SQLObject Documentation

Release 3.3.0

Ian Bicking and contributors

May 07, 2017

Contents

SQLObject is a popular *Object Relational Manager* for providing an object interface to your database, with tables as classes, rows as instances, and columns as attributes.

SQLObject includes a Python-object-based query language that makes SQL more abstract, and provides substantial database independence for applications.

CHAPTER 1

Documentation

Download SQLObject

The latest releases are always available on the Python Package Index, and is installable with pip or easy_install.

You can install the latest release with:

```
pip install -U SQLObject
```

or:

```
easy_install -U SQLObject
```

You can install the latest version of SQLObject with:

```
easy_install SQLObject==dev
```

You can install the latest bug fixing branch with:

```
easy_install SQLObject==bugfix
```

If you want to require a specific revision (because, for instance, you need a bugfix that hasn't appeared in a release), you can put this in your setuptools using `setup.py` file:

```
setup(...
  install_requires=["SQLObject==bugfix,>=0.7.1dev-r1485"],
)
```

This says that you *need* revision 1485 or higher. But it also says that you can aquire the "bugfix" version to try to get that. In fact, when you install `SQLObject==bugfix` you will be installing a specific version, and "bugfix" is just a kind of label for a way of acquiring the version (it points to a branch in the repository).

Repositories

The SQLObject git repositories are located at https://github.com/sqlobject and https://sourceforge.net/p/sqlobject/_list/git

Before switching to git development was performed at the Subversion repository that is no longer available.

SQLObject Community

SQLObject questions and discussion happens on the sqlobject-discuss mailing list.

Bugs should be submitted to the GitHub issue tracker or bug tracker at SourceForge, and patches as pull requests at GitHub or to the patch tracker at SF.

Development takes place in the git repositories. There are development docs, i.e the docs from the development branch (master). If you are interested in contributing you should read the Developer Guide.

The Author List tries to list all the major contributors.

One can also contribute to community-editable recipe/documentation site.

SQLObject Links

Contents

If you have a link you'd like added to this page, please submit a pull requests at GitHub or a bug report with the link and title at bug tracker.

Articles and Documentation

- Connecting databases to Python with SQLObject; an article at DeveloperWorks.
- DB migration using sqlobject-admin how-to.
- Using raw SQL with SQLObject and keeping the object-y goodness.
- Example of using SQLObject with web.py under mod_wsgi.

Open Source Projects

- Catwalk is a web-based SQLObject browser and object editor (included in TurboGears 1.0).
- Ultra Gleeper, a Recommendation Engine for Web Pages.
- Guten; an application for browsing, reading, and managing books from Project Gutenberg.

News

Contents:

SQLObject 3.3.0

Released 7 May 2017.

Features

- Support for Python 2.6 is declared obsolete and will be removed in the next release.

Minor features

- Convert scripts repository to devscripts subdirectory. Some of thses scripts are version-dependent so it's better to have them in the main repo.
- Test for __nonzero__ under Python 2, __bool__ under Python 3 in BoolCol.

Drivers (work in progress)

- Add support for PyODBC and PyPyODBC (pure-python ODBC DB API driver) for MySQL, PostgreSQL and MS SQL. Driver names are `pyodbc`, `pypyodbc` or `odbc` (try `pyodbc` and `pypyodbc`). There are some problems with pyodbc and many problems with pypyodbc.

Documentation

- Stop updating http://sqlobject.readthedocs.org/ - it's enough to have http://sqlobject.org/

Tests

- Run tests at Travis CI and AppVeyor with Python 3.6, x86 and x64.
- Stop running tests at Travis with Python 2.6.
- Stop running tests at AppVeyor with pymssql - too many timeouts and problems.

SQLObject 3.2.0

Released 11 Mar 2017.

Minor features

- Drop table name from `VACUUM` command in SQLiteConnection: SQLite doesn't vacuum a single table and SQLite 3.15 uses the supplied name as the name of the attached database to vacuum.
- Remove `driver` keyword from RdbhostConnection as it allows one driver `rdbhdb`.
- Add `driver` keyword for FirebirdConnection. Allowed values are 'fdb', 'kinterbasdb' and 'pyfirebirdsql'. Default is to test 'fdb' and 'kinterbasdb' in that order. pyfirebirdsql is supported but has problems.
- Add `driver` keyword for MySQLConnection. Allowed values are 'mysqldb', 'connector', 'oursql' and 'pymysql'. Default is to test for mysqldb only.
- Add support for MySQL Connector (pure python; binary packages are not at PyPI and hence are hard to install and test).
- Add support for oursql MySQL driver (only Python 2.6 and 2.7 until oursql author fixes Python 3 compatibility).
- Add support for PyMySQL - pure python mysql interface).
- Add parameter `timeout` for MSSQLConnection (usable only with pymssql driver); timeouts are in seconds.
- Remove deprecated ez_setup.py.

Drivers (work in progress)

- Extend support for PyGreSQL driver. There are still some problems.
- Add support for py-postgresql PostgreSQL driver. There are still problems with the driver.
- Add support for pyfirebirdsql.There are still problems with the driver.

Bug fixes

- Fix MSSQLConnection.columnsFromSchema: remove *(* and *)* from default value.
- Fix MSSQLConnection and SybaseConnection: insert default values into a table with just one IDENTITY column.
- Remove excessive NULLs from `CREATE TABLE` for MSSQL/Sybase.
- Fix concatenation operator for MSSQL/Sybase (it's +, not `||`).
- Fix MSSQLConnection.server_version() under Py3 (decode version to str).

Documentation

- The docs are now generated with Sphinx.
- Move `docs/LICENSE` to the top-level directory so that Github recognizes it.

Tests

- Rename `py.test` -> `pytest` in tests and docs.
- Great Renaming: fix `pytest` warnings by renaming `TestXXX` classes to `SOTestXXX` to prevent `pytest` to recognize them as test classes.
- Fix `pytest` warnings by converting yield tests to plain calls: yield tests were deprecated in `pytest`.
- Tests are now run at CIs with Python 3.5.
- Drop `Circle CI`.
- Run at Travis CI tests with Firebird backend (server version 2.5; drivers fdb and firebirdsql). There are problems with tests.
- Run tests at AppVeyor for windows testing. Run tests with MS SQL, MySQL, Postgres and SQLite backends; use Python 2.7, 3.4 and 3.5, x86 and x64. There are problems with MS SQL and MySQL.

SQLObject 3.1.0

Released 16 Aug 2016.

Features

- Add UuidCol.
- Add JsonbCol. Only for PostgreSQL. Requires psycopg2 >= 2.5.4 and PostgreSQL >= 9.2.
- Add JSONCol, a universal json column.

- For Python >= 3.4 minimal FormEncode version is now 1.3.1.
- If mxDateTime is in use, convert timedelta (returned by MySQL) to mxDateTime.Time.

Documentation

- Developer's Guide is extended to explain SQLObject architecture and how to create a new column type.
- Fix URLs that can be found; remove missing links.
- Rename reStructuredText files from *.txt to *.rst.

Source code

- Fix all *import* * using https://github.com/zestyping/star-destroyer.

Tests

- Tests are now run at Circle CI.
- Use pytest-cov for test coverage. Report test coverage via coveralls.io and codecov.io.
- Install mxDateTime to run date/time tests with it.

SQLObject 3.0.0

Released 1 Jun 2016.

Features

- Support for Python 2 and Python 3 with one codebase! (Python version >= 3.4 currently required.)

Minor features

- PyDispatcher (>=2.0.4) was made an external dependency.

Development

- Source code was made flake8-clean.

Documentation

- Documentation is published at http://sqlobject.readthedocs.org/ in Sphinx format.

Older news

SQLObject and Python 3

Contents

Changes between Python 2 and Python 3

There are a few changes in the behaviour of SQLObject on Python 3, due to the changed stings / bytes handling introduced in Python 3.0.

BLOBCol

In Python 3, BLOBCol now accepts and returns bytes, rather than strings as it did in Python 2.

StringCol

In Python 3, StringCol now accepts arbitrary Unicode strings.

UnicodeCol

The dbEncoding parameter to UnicodeCol has no effect in Python 3 code. This is now handled by the underlying database layer and is no longer exposed via SQLObject. The parameter is still available for those writing Python 2 compatible code.

Python 3 and MySQL

SQLObject is tested using mysqlclient as the database driver on Python 3. Note that the default encoding of MySQL databases is *latin1*, which can cause problems with general Unicode strings. We recommend specifying the character set as *utf8* when using MySQL to protect against these issues.

Using databases created with SQLObject and Python 2 in Python 3

For most cases, things should just work as before. The only issues should around UnicodeCol, as how this is handled has changed.

SQLite

The Python 3 sqlite driver expects Unicode columns to be encoded using utf8. Columns created using the default encoding on Python 2 should work fine, but columns created with a different encoding set using the dbEncoding parameter may cause problems.

Postgres

Postgres' behaviour is similar to sqlite. Columns created using the default encoding on Python 2 should work fine, but columns created with a different encoding set using the dbEncoding may cause problems.

MySQL

For MySQL, the results depend on whether the Python 2 database was using MySQLdb's Unicode mode or not.

If a character set was specified for the database using the charset parameter, such as:

```
mysql:///localhost/test?charset=latin1
```

Things should work provided the same character set is specified when using Python 3.

If a character set wasn't specified, then things may work if the character set is set to match the dbEncoding parameter used when defining the UnicodeCol.

SQLObject

Contents:

Credits

SQLObject is by Ian Bicking (ianb@colorstudy.com) and Contributors. The website is sqlobject.org.

License

Introduction

SQLObject is an *object-relational mapper* for Python programming language. It allows you to translate RDBMS table rows into Python objects, and manipulate those objects to transparently manipulate the database.

In using SQLObject, you will create a class definition that will describe how the object translates to the database table. SQLObject will produce the code to access the database, and update the database with your changes. The generated interface looks similar to any other interface, and callers need not be aware of the database backend.

SQLObject also includes a novel feature to avoid generating, textually, your SQL queries. This also allows non-SQL databases to be used with the same query syntax.

Requirements

Currently SQLObject supports MySQL via MySQLdb aka MySQL-python (called mysqlclient for Python 3), MySQL Connector, oursql, PyMySQL, PyODBC and PyPyODBC. For PostgreSQL psycopg2 or psycopg1 are recommended; PyGreSQL, py-postgresql, PyODBC and PyPyODBC are supported but have problems (not all tests passed). SQLite has a built-in driver or PySQLite. Firebird is supported via fdb or kinterbasdb; pyfirebirdsql is supported but has problems. MAX DB (also known as SAP DB) is supported via sapdb. Sybase via Sybase. MSSQL Server via pymssql (+ FreeTDS) or adodbapi (Win32).

Python 2.6, 2.7 or 3.4+ is required.

Compared To Other Database Wrappers

There are several object-relational mappers (ORM) for Python. We honestly can't comment deeply on the quality of those packages, but we'll try to place SQLObject in perspective.

Objects have built-in magic – setting attributes has side effects (it changes the database), and defining classes has side effects (through the use of metaclasses). Attributes are generally exposed, not marked private, knowing that they can be made dynamic or write-only later.

SQLObject creates objects that feel similar to normal Python objects. An attribute attached to a column doesn't look different than an attribute that's attached to a file, or an attribute that is calculated. It is a specific goal that you be able to change the database without changing the interface, including changing the scope of the database, making it more or less prominent as a storage mechanism.

This is in contrast to some ORMs that provide a dictionary-like interface to the database (for example, PyDO). The dictionary interface distinguishes the row from a normal Python object. We also don't care for the use of strings where an attribute seems more natural – columns are limited in number and predefined, just like attributes. (Note: newer version of PyDO apparently allow attribute access as well)

SQLObject is, to my knowledge, unique in using metaclasses to facilitate this seamless integration. Some other ORMs use code generation to create an interface, expressing the schema in a CSV or XML file (for example, MiddleKit, part of Webware). By using metaclasses you are able to comfortably define your schema in the Python source code. No code generation, no weird tools, no compilation step.

SQLObject provides a strong database abstraction, allowing cross-database compatibility (so long as you don't sidestep SQLObject).

SQLObject has joins, one-to-many, and many-to-many, something which many ORMs do not have. The join system is also intended to be extensible.

You can map between database names and Python attribute and class names; often these two won't match, or the database style would be inappropriate for a Python attribute. This way your database schema does not have to be designed with SQLObject in mind, and the resulting classes do not have to inherit the database's naming schemes.

Using SQLObject: An Introduction

Let's start off quickly. We'll generally just import everything from the `sqlobject` class:

```
>>> from sqlobject import *
```

Declaring a Connection

The connection URI must follow the standard URI syntax:

```
scheme://[user[:password]@]host[:port]/database[?parameters]
```

Scheme is one of `sqlite`, `mysql`, `postgres`, `firebird`, `interbase`, `maxdb`, `sapdb`, `mssql`, `sybase`.

Examples:

```
mysql://user:password@host/database
mysql://host/database?debug=1
postgres://user@host/database?debug=&cache=
postgres:///full/path/to/socket/database
postgres://host:5432/database
sqlite:///full/path/to/database
sqlite:/C:/full/path/to/database
sqlite:/:memory:
```

Parameters are: `debug` (default: False), `debugOutput` (default: False), `cache` (default: True), `autoCommit` (default: True), `debugThreading` (default: False), `logger` (default: None), `loglevel` (default: None), `schema` (default: None).

If you want to pass True value in a connection URI - pass almost any non-empty string, especially `yes`, `true`, `on` or `1`; an empty string or `no`, `false`, `off` or `0` for False.

There are also connection-specific parameters, they are listed in the appropriate sections.

Lets first set up a connection:

```
>>> import os
>>> db_filename = os.path.abspath('data.db')
>>> connection_string = 'sqlite:' + db_filename
>>> connection = connectionForURI(connection_string)
>>> sqlhub.processConnection = connection
```

The `sqlhub.processConnection` assignment means that all classes will, by default, use this connection we've just set up.

Declaring the Class

We'll develop a simple addressbook-like database. We could create the tables ourselves, and just have SQLObject access those tables, but let's have SQLObject do that work. First, the class:

```
>>> class Person(SQLObject):
...
...     firstName = StringCol()
...     middleInitial = StringCol(length=1, default=None)
...     lastName = StringCol()
```

Many basic table schemas won't be any more complicated than that. *firstName*, *middleInitial*, and *lastName* are all columns in the database. The general schema implied by this class definition is:

```
CREATE TABLE person (
    id INT PRIMARY KEY AUTO_INCREMENT,
    first_name TEXT,
    middle_initial CHAR(1),
    last_name TEXT
);
```

This is for SQLite or MySQL. The schema for other databases looks slightly different (especially the `id` column). You'll notice the names were changed from mixedCase to underscore_separated – this is done by the *style object*.

There are a variety of ways to handle names that don't fit conventions (see *Irregular Naming*). Now we'll create the table in the database:

```
>>> Person.createTable()
[]
```

We can change the type of the various columns by using something other than *StringCol*, or using different arguments. More about this in *Column Types*.

You'll note that the `id` column is not given in the class definition, it is implied. For MySQL databases it should be defined as `INT PRIMARY KEY AUTO_INCREMENT`, in Postgres `SERIAL PRIMARY KEY`, in SQLite as `INTEGER PRIMARY KEY AUTOINCREMENT`, and for other backends accordingly. You can't use tables with SQLObject that don't have a single primary key, and you must treat that key as immutable (otherwise you'll confuse SQLObject terribly).

You can *override the id name* in the database, but it is always called `.id` from Python.

Using the Class

Now that you have a class, how will you use it? We'll be considering the class defined above.

To create a new object (and row), use class instantiation, like:

```
>>> Person(firstName="John", lastName="Doe")
<Person 1 firstName='John' middleInitial=None lastName='Doe'>
```

Note: In SQLObject NULL/None does *not* mean default. NULL is a funny thing; it mean very different things in different contexts and to different people. Sometimes it means "default", sometimes "not applicable", sometimes "unknown". If you want a default, NULL or otherwise, you always have to be explicit in your class definition.

Also note that the SQLObject default isn't the same as the database's default (SQLObject never uses the database's default).

If you had left out `firstName` or `lastName` you would have gotten an error, as no default was given for these columns (`middleInitial` has a default, so it will be set to `NULL`, the database equivalent of `None`).

You can use the class method *.get()* to fetch instances that already exist:

```
>>> Person.get(1)
<Person 1 firstName='John' middleInitial=None lastName='Doe'>
```

When you create an object, it is immediately inserted into the database. SQLObject uses the database as immediate storage, unlike some other systems where you explicitly save objects into a database.

