Basic BioSQL with Biopython

Brad Chapman (chapmanb@arches.uga.edu)

February 19, 2003

1 Installing python database packages

In order to access databases from within python, you'll need to install python interfaces to the database you are using. This section describes the interfaces Biopython supports for using BioSQL and basic installation.

These are designed with a UNIX-like platform in mind (Linux, any of the various UNIXes, Mac OS X).

For all of these installations, you'll need to have at least a database client installed, and also need to have the development libraries and hearder files installed. On some systems that use packaging systems (Red Hat, Debian, OS X) this means that you'll need to install a development package (for example, mysql-dev or mysql-headers or mysql-libs) separately. If you start trying to install one of the python interfaces below and start getting errors are missing libraries or header files, you should check to make sure these are installed.

1.1 MySQL

MySQL access with python is fairly simple, as there is basically one adaptor which is regularly developed and which most people use.

1.1.1 MySQLdb

This interface is quite stable and completely DB API-2.0 compliant and seems to definitely be the one to use for MySQL access. The latest releases are available from: http://sourceforge.net/projects/mysql-python.

To install, first download and unpack the tar.gz file:

```
$ tar -xzvpf MySQL-python-0.9.2.tar.gz
MySQL-python-0.9.2/
MySQL-python-0.9.2/MySQLdb/
.....
```

Then change into the directory containing the MySQLdb code:

```
$ cd MySQL-python-0.9.2
```

MySQLdb uses the standard distutils mechanisms for installing programs, so you should be able to build the code with:

```
$ python setup.py build
running build
running build_py
.....
```

The `setup.py` file is pretty good at finding the MySQL libraries and header files on a number of platforms, so this should normally work well (see platform specific notes below for special cases).

Finally, to install the library you'll need to be root on the machine, and do:

```
#python setup.py install
running install
.....
```

To test that everything worked smoothly, you should be able to do:

```
$ python
Python 2.2.2 (#1, 01/12/03, 07:51:34)
[GCC Apple cpp-precomp 6.14] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import MySQLdb
>>>
```

If this works happily, then you should be all set.

Platform specific notes:

Mac OS-X – With the default setup.py file, I get the error:

```
ld: can't locate file for: -lmysqlclient_r
```

The setup.py file is apparently looking for a specially name thread-safe library. To fix this, I changed the line:

```
mysqlclient = thread_safe_library and "mysqlclient_r" or "mysqlclient"
```

to:

```
mysqlclient = thread_safe_library and "mysqlclient"
```

in the setup.py file.

1.2 PostgreSQL

In contrast to MySQL, PostgreSQL has a mess of python interfaces all of which are in varying states of completion and maintenance. The most frustrating problem with many of these is that they do not necessarily support the DB API-2.0 which makes them very difficult to use with standard python SQL access code.

Below are the currently supported interfaces. More could of course be added, but probably won't be unless someone gets very excited about them.

1.2.1 psycopg

psycopg is a nice PostgreSQL python interface developed along with the Zope project. It is available from http://www.zope.org/Members/fog/psycopg, and is fully DP API-2.0 compliant.

psycopg requires you to make the mxDateTime package installed for handling of dates. This comes in the mxBase set of packages which are also required for Biopython, and you can get it from http://www.egenix.com/files/python/eGenix-mx-Extensions.html#Download-mxBASE.

To begin installing, psycopg download the tar.gz file, unpack it, and change into the new directory created:

```
$ tar -xzvpf psycopg-1.0.14.tar.gz
psycopg-1.0.14/
psycopg-1.0.14/AUTHORS
....
$ cd psycopg-1.0.14
```

psycopg uses the autoconf build system, so it follows a configure/make/make install set of steps.

First, run the configuration:

```
$ ./configure
creating cache ./config.cache
checking for python... /sw/bin/python
checking python version... 2.2
....
```

If the configure has trouble finding the postgreSQL or mxDateTime include files ans libraries, a few useful flags to configure are:

`--with-postgres-libraries=DIR`,
`--with-postgres-includes=DIR`,
`--with-mxdatetime-includes=DIR`.

