PyORQ - Python Object Relational binding with Queries

Release 0.1

Roeland Rengelink

June 15, 2004

rengelink@sourceforge.net

Abstract

This is the reference documentation for PyORQ (Python Object Relational binding with Queries). PyORQ implements persistence for Python objects using a relational database (RDBMS) for storage.

PyORQ uses Python expressions to denote queries which can be automatically translated into SQL and then be executed by the backend. This leverages the full search capabilities of RDBMSs in an object-oriented programming environment. Contrary to other object-relational Python-SQL mappings, the user needs no knowledge of SQL to search the database.

Contents

1 Introduction **2**

2 Installation **3**
2.1 Running the tests . . . 3

3 Using PyORQ **3**
3.1 How to write persistent classes . . . 4
3.2 How to write queries . . . 6

4 `pyorq` — The base PyORQ package **7**
4.1 `pyorq.ptype` — Persistent objects . . . 7
4.2 `ptype.pprop` — Persistent properties . . . 8
4.3 `ptype.prel` — Relations . . . 9

5 pyorq.interface — The database interfaces **11**
5.1 pyorq.interface.db_base — The interface definition 11
5.2 pyorq.interface.postgresql_db — for PostgreSQL 11
5.3 pyorq.interface.mysql_db — for MySQL 11
5.4 pyorq.interface.sqlite_db — for SQLite 12

1 Introduction

PyORQ (Python Object Relational binding with Queries) implements persistence for Python objects using a relational database (RDBMS, e.g. PostgreSQL MySQL) for storage.

Object-relational mappings have been done before. They are relatively straightforward: classes map to tables, attributes map to columns and instances map to rows. However, fundamental to the object paradigm is that identity maps to state, and not the other way around. Hence, to search (i.e. map state to identity) one has to loop over the collection of all objects and examine their state. If the objects are in a persistent store, the objects need to be instantiated first which may be prohibitively expensive.

Traditionally there have been two solutions to this problem:

- Use persistent containers that use knowledge of the object's state to allow efficient searches (e.g. B-Trees). However, this essentially generalizes the notion of identity, and does not allow for arbitrary queries without instantiation.
- Use knowledge of the object-relational mapping to write SQL queries that return object identities, which can then be used to instantiate the results of the query. However, this means that the mechanism of the object-relational mapping becomes part of the interface, and requires the user to use SQL within his application.

The innovative aspect of PyORQ is the use of Python expressions to denote queries which can be automatically translated into SQL and then be executed by the backend. This leverages the full search capabilities of RDBMSs in an object-oriented programming environment. Contrary to other object-relational Python-SQL mappings, the user needs no knowledge of SQL to search the database.

v. 0.1 of PyORQ has the following features:

- A notation for describing persistent objects based on Python properties
- Automatic creation of tables based on the persistent object definition.
- A native Python notation to describe queries.
- Full support for object-oriented programming (encapsulation, inheritance)
  - Persistent objects may refer to other persistent objects and queries understand this.
  - References to objects of a particular type may also refer to subclasses of that type.
  - Queries on a type, may return subclasses of that type.
- Interfaces to several SQL backends, including:
  - PostgreSQL, using pyPgSQL by Billy G. Allie.
  - MySQL, using MySQL-Python by Andy Dustman.
  - SQLite, using PySQLite by Michael Owens and Gerhard Haring.

Some desirable features are still missing.

- PyORQ does not check if the definition of a previously created persistent object still matches the table definition.
  (Before modifying a persistent object, use `db.drop_table()` to remove the table).

- No support for multiple inheritance (Don't do it).
- Potential name-mangling problems are not checked.
  (Assume that persistent attributes are case-insensitive, and you should be OK).

See Also:

pyPgSQL
(http://pypgsql.sourceforge.net/)
: For the Python interface to PostgreSQL by Billy G. Allie.