Here's a longer example of using the class:

```
>>> p = Person.get(1)
>>> p
<Person 1 firstName='John' middleInitial=None lastName='Doe'>
>>> p.firstName
'John'
>>> p.middleInitial = 'Q'
>>> p.middleInitial
'Q'
>>> p2 = Person.get(1)
>>> p2
<Person 1 firstName='John' middleInitial='Q' lastName='Doe'>
```

```
>>> p is p2
True
```

Columns are accessed like attributes. (This uses the `property` feature of Python, so that retrieving and setting these attributes executes code). Also note that objects are unique – there is generally only one `Person` instance of a particular id in memory at any one time. If you ask for a person by a particular ID more than once, you'll get back the same instance. This way you can be sure of a certain amount of consistency if you have multiple threads accessing the same data (though of course across processes there can be no sharing of an instance). This isn't true if you're using *transactions*, which are necessarily isolated.

To get an idea of what's happening behind the surface, we'll give the same actions with the SQL that is sent, along with some commentary:

```
>>> # This will make SQLObject print out the SQL it executes:
>>> Person._connection.debug = True
>>> p = Person(firstName='Bob', lastName='Hope')
 1/QueryIns:  INSERT INTO person (first_name, middle_initial, last_name) VALUES ('Bob
↪', NULL, 'Hope')
 1/QueryR  :  INSERT INTO person (first_name, middle_initial, last_name) VALUES ('Bob
↪', NULL, 'Hope')
 1/COMMIT  :  auto
 1/QueryOne:  SELECT first_name, middle_initial, last_name FROM person WHERE ((person.
↪id) = (2))
 1/QueryR  :  SELECT first_name, middle_initial, last_name FROM person WHERE ((person.
↪id) = (2))
 1/COMMIT  :  auto
>>> p
<Person 2 firstName='Bob' middleInitial=None lastName='Hope'>
>>> p.middleInitial = 'Q'
 1/Query   :  UPDATE person SET middle_initial = ('Q') WHERE id = (2)
 1/QueryR  :  UPDATE person SET middle_initial = ('Q') WHERE id = (2)
 1/COMMIT  :  auto
>>> p2 = Person.get(1)
>>> # Note: no database access, since we're just grabbing the same
>>> # instance we already had.
```

Hopefully you see that the SQL that gets sent is pretty clear and predictable. To view the SQL being sent, add `?debug=true` to your connection URI, or set the `debug` attribute on the connection, and all SQL will be printed to the console. This can be reassuring, and we would encourage you to try it.

As a small optimization, instead of assigning each attribute individually, you can assign a number of them using the `set` method, like:

```
>>> p.set(firstName='Robert', lastName='Hope Jr.')
```

This will send only one `UPDATE` statement. You can also use *set* with non-database properties (there's no benefit, but it helps hide the difference between database and non-database attributes).

Selecting Multiple Objects

While the full power of all the kinds of joins you can do with a relational database are not revealed in SQLObject, a simple `SELECT` is available.

`select` is a class method, and you call it like (with the SQL that's generated):

```
>>> Person._connection.debug = True
>>> peeps = Person.select(Person.q.firstName=="John")
```

```
>>> list(peeps)
 1/Select   :  SELECT person.id, person.first_name, person.middle_initial, person.last_
↪name FROM person WHERE ((person.first_name) = ('John'))
 1/QueryR   :  SELECT person.id, person.first_name, person.middle_initial, person.last_
↪name FROM person WHERE ((person.first_name) = ('John'))
 1/COMMIT   :  auto
[<Person 1 firstName='John' middleInitial='Q' lastName='Doe'>]
```

This example returns everyone with the first name John.

Queries can be more complex:

```
>>> peeps = Person.select(
...          OR(Person.q.firstName == "John",
...             LIKE(Person.q.lastName, "%Hope%")))
>>> list(peeps)
 1/Select   :  SELECT person.id, person.first_name, person.middle_initial, person.last_
↪name FROM person WHERE (((person.first_name) = ('John')) OR (person.last_name LIKE (
↪'%Hope%')))
 1/QueryR   :  SELECT person.id, person.first_name, person.middle_initial, person.last_
↪name FROM person WHERE (((person.first_name) = ('John')) OR (person.last_name LIKE (
↪'%Hope%')))
 1/COMMIT   :  auto
[<Person 1 firstName='John' middleInitial='Q' lastName='Doe'>, <Person 2 firstName=
↪'Robert' middleInitial='Q' lastName='Hope Jr.'>]
```

You'll note that classes have an attribute `q`, which gives access to special objects for constructing query clauses. All attributes under `q` refer to column names and if you construct logical statements with these it'll give you the SQL for that statement. You can also create your SQL more manually:

```
>>> Person._connection.debug = False  # Need for doctests
>>> peeps = Person.select("""person.first_name = 'John' AND
...                          person.last_name LIKE 'D%'""")
```

You should use *MyClass.sqlrepr* to quote any values you use if you create SQL manually (quoting is automatic if you use `q`). You can use the keyword arguments *orderBy* to create `ORDER BY` in the select statements: *orderBy* takes a string, which should be the *database* name of the column, or a column in the form `Person.q.firstName`. You can use `"-colname"` or `DESC(Person.q.firstName)` to specify descending order (this is translated to DESC, so it works on non-numeric types as well), or call `MyClass.select().reversed()`. orderBy can also take a list of columns in the same format: `["-weight", "name"]`.

You can use the *sqlmeta* class variable *defaultOrder* to give a default ordering for all selects. To get an unordered result when *defaultOrder* is used, use `orderBy=None`. Select results are generators, which are lazily evaluated. So the SQL is only executed when you iterate over the select results, or if you use `list()` to force the result to be executed. When you iterate over the select results, rows are fetched one at a time. This way you can iterate over large results without keeping the entire result set in memory. You can also do things like `.reversed()` without fetching and reversing the entire result – instead, SQLObject can change the SQL that is sent so you get equivalent results.

You can also slice select results. This modifies the SQL query, so `peeps[:10]` will result in `LIMIT 10` being added to the end of the SQL query. If the slice cannot be performed in the SQL (e.g., peeps[:-10]), then the select is executed, and the slice is performed on the list of results. This will generally only happen when you use negative indexes.

In certain cases, you may get a select result with an object in it more than once, e.g., in some joins. If you don't want this, you can add the keyword argument `MyClass.select(..., distinct=True)`, which results in a `SELECT DISTINCT` call.

You can get the length of the result without fetching all the results by calling `count` on the result object, like

`MyClass.select().count()`. A `COUNT(*)` query is used – the actual objects are not fetched from the database. Together with slicing, this makes batched queries easy to write:

> start = 20 size = 10 query = Table.select() results = query[start:start+size] total = query.count() print "Showing page %i of %i" % (start/size + 1, total/size + 1)

Note: There are several factors when considering the efficiency of this kind of batching, and it depends very much how the batching is being used. Consider a web application where you are showing an average of 100 results, 10 at a time, and the results are ordered by the date they were added to the database. While slicing will keep the database from returning all the results (and so save some communication time), the database will still have to scan through the entire result set to sort the items (so it knows which the first ten are), and depending on your query may need to scan through the entire table (depending on your use of indexes). Indexes are probably the most important way to improve importance in a case like this, and you may find caching to be more effective than slicing.

In this case, caching would mean retrieving the *complete* results. You can use `list(MyClass.select(...))` to do this. You can save these results for some limited period of time, as the user looks through the results page by page. This means the first page in a search result will be slightly more expensive, but all later pages will be very cheap.

For more information on the where clause in the queries, see the SQLBuilder documentation.

q-magic

Please note the use of the *q* attribute in examples above. *q* is an object that returns special objects to construct SQL expressions. Operations on objects returned by *q-magic* are not evaluated immediately but stored in a manner similar to symbolic algebra; the entire expression is evaluated by constructing a string that is sent then to the backend.

For example, for the code:

```
>>> peeps = Person.select(Person.q.firstName=="John")
```

SQLObject doesn't evaluate firstName but stores the expression:

> Person.q.firstName=="John"

Later SQLObject converts it to the string `first_name = 'John'` and passes the string to the backend.

selectBy Method

An alternative to `.select` is `.selectBy`. It works like:

```
>>> peeps = Person.selectBy(firstName="John", lastName="Doe")
```

Each keyword argument is a column, and all the keyword arguments are ANDed together. The return value is a *SelectResults*, so you can slice it, count it, order it, etc.

Lazy Updates

By default SQLObject sends an `UPDATE` to the database for every attribute you set, or every time you call `.set()`. If you want to avoid this many updates, add `lazyUpdate = True` to your class *sqlmeta definition*. Then updates will only be written to the database when you call `inst.syncUpdate()` or `inst.sync()`: `.sync()` also refetches the data from the database, which `.syncUpdate()` does not do.

When enabled instances will have a property `.sqlmeta.dirty`, which indicates if there are pending updates. Inserts are still done immediately; there's no way to do lazy inserts at this time.

One-to-Many Relationships

An address book is nothing without addresses.

First, let's define the new address table. People can have multiple addresses, of course:

```
>>> class Address(SQLObject):
...
...     street = StringCol()
...     city = StringCol()
...     state = StringCol(length=2)
...     zip = StringCol(length=9)
...     person = ForeignKey('Person')
>>> Address.createTable()
[]
```

Note the column `person = ForeignKey("Person")`. This is a reference to a *Person* object. We refer to other classes by name (with a string). In the database there will be a `person_id` column, type `INT`, which points to the `person` column.

Note: The reason SQLObject uses strings to refer to other classes is because the other class often does not yet exist. Classes in Python are *created*, not *declared*; so when a module is imported the commands are executed. `class` is just another command; one that creates a class and assigns it to the name you give.

If class `A` referred to class `B`, but class `B` was defined below `A` in the module, then when the `A` class was created (including creating all its column attributes) the `B` class simply wouldn't exist. By referring to classes by name, we can wait until all the required classes exist before creating the links between classes.

We want an attribute that gives the addresses for a person. In a class definition we'd do:

```
class Person(SQLObject):
    ...
    addresses = MultipleJoin('Address')
```

But we already have the class. We can add this to the class in-place:

```
>>> Person.sqlmeta.addJoin(MultipleJoin('Address',
...                         joinMethodName='addresses'))
```

Note: In almost all cases you can modify SQLObject classes after they've been created. Having attributes that contain `*Col` objects in the class definition is equivalent to calling certain class methods (like `addColumn()`).

Now we can get the backreference with `aPerson.addresses`, which returns a list. An example:

```
>>> p.addresses
[]
>>> Address(street='123 W Main St', city='Smallsville',
...         state='MN', zip='55407', person=p)
<Address 1 ...>
>>> p.addresses
[<Address 1 ...>]
```

Note: MultipleJoin, as well as RelatedJoin, returns a list of results. It is often preferable to get a SelectResults object instead, in which case you should use SQLMultipleJoin and SQLRelatedJoin. The declaration of these joins is

unchanged from above, but the returned iterator has many additional useful methods.

Many-to-Many Relationships

For this example we will have user and role objects. The two have a many-to-many relationship, which is represented with the *RelatedJoin*.

```
>>> class User(SQLObject):
...
...     class sqlmeta:
...         # user is a reserved word in some databases, so we won't
...         # use that for the table name:
...         table = "user_table"
...
...     username = StringCol(alternateID=True, length=20)
...     # We'd probably define more attributes, but we'll leave
...     # that exercise to the reader...
...
...     roles = RelatedJoin('Role')
```

```
>>> class Role(SQLObject):
...
...     name = StringCol(alternateID=True, length=20)
...
...     users = RelatedJoin('User')
```

```
>>> User.createTable()
[]
>>> Role.createTable()
[]
```

Note: The sqlmeta class is used to store different kinds of metadata (and override that metadata, like table). This is new in SQLObject 0.7. See the section *Class sqlmeta* for more information on how it works and what attributes have special meanings.

And usage:

```
>>> bob = User(username='bob')
>>> tim = User(username='tim')
>>> jay = User(username='jay')
>>> admin = Role(name='admin')
>>> editor = Role(name='editor')
>>> bob.addRole(admin)
>>> bob.addRole(editor)
>>> tim.addRole(editor)
>>> bob.roles
[<Role 1 name='admin'>, <Role 2 name='editor'>]
>>> tim.roles
[<Role 2 name='editor'>]
>>> jay.roles
[]
>>> admin.users
[<User 1 username='bob'>]
```

```
>>> editor.users
[<User 1 username='bob'>, <User 2 username='tim'>]
```

In the process an intermediate table is created, `role_user`, which references both of the other classes. This table is never exposed as a class, and its rows do not have equivalent Python objects – this hides some of the nuisance of a many-to-many relationship.

By the way, if you want to create an intermediate table of your own, maybe with additional columns, be aware that the standard SQLObject methods add/removesomething may not work as expected. Assuming that you are providing the join with the correct joinColumn and otherColumn arguments, be aware it's not possible to insert extra data via such methods, nor will they set any default value.

Let's have an example: in the previous User/Role system, you're creating a UserRole intermediate table, with the two columns containing the foreign keys for the MTM relationship, and an additional DateTimeCol defaulting to datetime.datetime.now : that column will stay empty when adding roles with the addRole method. If you want to get a list of rows from the intermediate table directly add a MultipleJoin to User or Role class.

You may notice that the columns have the extra keyword argument *alternateID*. If you use `alternateID=True`, this means that the column uniquely identifies rows – like a username uniquely identifies a user. This identifier is in addition to the primary key (`id`), which is always present.

Note: SQLObject has a strong requirement that the primary key be unique and *immutable*. You cannot change the primary key through SQLObject, and if you change it through another mechanism you can cause inconsistency in any running SQLObject program (and in your data). For this reason meaningless integer IDs are encouraged – something like a username that could change in the future may uniquely identify a row, but it may be changed in the future. So long as it is not used to reference the row, it is also *safe* to change it in the future.

A alternateID column creates a class method, like `byUsername` for a column named `username` (or you can use the *alternateMethodName* keyword argument to override this). Its use:

```
>>> User.byUsername('bob')
<User 1 username='bob'>
>>> Role.byName('admin')
<Role 1 name='admin'>
```

Selecting Objects Using Relationships

An select expression can refer to multiple classes, like:

```
>>> Person._connection.debug = False # Needed for doctests
>>> peeps = Person.select(
...          AND(Address.q.personID == Person.q.id,
...              Address.q.zip.startswith('504')))
>>> list(peeps)
[]
>>> peeps = Person.select(
...          AND(Address.q.personID == Person.q.id,
...              Address.q.zip.startswith('554')))
>>> list(peeps)
[<Person 2 firstName='Robert' middleInitial='Q' lastName='Hope Jr.'>]
```

It is also possible to use the `q` attribute when constructing complex queries, like:

```
>>> Person._connection.debug = False  # Needed for doctests
>>> peeps = Person.select("""address.person_id = person.id AND
```

```
...                             address.zip LIKE '504%'""",
...                           clauseTables=['address'])
```

Note that you have to use `clauseTables` if you use tables besides the one you are selecting from. If you use the `q` attributes SQLObject will automatically figure out what extra classes you might have used.

Class sqlmeta

This new class is available starting with SQLObject 0.7 and allows specifying metadata in a clearer way, without polluting the class namespace with more attributes.

There are some special attributes that can be used inside this class that will change the behavior of the class that contains it. Those values are:

***table*:** The name of the table in the database. This is derived from `style` and the class name if no explicit name is given. If you don't give a name and haven't defined an alternative `style`, then the standard *MixedCase* to *mixed_case* translation is performed.

***idName*:** The name of the primary key column in the database. This is derived from `style` if no explicit name is given. The default name is `id`.

***idType*:** A function that coerces/normalizes IDs when setting IDs. This is `int` by default (all IDs are normalized to integers).

***style*:** A style object – this object allows you to use other algorithms for translating between Python attribute and class names, and the database's column and table names. See *Changing the Naming Style* for more. It is an instance of the *IStyle* interface.

***lazyUpdate*:** A boolean (default false). If true, then setting attributes on instances (or using `inst.set(.)` will not send `UPDATE` queries immediately (you must call `inst.syncUpdates()` or `inst.sync()` first).

***defaultOrder*:** When selecting objects and not giving an explicit order, this attribute indicates the default ordering. It is like this value is passed to `.select()` and related methods; see those method's documentation for details.

***cacheValues*:** A boolean (default true). If true, then the values in the row are cached as long as the instance is kept (and `inst.expire()` is not called).

If set to *False* then values for attributes from the database won't be cached. So every time you access an attribute in the object the database will be queried for a value, i.e., a `SELECT` will be issued. If you want to handle concurrent access to the database from multiple processes then this is probably the way to do so.

***registry*:** Because SQLObject uses strings to relate classes, and these strings do not respect module names, name clashes will occur if you put different systems together. This string value serves as a namespace for classes.

***fromDatabase*:** A boolean (default false). If true, then on class creation the database will be queried for the table's columns, and any missing columns (possible all columns) will be added automatically. Please be warned that not all connections fully implement database introspection.

***dbEncoding*:** *UnicodeCol* looks up *sqlmeta.dbEncoding* if *column.dbEncoding* is `None` (if *sqlmeta.dbEncoding* is `None` *UnicodeCol* looks up *connection.dbEncoding* and if *dbEncoding* isn't defined anywhere it defaults to `"utf-8"`). For Python 3 there must be one encoding for connection - do not define different columns with different encodings, it's not implemented.

