 'B1'), ('Terry Pratchett', 'Equal Rites', 'P4'), ('Neil Gaiman', 'Neverwhere', None)]
```

But, what if the data to fetch consists of a **huge** number of entries? Fetching all is not always a good idea.

FETCHING *SOME* DATA

To get a back of results we can use the cursor method `fetchmany`:

```
import sqlite3
connection = sqlite3.connect("my_books.db")
cur = connection.cursor()

# This only executes the statement
cur.execute("SELECT * FROM Books;")
# fetchmany gets a one or more values from the result
batch =  cur.fetchmany(3)
while len(batch) != 0:
    print(f"This batch: {batch}")
    batch = cur.fetchmany()
connection.close()
```

FETCHING ONE TUPLE AT A TIME

To get one tuple at a time we can use the cursor method `fetchone`:

```
import sqlite3
connection = sqlite3.connect("my_books.db")
cur = connection.cursor()

# This only executes the statement
cur.execute("SELECT * FROM Books;")
# Pick one tuple at a time
row =  cur.fetchone()
while None != row:
    print(f"Tuple: {row}")
    row = cur.fetchone()
connection.close()
```

ITERATING OVER CURSOR

The concept of a database *cursor* is similar to an iterator, and it can be used as such in Python:

```
import sqlite3
connection = sqlite3.connect("my_books.db")
cur = connection.cursor()

# This only executes the statement
cur.execute("SELECT * FROM Books;")
# For loop over result
for row in cur:
    # Each row is a tuple
    print(f"Tuple: {row}")

connection.close()
```

SECURITY - SQL INJECTION

This is **very important**, and affects any application designed to interface with databases

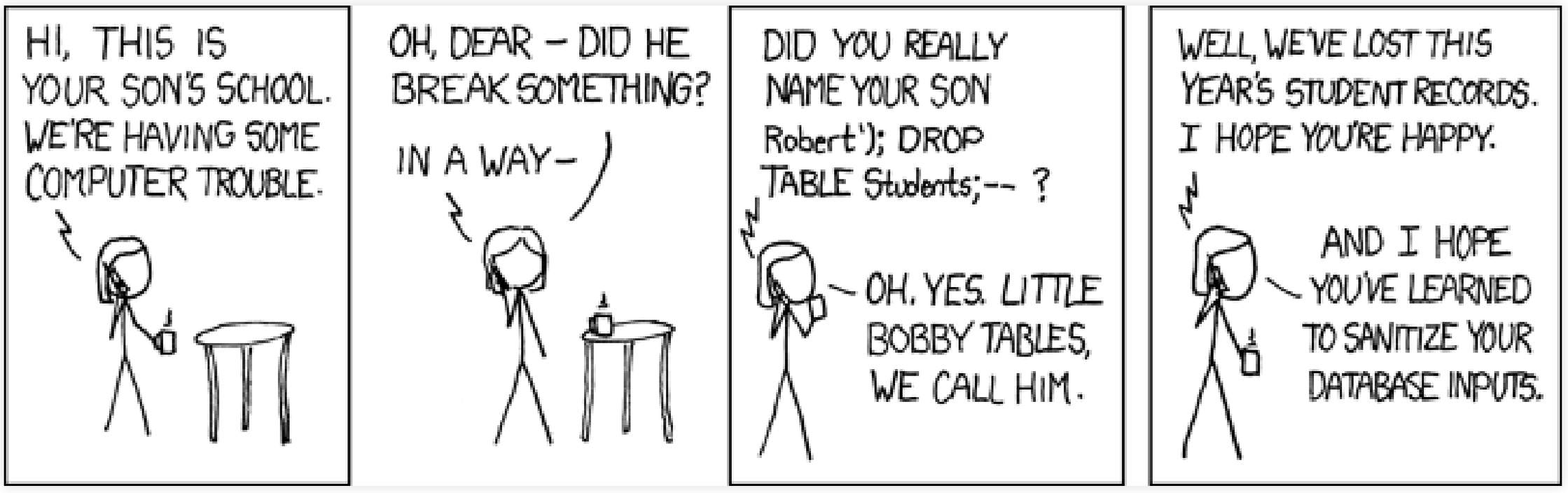
Common pattern:

1. User inputs something
2. Software builds SQL query
3. Executes query
4. Presents result to user

Dynamic construction of query based on user input. But what if the user inputs SQL code?

This is called **SQL injection**

SQL INJECTION - MANDATORY REFERENCE



(CC BY-NC 2.5 Randall Munroe https://xkcd.com/327/)

SQL INJECTION - AVOID IT

- **NEVER** insert the values of program variables by standard string concatenation
- **ALWAYS** use sqlite3 `execute` parameter substitution (or similar for other libraries/languages)
    - Put a '?' everywhere where you want to substitute, and then give a tuple of values

SMALL INTERACTIVE PROGRAM