For example:

```
$ ./configure --with-mxdatetime-includes=/sw/lib/python2.2/site-packages/mx/DateTime/mxDateTime
```

is necessary on Mac OS X since the mxDateTime header files are in a unusual location.

After the configure finished happily, you need to make the module:

```
$ make
gcc  -DNDEBUG -O3 -Wall -Wstrict-prototypes  -I/sw/include/python2.2
  -I/sw/lib/python2.2/config -DHAVE_CONFIG_H=1  -DHAVE_LIBCRYPTO=1
  -DHAVE_ASPRINTF=1  -I/sw/include/postgresql -I/sw/include/postgresql/server
  -I/sw/lib/python2.2/site-packages/mx/DateTime/mxDateTime
-DVERSION=\"1.0.14\" -DNDEBUG -D_REENTRANT -D_GNU_SOURCE   -c
././module.c -o ./module.o
....
```

Finally, become root and install the module:

```
# make install
Installing shared modules...
install -m 555 ./psycopgmodule.so /sw/lib/python2.2/site-packages
```

To test that everything worked okay, fire up python and import the newly created module:

```
$ python
Python 2.2.2 (#1, 01/12/03, 07:51:34)
[GCC Apple cpp-precomp 6.14] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import psycopg
>>>
```

No error messages here and you've done everything correctly.

Platform specific notes:

Mac OS-X – I had a couple of things I needed to change to get psycopg to compile happily. First the make complains with:

```
././module.c:156: only 1 arg to macro 'Dprintf' (2 expected)
```

There seems to have been kind of mistake in how this macro was defined for gnu C compilers. To fix the problem, I had to change line 107 in `module.h` from:

```
#define Dprintf(fmt, args...)
```

to:

```
static void Dprintf(const char *fmt, ...) {}
```

After this the make will proceed for a while, and then complains with:

```
ld: can't open: /sw/bin//sw/bin/python
```

This appears to be a glitch in how the Makefile is created. I had to change line 136 from:

```
PYTHON = /sw/bin/python
```

to:

```
PYTHON = python
```

After these two changes, everything compiled fine.

2 Getting Started

Once you've got your python database interface installed you're ready to get started with BioSQL. This section is intented to get you going with the basics in python. For in-depth information about the BioSQL schema itself, you should check out the standard BioSQL documentation which has extensive information about setting up databases, loading SQL and describing what exactly is going on with all those SQL tables. For specific Biopython use cases, see the cookbook items (hopefully to be written) below.

2.1 Prerequisites

Allow of the work in this section assumes that you have installed a database, a python binding to this database, and loaded the BioSQL schema into the database.

Addtionally, this is written with the assumption that you know a little about databases in general. The most important thing that is really assumed is that you understand the client/server model of databases. The basic idea is that there is a database server which actually physically contains the data you are working with. This server can be located on your machine, a machine across the room from you, or some computer half-way around the world. We will not be dealing with the server here – we will be dealing with a client that connects to this server.

Understanding this, for this example we are going to assume we have a server/client set up with the following information:

username – This is the user you are allowed to connect to the database server as. In this example, the username is: **chapmanb**

password – This is the password associated with the above use. For our example, this is: **biopython**

host – This is the hostname where the server computer is located. As mentioned above, this could be anywhere, but for this example we will use: **localhost**

db – The name of the database where the BioSQL schema has been loaded. For our example, this is: `biosql_example`

driver – The python database adaptor that you are using. The available drivers and their installation are described above. Obviously, the driver and your client have to match the database server type you will be dealing with. In our example, we are using a MySQL database with the driver: **MySQLdb**

To run these examples on your own, you'll need to know all of this information and adjust what is written below to get it to work for you.