The following attributes provide introspection but should not be set directly - see *Runtime Column and Join Changes* for dynamically modifying these class elements.

***columns*:** A dictionary of `{columnName: anSOColInstance}`. You can get information on the columns via this read-only attribute.

***columnList*:** A list of the values in `columns`. Sometimes a stable, ordered version of the columns is necessary; this is used for that.

***columnDefinitions*:** A dictionary like `columns`, but contains the original column definitions (which are not class-specific, and have no logic).

***joins*:** A list of all the Join objects for this class.

***indexes*:** A list of all the indexes for this class.

***createSQL*:** SQL queries run after table creation. createSQL can be a string with a single SQL command, a list of SQL commands, or a dictionary with keys that are dbNames and values that are either single SQL command string or a list of SQL commands. This is usually for ALTER TABLE commands.

There is also one instance attribute:

***expired*:** A boolean. If true, then the next time this object's column attributes are accessed a query will be run.

While in previous versions of SQLObject those attributes were defined directly at the class that will map your database data to Python and all of them were prefixed with an underscore, now it is suggested that you change your code to this new style. The old way was removed in SQLObject 0.8.

Please note: when using InheritedSQLObject, sqlmeta attributes don't get inherited, e.g. you can't access via the sqlmeta.columns dictionary the parent's class column objects.

Using sqlmeta

To use sqlmeta you should write code like this example:

```
class MyClass(SQLObject):

    class sqlmeta:
        lazyUpdate = True
        cacheValues = False

    columnA = StringCol()
    columnB = IntCol()

    def _set_attr1(self, value):
        # do something with value

    def _get_attr1(self):
        # do something to retrieve value
```

The above definition is creating a table `my_class` (the name may be different if you change the `style` used) with two columns called columnA and columnB. There's also a third field that can be accessed using `MyClass.attr1`. The sqlmeta class is changing the behavior of `MyClass` so that it will perform lazy updates (you'll have to call the `.sync()` method to write the updates to the database) and it is also telling that `MyClass` won't have any cache, so that every time you ask for some information it will be retrieved from the database.

j-magic

There is a magic attribute *j* similar to *q* with attributes for ForeignKey and SQLMultipleJoin/SQLRelatedJoin, providing a shorthand for the SQLBuilder join expressions to traverse the given relationship. For example, for a ForeignKey AClass.j.someB is equivalent to (AClass.q.someBID==BClass.q.id), as is BClass.j.someAs for the matching SQLMultipleJoin.

SQLObject Class

There is one special attribute - *_connection*. It is the connection defined for the table.

***_connection*:** The connection object to use, from *DBConnection*. You can also set the variable *__connection__* in the enclosing module and it will be picked up (be sure to define *__connection__* before your class). You can also pass a connection object in at instance creation time, as described in *transactions*.

If you have defined *sqlhub.processConnection* then this attribute can be omitted from your class and the sqlhub will be used instead. If you have several classes using the same connection that might be an advantage, besides saving a lot of typing.

Customizing the Objects

While we haven't done so in the examples, you can include your own methods in the class definition. Writing your own methods should be obvious enough (just do so like in any other class), but there are some other details to be aware of.

Initializing the Objects

There are two ways SQLObject instances can come into existence: they can be fetched from the database, or they can be inserted into the database. In both cases a new Python object is created. This makes the role of *__init__* a little confusing.

In general, you should not touch *__init__*. Instead use the *_init* method, which is called after an object is fetched or inserted. This method has the signature `_init(self, id, connection=None, selectResults=None)`, though you may just want to use `_init(self, *args, **kw)`. **Note:** don't forget to call `SQLObject._init(self, *args, **kw)` if you override the method!

Adding Magic Attributes (properties)

You can use all the normal techniques for defining methods in this class, including *classmethod*, *staticmethod*, and *property*, but you can also use a shortcut. If you have a method that's name starts with `_set_`, `_get_`, `_del_`, or `_doc_`, it will be used to create a property. So, for instance, say you have images stored under the ID of the person in the `/var/people/images` directory:

```
class Person(SQLObject):
    # ...

    def imageFilename(self):
        return 'images/person-%s.jpg' % self.id

    def _get_image(self):
        if not os.path.exists(self.imageFilename()):
            return None
        f = open(self.imageFilename())
        v = f.read()
        f.close()
        return v

    def _set_image(self, value):
        # assume we get a string for the image
        f = open(self.imageFilename(), 'w')
        f.write(value)
```

```
        f.close()

    def _del_image(self, value):
        # We usually wouldn't include a method like this, but for
        # instructional purposes...
        os.unlink(self.imageFilename())
```

Later, you can use the `.image` property just like an attribute, and the changes will be reflected in the filesystem by calling these methods. This is a good technique for information that is better to keep in files as opposed to the database (such as large, opaque data like images).

You can also pass an `image` keyword argument to the constructor or the *set* method, like `Person(..., image=imageText)`.

All of the methods (`_get_`, `_set_`, etc) are optional – you can use any one of them without using the others. So you could define just a `_get_attr` method so that `attr` was read-only.

Overriding Column Attributes

It's a little more complicated if you want to override the behavior of an database column attribute. For instance, imagine there's special code you want to run whenever someone's name changes. In many systems you'd do some custom code, then call the superclass's code. But the superclass (`SQLObject`) doesn't know anything about the column in your subclass. It's even worse with properties.

SQLObject creates methods like `_set_lastName` for each of your columns, but again you can't use this, since there's no superclass to reference (and you can't write `SQLObject._set_lastName(...)`, because the SQLObject class doesn't know about your class's columns). You want to override that `_set_lastName` method yourself.

To deal with this, SQLObject creates two methods for each getter and setter, for example: `_set_lastName` and `_SO_set_lastName`. So to intercept all changes to `lastName`:

```
class Person(SQLObject):
    lastName = StringCol()
    firstName = StringCol()

    def _set_lastName(self, value):
        self.notifyLastNameChange(value)
        self._SO_set_lastName(value)
```

Or perhaps you want to constrain a phone numbers to be actual digits, and of proper length, and make the formatting nice:

```
import re

class PhoneNumber(SQLObject):
    phoneNumber = StringCol(length=30)

    _garbageCharactersRE = re.compile(r'[\-\.\(\) ]')
    _phoneNumberRE = re.compile(r'^[0-9]+$')
    def _set_phoneNumber(self, value):
        value = self._garbageCharactersRE.sub('', value)
        if not len(value) >= 10:
            raise ValueError(
                'Phone numbers must be at least 10 digits long')
        if not self._phoneNumberRE.match(value):
            raise ValueError, 'Phone numbers can contain only digits'
        self._SO_set_phoneNumber(value)
```

```
    def _get_phoneNumber(self):
        value = self._SO_get_phoneNumber()
        number = '(%s) %s-%s' % (value[0:3], value[3:6], value[6:10])
        if len(value) > 10:
            number += ' ext.%s' % value[10:]
        return number
```

Note: You should be a little cautious when modifying data that gets set in an attribute. Generally someone using your class will expect that the value they set the attribute to will be the same value they get back. In this example we removed some of the characters before putting it in the database, and reformatted it on the way out. One advantage of methods (as opposed to attribute access) is that the programmer is more likely to expect this disconnect.

Also note while these conversions will take place when getting and setting the column, in queries the conversions will not take place. So if you convert the value from a "Pythonic" representation to a "SQLish" representation, your queries (when using `.select()` and `.selectBy()`) will have to be in terms of the SQL/Database representation (as those commands generate SQL that is run on the database).

Undefined attributes

There's one more thing worth telling, because you may something get strange results when making a typo. SQLObject won't ever complain or raise any error when setting a previously undefined attribute; it will simply set it, without making any change to the database, i.e: it will work as any other attribute you set on any Python class, it will 'forget' it is a SQLObject class.

This may sometimes be a problem: if you have got a 'name' attribute and you you write `a.namme="Victor"` once, when setting it, you'll get no error, no warning, nothing at all, and you may get crazy at understanding why you don't get that value set in your DB.

Reference

The instructions above should tell you enough to get you started, and be useful for many situations. Now we'll show how to specify the class more completely.

Col Class: Specifying Columns

The list of columns is a list of *Col* objects. These objects don't have functionality in themselves, but give you a way to specify the column.

***dbName*:** This is the name of the column in the database. If you don't give a name, your Pythonic name will be converted from mixed-case to underscore-separated.

***default*:** The default value for this column. Used when creating a new row. If you give a callable object or function, the function will be called, and the return value will be used. So you can give `DateTimeCol.now` to make the default value be the current time. Or you can use `sqlbuilder.func.NOW()` to have the database use the `NOW()` function internally. If you don't give a default there will be an exception if this column isn't specified in the call to *new*.

***defaultSQL*:** `DEFAULT` SQL attribute.

***alternateID*:** This boolean (default False) indicates if the column can be used as an ID for the field (for instance, a username), though it is not a primary key. If so a class method will be added, like

`byUsername` which will return that object. Use *alternateMethodName* if you don't like the `by*` name (e.g. `alternateMethodName="username"`).

The column should be declared `UNIQUE` in your table schema.

***unique*:** If true, when SQLObject creates a table it will declare this column to be `UNIQUE`.

***notNone*:** If true, None/`NULL` is not allowed for this column. Useful if you are using SQLObject to create your tables.

***sqlType*:** The SQL type for this column (like `INT`, `BOOLEAN`, etc). You can use classes (defined below) for this, but if those don't work it's sometimes easiest just to use *sqlType*. Only necessary if SQLObject is creating your tables.

***validator*:** formencode-like validator. Making long story short, this is an object that provides `to_python()` and `from_python()` to validate *and* convert (adapt or cast) the values when they are read/written from/to the database. You should see formencode validator documentation for more details. This validator is appended to the end of the list of the list of column validators. If the column has a list of validators their `from_python()` methods are ran from the beginnig of the list to the end; `to_python()` in the reverse order. That said, `from_python()` method of this validator is called last, after all validators in the list; `to_python()` is called first.

***validator2*:** Another validator. It is inserted in the beginning of the list of the list of validators, i.e. its `from_python()` method is called first; `to_python()` last.

Column Types

The *ForeignKey* class should be used instead of *Col* when the column is a reference to another table/class. It is generally used like `ForeignKey('Role')`, in this instance to create a reference to a table *Role*. This is largely equivalent to `Col(foreignKey='Role', sqlType='INT')`. Two attributes will generally be created, `role`, which returns a *Role* instance, and `roleID`, which returns an integer ID for the related role.

There are some other subclasses of *Col*. These are used to indicate different types of columns, when SQLObject creates your tables.

***BLOBCol*:** A column for binary data. Presently works only with MySQL, PostgreSQL and SQLite backends.

***BoolCol*:** Will create a `BOOLEAN` column in Postgres, or `INT` in other databases. It will also convert values to `"t"`/`"f"` or `0/1` according to the database backend.

***CurrencyCol*:** Equivalent to `DecimalCol(size=10, precision=2)`. WARNING: as DecimalCol MAY NOT return precise numbers, this column may share the same behavior. Please read the DecimalCol warning.

***DateTimeCol*:** A date and time (usually returned as an datetime or mxDateTime object).

***DateCol*:** A date (usually returned as an datetime or mxDateTime object).

***TimeCol*:** A time (usually returned as an datetime or mxDateTime object).

***TimestampCol*:** Supports MySQL TIMESTAMP type.

***DecimalCol*:** Base-10, precise number. Uses the keyword arguments *size* for number of digits stored, and *precision* for the number of digits after the decimal point. WARNING: it may happen that DecimalCol values, although correctly stored in the DB, are returned as floats instead of decimals. For example, due to the type affinity SQLite stores decimals as integers or floats (NUMERIC storage class). You should test with your database adapter, and you should try importing the Decimal type and your DB adapter before importing SQLObject.

***DecimalStringCol*:** Similar to *DecimalCol* but stores data as strings to work around problems in some drivers and type affinity problem in SQLite. As it stores data as strings the column cannot be used in SQL expressions (column1 + column2) and probably will has problems with ORDER BY.

***EnumCol*:** One of several string values – give the possible strings as a list, with the *enumValues* keyword argument. MySQL has a native `ENUM` type, but will work with other databases too (storage just won't be as efficient).

For PostgreSQL, EnumCol's are implemented using check constraints. Due to the way PostgreSQL handles check constraints involving NULL, specifying None as a member of an EnumCol will effectively mean that, at the SQL level, the check constraint will be ignored (see http://archives.postgresql.org/pgsql-sql/2004-12/msg00065.php for more details).

***SetCol*:** Supports MySQL SET type.

***FloatCol*:** Floats.

***ForeignKey*:** A key to another table/class. Use like `user = ForeignKey('User')`. It can check for referential integrity using the keyword argument *cascade*, please see *ForeignKey* for details.

***IntCol*:** Integers.

***JsonbCol*:** A column for jsonb objects. Only supported on Postgres. Any Python object that can be serialized with json.dumps can be stored.

***JSONCol*:** A universal json column that converts simple Python objects (None, bool, int, float, long, dict, list, str/unicode to/from JSON using json.dumps/loads. A subclass of StringCol.

***PickleCol*:** An extension of BLOBCol; this column can store/retrieve any Python object; it actually (un)pickles the object from/to string and stores/retrieves the string. One can get and set the value of the column but cannot search (use it in WHERE).

***StringCol*:** A string (character) column. Extra keywords:

***length*:** If given, the type will be something like `VARCHAR(length)`. If not given, then `TEXT` is assumed (i.e., lengthless).

***varchar*:** A boolean; if you have a length, differentiates between `CHAR` and `VARCHAR`, default True, i.e., use `VARCHAR`.

***UnicodeCol*:** A subclass of *StringCol*. Also accepts a *dbEncoding* keyword argument, it defaults to `None` which means to lookup *dbEncoding* in *sqlmeta* and connection, and if *dbEncoding* isn't defined anywhere it defaults to `"utf-8"`. Values coming in and out from the database will be encoded and decoded. **Note**: there are some limitations on using UnicodeCol in queries:

- only simple q-magic fields are supported; no expressions;
- only == and != operators are supported;

The following code works:

```
MyTable.select(u'value' == MyTable.q.name)
MyTable.select(MyTable.q.name != u'value')
MyTable.select(OR(MyTable.q.col1 == u'value1', MyTable.q.col2 != u'value2'))
MyTable.selectBy(name = u'value')
MyTable.selectBy(col1=u'value1', col2=u'value2')
MyTable.byCol1(u'value1') # if col1 is an alternateID
```

The following does not work:

```
MyTable.select((MyTable.q.name + MyTable.q.surname) == u'value')
```

In that case you must apply the encoding yourself:

```
MyTable.select((MyTable.q.name + MyTable.q.surname) == u'value'.
↪encode(dbEncoding))
```

***UuidCol*:** A column for UUID. On Postgres uses 'UUID' data type, on all other backends uses VARCHAR(36).

Relationships Between Classes/Tables

ForeignKey

You can use the *ForeignKey* to handle foreign references in a table, but for back references and many-to-many relationships you'll use joins.

ForeignKey allows you to specify referential integrity using the keyword *cascade*, which can have these values:

***None*:** No action is taken on related deleted columns (this is the default). Following the Person/Address example, if you delete the object *Person* with id 1 (John Doe), the *Address* with id 1 (123 W Main St) will be kept untouched (with `personID=1`).

***False*:** Deletion of an object that has other objects related to it using a *ForeignKey* will fail (sets `ON DELETE RESTRICT`). Following the Person/Address example, if you delete the object *Person* with id 1 (John Doe) a *SQLObjectIntegrityError* exception will be raised, because the *Address* with id 1 (123 W Main St) has a reference (`personID=1`) to it.

***True*:** Deletion of an object that has other objects related to it using a *ForeignKey* will delete all the related objects too (sets `ON DELETE CASCADE`). Following the Person/Address example, if you delete the object *Person* with id 1 (John Doe), the *Address* with id 1 (123 W Main St) will be deleted too.

***'null'*:** Deletion of an object that has other objects related to it using a *ForeignKey* will set the *ForeignKey* column to *NULL/None* (sets `ON DELETE SET NULL`). Following the Person/Address example, if you delete the object *Person* with id 1 (John Doe), the *Address* with id 1 (123 W Main St) will be kept but the reference to person will be set to *NULL/None* (`personID=None`).

MultipleJoin and SQLMultipleJoin: One-to-Many

See *One-to-Many Relationships* for an example of one-to-many relationships.

MultipleJoin returns a list of results, while SQLMultipleJoin returns a SelectResults object.

Several keyword arguments are allowed to the *MultipleJoin* constructor:

***joinColumn*:** The column name of the key that points to this table. So, if you have a table `Product`, and another table has a column `ProductNo` that points to this table, then you'd use `joinColumn="ProductNo"`. WARNING: the argument you pass must conform to the column name in the database, not to the column in the class. So, if you have a SQLObject containing the `ProductNo` column, this will probably be translated into `product_no_id` in the DB (`product_no` is the normal uppercase- to-lowercase + underscores SQLO Translation, the added _id is just because the column referring to the table is probably a ForeignKey, and SQLO translates foreign keys that way). You should pass that parameter.