MySQL-Python
(http://sourceforge.net/projects/mysql-python)
: For the Python interface to MySQL by Andy Dustman.

PySQLite
(http://sourceforge.net/projects/pysqlite)
: For the Python interface to SQLite by Michael Owens and Gerhard Haring.

2 Installation

To install PyORQ, unpack the tar file you downloaded, and run `python setup.py install`. For example, on Linux:

```
tar -xvzf PyORQ-0.1.tar.gz
cd PyORQ-0.1
python setup.py install
```

This assumes that you have write privileges in the site-packages directory of your Python distribution. If you don't have the necessary permissions, or if you just want to test PyORQ, you can also just add the directory that was created when unpacking PyORQ to your `PYTHONPATH` environment variable.

PyORQ assumes that you have access to one of the supported databases (PostgreSQL, MySQL or SQLite), and that you have installed the corresponding Python interface (pyPgSQL, MySQL-Python, or PySQLite).

In the case of MySQL and PostgreSQL you will also need to know your username, password and the name of the database, and, if the server is not located on your machine, the name of the host.

2.1 Running the tests

If you wish, you can run the unit tests supplied with the source distribution.

Make sure that PyORQ is installed, or that the installation directory is added to your `PYTHONPATH` environment variable. If you plan to use either the MySQL interface and/or the PostgreSQL interface, make sure that they contain a database 'testdb', and that your login-name is also your username for these databases.

Change directory to 'test' and run

```
python query_test
```

This tests, for each available interface, whether the low-level library is available and whether you can connect to the database. If successful, it will populate the database with some test data, and execute a number of queries.

3 Using PyORQ

PyORQ contains two packages, `pyorq` and `pyorq.interface`. The `pyorq` package contains the following modules:

- `pyorq.ptype` defines `pobject`, the base class for all persistent objects, and `ptype` the metaclass of `pobject`.
- `pyorq.pprop` defines persistent properties.
- `pyorq.prel` implements the relational algebra.

You can safely import the objects that constitute the public interface of PyORQ (`pobject` and the persistent property objects), using:

```
from pyorq import *
```

The `pyorq.interface` package contains the interfaces to the database backends. It contains a number of modules:

- `pyorq.interface.db_base` contains the base class which defines the public interface of the database wrappers.
- `pyorq.interface.nodb` contains a database wrapper without persistent store (basically only a cache, used for testing).
- `pyorq.interface.sql_db` contains the base class for SQL databases.
- `pyorq.interface.postgresql_db` contains the PostgreSQL interface.
- `pyorq.interface.mysql_db` contains the MySQL interface.
- `pyorq.interface.sqlite_db` contains the SQLite interface.

3.1 How to write persistent classes

A persistent class must:

1. derive from `pobject`, or another persistent class
2. have an attribute `database`, referring to an instance of one of the database interfaces
3. define one or more persistent attributes.

PyORQ supports the following persistent properties for built in python types:

Property	Python type	default
`pint()`	`int`	`0`
`pfloat()`	`float`	`0.0`
`pstr()`	`str`	`''`
`pdate()`	`datetime.date`	`datetime.date(1,1,1)`
`ptime()`	`datetime.time`	`datetime.time()`
`pdatetime()`	`datetime.datetime`	`datetime.datetime(1,1,1)`

You can provide alternative defaults to the constructor of the property objects. The main purpose of defaults is to avoid `NULL`s in the database.

Note that the datetime properties currently don't support time zones.