2.2 Connecting to a BioSQL database

This section describes the basic steps to connect with a BioSQL database. Right now we are not assuming the database needs to be loaded with any information (that comes next), but rather just enumerating the very basic steps for connecting with the database with the biopython implementation of BioSQL.

Okay, now that all the preliminary ramblings are out of the way, let's get started actually doing this. First, we load the Biopython implementation of BioSQL:

```
>>> from BioSQL import BioSeqDatabase
```

Now the second and final step – connect with the database:

```
>>> server = BioSeqDatabase.open_database(driver = "MySQLdb", user = "chapmanb",
...              passwd = "biopython", host = "localhost", db = "biosql_example")
```

There you go – you've got a connection. Hmmm, wonder why I did so much rambling for such a simple thing? So does my psychologist.

Now that we've got the connecting to the database out of the way, let's do some useful work with this database.

2.3 Loading a GenBank file into the database

Now that we've got a connection to our database, the next logical thing to do is load some information into it. For this example, we are going to assume we have a GenBank file on our computer called `cor6_6.gb` that we are going to work with. This is just a file of GenBank sequences, and in this case contains a bunch of cold resistance related genes from various plants.

We want to load this file into the BioSQL server so we can query it. The first thing we want to do is create a BioSQL database on the server to cold this information. In this case, we'll simply call it `cold`. Again, this is a one-liner:

```
>>> db = server.new_database("cold")
```

Now we want to do the loading of the file into the database. The Biopython implementation works by taking a standard Iterator object that returns Biopython SeqFeature objects and then doing the loading. If that last sentence doesn't make any sense at all, then you should go check out the standard Biopython documentation which explains it all in excruciating detail. So, we need to set up our Iterator for our GenBank file, which we can do with the following code:

```
>>> from Bio import GenBank
>>> parser = GenBank.FeatureParser()
>>> iterator = GenBank.Iterator(open("cor6_6.gb"), parser)
```

With this iterator, the loading of the database is another one-liner:

```
>>> db.load(iterator)
6
```

And the GenBank file is loaded into the database. Notice that the load function returns the number of records loaded (6 in this case). This is useful for sanity checking to make sure that you didn't try to load a massive file and end up with a result like 3.

2.4 Retrieving SeqRecord objects

Now that our database is loaded with information, the next logical step is retrieving information from that database. Let's start with our initial connection to the database server that we got when we connected to it above. The first step is getting a direct connection to our `cold` database. The server is set up to act like a python dictionary, so we get our `cold` database with the following code:

```
>>> db = server["cold"]
```

Now that we've got the database, we can retrieve a record based on accession numbers. So, to get a record for the Arabidopsis kin2 gene (contained in the `cor6_6.gb` file), we simply do:

```
>>> record = db.lookup(accession = "X62281")
```

The `record` that we get back models as exactly as possible a standard SeqRecord object (again, referring to the Biopython documentation is a great idea if you don't understand these standard Biopython objects). So we can do things like retrieve basic information about this sequence in the standard ways:

```
>>> record.id
'X62281'
>>> record.name
'ATKIN2'
>>> record.description
'A.thaliana kin2 gene.'
```

We can deal with the sequence:

```
>>> sequence = record.seq
>>> sequence.alphabet
IUPACUnambiguousDNA()
>>> sequence[:5].tostring()
'ATTTG'
```

We can deal with features:

```
>>> feature = record.features[0]
>>> feature.type
'source'
>>> print feature.location
(0..880)
>>> feature.qualifiers
{'strain': ['ssp. L. Heynh, Colombia'], 'organism': ['Arabidopsis
thaliana'], 'db_xref': ['taxon:3702']}
```

And so on and so in. The basic idea is that you should be able to deal with a sequence with annotations stored in a database in exactly the same way you deal with sequences in flatfiles. All of the database access happens under the hood so you don't even have to think about SQL or other nastiness, but you get all the advantages of a database.

3 Python Cookbook Code

Doing various fun things to demonstrate the usefulness of BioSQL. I suppose I could go ahead and write something here.