***orderBy*:** Like the *orderBy* argument to *select()*, you can specify the order that the joined objects should be returned in. *defaultOrder* will be used if not specified; `None` forces unordered results.

***joinMethodName*:** When adding joins dynamically (using the class method *addJoin*), you can give the name of the accessor for the join. It can also be created automatically, and is normally implied (i.e., `addresses = MultipleJoin(...)` implies `joinMethodName="addresses"`).

RelatedJoin and SQLRelatedJoin: Many-to-Many

See *Many-to-Many Relationships* for examples of using many-to-many joins.

RelatedJoin returns a list of results, while SQLRelatedJoin returns a SelectResults object.

RelatedJoin has all the keyword arguments of *MultipleJoin*, plus:

***otherColumn*:** Similar to *joinColumn*, but referring to the joined class. Same warning about column name.

***intermediateTable*:** The name of the intermediate table which references both classes. WARNING: you should pass the database table name, not the SQLO class representing.

***addRemoveName*:** In the *user/role example*, the methods *addRole(role)* and *removeRole(role)* are created. The `Role` portion of these method names can be changed by giving a string value here.

***createRelatedTable*:** default: `True`. If `False`, then the related table won't be automatically created; instead you must manually create it (e.g., with explicit SQLObject classes for the joins). New in 0.7.1.

Note: Let's suppose you have SQLObject-inherited classes Alpha and Beta, and an AlphasAndBetas used for the many-to-many relationship. AlphasAndBetas contains the alphaIndex Foreign Key column referring to Alpha, and the betaIndex FK column referring to Beta. if you want a 'betas' RelatedJoin in Alpha, you should add it to Alpha passing 'Beta' (class name!) as the first parameter, then passing 'alpha_index_id' as joinColumn, 'beta_index_id' as otherColumn, and 'alphas_and_betas' as intermediateTable.

An example schema that requires the use of *joinColumn*, *otherColumn*, and *intermediateTable*:

```
CREATE TABLE person (
    id SERIAL,
    username VARCHAR(100) NOT NULL UNIQUE
);

CREATE TABLE role (
    id SERIAL,
    name VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE assigned_roles (
    person INT NOT NULL,
    role INT NOT NULL
);
```

Then the usage in a class:

```
class Person(SQLObject):
    username = StringCol(length=100, alternateID=True)
    roles = RelatedJoin('Role', joinColumn='person', otherColumn='role',
                        intermediateTable='assigned_roles')
class Role(SQLObject):
    name = StringCol(length=50, alternateID=True)
    roles = RelatedJoin('Person', joinColumn='role', otherColumn='person',
                        intermediateTable='assigned_roles')
```

SingleJoin: One-to-One

Similar to *MultipleJoin*, but returns just one object, not a list.

Connection pooling

Connection object acquires a new low-level DB API connection from the pool and stores it; the low-level connection is removed from the pool; "releasing" means "return it to the pool". For single-threaded programs there is one connection in the pool.

If the pool is empty a new low-level connection opened; if one has disabled pooling (by setting conn._pool = None) the connection will be closed instead of returning to the pool.

Transactions

Transaction support in SQLObject is left to the database. Transactions can be used like:

```
conn = DBConnection.PostgresConnection('yada')
trans = conn.transaction()
p = Person.get(1, trans)
p.firstName = 'Bob'
trans.commit()
p.firstName = 'Billy'
trans.rollback()
```

The `trans` object here is essentially a wrapper around a single database connection, and *commit* and *rollback* just pass that message to the low-level connection.

One can call as much `.commit()`'s, but after a `.rollback()` one has to call `.begin()`. The last `.commit()` should be called as `.commit(close=True)` to release low-level connection back to the connection pool.

You can use SELECT FOR UPDATE in those databases that support it:

```
Person.select(Person.q.name=="value", forUpdate=True, connection=trans)
```

Method `sqlhub.doInTransaction` can be used to run a piece of code in a transaction. The method accepts a callable and positional and keywords arguments. It begins a transaction using its `processConnection` or `threadConnection`, calls the callable, commits the transaction and closes the underlying connection; it returns whatever the callable returned. If an error occurs during call to the callable it rolls the transaction back and reraise the exception.

Automatic Schema Generation

All the connections support creating and dropping tables based on the class definition. First you have to prepare your class definition, which means including type information in your columns.

Indexes

You can also define indexes for your tables, which is only meaningful when creating your tables through SQLObject (SQLObject relies on the database to implement the indexes). You do this again with attribute assignment, like:

```
firstLastIndex = DatabaseIndex('firstName', 'lastName')
```

This creates an index on two columns, useful if you are selecting a particular name. Of course, you can give a single column, and you can give the column object (`firstName`) instead of the string name. Note that if you use `unique` or `alternateID` (which implies `unique`) the database may make an index for you, and primary keys are always indexed.

If you give the keyword argument `unique` to *DatabaseIndex* you'll create a unique index – the combination of columns must be unique.

You can also use dictionaries in place of the column names, to add extra options. E.g.:

```
lastNameIndex = DatabaseIndex({'expression': 'lower(last_name)'})
```

In that case, the index will be on the lower-case version of the column. It seems that only PostgreSQL supports this. You can also do:

```
lastNameIndex = DatabaseIndex({'column': lastName, 'length': 10})
```

Which asks the database to only pay attention to the first ten characters. Only MySQL supports this, but it is ignored in other databases.

Creating and Dropping Tables

To create a table call *createTable*. It takes two arguments:

***ifNotExists*:** If the table already exists, then don't try to create it. Default False.

***createJoinTables*:** If you used *Many-to-Many relationships*, then the intermediate tables will be created (but only for one of the two involved classes). Default True.

dropTable takes arguments *ifExists* and *dropJoinTables*, self-explanatory.

Dynamic Classes

SQLObject classes can be manipulated dynamically. This leaves open the possibility of constructing SQLObject classes from an XML file, from database introspection, or from a graphical interface.

Automatic Class Generation

SQLObject can read the table description from the database, and fill in the class columns (as would normally be described in the *_columns* attribute). Do this like:

```
class Person(SQLObject):
    class sqlmeta:
        fromDatabase = True
```

You can still specify columns (in *_columns*), and only missing columns will be added.

Runtime Column and Join Changes

You can add and remove columns to your class at runtime. Such changes will effect all instances, since changes are made in place to the class. There are two methods of the *class sqlmeta object*, *addColumn* and *delColumn*, both of which take a *Col* object (or subclass) as an argument. There's also an option argument *changeSchema* which, if True, will add or drop the column from the database (typically with an `ALTER` command).

When adding columns, you must pass the name as part of the column constructor, like `StringCol("username", length=20)`. When removing columns, you can either use the Col object (as found in *sqlmeta.columns*, or which you used in *addColumn*), or you can use the column name (like `MyClass.delColumn("username")`). You can also add *Joins*, like `MyClass.addJoin(MultipleJoin("MyOtherClass"))`, and remove joins with *delJoin*. *delJoin* does not take strings, you have to get the join object out of the *sqlmeta.joins* attribute.

Legacy Database Schemas

Often you will have a database that already exists, and does not use the naming conventions that SQLObject expects, or does not use any naming convention at all.

SQLObject requirements

While SQLObject tries not to make too many requirements on your schema, some assumptions are made. Some of these may be relaxed in the future.

All tables that you want to turn into a class need to have an integer primary key. That key should be defined like:

MySQL: `INT PRIMARY KEY AUTO_INCREMENT`

Postgres: `SERIAL PRIMARY KEY`

SQLite: `INTEGER PRIMARY KEY AUTOINCREMENT`

SQLObject does not support primary keys made up of multiple columns (that probably won't change). It does not generally support tables with primary keys with business meaning – i.e., primary keys are assumed to be immutable (that won't change).

At the moment foreign key column names must end in `"ID"` (case-insensitive). This restriction will probably be removed in the next release.

Workaround for primary keys made up of multiple columns

If the database table/view has ONE NUMERIC Primary Key then sqlmeta - idName should be used to map the table column name to SQLObject id column.

If the Primary Key consists only of number columns it is possible to create a virtual column `id` this way:

Example for Postgresql:

> select '1'||lpad(PK1,max_length_of_PK1,'0')||lpad(PK2,max_length_of_PK2,'0')||...||lpad(PKn,max_length_of_PKn,'0') as "id", column_PK1, column_PK2, .., column_PKn, column... from table;

Note:

- The arbitrary '1' at the beginning of the string to allow for leading zeros of the first PK.
- The application designer has to determine the maximum length of each Primary Key.

This statement can be saved as a view or the column can be added to the database table, where it can be kept up to date with a database trigger.

Obviously the "view" method does generally not allow insert, updates or deletes. For Postgresql you may want to consult the chapter "RULES" for manipulating underlying tables.

For an alphanumeric Primary Key column a similar method is possible:

Every character of the lpaded PK has to be transfered using ascii(character) which returns a 3digit number which can be concatenated as shown above.

Caveats:

- this way the `id` may become a very large integer number which may cause troubles elsewhere.
- no performance loss takes place if the where clauses specifies the PK columns.

Example: CD-Album * Album: PK=ean * Tracks: PK=ean,disc_nr,track_nr

The database view to show the tracks starts:

> SELECT ean||lpad("disc_nr",2,'0')||lpad("track_nr",2,'0') as id, ... Note: no leading '1' and no padding necessary for ean numbers

Tracks.select(Tracks.q.ean==id) ... where id is the ean of the Album.

Changing the Naming Style

By default names in SQLObject are expected to be mixed case in Python (like `mixedCase`), and underscore-separated in SQL (like `mixed_case`). This applies to table and column names. The primary key is assumed to be simply `id`.

Other styles exist. A typical one is mixed case column names, and a primary key that includes the table name, like `ProductID`. You can use a different *Style* object to indicate a different naming convention. For instance:

```
class Person(SQLObject):
    class sqlmeta:
        style = MixedCaseStyle(longID=True)
    firstName = StringCol()
    lastName = StringCol()
```

If you use `Person.createTable()`, you'll get:

```
CREATE TABLE Person (
    PersonID INT PRIMARY KEY,
    FirstName Text,
    LastName Text
)
```

The *MixedCaseStyle* object handles the initial capitalization of words, but otherwise leaves them be. By using `longID=True`, we indicate that the primary key should look like a normal reference (`PersonID` for *MixedCaseStyle*, or `person_id` for the default style).

If you wish to change the style globally, assign the style to the connection, like:

```
__connection__.style = MixedCaseStyle(longID=True)
```

Irregular Naming

This is now covered in the *Class sqlmeta* section.

Non-Integer Keys

While not strictly a legacy database issue, this fits into the category of "irregularities". If you use non-integer keys, all primary key management is up to you. You must create the table yourself (SQLObject can create tables with int or str IDs), and when you create instances you must pass a `id` keyword argument into constructor (like `Person(id='555-55-5555', ...)`).

DBConnection: Database Connections

The *DBConnection* module currently has six external classes, *MySQLConnection*, *PostgresConnection*, *SQLiteConnection*, *SybaseConnection*, *MaxdbConnection*, *MSSQLConnection*.

You can pass the keyword argument *debug* to any connector. If set to true, then any SQL sent to the database will also be printed to the console.

You can additionally pass *logger* keyword argument which should be a name of the logger to use. If specified and *debug* is `True`, SQLObject will write debug print statements via that logger instead of printing directly to console. The argument *loglevel* allows to choose the logging level - it can be `debug`, `info`, `warning`, `error`, `critical` or `exception`. In case *logger* is absent or empty SQLObject uses `print`'s instead of logging; *loglevel* can be `stdout` or `stderr` in this case; default is `stdout`.

To configure logging one can do something like that:

```
import logging
logging.basicConfig(
    filename='test.log',
    format='[%(asctime)s] %(name)s %(levelname)s: %(message)s',
    level=logging.DEBUG,
)
log = logging.getLogger("TEST")
log.info("Log started")

__connection__ = "sqlite:/:memory:?debug=1&logger=TEST&loglevel=debug"
```

The code redirects SQLObject debug messages to *test.log* file.

MySQL

MySQLConnection takes the keyword arguments *host*, *port*, *db*, *user*, and *password*, just like *MySQLdb.connect* does.

MySQLConnection supports all the features, though MySQL only supports *transactions* when using the InnoDB backend; SQLObject can explicitly define the backend using `sqlmeta.createSQL`.

Supported drivers are `mysqldb`, `connector`, `oursql` and `pymysql`, `pyodbc`, `pypyodbc` or `odbc` (try `pyodbc` and `pypyodbc`); defualt is `mysqldb`.

Keyword argument `conv` allows to pass a list of custom converters. Example:

```
import time
import sqlobject
import MySQLdb.converters

def _mysql_timestamp_converter(raw):
        """Convert a MySQL TIMESTAMP to a floating point number representing
        the seconds since the Un*x Epoch. It uses custom code the input seems
        to be the new (MySQL 4.1+) timestamp format, otherwise code from the
        MySQLdb module is used."""
        if raw[4] == '-':
            return time.mktime(time.strptime(raw, '%Y-%m-%d %H:%M:%S'))
        else:
            return MySQLdb.converters.mysql_timestamp_converter(raw)

conversions = MySQLdb.converters.conversions.copy()
conversions[MySQLdb.constants.FIELD_TYPE.TIMESTAMP] = _mysql_timestamp_converter

MySQLConnection = sqlobject.mysql.builder()
connection = MySQLConnection(user='foo', db='somedb', conv=conversions)
```

Connection-specific parameters are: `unix_socket`, `init_command`, `read_default_file`, `read_default_group`, `conv`, `connect_timeout`, `compress`, `named_pipe`, `use_unicode`, `client_flag`, `local_infile`, `ssl_key`, `ssl_cert`, `ssl_ca`, `ssl_capath`, `charset`.

Postgres

PostgresConnection takes a single connection string, like `"dbname=something user=some_user"`, just like *psycopg.connect*. You can also use the same keyword arguments as for *MySQLConnection*, and a dsn string will be constructed.

PostgresConnection supports transactions and all other features.

The user can choose a DB API driver for PostgreSQL by using a `driver` parameter in DB URI or PostgresConnection that can be a comma-separated list of driver names. Possible drivers are: `psycopg2`, psycopg1, `psycopg` (tries psycopg2 and psycopg1), `pygresql`, `pygresql`, `pypostgresql`, `pyodbc`, `pypyodbc` or `odbc` (try `pyodbc` and `pypyodbc`). Default is `psycopg`.

Connection-specific parameters are: `sslmode`, `unicodeCols`, `schema`, `charset`.

SQLite

SQLiteConnection takes the a single string, which is the path to the database file.

SQLite puts all data into one file, with a journal file that is opened in the same directory during operation (the file is deleted when the program quits). SQLite does not restrict the types you can put in a column – strings can go in integer columns, dates in integers, etc.

SQLite may have concurrency issues, depending on your usage in a multi-threaded environment.

The user can choose a DB API driver for SQLite by using a `driver` parameter in DB URI or SQLiteConnection that can be a comma-separated list of driver names. Possible drivers are: `pysqlite2` (alias `sqlite2`), `sqlite3`, `sqlite` (alias `sqlite1`). Default is to test pysqlite2, sqlite3 and sqlite in that order.

Connection-specific parameters are: `encoding`, `mode`, `timeout`, `check_same_thread`, `use_table_info`.

Firebird

FirebirdConnection takes the arguments *host*, *db*, *user* (default `"sysdba"`), *password* (default `"masterkey"`).

Firebird supports all the features. Support is still young, so there may be some issues, especially with concurrent access, and especially using lazy selects. Try `list(MyClass.select())` to avoid concurrent cursors if you have problems (using `list()` will pre-fetch all the results of a select).

Firebird support `fdb`, `kinterbasdb` or `firebirdsql` drivers. Default are `fdb` and `kinterbasdb`.

There could be a problem if one tries to connect to a server running on w32 from a program running on Unix; the problem is how to specify the database so that SQLObject correctly parses it. Vertical bar is replaces by a semicolon only on a w32. On Unix a vertical bar is a pretty normal character and must not be processed.

The most correct way to fix the problem is to connect to the DB using a database name, not a file name. In the Firebird a DBA can set an alias instead of database name in the aliases.conf file

Example from Firebird 2.0 Administrators Manual:

```
# fbdb1 is on a Windows server:
fbdb1 = c:\Firebird\sample\Employee.fdb
```

Now a program can connect to firebird://host:port/fbdb1.

One can edit aliases.conf whilst the server is running. There is no need to stop and restart the server in order for new aliases.conf entries to be recognised.

If you are using indexes and get an error like *key size exceeds implementation restriction for index*, see this page to understand the restrictions on your indexing.

Connection-specific parameters are: `dialect`, `role`, `charset`.

Sybase

SybaseConnection takes the arguments *host*, *db*, *user*, and *password*. It also takes the extra boolean argument *locking* (default True), which is passed through when performing a connection. You may use a False value for *locking* if you are not using multiple threads, for a slight performance boost.