This is a valid persistent class:

```
from pyorq import *
from pyorq.interface import mysql_db

db = mysql_db.mysql_db(database='testdb')

class myclass(pobject)
    database = db
    a = pint()      # default = 0
    b = pfloat(7.3) # default = 7.3
```

You use a persistent class just like any other. You can instantiate an object and assign values to the persistent properties. For example:

```
m = myclass()
m.a = 5
m.b = 22.4
```

You can store the instance to the database using its `commit` method. When the instance is committed, it obtains an *object identifier* (`oid`).

```
print m.oid     # prints None
m.commit()
print m.oid    # print the object identifier
```

Object identifiers are used internally to relate Python instances with table rows. You can use the identifier to retrieve an object from the database.

```
m_oid = m.oid
del m
new_m = myclass(oid=m_oid)
print new_m.a, new_m.b     # prints '5 22.4'
```

The database ensures that at any time there is only one object that corresponds to a given identifier. This means that:

```
a = myclass(oid=m_oid)
b = myclass(oid=m_oid)
print a is b              # prints True
```

Of course, the normal way to retrieve instances from the database is through queries (see below).

Persistent classes may contain attributes that refer to other persistent classes, using the property `pref(cls)`. The argument of `pref` defines which class the persistent attribute refers to. Objects that are assigned to this attribute must be instances of that class, or one of its subclasses.

This is a persistent class with a reference to the previously defined `myclass`.

```
class newclass(pobject):
    database = db
    r = pref(myclass)
```

The default value for a persistent reference is None. You can also provide an instance of the referred class as a default, or, provide a tuple `(cls, oid)`, that will create an instance `cls(oid=oid)`, when an attribute is first

read.

For example:

```
class newclass(pobject):
    database = db
    r = pref(myclass, (myclass, None))

n = newclass()
print m.r
```

will print the representation of a new instance of `myclass`, created using `myclass(oid=None)`.

Persistent classes are subject to the following rules:

- All persistent classes in a schema must refer to the same database instance
- The names of persistent classes and persistent attributes are case-insensitive (The databases use lowercase versions of the names for the table and column names).
- Derived classes cannot redefine persistent properties from the base class.
- PyORQ does not support multiple inheritance yet, although you can use mixins
- You can define constructors (`__init__` method), but constructors will not be called when objects are retrieved from the database.
- You can define `__new__`, but the method will not be called, when retrieving objects from the database.

The requirement that all persistent classes must refer to the same database instance is best satisfied by using some sort of singleton pattern. I prefer using modules for that. For example:

```
# module mydb
from pyorq.interface.postgresql_db import postgresql_db
db = postgresql_db(database='mydb')
```

which can then be used in the different modules that define my database schema as:

```
from pyorq import *
from mydb import db

class Thing(pobject):
    database = db
    ...
```

3.2 How to write queries

To retrieve objects from the database you write queries. The key idea behind our notation for queries is that **a class represents the set of all its instances**, and that **a class attribute that refers to a persistent property says something about all the instances in this set**. Hence, queries are expressions with persistent properties as arguments that contain comparisons.

This is a valid query:

```
myclass.a == 5
```

This query returns an iterator. For each instance `i`, yielded by the iterator, `isinstance(i, myclass) and i.a==5` is `True`. If myclass has subclasses, then this query may also return instances of those subclasses. The constraint `isinstance(i, myclass)` implies that `i` may be an instance of a subclass of `myclass`.

Queries understand references.

This query

```
newclass.r.a == 5
```

yields instances `i` for which `isinstance(i, newclass) and i.r.a==5` is `True`. Implicitly, this means that `isinstance(i.r, myclass)` is also `True`.

The query is equivalent to the generator:

```
def f()
    for m in myclass.a == 5:
        for n in newclass.r == m:
            yield n
```

Note that comparison of persistent instances (as in `newclass.r == m`) implies comparison by identity (i.e.: `is`).

Also note that if a class refers to a class with many subclasses, then the query will have to consider many possible relations between referring object and referred objects.

Queries use the bitwise operators `~`, `|`, and `&` to represent the logical operators `not`, `or`, and `and` respectively. Note that bitwise operators have a higher precedence than comparison operators (contrary to logical operators). Make sure that you properly parenthesize the terms in your queries!

The most important rule in queries is that they should have only one 'free variable'. This is an illegal query:

```
(A.a == 1) & (B.b == 2)
```

because it is not clear if this query should return instances of type `A` or instances of type `B`.

