Accessing databases from other languages

Reading / References

- SQLAlchemy Expression Language Tutorial[1]
- SQLAlchemy Core[2]
- SQL Injection[3]
- SQLAlchemy Describing Databases[4]
- SQLAlchemy Column and Data Types[5]

Notes

SQL Injection

We often need to communicate with databases from within a language like Python. One way is to form an SQL query and then communicate with the database server to submit that query. There are many reasons to not do that. One of these is the various forms of **SQL injection**, that have catastrophic consequences and are harder to guard against when writing your own query code.

Let's consider a simple example. We have someone type in their name in a web form, then query the database about their information. We may have in mind a query like:

```
SELECT * FROM users WHERE name = 'usernameHere';
```

Our script is in Python, and we'll need to create that string. We may do something like this:

```
username = .... # We've read the username from a webpage. User provided it
query = "SELECT * FROM users WHERE name ='" + username + "';"
```

If we are not careful, the provided username might be something like: `'; DROP TABLE users; --`. In that case the query we are sending to SQL would be:

```
SELECT * FROM users WHERE name = ''; DROP TABLE users; --';
```

This would effectively execute the DROP TABLES command on the database, deleting our entire database. You must always take care to "clean up" your input and not blindly feed it into an SQL query. This is called **sanitizing**.

> Always sanitize user input!

This is easier to do when you use built-in libraries for database access. We will learn exactly one such library for Python, called SQLAlchemy. But we would be remiss if we didn't first link to this awesome and relevant xkcd comic:

[1] https://docs.sqlalchemy.org/en/latest/core/tutorial.html
[2] https://docs.sqlalchemy.org/en/latest/core/index.html
[3] https://en.wikipedia.org/wiki/SQL_injection
[4] http://docs.sqlalchemy.org/en/latest/core/metadata.html
[5] http://docs.sqlalchemy.org/en/latest/core/types.html

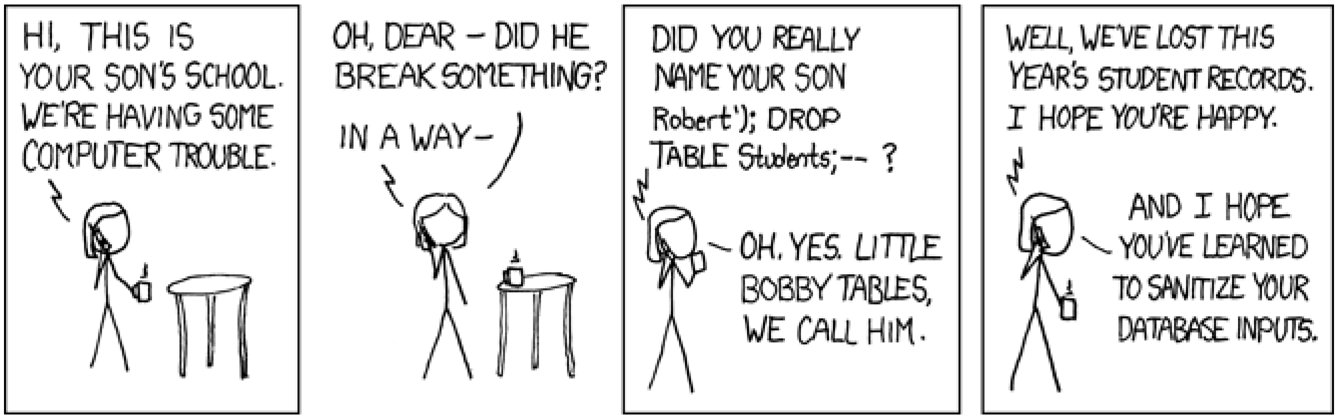
HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Figure 1: Sanitize your inputs!

SQLAlchemy

There are many different libraries to use in order to interface with SQL databases. We will see the basics of SQLAlchemy in this section. SQLAlchemy has two main user-facing components:

SQLAlchemy ORM This offers a way to link objects directly to database tables, and work with Python as if we only had normal objects around. We will examine this later.

