Using Python+SQLAlchemy+Elixir to script a database

**Note:** if you are new to Python development, make sure you have [*pip*](https://pypi.python.org/pypi/pip) installed. In addition, I highly recommend you use [*virutalenv*](http://www.virtualenv.org/) to keep your Python environments separate. Read more [*here*](http://www.davidfischer.name/2010/04/why-you-should-be-using-pip-and-virtualenv/).

# SQLAlchemy+Elixir

[SQLAlchemy](http://www.sqlalchemy.org/) is a Python SQL toolkit (the Core) and Object Relationship Mapper (ORM) that aims to ease developer interaction with relational databases in Python. The Core abstracts interaction with a wide variety of relational databases (such as MySQL, Postegres, and SQLite3), while the ORM provides a way to map data inside a relational database to native Python data types, providing an easy way to read and write database tables using native Python code.

[Elixir](http://elixir.ematia.de/trac/wiki) is a thin wrapper on top of SQLAlchemy, providing a simple syntax for mapping Python classes to logical objects in a database to Python objects.

We will be using these two libraries to make it syntactically easy to inspect and edit a database programmatically, essentially making a database scriptable (without actually having to write any SQL)

# Installing dependencies

**Note:** this tutorial assumes you already have a database server set up on your machine.

First of all you should have Python 2.7 installed. Next, if you haven’t already, install pip, for installing and managing Python packages, and virtualenv, for managing Python environments. Start a new virtualenv and we’ll start hacking.

We’ll start by installing SQLAlchemy and Elixir:

$ sudo pip install SQLAlchemy==0.7.8

$ sudo pip install Elixir

*(Elixir is not yet compatible with the latest SQLAlchemy (0.8) so we’ll have to specify a slightly older version here.)*

We’ll also need to install a database driver for Python to interface with MySQL (or whatever database you choose):

$ sudo pip install MySQL-python # for MySQL

$ sudo pip install psycopg2 # for postgres

**Note:** for MySQL and Postgres, you will need a C compiler and associated toolchain in order to build the appropriate Python driver. On Mac OS X this means you will need XCode. On Windows, you will need Visual Studio and its C compiler. If you don’t want to go through the trouble, you can use SQLite.

# Connecting to the database

Boot up your python shell. We’ll start by importing both libraries:

>>> from sqlalchemy import \*

>>> from elixir import \*

For the purpose of this tutorial, I have a MySQL server accepting connections port 5432 (the default) with a database called oldhandbook, owned by a user, bfa with password ‘gtca’.

First we’ll start by setting up our database connection (you can replace the parameters in the metadata.bind string with whatever you need to connect your own database):

>>> metadata.bind = ‘mysql://bfa:gtca@localhost:5432/oldhandbook’

>>> metadata.bind.connect()

<sqlalchemy.engine.base.Connection object at 0x103460f50>

If you get something like <sqlalchemy.engine.base.Connection object at 0x103460f50> back, that means you’ve successfully connected the database!

# Playing with the data

Elixir allows you to interact with the data via Python classes. Generally speaking, we will always create one Python class to correspond to one table in the database.

In my example database, oldhandbook, there is a table called cities, which contains some fields such as name, state, and citywebsite.

We will create a class, Cities, to correspond to this table, and activate it:

>>> # I’m doing this in the Python shell

>>> class Cities(Entity):

... using\_options(tablename='cities', autoload=True)

...

>>> setup\_all()

>>> create\_all()

>>> Cities

<class '\_\_main\_\_.Cities'>

SQLAlchemy and Elixir are typically used to define and create databases (you can see some examples [here](http://docs.sqlalchemy.org/en/rel_0_8/orm/tutorial.html#declare-a-mapping) and [here](http://elixir.ematia.de/trac/wiki/TutorialDivingIn)), in which case we would define some columns for the table. However, since our goal is simply to inspect and edit an existing table, we simply give it using\_options(tablename='cities', autoload=True), telling it to use SQLAlchemy’s autoload feature to populate this Python class with columns from the table.

We can create as many classes as we like, for as many tables in our database. Then we call setup\_all() and create\_all() to read from the databse and create the class-to-table mappings.

**Note:** this is a very bare-bones setup. SQLAlchemy and Elixir are very, **very**, powerful tools, with many options and features. For example, when doing autoloading, you may want to specify many-to-one/many-to-many relationships, and SQLAlchemy will enable you to follow those relationships in native Python. For more information, look into the SQLAlchemy and Elixir documentation.

Now we can start playing with our Cities object. Here are just some examples of the things you can do:

>>> Cities.query.all() # returns all rows in this table

[<\_\_main\_\_.Cities object at 0x1034bde90>, <\_\_main\_\_.Cities object at 0x1034bdf50>, <\_\_main\_\_.Cities object at 0x1034bdfd0>, <\_\_main\_\_.Cities object at 0x1034cd090>, <\_\_main\_\_.Cities object at 0x1034cd150>, <\_\_main\_\_.Cities object at 0x1034cd210>, <\_\_main\_\_.Cities object at 0x1034cd2d0>, <\_\_main\_\_.Cities object at 0x1034cd390>, <\_\_main\_\_.Cities object at 0x1034cd450>...]

>>> Cities.query.first() # returns the first row as object

<\_\_main\_\_.Cities object at 0x1034bde90>

>>> Cities.query.first().name # retrieve fields by name

'Birmingham'

>>> Cities.query.first().state

'AL'

>>> Cities.query.first().citywebsite

'http://www.informationbirmingham.com/'

>>> # easily loop over all row data

>>> for city in Cities.query.all():

... print city.name

...

Birmingham

Flagstaff

Santa Barbara

Stockton

Storrs

Tallahassee

...

>>> # programmatically filter row data

>>> for bigcity in Cities.query.filter(Cities.citypopulation > 100000).all():

... print bigcity.name

...

Birmingham

Stockton

Tallahassee

Indianapolis

Lexington

Lowell

Worcester

Minneapolis

St. Louis

Cincinnati

Chattanooga

Richmond

Vancouver

Lincoln

Guelph-Hamilton

Halifax

Waco

Boston

>>> # search for data

>>> boston = Cities.query.filter\_by(name=u'Boston').one()

>>> boston.state

'MA'

>>> # edit existing data

>>> boston.name = 'Bahston'

>>> # changes don’t get written until you explicitly commit

>>> # this will commit all changes made thus far

>>> session.commit()

>>> # go check your db

>>> # create new rows

>>> berkeley = Cities(name="Berkeley", state="CA")

>>> session.commit()

>>> # SQLAlchemy will automatically create increment the pk