Queries also understand the arithmetic operators `+`, `-`, `/`, and `*`. This is a valid query:

```
(A.x + A.b.y + 17) <= (A.y * A.b.x * 2)
```

4 `pyorq` — The base PyORQ package

4.1 `pyorq.ptype` — Persistent objects

class **`ptype`**(*name, bases, attributes*)

This is the metaclass for persistent classes.

The purpose of `ptype` is to register newly defined classes with the database and to delegate object instantiation to the database. `ptype` derives from the build in `type`.

The constructor performs bookkeeping operations on persistent properties, and registers the class with the database.

`__call__`(**args, **kwargs*)

This method delegates object instantiation to the database

The following methods on `ptype` behave as if they are class-methods on `pobject` (see below).

`all_sublasses`()

Recursively generate the class and all of its subclasses

`make_new_instance`(**args, **kwargs*)

Create a new instance, passing `args` and `kwargs` to the constructor

`make_old_instance`(*oid*)

Create a 'bare' instance, bypassing the object constructor, for an object that will be retrieved from the

database

class **pobject**

This is be the base class for all persistent classes.

A persistent class should:

1. be a subclass of `pobject`
2. define an attribute `database`, referring to a database interface
3. define one or more persistent attributes.

__metaclass__

The metaclass of all persistent classes is `ptype` (q.v.).

commit()

Commit the persistent object to the database

First, recursively, commit all objects that the object refers to, then commit the object itself.

4.2 ptype.pprop — Persistent properties

class **pprop**

The base class for all persistent properties

All persistent properties have the following attributes

name

The attribute name that refers to the property. This value is set upon initialization of the class by the metaclass.

key

The key that is used to store the value of the property in the instance `__dict__`. This value is set upon initialization of the class by the metaclass.

ptype

The type of the persistent property. Assignment of a value `val` is type checked using `isinstance(val, ptype)`

default

The default value of the persistent property. The property getter returns the default, if the values is not found in the instance `__dict__`.

class **pval**(*default=None*)

The base class for persistent (atomic) values

__get__(*instance, cls=None*)

The getter.

If called as `instance.attr`, return the value of the property. If called as `cls.attr`, return a `prel.value_attr` object.

The following classed derive from `pval`:

Property	ptype	default
pint	int	0
pfloat	float	0.0
pstr	str	''
pdate	datetime.date	datetime.date(1,1,1)
ptime	datetime.time	datetime.time()
pdatetime	datetime.datetime	datetime.datetime(1,1,1)

class **pref**(*cls, default=None*)

A reference to another persistent class `cls`.

__get__(*instance, cls=None*)

The getter.

If called as `instance.attr`, return the value of the property. If the value is a tuple `(cls, oid)` then it first instantiates the object using `cls(oid=oid)`.

If called as `cls.attr`, return a `prel.ref_attr` object.

4.3 ptype.prel — Relations

This module implements the relational algebra for PyORQ.

Ths module is used internally by PyORQ, You don't need this information to work with queries

A more or less formal definition is given by the following grammar:

```
relation   := rel_not | rel_and | rel_or | comparions
rel_not    := '~' relation
rel_and    := relation '&' relation
rel_or     := relation '|' relation
comparison := expr cmp_op expr
cmp_op     := '==' | '!=' | '<' | '<=' | '>' | '>>'
expression := min_expr | bin_expr | term
min_expr   := '-' expr
bin_expr   := expr bin_op expr
bin_op     := '+' | '-' | '*' | '/'
term       := value_attr | ref_attr | int | long | float | str ...
```

All classes in this module define the following methods:

py_repr()
: Build a representation that can be evaluated by python. Used by the nodb() interface to build a list comprehension. that should produce the same result as an SQL query

sql_repr(*db*)
: Build a representation that can be used in a WHERE clause. The db-argument is used to call back into the database interface to build representations for builtin values

free_variable()
: Returns the free variable of the relation or the expression

updated_bound_variables(*d*)
: Used to find the join clauses necessary to build the cross reference between tables. A query of the form `A.b.c.d == 3`, produces a bound-variable dict `{('_x', 'b'): A.b.ptype, ('_x', 'b', 'c'): A.b.ptype.c.ptype}` This dictionary is used to produce aliases and join clauses.

