Accessing databases from other languages

Reading / References

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

Notes

SQL Injection

We often need to communicate with databases from within a language like Python. One ways 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 coming:

[1] http://docs.sqlalchemy.org/en/rel_0_9/core/index.html
[2] https://en.wikipedia.org/wiki/SQL_injection
[3] http://docs.sqlalchemy.org/en/rel_0_9/core/metadata.html
[4] http://docs.sqlalchemy.org/en/rel_0_9/core/types.html

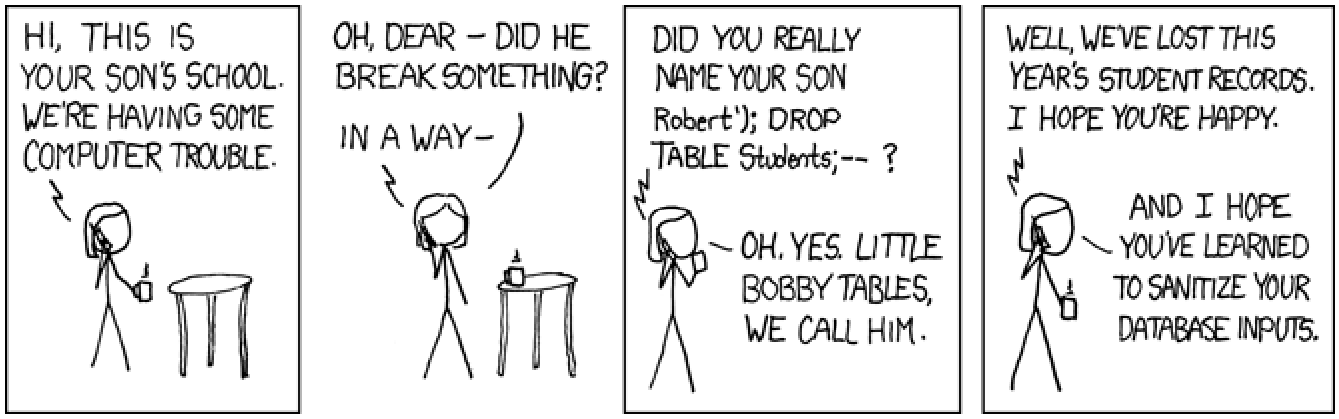
HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE 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 a more direct access to the database, including an SQL Expression Language interface that allows us to write queries in Python.

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 We start by importing SQLAlchemy and contacting the database:

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

engineUrl = ('mysql+mysqldb://' + vault['username'] +
             ':' + vault['password'] +
             '@' + vault['server'] +
             '/' + vault['schema'])

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

Creating or specifying tables In order to do further work with the database, we need to describe the tables. Let us first discuss in SQL terms what we want. We will be looking at some Twitter data. So we might have the following tables:

```
CREATE TABLE tw_users (
     id INT PRIMARY KEY,
     screen_name VARCHAR(140),
     name VARCHAR(140)
);
CREATE TABLE tw_tweets (
     id INT  PRIMARY KEY,
     text VARCHAR(150) NOT NULL,
     created TIMESTAMP NOT NULL,
     user INT NOT NULL,
     CONSTRAINT FK_user FOREIGN KEY (user) REFERENCES tw_users(id)
);
CREATE TABLE tw_mentions (
     tweet_id INT,
     user_id INT,
     PRIMARY KEY (tweet_id, user_id),
     CONSTRAINT FK_mention_tweet FOREIGN KEY (tweet_id) REFERENCES tw_tweets(id),
     CONSTRAINT FK_mention_user FOREIGN KEY (user_id) REFERENCES tw_users(id)
);
CREATE TABLE tw_hashtags (
     tweet_id INT,
     hashtag VARCHAR(140),
     PRIMARY KEY (tweet_id, hashtag),
     CONSTRAINT FK_hashtag_tweet FOREIGN KEY (tweet_id) REFERENCES tw_tweets(id),
);
```

This is how we would create these tables in MySQL. Take a moment to study them and make sure they make sense. We will instead learn how to describe the same information in SQLAlchemy.

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

```
metadata = Metadata()
dbusers = Table('tw_users', metadata,
   Column('id', Integer, primary_key = True),
   Column('screen_name', String(140)),
   Column('name', String(140))
)
dbtweets = Table('tw_tweets', metadata,
   Column('id', Integer, primary_key = True),
   Column('text', String(150), nullable = False),
   Column('created', DateTime(timezone = True), nullable = False),
   Column('user', Integer, ForeignKey('tw_users.id'), nullable = False)
)
dbmentions = Table('tw_mentions', metadata,
   Column('tweet_id', Integer, ForeignKey('tw_tweets.id'), primary_key = True),
   Column('user_id', Integer, ForeignKey('tw_users.id'), primary_key = True)
)
dbhashtags = Table('tw_hashtags', metadata,
   Column('tweet_id', Integer, ForeignKey('tw_tweets.id'), primary_key = True),
   Column('hashtag', String(140), primary_key = True)
)
# Create these tables if they do not exist
metadata.create_all(engine)
```