SQLAlchemy Core Core offers more direct access to the database, including an SQL Expression Language interface that allows us to write queries in Python. The huge advantage of this is that we can let the library worry about sanitizing the input, and we can also more easily create dynamic queries.

We will spend this section looking at parts of the core, and specifically the expression language. We will use this also as an opportunity to revisit the Twitter API and store tweets in the database.

Connecting Typically an interaction of SQLAlchemy with a database involves a number of steps:

- Setting up the database parameters so that SQLAlchemy can properly authenticate with the database. In SQLAlchemy this is typically called “setting up an engine”.
- Setting up the **metadata**, i.e. telling SQLAlchemy about the various tables that exist in the database and how they should relate to Python entities. This can be omitted if the tables already exist, but is important if the tables must be created by SQLAlchemy
- Using the **engine** object to establish a **connection** with the database
- **Preparing** an SQL query.
- **Executing** an SQL query
- Processing the query **results**.

We start by importing SQLAlchemy and setting up the database engine:

```
from sqlalchemy import *
## Reading database keys
import json
with open('keys.json', 'r') as f:
   vault = json.loads(f.read())['vault']

engineString = 'mysql+mysqldb://{username}:{password}@{server}/{schema}'
engineUrl = engineString.format(**vault)  # Learn about * and ** !!!

# Establishing a specific database connection
engine = create_engine(engineUrl, echo = True)
# Now engine can be used to interact with the database
```

This engine will be our connection to the database.

In order to make the above work, you will need to add some appropriate entries to the keys.json file. Your values will be different of course:

```
   "vault": {
      "username": "skiadas",
      "password": "....",
      "server": "vault.hanover.edu",
      "schema": "skiadas"
   }
```

To make sure that the engine is set up properly, the following will create an actual connection to the database:

```
conn = engine.connect()
conn
```

Creating or specifying tables In order to do further work with the database, we need to describe the tables (If we were using an existing database, can also let the system "infer" the table structure from the database).

Let us first discuss in SQL terms what we want. Let's consider the student enrollments tables we have been using. Here's how those were defined:

```
CREATE TABLE students (
     id      INT  UNIQUE   NOT NULL AUTO_INCREMENT,
     login VARCHAR(20) UNIQUE NOT NULL,
     first VARCHAR(20),
     last  VARCHAR(20),
     credits INT DEFAULT 0,
     gpa      DOUBLE DEFAULT 0,
     PRIMARY KEY (id)
);

CREATE TABLE courses (
     id       INT  UNIQUE   NOT NULL AUTO_INCREMENT,
     prefix CHAR(4) NOT NULL,
     no       INT NOT NULL,
     title   VARCHAR(55) NOT NULL,
     credits INT NOT NULL DEFAULT 4,
     UNIQUE KEY fullCode (prefix, no),
     PRIMARY KEY (id)
 );

 CREATE TABLE enrollments (
     student_id INT NOT NULL,
     course_id INT NOT NULL,
     letter_grade  CHAR(2) DEFAULT NULL,
     point_grade   DOUBLE DEFAULT NULL,
     FOREIGN KEY (student_id) REFERENCES students(id) ON DELETE CASCADE,
     FOREIGN KEY (course_id)  REFERENCES courses(id) ON DELETE CASCADE,
     PRIMARY KEY (student_id, course_id)
 );
```

This is how we would create these tables in MySQL. These should hopefully be familiar to you by now. We will instead learn how to describe the same information in SQLAlchemy.

One of the challenges we will encounter is the AUTO_INCREMENT part. SQLAlchemy tries to be “implementation agnostic”. So you write your code once and it runs on all databases. But not all databases have the auto-increment feature. Some don’t have anything like it, while others like Firebird use a SEQUENCE system.