It uses the Sybase module.

Connection-specific parameters are: `locking`, `autoCommit`.

MAX DB

MAX DB, also known as SAP DB, is available from a partnership of SAP and MySQL. It takes the typical arguments: *host*, *database*, *user*, *password*. It also takes the arguments *sqlmode* (default `"internal"`), *isolation*, and *timeout*, which are passed through when creating the connection to the database.

It uses the sapdb module.

Connection-specific parameters are: `autoCommit`, `sqlmode`, `isolation`, `timeout`.

MS SQL Server

The *MSSQLConnection* objects wants to use new style connection strings in the format of

mssql://user:pass@host:port/db

This will then be mapped to either the correct driver format. If running SQL Server on a "named" port, make sure to specify the port number in the URI.

The two drivers currently supported are adodbapi and pymssql.

The user can choose a DB API driver for MSSQL by using a `driver` parameter in DB URI or MSSQLConnection that can be a comma-separated list of driver names. Possible drivers are: `adodb` (alias `adodbapi`) and `pymssql`. Default is to test `adodbapi` and `pymssql` in that order.

Connection-specific parameters are: `autoCommit`, `timeout`.

Events (signals)

Signals are a mechanism to be notified when data or schema changes happen through SQLObject. This may be useful for doing custom data validation, logging changes, setting default attributes, etc. Some of what signals can do is also possible by overriding methods, but signals may provide a cleaner way, especially across classes not related by inheritance.

Example:

```
from sqlobject.events import listen, RowUpdateSignal, RowCreatedSignal
from model import Users

def update_listener(instance, kwargs):
    """keep "last_updated" field current"""
    import datetime
    # BAD method 1, causes infinite recursion?
    # instance should be read-only
    instance.last_updated = datetime.datetime.now()
    # GOOD method 2
    kwargs['last_updated'] = datetime.datetime.now()
```

```
def created_listener(instance, kwargs, post_funcs):
    """email me when new users added"""
    # email() implementation left as an exercise for the reader
    msg = "%s just was just added to the database!" % kwargs['name']
    email(msg)

listen(update_listener, Users, RowUpdateSignal)
listen(created_listener, Users, RowCreatedSignal)
```

Exported Symbols

You can use `from sqlobject import *`, though you don't have to. It exports a minimal number of symbols. The symbols exported:

From *sqlobject.main*:

- *NoDefault*
- *SQLObject*
- *getID*
- *getObject*

From *sqlobject.col*: * *Col* * *StringCol* * *IntCol* * *FloatCol* * *KeyCol* * *ForeignKey* * *EnumCol* * *SetCol* * *DateTimeCol* * *DateCol* * *TimeCol* * *TimestampCol* * *DecimalCol* * *CurrencyCol*

From *sqlobject.joins*: * *MultipleJoin* * *RelatedJoin*

From *sqlobject.styles*: * *Style* * *MixedCaseUnderscoreStyle* * *DefaultStyle* * *MixedCaseStyle*

From *sqlobject.sqlbuilder*:

- *AND*
- *OR*
- *NOT*
- *IN*
- *LIKE*
- *DESC*
- *CONTAINSSTRING*
- *const*
- *func*

LEFT JOIN and other JOINs

First look in the FAQ, question "How can I do a LEFT JOIN?"

Still here? Well. To perform a JOIN use one of the JOIN helpers from SQLBuilder. Pass an instance of the helper to .select() method. For example:

```
from sqlobject.sqlbuilder import LEFTJOINOn
MyTable.select(
    join=LEFTJOINOn(Table1, Table2,
                    Table1.q.name == Table2.q.value))
```

will generate the query:

```
SELECT my_table.* FROM my_table, table1
LEFT JOIN table2 ON table1.name = table2.value;
```

If you want to join with the primary table - leave the first table None:

```
MyTable.select(
    join=LEFTJOINOn(None, Table1,
                    MyTable.q.name == Table1.q.value))
```

will generate the query:

```
SELECT my_table.* FROM my_table
LEFT JOIN table2 ON my_table.name = table1.value;
```

The join argument for .select() can be a JOIN() or a sequence (list/tuple) of JOIN()s.

Available joins are JOIN, INNERJOIN, CROSSJOIN, STRAIGHTJOIN, LEFTJOIN, LEFTOUTERJOIN, NATURALJOIN, NATURALLEFTJOIN, NATURALLEFTOUTERJOIN, RIGHTJOIN, RIGHTOUTERJOIN, NATURALRIGHTJOIN, NATURALRIGHTOUTERJOIN, FULLJOIN, FULLOUTERJOIN, NATURALFULLJOIN, NATURALFULLOUTERJOIN, INNERJOINOn, LEFTJOINOn, LEFTOUTERJOINOn, RIGHTJOINOn, RIGHTOUTERJOINOn, FULLJOINOn, FULLOUTERJOINOn, INNERJOINUsing, LEFTJOINUsing, LEFTOUTERJOINUsing, RIGHTJOINUsing, RIGHTOUTERJOINUsing, FULLJOINUsing, FULLOUTERJOINUsing.

How can I join a table with itself?

Use Alias from SQLBuilder. Example:

```
from sqlobject.sqlbuilder import Alias
alias = Alias(MyTable, "my_table_alias")
MyTable.select(MyTable.q.name == alias.q.value)
```

will generate the query:

```
SELECT my_table.* FROM my_table, my_table AS my_table_alias
WHERE my_table.name = my_table_alias.value;
```

Can I use a JOIN() with aliases?

Sure! That's a situation the JOINs and aliases were primary developed for. Code:

```
from sqlobject.sqlbuilder import LEFTJOINOn, Alias
alias = Alias(OtherTable, "other_table_alias")
MyTable.select(MyTable.q.name == OtherTable.q.value,
    join=LEFTJOINOn(MyTable, alias, MyTable.col1 == alias.q.col2))
```

will result in the query:

```
SELECT my_table.* FROM other_table,
    my_table LEFT JOIN other_table AS other_table_alias
WHERE my_table.name == other_table.value AND
    my_table.col1 = other_table_alias.col2.
```

Subqueries (subselects)

You can run queries with subqueries (subselects) on those DBMS that can do subqueries (MySQL supports subqueries from version 4.1).

Use corresponding classes and functions from SQLBuilder:

```
from sqlobject.sqlbuilder import EXISTS, Select
select = Test1.select(EXISTS(Select(Test2.q.col2, where=(Outer(Test1).q.col1 == Test2.
↪q.col2))))
```

generates the query:

```
SELECT test1.id, test1.col1 FROM test1 WHERE
EXISTS (SELECT test2.col2 FROM test2 WHERE (test1.col1 = test2.col2))
```

Note the usage of Outer - it is a helper to allow referring to a table in the outer query.

Select() is used instead of .select() because you need to control what columns the inner query returns.

Available queries are `IN()`, `NOTIN()`, `EXISTS()`, `NOTEXISTS()`, `SOME()`, `ANY()` and `ALL()`. The last 3 are used with comparison operators, like this: `somevalue = ANY(Select(...))`.

Utilities

Some useful utility functions are included with SQLObject. For more information see their module docstrings.

- sqlobject.util.csvexport

SQLBuilder

For more information on SQLBuilder, read the SQLBuilder Documentation.

SQLObject FAQ

Contents

SQLExpression

In *SomeTable.select(SomeTable.q.Foo > 30)* why doesn't the inner parameter, *SomeTable.q.Foo > 30*, get evaluated to some boolean value?

q is an object that returns special attributes of type *sqlbuilder.SQLExpression*. SQLExpression is a special class that overrides almost all Python magic methods and upon any operation instead of evaluating it constructs another instance of SQLExpression that remembers what operation it has to do. This is similar to a symbolic algebra. Example:

SQLExpression("foo") > 30

produces SQLExpression("foo", ">", 30) (well, it really produces SQLExpression(SQLExpression("foo")...))

How does the select(...) method know what to do?

In short, select() recursively evaluates the top-most SQLExpression to a string:

SQLExpression("foo", ">", 30) => "foo > 30"

and passes the result as a string to the SQL backend.

The longer but more detailed and correct explanation is that select() produces an instance of SelectResults class that upon being iterated over produces an instance of Iteration class that upon calling its next() method (it is iterator!) constructs the SQL query string, passes it to the backend, fetches the results, wraps every row as SQLObject instance and passes them back to the user.

For the details of the implementation see sqlobject/main.py for SQLObject, sqlobject/sqlbuilder.py for SQLExpression, sqlobject/dbconnection.py for DBConnection class (that constructs the query strings) and Iteration class, and different subdirectories of sqlobject for concrete implementations of connection classes - different backends require different query strings.

Why there is no __len__?

There are reasons why there is no __len__ method, though many people think having those make them feel more integrated into Python.

One is that len(foo) is expected to be fast, but issuing a COUNT query can be slow. Worse, often this causes the database to do essentially redundant work when the actual query is performed (generally taking the len of a sequence is followed by accessing items from that sequence).

Another is that list(foo) implicitly tries to do a len first, as an optimization (because len is expected to be cheap – see previous point). Worse, it swallows *all* exceptions that occur during that call to __len__, so if it fails (e.g. there's a typo somewhere in the query), the original cause is silently discarded, and instead you're left with mysterious errors like "current transaction is aborted, commands ignored until end of transaction block" for no apparent reason.

How can I do a LEFT JOIN?

The short: you can't. You don't need to. That's a relational way of thinking, not an object way of thinking. But it's okay! It's not hard to do the same thing, even if it's not with the same query.

For these examples, imagine you have a bunch of customers, with contacts. Not all customers have a contact, some have several. The left join would look like:

```
SELECT customer.id, customer.first_name, customer.last_name,
       contact.id, contact.address
FROM customer
LEFT JOIN contact ON contact.customer_id = customer.id
```

Simple

```
for customer in Customer.select():
    print customer.firstName, customer.lastName
    for contact in customer.contacts:
        print '   ', contact.phoneNumber
```

The effect is the same as the left join – you get all the customers, and you get all their contacts. The problem, however, is that you will be executing more queries – a query for each customer to fetch the contacts – where with the left join you'd only do one query. The actual amount of information returned from the database will be the same. There's a good chance that this won't be significantly slower. I'd advise doing it this way unless you hit an actual performance problem.

Efficient

Lets say you really don't want to do all those queries. Okay, fine:

```
custContacts = {}
for contact in Contact.select():
    custContacts.setdefault(contact.customerID, []).append(contact)
for customer in Customer.select():
    print customer.firstName, customer.lastName
    for contact in custContacts.get(customer.id, []):
        print '   ', contact.phoneNumber
```

This way there will only be at most two queries. It's a little more crude, but this is an optimization, and optimizations often look less than pretty.

But, say you don't want to get everyone, just some group of people (presumably a large enough group that you still need this optimization):

```
query = Customer.q.firstName.startswith('J')
custContacts = {}
for contact in Contact.select(AND(Contact.q.customerID == Customer.q.id,
                                  query)):
    custContacts.setdefault(contact.customerID, []).append(contact)
for customer in Customer.select(query):
    print customer.firstName, customer.lastName
    for contact in custContacts.get(customer.id, []):
        print '   ', contact.phoneNumber
```

SQL-wise

Use LEFTJOIN() from SQLBuilder.

How can I join a table with itself?

Use Alias from SQLBuilder. See example.

How can I define my own intermediate table in my Many-to-Many relationship?

Note: In User and Role, SQLRelatedJoin is used with createRelatedTable=False so the intermediate table is not created automatically. We also set the intermediate table name with intermediateTable='user_roles'. UserRoles is the definition of our intermediate table. UserRoles creates a unique index to make sure we don't have duplicate data in the database. We also added an extra field called active which has a boolean value. The active column might be used to activate/deactivate a given role for a user in this example. Another common field to add in this an intermediate table might be a sort field. If you want to get a list of rows from the intermediate table directly add a MultipleJoin to User or Role class.

We'll expand on the User and Role example and define our own UserRoles class which will be the intermediate table for the User and Role Many-to-Many relationship.

Example:

```
>>> class User(SQLObject):
...     class sqlmeta:
...         table = "user_table"
...     username = StringCol(alternateID=True, length=20)
...     roles = SQLRelatedJoin('Role',
...         intermediateTable='user_roles',
...         createRelatedTable=False)

>>> class Role(SQLObject):
...     name = StringCol(alternateID=True, length=20)
...     users = SQLRelatedJoin('User',
...         intermediateTable='user_roles',
...         createRelatedTable=False)

>>> class UserRoles(SQLObject):
...     class sqlmeta:
...         table = "user_roles"
...     user = ForeignKey('User', notNull=True, cascade=True)
...     role = ForeignKey('Role', notNull=True, cascade=True)
```

```
...     active = BoolCol(notNull=True, default=False)
...     unique = index.DatabaseIndex(user, role, unique=True)
```

How Does Inheritance Work?

SQLObject is not intended to represent every Python inheritance structure in an RDBMS – rather it is intended to represent RDBMS structures as Python objects. So lots of things you can do in Python you can't do with SQLObject classes. However, some form of inheritance is possible.

One way of using this is to create local conventions. Perhaps:

```
class SiteSQLObject(SQLObject):
    _connection = DBConnection.MySQLConnection(user='test', db='test')
    _style = MixedCaseStyle()

    # And maybe you want a list of the columns, to autogenerate
    # forms from:
    def columns(self):
        return [col.name for col in self._columns]
```

Since SQLObject doesn't have a firm introspection mechanism (at least not yet) the example shows the beginnings of a bit of ad hoc introspection (in this case exposing the `_columns` attribute in a more pleasing/public interface).

However, this doesn't relate to *database* inheritance at all, since we didn't define any columns. What if we do?

```
class Person(SQLObject):
    firstName = StringCol()
    lastName = StringCol()

class Employee(Person):
    position = StringCol()
```

Unfortunately, the resultant schema probably doesn't look like what you might have wanted:

```
CREATE TABLE person (
    id INT PRIMARY KEY,
    first_name TEXT,
    last_name TEXT
);

CREATE TABLE employee (
    id INT PRIMARY KEY
    first_name TEXT,
    last_name TEXT,
    position TEXT
)
```

All the columns from `person` are just repeated in the `employee` table. What's more, an ID for a Person is distinct from an ID for an employee, so for instance you must choose `ForeignKey("Person")` or `ForeignKey("Employee")`, you can't have a foreign key that sometimes refers to one, and sometimes refers to the other.

Altogether, not very useful. You probably want a `person` table, and then an `employee` table with a one-to-one relation between the two. Of course, you can have that, just create the appropriate classes/tables – but it will appear as two distinct classes, and you'd have to do something like `Person(1).employee.position`. Of course, you can always create the necessary shortcuts, like:

```
class Person(SQLObject):
    firstName = StringCol()
    lastName = StringCol()

    def _get_employee(self):
        value = Employee.selectBy(person=self)
        if value:
            return value[0]
        else:
            raise AttributeError, '%r is not an employee' % self
    def _get_isEmployee(self):
        value = Employee.selectBy(person=self)
        # turn into a bool:
        return not not value
    def _set_isEmployee(self, value):
        if value:
            # Make sure we are an employee...
            if not self.isEmployee:
                Empoyee.new(person=self, position=None)
        else:
            if self.isEmployee:
                self.employee.destroySelf()
    def _get_position(self):
        return self.employee.position
    def _set_position(self, value):
        self.employee.position = value

class Employee(SQLObject):
    person = ForeignKey('Person')
    position = StringCol()
```

There is also another kind of inheritance. See Inheritance.html

Composite/Compound Attributes

A composite attribute is an attribute formed from two columns. For example:

```
CREATE TABLE invoice_item (
    id INT PRIMARY KEY,
    amount NUMERIC(10, 2),
    currency CHAR(3)
);
```

Now, you'll probably want to deal with one amount/currency value, instead of two columns. SQLObject doesn't directly support this, but it's easy (and encouraged) to do it on your own:

```
class InvoiceItem(SQLObject):
    amount = Currency()
    currency = StringChar(length=3)

    def _get_price(self):
        return Price(self.amount, self.currency)
    def _set_price(self, price):
        self.amount = price.amount
        self.currency = price.currency

class Price(object):
```

```
    def __init__(self, amount, currency):
        self._amount = amount
        self._currency = currency

    def _get_amount(self):
        return self._amount
    amount = property(_get_amount)

    def _get_currency(self):
        return self._currency
    currency = property(_get_currency)

    def __repr__(self):
        return '<Price: %s %s>' % (self.amount, self.currency)
```

You'll note we go to some trouble to make sure that `Price` is an immutable object. This is important, because if `Price` wasn't and someone changed an attribute, the containing `InvoiceItem` instance wouldn't detect the change and update the database. (Also, since `Price` doesn't subclass `SQLObject`, we have to be explicit about creating properties) Some people refer to this sort of class as a *Value Object*, that can be used similar to how an integer or string is used.