Inserting data We will now show how we can use SQLAlchemy to add new entries into the database. We will read tweet data as we did in the second assignment. We will then create list comprehensions that extra tweet information, user information and hashtag/mentions from a list of tweets. Finally, we do mass inserts of the created lists into our tables:

```
conn = engine.connect()
from datetime import datetime
time_format = '%a␣%b␣%d␣%H:%M:%S␣+0000␣%Y'

# The following 4 methods obtain lists of values that we would
# want to insert into the database, starting from a list of tweets.
#
# Returns a list of all users (possibly with duplicates)
def getAllUsers(tweets):
   allUsers = ([ tweet['user'] for tweet in tweets ] +
               [  user
                  for tweet in tweets
                  for user in tweet['entities']['user_mentions']
               ])
   return [
      {
         'id': user['id'],
         'screen_name': user['screen_name'],
         'name': user['name']
      }
      for user in allUsers
   ]

# Returns a list of mentions ready to go in database
def getAllMentions(tweets):
   return [
      { 'tweet_id': tweet['id'], 'user_id': user['id'] }
      for tweet in tweets
      for user in tweet['entities']['user_mentions']
   ]

# Returns tweet-hashtag pairs
def getAllHashtags(tweets):
   return [
      { 'tweet_id': tweet['id'], 'hashtag': tag['text'] }
      for tweet in tweets
```

```
        for tag in tweet['entities']['hashtags']
    ]

# Returns simplified tweet entries
def getSimpleTweets(tweets):
    return [
        {
            'id': tweet['id'],
            'created': datetime.strptime(tweet['created_at'], time_format),
            'text': tweet['text'],
            'user': tweet['user']['id']
        }
        for tweet in tweets
    ]

## Processes the tweets
def processTweets(tweets):
    conn.execute(dbusers.insert().prefix_with('IGNORE'), getAllUsers(tweets))
    conn.execute(dbtweets.insert().prefix_with('IGNORE'), getSimpleTweets(tweets))
    conn.execute(dbmentions.insert().prefix_with('IGNORE'), getAllMentions(tweets))
    conn.execute(dbhashtags.insert().prefix_with('IGNORE'), getAllHashtags(tweets))

processTweets(hillary)
processTweets(donald)
```

The relevant SQLALchemy methods are in the `processTweets` function, where `insert` is used to create an INSERT query, then `execute` executes that query in a connection, passing it the lists of values to add.

At the end of this you should have a rich database of tweets to work with.

Querying the data Now that we have a rich dataset, let us try to do some queries. We'll write them in SQL first as practice. Here they are:

1. We want to look at the hashtags list, and see how many times each tag occurs, then order by those counts, starting from the highest.
2. We want to look at how many hashtags each candidate has used. So the resulting table would have only two rows, and two columns for candidate names and tag counts.
3. We want to look at hashtag counts further broken by candidate. So for each candidate and each tag it would list the number of times the candidate used that tag.
4. This is similar to 3, but with a twist: For each hashtag it should show: the hashtag, the number of times Clinton used it, and the number of times Trump used it. For extra challenge, order the hashtags according to the total count amongst both candidates.

Do not read further down until you have worked the above queries in SQL Workbench.

Here is how we would do the first query in SQL:

```
SELECT hashtag, COUNT(*) as no_tweets
FROM tw_hashtags
GROUP BY hashtag
ORDER BY no_tweets DESC;
```

Now we want to do the same query with SQLAlchemy. Here is how it would look like:

```
tag_counts = select([ dbhashtags.c.hashtag,
                      func.count(dbhashtags.c.tweet_id).label('no_tweets')
                    ]).group_by('hashtag').order_by(desc('no_tweets'))
results = conn.execute(tag_counts).fetchall()
```

For the second query, we would have:

```
select u.name, count(*) AS hashtags
FROM tw_hashtags h
JOIN tw_tweets t ON t.id = h.tweet_id
JOIN tw_users u ON u.id = t.user
GROUP BY u.name;
```

And here is the same query in Python:

```
tag_candidates = select([ dbusers.c.name,
                      func.count(dbtweets.c.id).label('no_tweets')
                    ]).select_from(dbhashtags.join(dbtweets).join(dbusers)
                    ).group_by('name').order_by(desc('no_tweets'))
results = conn.execute(tag_candidates).fetchall()
```

And here is the third query:

```
SELECT name, hashtag, COUNT(DISTINCT tweet_id) as no_tweets
FROM tw_hashtags h, tw_tweets t, tw_users u
WHERE h.tweet_id = t.id
AND u.id = t.user
GROUP BY hashtag, u.name
ORDER BY no_tweets DESC;
```

And in Python:

```
tag_and_candidates = select([
    dbusers.c.name, dbhashtags.c.hashtag,
    func.count(dbtweets.c.id).label('no_tweets')
]).where(
    dbusers.c.id == dbtweets.c.user
).where(
    dbtweets.c.id == dbhashtags.c.tweet_id
).group_by('name').group_by('hashtag').order_by(desc('no_tweets'))
results = conn.execute(tag_and_candidates).fetchall()
```

Look at the SQLAlchemy expression tutorial[5] to learn more about what else is available.

[5] http://docs.sqlalchemy.org/en/rel_0_9/core/tutorial.html