Such information is referred to as **metadata**. We start with creating a “Metadata object” with the Metadata constructor, then use the Table and Column methods to add specifications:

```
metadata = MetaData()
metadata.bind(engine)

tblStudents = Table('en_students', metadata,
   Column('id', Integer, primary_key = True),
   Column('login', String(20), unique=True, nullable=False),
   Column('first', String(20)),
   Column('last', String(20)),
   Column('credits', Integer, default=0),
   Column('gpa', Float(precision=32), default=0)
)

tblCourses = Table('en_courses', metadata,
   Column('id', Integer, primary_key=True),
   Column('prefix', String(4), nullable=False),
   Column('no', String(20), nullable=False),
   Column('title', String(55), nullable=False),
   Column('credits', Integer, nullable=False, default=4),
   UniqueConstraint('prefix', 'no', name="fullCode")
)

tblEnrollments = Table('en_enrollments', metadata,
   Column('student_id', Integer,
      ForeignKey("en_students.id", ondelete="CASCADE"),
      nullable=False),
   Column('course_id', Integer,
      ForeignKey("en_courses.id", ondelete="CASCADE"),
      nullable=False),
   Column('letter_grade', String(2)),
   Column('point_grade', Float(32)),
   PrimaryKeyConstraint('student_id', 'course_id')
)

# drop the tables if they existed already. Don't always need this.
metadata.drop_all(engine)

# Create these tables if they do not exist
metadata.create_all(engine)
```

Inserting data Now let's look into creating some students. Typically the steps for inserting new values would be:

1. Make sure you have a **connection** object.
2. Create a dictionary object containing the values of the tuple you want to insert, or create a table of such objects for inserting multiple values.
3. Create an **insert object** for the table you want to insert in.
4. Use the connection to **execute** an insert using the insert object and the values.

So let's take a look at how this might look in our case:

```
conn = engine.connect()   # Only if it doesn't already exist
users = [
    { "login": "somebodyj1", "first": "Joe", "last": "Somebody" },
    { "login": "somebodyj2", "first": "Joel", "last": "Somebody" },
    { "login": "otherp1", "first": "Peter", "last": "Other" },
    { "login": "otherm1", "first": "Mary", "last": "Other" },
    { "login": "doem1", "first": "Mary", "last": "Doe" },
    { "login": "doep1", "first": "Peter", "last": "Doe" },
    { "login": "doed1", "first": "David", "last": "Doe" }
]
ins = tblStudents.insert()  # Create an "insert object"
result = conn.execute(ins, users)    # Execute the insert on a connection
dir(result)          # examine what properties the result object has
result.rowcount      #  7
```

Let's do the same for the courses:

```
courses = [
    { "prefix": "MAT", "no": 121, "title": "Calculus␣1" },
    { "prefix": "CS", "no": 220, "title": "Intro␣to␣CS" },
    { "prefix": "MAT", "no": 122, "title": "Calculus␣2" },
    { "prefix": "MAT", "no": 221, "title": "Calculus␣3" },
    { "prefix": "CS", "no": 223, "title": "Data␣Structures" }
]
ins = tblCourses.insert()
result = conn.execute(ins, courses)
```

We will discuss the INSERT-SELECT variant later. Let us now turn to querying the data.

Querying the data Now let us discuss how we can query the database for information from within SQLAlchemy. We start with some basic queries. For example, let's see how we would do a basic "select all" query:

```
SELECT * FROM students;
```

In SQLAlchemy, you would break this into steps:

1. Prepare a **select** object for one or more tables.
2. Add any extra **conditions** to that object.
3. **Execute** the object on an active **connection** object.
4. **Process** the results as you would process a list.

Let's take a look:

```
s = select([tblStudents])
result = conn.execute(s)
result.fetchall()    # A list of tuples
# The result object is enumerable
result = conn.execute(s)
for student in result:
    print(student)

result.close()    # Done using it
```

One important thing to notice is that the result object acts as what is called a **DBAPI cursor**[6]: It offers you an iterative pattern over the results set, but once you process the set it then closes and is not available again. You need to execute a new query to process the list a second time, unless you stored the result of the iteration in some way (for example stored the result of result.fetchall).