You could also use a mutable composite class:

```
class Address(SQLObject):
    street = StringCol()
    city = StringCol()
    state = StringCol(length=2)

    latitude = FloatCol()
    longitude = FloatCol()

    def _init(self, id):
        SQLObject._init(self, id)
        self._coords = SOCoords(self)

    def _get_coords(self):
        return self._coords

class SOCoords(object):
    def __init__(self, so):
        self._so = so

    def _get_latitude(self):
        return self._so.latitude
    def _set_latitude(self, value):
        self._so.latitude = value
    latitude = property(_get_latitude, set_latitude)

    def _get_longitude(self):
        return self._so.longitude
    def _set_longitude(self, value):
        self._so.longitude = value
    longitude = property(_get_longitude, set_longitude)
```

Pretty much a proxy, really, but `SOCoords` could contain other logic, could interact with non-SQLObject-based latitude/longitude values, or could be used among several objects that have latitude/longitude columns.

Non-Integer IDs

Yes, you can use non-integer IDs.

If you use non-integer IDs, you will not be able to use automatic `CREATE TABLE` generation (i.e., `createTable`); SQLObject can create tables with int or str IDs. You also will have to give your own ID values when creating an object, like:

```
color = Something(id="blue", r=0, b=100, g=0)
```

IDs are, and always will in future versions, be considered immutable. Right now that is not enforced; you can assign to the `id` attribute. But if you do you'll just mess everything up. This will probably be taken away sometime to avoid possibly confusing bugs (actually, assigning to `id` is almost certain to cause confusing bugs).

If you are concerned about enforcing the type of IDs (which can be a problem even with integer IDs) you may want to do this:

```
def Color(SQLObject):
    def _init(self, id, connection=None):
        id = str(id)
        SQLObject._init(self, id, connection)
```

Instead of `str()` you may use `int()` or whatever else you want. This will be resolved in a future version when ID column types can be declared like other columns.

Additionally you can set idType=str in you SQLObject class.

Binary Values

Binary values can be difficult to store in databases, as SQL doesn't have a widely-implemented way to express binaries as literals, and there's differing support in database.

The module sqlobject.col defines validators and column classes that to some extent support binary values. There is BLOBCol that extends StringCol and allow to store binary values; currently it works only with PostgreSQL and MySQL. PickleCol extends BLOBCol and allows to store any object in the column; the column, naturally, pickles the object upon assignment and unpickles it upon retrieving the data from the DB.

Another possible way to keep binary data in a database is by using encoding. Base 64 is a good encoding, reasonably compact but also safe. As an example, imagine you want to store images in the database:

```
class Image(SQLObject):

    data = StringCol()
    height = IntCol()
    width = IntCol()

    def _set_data(self, value):
        self._SO_set_data(value.encode('base64'))

    def _get_data(self, value):
        return self._SO_get_data().decode('base64')
```

Reloading Modules

If you've tried to reload a module that defines SQLObject subclasses, you've probably encountered various odd errors. The short answer: you can't reload these modules.

The long answer: reloading modules in Python doesn't work very well. Reloading actually means *re-running* the module. Every `class` statement creates a class – but your old classes don't disappear. When you reload a module, new classes are created, and they take over the names in the module.

SQLObject, however, doesn't search the names in a module to find a class. When you say `ForeignKey('SomeClass')`, SQLObject looks for any SQLObject subclass anywhere with the name `SomeClass`. This is to avoid problems with circular imports and circular dependencies, as tables have forward- and back-references, and other circular dependencies. SQLObject resolves these dependencies lazily.

But when you reload a module, suddenly there will be two SQLObject classes in the process with the same name. SQLObject doesn't know that one of them is obsolete. And even if it did, it doesn't know every other place in the system that has a reference to that obsolete class.

For this reason and several others, reloading modules is highly error-prone and difficult to support.

Python Keywords

If you have a table column that is a Python keyword, you should know that the Python attribute doesn't have to match the name of the column. See Irregular Naming in the documentation.

Lazy Updates and Insert

Lazy updates allow you to defer sending `UPDATES` until you synchronize the object. However, there is no way to do a lazy insert; as soon as you create an instance the `INSERT` is executed.

The reason for this limit is that each object needs a database ID, and in many databases you cannot attain an ID until you create a row.

Mutually referencing tables

How can I create mutually referencing tables? For the code:

```
class Person(SQLObject):
    role = ForeignKey("Role")

class Role(SQLObject):
    person = ForeignKey("Person")

Person.createTable()
Role.createTable()
```

Postgres raises ProgrammingError: ERROR: relation "role" does not exist.

The correct way is to delay constraints creation until all tables are created:

```
class Person(SQLObject):
    role = ForeignKey("Role")

class Role(SQLObject):
    person = ForeignKey("Person")

constraints = Person.createTable(applyConstraints=False)
constraints += Role.createTable(applyConstraints=False)

for constraint in constraints:
    connection.query(constraint)
```

What about GROUP BY, UNION, etc?

In short - not every query can be represented in SQLObject. SQLOBject's objects are instances of "table" clasess:

```
class MyTable(SQLObject):
    ...

my_table_row = MyTable.get(id)
```

Now my_table_row is an instance of MyTable class and represents a row in the my_table table. But for a statement with GROUP BY like this:

```
SELECT my_column, COUNT(*) FROM my_table GROUP BY my_column;
```

there is no table, there is no corresponding "table" class, and SQLObject cannot return a list of meaningful objects.

You can use a lower-level machinery available in SQLBuilder.

How to do mass-insertion?

Mass-insertion using high-level API in SQLObject is slow. There are many reasons for that. First, on creation SQLObject instances pass all values through validators/converters which is convenient but takes time. Second, after an INSERT query SQLObject executes a SELECT query to get back autogenerated values (id and timestamps). Third, there is caching and cache maintaining. Most of this is unnecessary for mass-insertion, hence high-level API is unsuitable.

Less convenient (no validators) but much faster API is Insert from SQLBuilder.

How can I specify the MySQL engine to use, or tweak other SQL-engine specific features?

You can *ALTER* the table just after creation using the `sqlmeta` attribute `createSQL`, for example:

```
class SomeObject(SQLObject):
    class sqlmeta:
        createSQL = { 'mysql' : 'ALTER TABLE some_object ENGINE InnoDB' }
    # your columns here
```

Maybe you want to specify the charset too? No problem:

```
class SomeObject(SQLObject):
    class sqlmeta:
        createSQL = { 'mysql' : [
            'ALTER TABLE some_object ENGINE InnoDB',
            '''ALTER TABLE some_object CHARACTER SET utf8
                COLLATE utf8_estonian_ci''']
            }
```

SQLBuilder

Contents

A number of variables from SQLBuilder are included with `from sqlobject import *` – see the relevant SQLObject documentation for more. Its functionality is also available through the special `q` attribute of *SQLObject* classes.

SQLExpression

SQLExpression uses clever overriding of operators to make Python expressions build SQL expressions – so long as you start with a Magic Object that knows how to fake it.

With SQLObject, you get a Magic Object by accessing the `q` attribute of a table class – this gives you an object that represents the field. All of this is probably easier to grasp in an example:

```
>>> from sqlobject.sqlbuilder import *
>>> person = table.person
# person is now equivalent to the Person.q object from the SQLObject
# documentation
>>> person
person
>>> person.first_name
person.first_name
>>> person.first_name == 'John'
person.first_name = 'John'
>>> name = 'John'
>>> person.first_name != name
person.first_name != 'John'
>>> AND(person.first_name == 'John', person.last_name == 'Doe')
(person.first_name = 'John' AND person.last_name = 'Doe')
```

Most of the operators work properly: <, >, <=, >=, !=, ==, +, -, /, *, **, %. However, `and`, `or`, and `not` **do not work**. You can use &, |, and ~ instead – but be aware that these have the same precedence as multiplication. So:

```
# This isn't what you want:
>> person.first_name == 'John' & person.last_name == 'Doe'
(person.first_name = ('John' AND person.last_name)) = 'Doe')
# This is:
>> (person.first_name == 'John') & (person.last_name == 'Doe')
((person.first_name = 'John') AND (person.last_name == 'Doe'))
```

SQLBuilder also contains the functions `AND`, `OR`, and `NOT` which also work – I find these easier to work with. `AND` and `OR` can take any number of arguments.

You can also use `.startswith()` and `.endswith()` on an SQL expression – these will translate to appropriate `LIKE` statements and all `%` quoting is handled for you, so you can ignore that implementation detail. There is also a `LIKE` function, where you can pass your string, with `%` for the wildcard, as usual.

If you want to access an SQL function, use the `func` variable, like:

```
>> person.created < func.NOW()
```

To pass a constant, use the `const` variable which is actually an alias for func.

SQL statements

SQLBuilder implements objects that execute SQL statements. SQLObject uses them internally in its higher-level API, but users can use this mid-level API to execute SQL queries that are not supported by the high-level API. To use these objects first construct an instance of a statement object, then ask the connection to convert the instance to an SQL query and finally ask the connection to execute the query and return the results. For example, for `Select` class:

```
>>> from sqlobject.sqlbuilder import *
>> select = Select(['name', 'AVG(salary)'], staticTables=['employees'],
>>     groupBy='name') # create an instance
>> query = connection.sqlrepr(select) # Convert to SQL string:
>>     # SELECT name, AVG(salary) FROM employees GROUP BY name
>> rows = connection.queryAll(query) # Execute the query
>>     # and get back the results as a list of rows
>>     # where every row is a sequence of length 2 (name and average salary)
```

Select

A class to build `SELECT` queries. Accepts a number of parameters, all parameters except *items* are optional. Use `connection.queryAll(query)` to execute the query and get back the results as a list of rows.

***items*:** A string, an SQLExpression or a sequence of strings or SQLExpression's, represents the list of columns. If there are q-values SQLExpression's `Select` derives a list of tables for SELECT query.

***where*:** A string or an SQLExpression, represents the `WHERE` clause.

***groupBy*:** A string or an SQLExpression, represents the `GROUP BY` clause.

***having*:** A string or an SQLExpression, represents the `HAVING` part of the `GROUP BY` clause.

***orderBy*:** A string or an SQLExpression, represents the `ORDER BY` clause.

***join*:** A (list of) JOINs (`LEFT JOIN`, etc.)

***distinct*:** A bool flag to turn on `DISTINCT` query.

***start*, *end*:** Integers. The way to calculate `OFFSET` and `LIMIT`.

***limit*:** An integer. *limit*, if passed, overrides *end*.

***reversed*:** A bool flag to do `ORDER BY` in the reverse direction.

***forUpdate*:** A bool flag to turn on `SELECT FOR UPDATE` query.

***staticTables*:** A sequence of strings or SQLExpression's that name tables for `FROM`. This parameter must be used if *items* is a list of strings from which Select cannot derive the list of tables.

Insert

A class to build `INSERT` queries. Accepts a number of parameters. Use `connection.query(query)` to execute the query.

***table*:** A string that names the table to `INSERT` into. Required.

***valueList*:** A list of (key, value) sequences or {key: value} dictionaries; keys are column names. Either *valueList* or *values* must be passed, but not both. Example:

```
>> insert = Insert('person', valueList=[('name', 'Test'), ('age', 42)])
       # or
>> insert = Insert('person', valueList=[{'name': 'Test'}, {'age': 42}])
>> query = connection.sqlrepr(insert)
       # Both generate the same query:
       # INSERT INTO person (name, age) VALUES ('Test', 42)
>> connection.query(query)
```

***values*:** A dictionary {key: value}; keys are column names. Either *valueList* or *values* must be passed, but not both. Example:

```
>> insert = Insert('person', values={'name': 'Test', 'age': 42})
>> query = connection.sqlrepr(insert)
       # The query is the same
       # INSERT INTO person (name, age) VALUES ('Test', 42)
>> connection.query(query)
```

Instances of the class work fast and thus are suitable for mass-insertion. If one needs to populate a database with SQLObject running a lot of `INSERT` queries this class is the way to go.

Update

A class to build `UPDATE` queries. Accepts a number of parameters. Use `connection.query(query)` to execute the query.

***table*:** A string that names the table to `UPDATE`. Required.

***values*:** A dictionary {key: value}; keys are column names. Required.

***where*:** An optional string or SQLExpression, represents the `WHERE` clause.

Example:

```
>> update = Update('person',
>>     values={'name': 'Test', 'age': 42}, where='id=1')
>> query = connection.sqlrepr(update)
       # UPDATE person SET name='Test', age=42 WHERE id=1
>> connection.query(query)
```

Delete

A class to build `DELETE FROM` queries. Accepts a number of parameters. Use `connection.query(query)` to execute the query.

***table*:** A string that names the table to `UPDATE`. Required.

***where*:** An optional string or an SQLExpression, represents the `WHERE` clause. Required. If you need to delete all rows pass `where=None`; this is a safety measure.

Example:

```
>> update = Delete('person', where='id=1')
>> query = connection.sqlrepr(update)
       # DELETE FROM person WHERE id=1
>> connection.query(query)
```

Union

A class to build `UNION` queries. Accepts a number of parameters - `Select` queries. Use `connection.queryAll(query)` to execute the query and get back the results.

Example:

```
>> select1 = Select(['min', func.MIN(const.salary)], staticTables=['employees'])
>> select2 = Select(['max', func.MAX(const.salary)], staticTables=['employees'])
>> union = Union(select1, select2)
>> query = connection.sqlrepr(union)
       # SELECT 'min', MIN(salary) FROM employees
       #    UNION
       # SELECT 'max', MAX(salary) FROM employees
>> rows = connection.queryAll(query)
```

Nested SQL statements (subqueries)

There are a few special operators that receive as parameter SQL statements. These are `IN`, `NOTIN`, `EXISTS`, `NOTEXISTS`, `SOME`, `ANY` and `ALL`. Consider the following example: You are interested in removing records from a table using deleteMany. However, the criterion for doing so depends on another table.

You would expect the following to work:

```
>> PersonWorkplace.deleteMany(where=
   ((PersonWorkplace.q.WorkplaceID==Workplace.q.id) &
   (Workplace.q.id==SOME_ID)))
```

But this doesn't work! However, you can't do a join in a deleteMany call. To work around this issue, use `IN`:

```
>> PersonWorkplace.deleteMany(where=
   IN(PersonWorkplace.q.WorkplaceID,
   Select(Workplace.q.id, Workplace.q.id==SOME_ID)))
```

SelectResults: Using Queries

Contents:

Overview

SelectResults are returned from `.select` and `.selectBy` methods on SQLObject classes, and from `SQLMultipleJoin`, and `SQLRelatedJoin` accessors on SQLObject instances.

SelectResults are generators, which are lazily evaluated. The SQL is only executed when you iterate over the SelectResults, fetching rows one at a time. This way you can iterate over large results without keeping the entire result set in memory. You can also do things like `.reversed()` without fetching and reversing the entire result – instead, SQLObject can change the SQL that is sent so you get equivalent results.

Note: To retrieve the results all at once use the python idiom of calling `list()` on the generator to force execution and convert the results to a stored list.

You can also slice SelectResults. This modifies the SQL query, so `peeps[:10]` will result in `LIMIT 10` being added to the end of the SQL query. If the slice cannot be performed in the SQL (e.g., peeps[:-10]), then the select is executed, and the slice is performed on the list of results. This will generally only happen when you use negative indexes.

In certain cases, you may get a SelectResults instance with an object in it more than once, e.g., in some joins. If you don't want this, you can add the keyword argument `MyClass.select(..., distinct=True)`, which results in a `SELECT DISTINCT` call.

You can get the length of the result without fetching all the results by calling `count` on the result object, like `MyClass.select().count()`. A `COUNT(*)` query is used – the actual objects are not fetched from the database. Together with slicing, this makes batched queries easy to write:

```
start = 20
size = 10
```

```
query = Table.select()
results = query[start:start+size]
total = query.count()
print "Showing page %i of %i" % (start/size + 1, total/size + 1)
```

Note: There are several factors when considering the efficiency of this kind of batching, and it depends very much how the batching is being used. Consider a web application where you are showing an average of 100 results, 10 at a time, and the results are ordered by the date they were added to the database. While slicing will keep the database from returning all the results (and so save some communication time), the database will still have to scan through the entire result set to sort the items (so it knows which the first ten are), and depending on your query may need to scan through the entire table (depending on your use of indexes). Indexes are probably the most important way to improve importance in a case like this, and you may find caching to be more effective than slicing.

In this case, caching would mean retrieving the *complete* results. You can use `list(MyClass.select(...))` to do this. You can save these results for some limited period of time, as the user looks through the results page by page. This means the first page in a search result will be slightly more expensive, but all later pages will be very cheap.

Retrieval Methods

Iteration

As mentioned in the overview, the typical way to access the results is by treating it as a generator and iterating over it (in a loop, by converting to a list, etc).

`getOne(default=optional)`

In cases where your restrictions cause there to always be a single record in the result set, this method will return it or raise an exception: SQLObjectIntegrityError if more than one result is found, or SQLObjectNotFound if there are actually no results, unless you pass in a default like `.getOne(None)`.