```
import sqlite3
connection = sqlite3.connect("my_books.db")
cur = connection.cursor()
author = input("Author: ")
title = input("Title: ")
cur.execute("""SELECT * FROM Books
               WHERE author = ? AND title = ?;"""
            , (author,title))
book = cur.fetchone()
if None == book:
    print("No such book.")
elif None == book[2]:
    print("No shelf for book.")
else:
    print(f"Shelf: {book[2]}")
connection.close()
```

DEALING WITH ERRORS

If something goes wrong with the call to sqlite3 an *exception* will be raised in your Python program. The type of exception depends on the error, but all errors raised by sqlite3 is subclasses of `sqlite3.Error` (you can read about them here).

EXCEPTIONS EXAMPLE PROGRAMMING ERROR

```
import sqlite3
connection = sqlite3.connect("my_books.db")
cur = connection.cursor()
shelf = input("Shelf: ")
cur.execute("SELECT name from Books WHERE shelf = ?;",
            (shelf,))
authors_on_shelf = cur.fetchall()

if 0 == len(authors_on_shelf):
    print("That shelf is empty")
else:
    print("Authors on that shelf:")
    for name in authors_on_shelf:
        print(name[0])

connection.commit()
connection.close()
```

```
Traceback (most recent call last):
  File "shelf.py", line 5, in <module>
    cur.execute("SELECT name from Books WHERE shelf = ?;", (shelf,))
sqlite3.OperationalError: no such column: name
```

CATCHING EXCEPTIONS

The most coarse grained treatment is a try/except around everything.

```
import sqlite3
try:
    connection = sqlite3.connect("my_books.db")
    cur = connection.cursor()
    shelf = input("Shelf: ")
    cur.execute("SELECT name from Books WHERE shelf = ?;",
                (shelf,))
    authors_on_shelf = cur.fetchall()

    if 0 == len(authors_on_shelf):
        print("That shelf is empty")
    else:
        print("Authors on that shelf:")
        for name in authors_on_shelf:
            print(name[0])
    connection.close()
except sqlite3.Error as e:
    print("An error occurred while accessing the database.")
    if None != connection:
        connection.close()
```

ISSUES WITH INPUT

```
import sqlite3, csv
connection = sqlite3.connect("my_books.db")
cur = connection.cursor()
# Open file of comma separated values
with open("booklist.csv") as f:
    # Parse it as comma-separated values.
    books = csv.reader(f)
    # For every row, insert into database
    cur.executemany("""INSERT INTO BOOKS
                       VALUES (?,?,?);""",
                    books)
connection.commit()
connection.close()
```

ISSUES WITH INPUT, CONTINUED

Assume `booklist.csv` contains the following lines

```
Rebecca Goldberger Newstein, Plato at the Googleplex, N3
Ursula K, LeGuin, The Left Hand of Darkness, L1
```

Executing the previous program will result in:

```
sqlite3.ProgrammingError: Incorrect number of bindings supplied.
The current statement uses 3, and there are 4 supplied.
```

- Lessons:
  - Check your input before SQL call if possible
  - Yet, don't exclude the possibility of SQL errors (you can't check everything)
  - Have a plan for how to deal with errors at different points