Another really important thing is that while the results are tuples, they are actually named tuples:

```
result = conn.execute(s)
student = result.fetchone()  # Just grabbing one match
student.keys()      # all the keys
student['login']    # value for key "login"
student.login       # also works
```

Let's now do a specific query for some fields/columns only:

```
SELECT last, first FROM students;
```

In SQLAlchemy that might look like so:

```
s = select([tblStudents.c.last, tblStudents.c.first])
result = conn.execute(s)
result.fetchall()
result = conn.execute(s)
for last, first in result:
    print(last + ",␣" + first)
```

[6]https://www.python.org/dev/peps/pep-0249/#cursor-objects

Modifying the select object

We can modify the select object to add other components. This is usually done by so-called “method chaining”: We add on method calls one after the other, and each one modifies and returns the object. For example we can add a condition for avoiding duplicates (DISTINCT):

```
s = select([tblStudents.c.last]).distinct()
```

Or let’s suppose we want to get all students whose last name is "Somebody". We did this in MySQL via:

```
SELECT first
FROM students
WHERE last = "Somebody";
```

In SQLAlchemy, we would use the where method that can be tacked on to a select object:

```
s = select([tblStudents]).\
     where(tblStudents.c.last=="Somebody")
```

Let’s scale things up! We want to add two conditions:

```
SELECT *
FROM students
WHERE first = "Joe"
AND last = "Somebody";
```

In Python:

```
s = select([tblStudents]).\
     where(tblStudents.c.first=="Joe").\
     where(tblStudents.c.last=="Somebody");
```

Or we can add a GROUP BY clause:

```
SELECT *
FROM students
ORDER BY last, first;
```

In Python:

```
s = select([tblStudents]).\
     order_by(tblStudents.c.last, tblStudents.c.first);
```

Insert with Select

Let’s see how we can do an INSERT querty that uses a SELECT query to determine the values. For example we had the following:

```
INSERT INTO enrollments (student_id, course_id)
SELECT id, 1
FROM students;
```

In Python, this would become:

```
ins = tblEnrollments.insert().\
        from_select([tblEnrollments.c.student_id, tblEnrollments.c.course_id],
                     select([tblStudents.c.id, literal(1)]));
```

Performing joins

We can perform joins in SQLAlchemy in two different ways, just as in MySQL. In MySQL, a join can be made implicitly by combining tables via appropriate where clauses:

```
SELECT s.last, s.first, c.prefix, c.no
FROM students s, enrollments e, courses c
WHERE e.student_id = s.id
AND   e.course_id  = c.id;
```

In Python:

```
s = select([tblStudents.c.last, tblStudents.c.first,
            tblCourses.c.prefix, tblCourses.c.no]).\
        where(tblEnrollments.c.student_id == tblStudents.c.id).\
        where(tblEnrollments.c.course_id == tblCourses.c.id);
```

There is also an alternative way to do multiple where steps, using and_:

```
s = select([tblStudents.c.last, tblStudents.c.first,
            tblCourses.c.prefix, tblCourses.c.no]).\
        where(and_(
            tblEnrollments.c.student_id == tblStudents.c.id,
            tblEnrollments.c.course_id == tblCourses.c.id
        ));
```

We also had another way of performing joins, with the JOIN ... ON construct. Let's look at the same example in that setup:

```
SELECT s.last, s.first, c.prefix, c.no
FROM enrollments e
JOIN students s ON e.student_id = s.id
JOIN courses c ON e.course_id  = c.id;
```

One of the nice things about SQLAlchemy is that it will automatically figure out which fields to compare in the ON portion, by examining the FOREIGN KEY restrictions. So we can omit that. We do however need an extra step to get our join started, via the select_from construct:

```
s = select([tblStudents.c.last, tblStudents.c.first,
            tblCourses.c.prefix, tblCourses.c.no]).\
        select_from(tblStudents.\
                    join(tblEnrollments).\
                    join(tblCourses));
```

We can certainly also do these steps in parts, a value of using Python instead of MySQL directly, and taking advantage of the fact that Python has objects representing the various SQL elements. So for example we can give a name to the triple join and the selected columns:

```
allData = tblStudents.join(tblEnrollments).join(tblCourses);
columns = [tblStudents.c.last, tblStudents.c.first,
            tblCourses.c.prefix, tblCourses.c.no];
s = select(columns).select_from(allData);
```