Cloning Methods

These methods return a modified copy of the SelectResults instance they are called on, so successive calls can chained, eg `results = MyClass.selectBy(city='Boston').filter(MyClass.q.commute_distance>10).orderBy('vehicle_mileage')` or used independently later on.

`orderBy(column)`

Takes a string column name (optionally prefixed with '-' for DESCending) or a SQLBuilder expression.

`limit(num)`

Only return first num many results. Equivalent to results[:num] slicing.

`lazyColumns(v)`

Only fetch the IDs for the results, the rest of the columns will be retrieved when attributes of the returned instances are accessed.

reversed()

Reverse-order. Alternative to calling orderBy with SQLBuilder.DESC or '-'.

distinct()

In SQL, SELECT DISTINCT, removing duplicate rows.

filter(expression)

Add additional expressions to restrict result set. Takes either a string static SQL expression valid in a WHERE clause, or a SQLBuilder expression. ANDed with any previous expressions.

Aggregate Methods

These return column values (strings, numbers, etc) not new SQLResults instances, by making the appropriate SQL query (the actual result rows are not retrieved). Any that take a column can also take a SQLBuilder column instance, e.g. `MyClass.q.size`.

count()

Returns the length of the result set, by a SQL `SELECT COUNT(...)` query.

sum(column)

The sum of values for `column` in the result set.

min(column)

The minimum value for `column` in the result set.

max(column)

The maximum value for `column` in the result set.

avg(column)

The average value for the `column` in the result set.

Traversal to related SQLObject classes

throughTo.join_name and throughTo.foreign_key_name

This accessor lets you retrieve the objects related to your SelectResults by either a join or foreign key relationship, in the same manner as the cloning methods above. For instance:

```
Schools.select(Schools.q.student_satisfaction>90).throughTo.teachers
```

returns a SelectResults of Teachers of Schools with satisfied students, assuming Schools has a SQLMultipleJoin or SQLRelatedJoin attribute named `teachers`. Similarily, with a self-joining foreign key named `father`:

```
Person.select(Person.q.name=='Steve').throughTo.father.throughTo.father
```

returns a SelectResults of Persons who are the paternal grandfather of someone named `Steve`.

The sqlobject-admin Tool

author Ian Bicking <ianb@colorstudy.com>

revision Rev

date $LastChangedDate$

Contents

Warning: This document isn't entirely accurate; some of what it describes are the intended features of the tool, not the actual features.

Particularly inaccurate is how modules and classes are found.

Introduction

The `sqlobject-admin` tool included with SQLObject allows you to manage your database as defined with SQLObject classes.

Some of the features include creating tables, checking the status of the database, recording a version of a schema, and updating the database to match the version of the schema in your code.

To see a list of commands run `sqlobject-admin help`. Each sub-command has `-h` option which explains the details of that command.

Common Options

Many of the commands share some common options, mostly for finding the database and classes.

`-c CONNECTION` or `--connection=CONNECTION`:

> This takes an argument, the connection string for the database. This overrides any connection the classes have (if they are hardwired to a connection).

`-f FILENAME` or `--config-file=FILENAME`:

> This is a configuration file from which to get the connection. This configuration file should be a Python-syntax file that defines a global variable `database`, which is the connection string for the database.

`-m MODULE` or `--module=MODULE`:

> A module to look in for classes. `MODULE` is something like `myapp.amodule`. Remember to set your `$PYTHONPATH` if the module can't be imported. You can provide this argument multiple times.

`-p PACKAGE` or `--package=PACKAGE`:

> A package to look in. This looks in all the modules in this class and subclasses for SQLObject classes.

`--class=CLASSMATCH`:

> This *restricts* the classes found to the matching classes. You may use wildcards. You can provide multiple `--class` arguments, and if any pattern matches the class will be included.

`--egg=EGG_SPEC`:

> This is an Egg description that should be loaded. So if you give `--egg=ProjectName` it'll load that egg, and look in `ProjectName.egg-info/sqlobject.txt` for some settings (like `db_module` and `history_dir`).

When finding SQLObject classes, we look in the modules for classes that belong to the module – so if you import a class from another module it won't be "matched". You have to indicate its original module.

If classes have to be handled in a specific order, create a `soClasses` global variable that holds a list of the classes. This overrides the module restrictions. This is important in databases with referential integrity, where dependent tables can't be created before the tables they depend on.

Simple Commands

The `create` Command

This finds the tables and creates them. Any tables that exist are simply skipped.

It also collects data from sqlmeta.createSQL (added in svn trunk) and runs the queries after table creation. createSQL can be a string with a single SQL command, a list of SQL commands, or a dictionary with keys that are dbNames and values that are either single SQL command string or a list of SQL commands. An example follows:

```
class MyTable(SQLObject):
    class sqlmeta:
        createSQL = {'postgres': [
            "ALTER TABLE my_table ADD CHECK(my_field != '');",
            ]}
    myField = StringCol()
```

The `sql` Command

This shows the SQL to create all the tables.

The `drop` Command

Drops tables! Missing tables are skipped.

The `execute` Command

This executes an arbitrary SQL expression. This is mostly useful if you want to run a query against a database described by a SQLObject connection string. Use `--stdin` if you want to pipe commands in; otherwise you give the commands as arguments.

The `list` Command

Lists out all the classes found. This can help you figure out what classes you are dealing with, and if there's any missing that you expected.

The `status` Command

This shows if tables are present in the database. If possible (it depends on the database) it will also show if the tables are missing any columns, or have any extra columns, when compared to the table the SQLObject class describes. It doesn't check column types, indexes, or constraints. This feature may be added in the future.

Versioning & Upgrading

There's two commands related to storing the schema and upgrading the database: `record` and `upgrade`.

The idea is that you record each iteration of your schema, and this gets a version number. Something like `2003-05-04a`. If you are using source control you'll check all versions into your repository; you don't overwrite one with the next.

In addition to the on-disk record of the different schemas you have gone through, the database itself contains a record of what version it is at. By having all the versions available at once, we can upgrade from any version. But more on that later

Basic Usage

Here's a quick summary of how you use these commands:

1. In project where you've never used `sqlobject-admin` before, you run `sqlobject-admin record --output-dir=sqlobject-history`. If your active database is up-to-date with the code, then the tool will add a `sqlobject_db_version` table to the database with the current version.
2. Now, make some updates to your code. Don't update the database! (You could, but for now it's more fun if you don't.)
3. Run `sqlobject-admin record --edit`. A new version will be created, and an editor will be opened up.

The `record` Command

Record will take the SQL `CREATE` statements for your tables, and output them in new version. It creates the version by using the ISO-formatted date (YYYY-MM-DD) and a suffix to make it unique. It puts each table in its own file.

This normally doesn't touch the database at all – it only records the schema as defined in your code, regardless of the database. In fact, I recommend calling `record` *before* you update your database.

The `upgrade` Command

Future

- Get `record` to do `svn cp` when creating a new version, then write over those files; this way the version control system will have nice diffs.
- An option to `record` the SQL for multiple database backends at once (now only the active backend is recorded).
- An option to upgrade databases with Python scripts instead of SQL commands. Or a little of both.
- Review all the verbosity, maybe add logging, review simulation.
- Generate simple `ALTER` statements for upgrade scripts, to give people something to work with. Maybe.
- A command to trim versions, by merging upgrade scripts.

Contents

Inheritance

Why

Imagine you have a list of persons, and every person plays a certain role. Some persons are students, some are professors, some are employees. Every role has different attributes. Students are known by their department and year. Professors have a department (some attributes are common for all or some roles), timetable and other attributes.

How does one implement this in SQLObject? Well, the obvious approach is to create a table Person with a column that describes or name the role, and a table for an every role. Then one must write code that interprets and dereferences the role column.

Well, the inheritance machinery described below does exactly this! Only it does it automagically and mostly transparent to the user.

First, you create a table Person. Nothing magical here:

```
class Person(SQLObject):
    name = StringCol()
    age = FloatCol()
```

Now you need a hierarchy of roles:

```
class Role(InheritableSQLObject):
    department = StringCol()
```

The magic starts here! You inherit the class from the special root class `InheritableSQLObject` and provide a set of attributes common for all roles. Other roles must be inherited from Role:

```
class Student(Role):
    year = IntCol()

class Professor(Role):
    timetable = StringCol()
```

Now you want a column in Person that can be interpreted as the role. Easy:

```
class Person(SQLObject):
    name = StringCol()
    age = FloatCol()
    role = ForeignKey("Role")
```

That's all, really! When asked for its role, Person returns the value of its .role attribute dereferenced and interpreted. Instead of returning an instance of class Role it returns an instance of the corresponding subclass - a Student or a Professor.

This is a brief explanation based on a task people meet most often, but of course it can be used far beyond the person/role task. I also omitted all details in the explanation. Now look at the real working program:

```
from sqlobject import *
from sqlobject.inheritance import InheritableSQLObject

__connection__ = "sqlite:/:memory:"

class Role(InheritableSQLObject):
   department = StringCol()

class Student(Role):
   year = IntCol()

class Professor(Role):
   timetable = StringCol(default=None)

class Person(SQLObject):
   name = StringCol()
   age = FloatCol()
   role = ForeignKey("Role", default=None)
```

```
Role.createTable()
Student.createTable()
Professor.createTable()
Person.createTable()

first_year = Student(department="CS", year=1)
lecturer = Professor(department="Mathematics")

student = Person(name="A student", age=21, role=first_year)
professor = Person(name="A professor", age=42, role=lecturer)

print student.role
print professor.role
```

It prints:

```
<Student 1 year=1 department='CS'>
<Professor 2 timetable=None department='Mathematics'>
```

You can get the list of all available roles:

```
print list(Role.select())
```

It prints:

```
[<Student 1 year=1 department='CS'>, <Professor 2 timetable=None department=
↪'Mathematics'>]
```

Look - you have gotten a list of Role's subclasses!

If you add a MultipleJoin column to Role, you can list all persons for a given role:

```
class Role(InheritableSQLObject):
    department = StringCol()
    persons = MultipleJoin("Person")

for role in Role.select():
    print role.persons
```

It prints:

```
[<Person 1 name='A student' age=21.0 roleID=1>]
[<Person 2 name='A professor' age=42.0 roleID=2>]
```

If you you want your persons to have many roles, use RelatedJoin:

```
class Role(InheritableSQLObject):
    department = StringCol()
    persons = RelatedJoin("Person")

class Student(Role):
    year = IntCol()

class Professor(Role):
    timetable = StringCol(default=None)

class Person(SQLObject):
```

```
    name = StringCol()
    age = FloatCol()
    roles = RelatedJoin("Role")

Role.createTable()
Student.createTable()
Professor.createTable()
Person.createTable()

first_year = Student(department="CS", year=1)
lecturer = Professor(department="Mathematics")

student = Person(name="A student", age=21)
student.addRole(first_year)
professor = Person(name="A professor", age=42)
professor.addRole(lecturer)

print student.roles
print professor.roles

for role in Role.select():
    print role.persons
```

It prints:

```
[<Student 1 year=1 department='CS'>]
[<Professor 2 timetable=None department='Mathematics'>]
[<Person 1 name='A student' age=21.0>]
[<Person 2 name='A professor' age=42.0>]
```

Who, What and How

Daniel Savard has implemented inheritance for SQLObject. In terms of ORM this is a kind of vertical inheritance. The only difference is that objects reference their leaves, not parents. Links to parents are reconstructed at run-time using the hierarchy of Python classes.

- As suggested by Ian Bicking, each child class now has the same ID as the parent class. No more need for childID column and parent foreignKey (and a small speed boost).
- No more need to call getSubClass, as the 'latest' child will always be returned when an instance of a class is created.
- This version now seems to work correctly with addColumn, delColumn, addJoin and delJoin.

The following code:

```
from sqlobject.inheritance import InheritableSQLObject
class Person(InheritableSQLObject):
    firstName = StringCol()
    lastName = StringCol()

class Employee(Person):
    _inheritable = False
    position = StringCol()
```

will generate the following tables:

```
CREATE TABLE person (
    id INT PRIMARY KEY,
    child_name TEXT,
    first_name TEXT,
    last_name TEXT
);

CREATE TABLE employee (
    id INT PRIMARY KEY,
    position TEXT
)
```

A new class attribute `_inheritable` is added. When this new attribute is set to 1, the class is marked 'inheritable' and a new columns will automatically be added: childName (TEXT).

Each class that inherits from a parent class will get the same ID as the parent class. So, there is no need to keep track of parent ID and child ID, as they are the same.

The column childName will contain the name of the child class (for example 'Employee'). This will permit a class to always return its child class if available (a person that is also an employee will always return an instance of the employee class).

For example, the following code:

```
p = Person(firstName='John', lastName='Doe')
e = Employee(firstName='Jane', lastName='Doe', position='Chief')
p2 = Person.get(1)
```

Will create the following data in the database:

```
*Person*
id
    child_name
    first_name
    last_name
0
    Null
    John
    Doe
1
    Employee
    Jane
    Doe


*Employee*
id
    position
1
    Chief
```

You will still be able to ask for the attribute normally: e.firstName will return Jane and setting it will write the new value in the person table.

If you use p2, as p2 is a person object, you will get an employee object. person(0) will return a Person instance and will have the following attributes: firstName and lastName. person(1) or employee(1) will both return the same Employee instance and will have the following attributes: firstName, lastName and position.

Also, deleting a person or an employee that is linked will destroy both entries as one would expect.

The SQLObject q magic also works. Using these selects is valid:

```
Employee.select(AND(Employee.q.firstName == 'Jane', Employee.q.position == 'Chief'))␣
↪will return Jane Doe
Employee.select(AND(Person.q.firstName == 'Jane', Employee.q.position == 'Chief'))␣
↪will return Jane Doe
Employee.select(Employee.q.lastName == 'Doe') will only return Jane Doe (as Joe isn't␣
↪an employee)
Person.select(Person.q.lastName == 'Doe') will return both entries.
```

The SQL 'where' clause will contain additional clauses when used with 'inherited' classes. These clauses are the link between the id and the parent id. This will look like the following request:

```
SELECT employee.id, person.first_name, person.last_name
FROM person, employee WHERE person.first_name = 'Jane'
AND employee.position = 'Chief' AND person.id = employee.id
```

Limitations and notes

- Only single inheritance will work. It is not possible to inherit from multiple SQLObject classes.
- It is possible to inherit from an inherited class and this will work well. In the above example, you can have a Chief class that inherits from Employee and all parents attributes will be available through the Chief class.
- You may not redefine columns in an inherited class (this will raise an exception).
- If you don't want 'childName' columns in your last class (one that will never be inherited), you must set '_inheritable' to False in this class.
- The inheritance implementation is incompatible with lazy updates. Do not set lazyUpdate to True. If you need this, you have to patch SQLObject and override many methods - _SO_setValue(), sync(), syncUpdate() at least. Patches will be gladly accepted.
- You'd better restrain yourself to simple use cases. The inheritance implementation is easily choked on more complex cases.
- A join between tables inherited from the same parent produces incorrect result due to joins to the same parent table (they must use different aliases).
- Inheritance works in two stages - first it draws the IDs from the parent table and then it draws the rows from the children tables. The first stage could fail if you try to do complex things. For example, Children.select(orderBy=Children.q.column, distinct=True) could fail because at the first stage inheritance generates a SELECT query for the parent table with ORDER BY the column from the children table.
- I made it because I needed to be able to have automatic inheritance with linked tables.
- This version works for me; it may not work for you. I tried to do my best but it is possible that I broke some things... So, there is no warranty that this version will work.
- Thanks to Ian Bicking for SQLObject; this is a wonderful python module.
- Although all the attributes are inherited, the same does not apply to sqlmeta data. Don't try to get a parent column via the sqlmeta.columns dictionary of an inherited InheritableSQLObject: it will raise a KeyError. The same applies to joins: the sqlmeta.joins list will be empty in an inherited InheritableSQLObject if a join has been defined in the parent class, even though the join method will work correctly.
- If you have suggestion, bugs, or patch to this patch, you can contact the SQLObject team: <sqlobject-discuss at lists.sourceforge.net>

Contents

Versioning

Why

You have a table where rows can be altered, such as a table of wiki pages. You want to retain a history of changes for auditing, backup, or tracking purposes.

You could write a decorator that stores old versions and manage all access to this object through it. Or you could take advantage of the event system in SQLObject 0.8+ and just catch row accesses to the object. SQLObject's versioning module does this for you. And it even works with inheritance!

How

Here's how to set it up:

```
class MyClass(SQLObject):
    name = StringCol()
    versions = Versioning()
```

To use it, just create an instance as usual:

```
mc = MyClass(name='fleem')
```

Then make some changes and check out the results:

```
mc.set(name='morx')
assert mc.versions[0].name == 'fleem'
```

You can also restore to a previous version:

```
mc.versions[0].restore()
assert mc.name == "fleem"
```

Inheritance

There are three ways versioning can be used with inheritance:

1. Parent versioned, children unversioned:

```
class Base(InheritableSQLObject):
    name = StringCol()
    versions = Versioning()

class Child(Base):
    toy = StringCol()
```

In this case, when changes are made to an instance of Base, new versions are created. But when changes are made to an instance of Child, no new versions are created.

2. Children versioned, parents unversioned.

In this case, when changes are made to an instance of Child, new versions are created. But when changes are made to an instance of Base, no new versions are created. The version data for Child contains all of the columns from child and from base, so that a full restore is possible.

3. Both children and parents versioned.

In this case, changes to either Child or Base instances create new versions, but in different tables. Child versions still contain all Base data, and a change to a Child only creates a new Child version, not a new Base version.

Version Tables

Versions are stored in a special table which is created when the table for a versioned class is created. Version tables are not altered when the main table is altered, so if you add a column to your main class, you will need to manually add the column to your version table.

Views and SQLObjects

In general, if your database backend supports defining views you may define them outside of SQLObject and treat them as a regular table when defining your SQLObject class.

ViewSQLObject

The rest of this document is experimental.

`from sqlobject.views import *`

`ViewSQLObject` is an attempt to allow defining views that allow you to define a SQL query that acts like a SQLObject class. You define columns based on other SQLObject classes .q SQLBuilder columns, have columns that are aggregates of other columns, and join multiple SQLObject classes into one and add restrictions using SQLBuilder expressions.

The resulting classes are currently read only, if you find use for this idea please bring discussion to the mailing list.

A short example from the tests will suffice for now.

Base classes:

```
class PhoneNumber(SQLObject):
  number = StringCol()
  calls = SQLMultipleJoin('PhoneCall')
  incoming = SQLMultipleJoin('PhoneCall', joinColumn='toID')

class PhoneCall(SQLObject):
```

```
  phoneNumber = ForeignKey('PhoneNumber')
  to = ForeignKey('PhoneNumber')
  minutes = IntCol()
```

View classes:

```
class ViewPhoneCall(ViewSQLObject):
  class sqlmeta:
      idName = PhoneCall.q.id
      clause = PhoneCall.q.phoneNumberID==PhoneNumber.q.id

  minutes = IntCol(dbName=PhoneCall.q.minutes)
  number = StringCol(dbName=PhoneNumber.q.number)
  phoneNumber = ForeignKey('PhoneNumber', dbName=PhoneNumber.q.id)
  call = ForeignKey('PhoneCall', dbName=PhoneCall.q.id)

class ViewPhone(ViewSQLObject):
  class sqlmeta:
      idName = PhoneNumber.q.id
      clause = PhoneCall.q.phoneNumberID==PhoneNumber.q.id

  minutes = IntCol(dbName=func.SUM(PhoneCall.q.minutes))
  numberOfCalls = IntCol(dbName=func.COUNT(PhoneCall.q.phoneNumberID))
  number = StringCol(dbName=PhoneNumber.q.number)
  phoneNumber = ForeignKey('PhoneNumber', dbName=PhoneNumber.q.id)
  calls = SQLMultipleJoin('PhoneCall', joinColumn='phoneNumberID')
  vCalls = SQLMultipleJoin('ViewPhoneCall', joinColumn='phoneNumberID')
```

SQLObject Developer Guide

Contents

These are some notes on developing SQLObject.

Development Installation

First install FormEncode:

```
$ git clone git://github.com/formencode/formencode.git
$ cd formencode
$ sudo python setup.py develop
```

Then do the same for SQLObject:

```
$ git clone git clone git://github.com/sqlobject/sqlobject.git
$ cd sqlobject
$ sudo python setup.py develop
```

Or rather fork it and clone your fork. To develop a feature or a bugfix create a separate branch, push it to your fork and create a pull request to the original repo. That way CI will be triggered to test your code.

Voila! The packages are globally installed, but the files from the checkout were not copied into `site-packages`. See setuptools for more.

Architecture

There are three main kinds of objects in SQLObject: tables, columns and connections.

Tables-related objects are in sqlobject/main.py module. There are two main classes: `SQLObject` and `sqlmeta`; the latter is not a metaclass but a parent class for `sqlmeta` attribute in every class - the authors tried to move there all attributes and methods not directly related to columns to avoid cluttering table namespace.

Connections are instances of `DBConnection` class (from sqlobject/dbconnection.py) and its concrete descendants. `DBConnection` contains generic code for generating SQL, working with transactions and so on. Concrete connection classes (like `PostgresConnection` and `SQLiteConnection`) provide backend-specific functionality.

Columns, validators and converters

Columns are instances of classes from sqlobject/col.py. There are two classes for every column: one is for user to include into an instance of SQLObject, an instance of the other is automatically created by SQLObject's metaclass. The two classes are usually named `Col` and `SOCol`; for example, `BoolCol` and `SOBoolCol`. User-visible classes, descendants of `Col`, seldom contain any code; the main code for a column is in `SOCol` descendants and in validators.

Every column has a list of validators. Validators validate input data and convert input data to python data and back. Every validator must have methods `from_python` and `to_python`. The former converts data from python to internal representation that will be converted by converters to SQL strings. The latter converts data from SQL data to python. Also please bear in mind that validators can receive `None` (for SQL `NULL`) and `SQLExpression` (an object that represents SQLObject expressions); both objects must be passed unchanged by validators.

Converters from sqlobject/converters.py aren't visible to users. They are used behind the scene to convert objects returned by validators to backend-specific SQL strings. The most elaborated converter is `StringLikeConverter`. Yes, it converts strings to strings. It converts python strings to SQL strings using backend-specific quoting rules.

Let look into `BoolCol` as an example. The very `BoolCol` doesn't have any code. `SOBoolCol` has a method to create `BoolValidator` and methods to create backend-specific column type. `BoolValidator` has identical methods `from_python` and `to_python`; the method passes `None`, `SQLExpression` and bool values unchanged; int and objects that have method `__nonzero__` (`__bool__` in Python 3) are converted to bool; other objects trigger validation error. Bool values that are returned by call to `from_python` will be converted to SQL strings by `BoolConverter`; bool values from `to_python` (is is supposed they are originated from the backend via DB API driver) are passed to the application.

Objects that are returned from `from_python` must be registered with converters. Another approach for `from_python` is to return an object that has `__sqlrepr__` method. Such objects convert to SQL strings themselves, converters are not used.

Branch workflow

Initially `SQLObject` was being developed using `Subversion`. Even after switching to git development process somewhat preserves the old workflow.

The `trunk`, called `master` in git, is the most advanced and the most unstable branch. It is where new features are applied. Bug fixes are applied to `oldstable` and `stable` branches and are merged upward – from `oldstable` to `stable` and from `stable` to `master`.

Style Guide

Generally you should follow the recommendations in PEP 8, the Python Style Guide. Some things to take particular note of:

- With a few exceptions sources must be flake8-clean (and hence pep8-clean). Please consider using pre-commit hook installed by running `flake8 --install-hook`.
- **No tabs**. Not anywhere. Always indent with 4 spaces.
- We don't stress too much on line length. But try to break lines up by grouping with parenthesis instead of with backslashes (if you can). Do asserts like:

```
assert some_condition(a, b), (
    "Some condition failed, %r isn't right!" % a)
```

- But if you are having problems with line length, maybe you should just break the expression up into multiple statements.
- Blank lines between methods, unless they are very small and closely bound to each other.
- *Never* use the form `condition and trueValue or falseValue`. Break it out and use a variable.
- Careful of namespace pollution. SQLObject does allow for `from sqlobject import *` so names should be fairly distinct, or they shouldn't be exported in `sqlobject.__init__`.
- We're very picky about whitespace. There's one and only one right way to do it. Good examples:

```
short = 3
longerVar = 4

if x == 4:
    do stuff

func(arg1='a', arg2='b')
func((a + b)*10)
```

Bad examples:

```
short    =3
longerVar=4

if x==4: do stuff

func(arg1 = 'a', arg2 = 'b')
func(a,b)
func( a, b )
[ 1, 2, 3 ]
```

To us, the poor use of whitespace seems lazy. We'll think less of your code (justified or not) for this very trivial reason. We will fix all your code for you if you don't do it yourself, because we can't bear to look at sloppy whitespace.

- Use `@@` to mark something that is suboptimal, or where you have a concern that it's not right. Try to also date it and put your username there.
- Docstrings are good. They should look like:

```
class AClass(object):
    """
    doc string...
    """
```

  Don't use single quotes (''`). Don't bother trying make the string less vertically compact.

- Comments go right before the thing they are commenting on.
- Methods never, ever, ever start with capital letters. Generally only classes are capitalized. But definitely never methods.
- mixedCase is preferred.
- Use `cls` to refer to a class. Use `meta` to refer to a metaclass (which also happens to be a class, but calling a metaclass `cls` will be confusing).
- Use `isinstance` instead of comparing types. E.g.:

```
if isinstance(var, str): ...
# Bad:
if type(var) is StringType: ...
```

- Never, ever use two leading underscores. This is annoyingly private. If name clashes are a concern, use name mangling instead (e.g., `_SO_blahblah`). This is essentially the same thing as double-underscore, only it's transparent where double underscore obscures.
- Module names should be unique in the package. Subpackages shouldn't share module names with sibling or parent packages. Sadly this isn't possible for `__init__`, but it's otherwise easy enough.
- Module names should be all lower case, and probably have no underscores (smushedwords).

Testing

Tests are important. Tests keep everything from falling apart. All new additions should have tests.

Testing uses pytest, an alternative to `unittest`. It is available at http://pytest.org/ and https://pypi.python.org/pypi/pytest. Read its getting started document for more.

To actually run the test, you have to give it a database to connect to. You do so with the option `-D`. You can either give a complete URI or one of several shortcuts like `mysql` (these shortcuts are defined in the top of `tests/dbtest.py`).

All the tests are modules in `sqlobject/tests`. Each module tests one kind of feature, more or less. If you are testing a module, call the test module `tests/test_modulename.py` – only modules that start with `test_` will be picked up by pytest.

The "framework" for testing is in `tests/dbtest`. There's a couple of important functions:

`setupClass(soClass)` creates the tables for the class. It tries to avoid recreating tables if not necessary.

`supports(featureName)` checks if the database backend supports the named feature. What backends support what is defined at the top of `dbtest`.

If you `import *` you'll also get pytest's version of raises, an `inserts` function that can create instances for you, and a couple miscellaneous functions.

If you submit a patch or implement a feature without a test, we'll be forced to write the test. That's no fun for us, to just be writing tests. So please, write tests; everything at least needs to be exercised, even if the tests are absolutely complete.

We now use Travis CI and AppVeyor to run tests. See the statuses: To avoid triggering unnecessary test run at CI services add text [skip ci] or `[ci skip]` anywhere in your commit messages for commits that don't change code (documentation updates and such).

We use coverage.py to measures code coverage by tests and upload the result for analyzis to Coveralls and Codecov:

Documentation

Please write documentation. Documentation should live in the docs/ directory in reStructuredText format. We use Sphinx to convert docs to HTML.

Authors

SQLObject was originally written by Ian Bicking <ianb@colorstudy.com>.

Contributions have been made by:

- Frank Barknecht <fbar at footils.org>
- Bud P. Bruegger <bug at sistema.it>
- David M. Cook <dave at davidcook.org>
- Luke Opperman <luke at metathusalan.com>
- James Ralston <jralston at hotmail.com>
- Sidnei da Silva <sidnei at awkly.org>
- Brad Bollenbach <brad at bbnet.ca>
- Daniel Savard, Xsoli Inc <sqlobject at xsoli.com>
- alexander smishlajev <alex at ank-sia.com>
- Yaroslav Samchuk <yarcat at ank-sia.com>
- Runar Petursson <grunar at gmail.com>
- J Paulo Fernandes Farias <jpaulofarias at gmail.com>
- Shahms King <shahms at shahms.com>
- David Turner, The Open Planning Project
- Dan Pascu <dan at ag-projects.com>
- Diez B. Roggisch <deets at web.de>
- Christopher Singley <csingley at gmail.com>
- David Keeney <dkeeney at rdbhost.com>
- Daniel Fetchinson <fetchinson at googlemail.com>
- Neil Muller <drnlmuller+sqlobject at gmail.com>

- Petr Jakes <petr.jakes at tpc.cz>
- Ken Lalonde
- Andrew Ziem <ahz001 at gmail.com>
- Andrew Trusty <atrusty at gatech.edu>
- Ian Cordasco <graffatcolmingov at gmail.com>
- Lukasz Dobrzanski <lukasz.m.dobrzanski at gmail.com>
- Gregor Horvath <gh at gregor-horvath.com>
- Nathan Edwards <nje5 at georgetown.edu>
- Lutz Steinborn <l.steinborn at 4c-gmbh.de>
- Oleg Broytman <phd@phdru.name>

TODO

- Stop supporting Python 2.6. Fix dbconnection.py and joins.py.
- PyPy.
- Quote table/column names that are reserved keywords (order => "order", values => *values* for MySQL).
- RelatedJoin.hasOther(otherObject[.id])
- createParamsPre/Post:

```
class MyTable(SQLObject):
    class sqlmeta:
        createParamsPre = 'TEMPORARY IF NOT EXISTS'
        createParamsPre = {temporary: True, ifNotExists: True,
                           'postgres': 'LOCAL'}
        createParamsPost = 'ENGINE InnoDB'
        createParamsPost = {'mysql': 'ENGINE InnoDB',
                            'postgres': 'WITH OIDS'}
```

- SQLObject.fastInsert().
- IntervalCol
- TimedeltaCol
- Cached join results.
- Invert tests isinstance(obj, (tuple, list)) to not isinstance(obj, basestr) to allow any iterable.
- Always use .lazyIter().
- Optimize Iteration.next() - use cursor.fetchmany().
- Generators instead of loops (fetchall => fetchone).
- Cache columns in sqlmeta.getColumns(); reset the cache on add/del Column/Join.
- Make ConnectionHub a context manager instead of .doInTransaction().
- Make version_info a namedtuple.
- Expression columns - in SELECT but not in INSERT/UPDATE. Something like this:

```
class MyClass(SQLObject):
    function1 = ExpressionCol(func.my_function(MyClass.q.col1))
    function2 = ExpressionCol('sum(col2)')
```

- A hierarchy of exceptions. SQLObject should translate exceptions from low-level drivers to a consistent set of high-level exceptions.
- Memcache.
- Refactor `DBConnection` to use parameterized queries instead of generating query strings.
- PREPARE/EXECUTE.
- Protect all .encode(), catch UnicodeEncode exceptions and reraise Invalid.
- More kinds of joins, and more powerful join results (closer to how *select* works).
- Better joins - automatic joins in .select() based on ForeignKey/MultipleJoin/RelatedJoin.
- Deprecate, then remove connectionForOldURI.
- Switch from setuptools to distribute.
- Support PyODBC driver for all backends.
- dbms is a DB API wrapper for DB API drivers for IBM DB2, Firebird, MSSQL Server, MySQL, Oracle, PostgreSQL, SQLite and ODBC.
- dict API: use getitem interface for column access instead of getattr; reserve getattr for internal attributes only; this helps to avoid collisions with internal attributes.
- Or move column values access to a separate namespace, e.g. .c: row.c.column.
- More documentation.
- RSS 2.0 and Atom news feeds.
- Use DBUtils, especially SolidConnection.
- `_fromDatabase` currently doesn't support IDs that don't fit into the normal naming scheme. It should do so. You can still use `_idName` with `_fromDatabase`.
- More databases supported. There has been interest and some work in the progress for Oracle. IWBN to have Informix and DB2 drivers.
- Better transaction support – right now you can use transactions for the database, but objects aren't transaction-aware, so non-database persistence won't be able to be rolled back.
- Optimistic locking and other techniques to handle concurrency.
- Profile of SQLObject performance to identify bottlenecks.
- Increase hooks with FormEncode validation and form generation package, so SQLObject classes (read: schemas) can be published for editing more directly and easily. (First step: get Schema-generating method into sqlmeta class)
- Merge SQLObject.create*, .create*SQL methods with DBPI.create* methods.
- Made SQLObject unicode-based instead of just unicode-aware. All internal processing should be done with unicode strings, conversion to/from ascii strings should happen for non-unicode DB API drivers.
- Allow to override ConsoleWriter/LogWriter classes and makeDebugWriter function.
- Type annotations and mypy tests.

CHAPTER 2

Example

This is just a snippet that creates a simple class that wraps a table:

```
>>> from sqlobject import *
>>>
>>> sqlhub.processConnection = connectionForURI('sqlite:/:memory:')
>>>
>>> class Person(SQLObject):
...     fname = StringCol()
...     mi = StringCol(length=1, default=None)
...     lname = StringCol()
...
>>> Person.createTable()
```

SQLObject supports most database schemas that you already have, and can also issue the `CREATE` statement for you (seen in `Person.createTable()`).

Here's how you'd use the object:

```
>>> p = Person(fname="John", lname="Doe")
>>> p
<Person 1 fname='John' mi=None lname='Doe'>
>>> p.fname
'John'
>>> p.mi = 'Q'
>>> p2 = Person.get(1)
>>> p2
<Person 1 fname='John' mi='Q' lname='Doe'>
>>> p is p2
True
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search