The module defines the following classes

class expression()
: Expression object define arithmetic operations that return new expression object. Expression object define comparison operators that return relation objects.

class relation()
: Relation objects define logical operators (using the bitwise operators) that return new relation objects.

  __iter__()
  : Yield all instances that satisfy the relation.

    The iterator loops over all subclasses of the free variable of the relation. The database evaluates the relation for each subclass

The following classes are derived from `expression`.

class value_attr(*parent, prop*)
: A reference to a persistent value.

  This object is returned by the getter of property `prop` if the attribute is accessed as a class attribute. In this case `parent` refers to the class. The object may also be returned by `ref_attr.__getattr__(attr)` if `attr` is as persistent value of the referred property. In this case the parent will be the `ref_attr` object.

class ref_attr(*parent, prop*)
: A reference to a persistent class.

  This object is returned by the getter of property `prop` if the attribute is accessed as a class attribute. In this case `parent` refers to the class. The object may also be returned by `ref_attr.__getattr__(attr)`

if `attr` is as persistent reference of the referred property. In this case the parent will be the `ref_attr` object.

`__getattr__`(*attr*)
Used to build chains of references to references.

If attr is a persistent value of the persistent class that `ref_attr` refers to, return a new `value_attr` object. If it is a persistent ref, return a `ref_attr` object. Otherwise, raise an `AttributeError`.

Comparisons of `ref_attr` objects with instances implies identity comparison.

`__eq__`(*other*)
Returns a `comp_is` object.

`__ne__`(*other*)
Returns a `comp_not_is` object.

class `term`(*value*)
An expression object, containing a `value` of one of the built in Python types.

class `expr_min`(*arg*)
Produced by `-arg`

class `expr_add`(*lhs*, *rhs*)
Produced by `lhs + rhs`

class `expr_sub`(*lhs*, *rhs*)
Produced by `lhs - rhs`

class `expr_mul`(*lhs*, *rhs*)
Produced by `lhs * rhs`

class `expr_div`(*lhs*, *rhs*)
Produced by `lhs / rhs`

The following classed are derived from `relation`.

class `comp_eq`(*lhs*, *rhs*)
Produced by `lhs == rhs`

class `comp_ne`(*lhs*, *rhs*)
Produced by `lhs != rhs`

class `comp_ge`(*lhs*, *rhs*)
Produced by `lhs >= rhs`

class `comp_gt`(*lhs*, *rhs*)
Produced by `lhs > rhs`

class `comp_le`(*lhs*, *rhs*)
Produced by `lhs <= rhs`

class `comp_lt`(*lhs*, *rhs*)
Produced by `lhs < rhs`

class `comp_is`(*lhs*, *rhs*)
Produced by `lhs == rhs` if `lhs` or `rhs` is a `ref_attr`

class `comp_not_is`(*lhs*, *rhs*)
Produced by `lhs != rhs` if `lhs` or `rhs` is a `ref_attr`

class `rel_and`(*lhs*, *rhs*)
Produced by `lhs &rhs`

class `rel_or`(*lhs*, *rhs*)
Produced by `lhs | rhs`

class `rel_not`(*arg*)
Produced by `~arg`

5 pyorq.interface — The database interfaces

5.1 pyorq.interface.db_base — The interface definition

This module defines the abstract base class for all database interfaces.

Ths module is used internally by PyORQ, You don't need this information to use the databases

class db_base

The following methods constitute the external interface of the database interfaces

register_class(*cls*)

Register class cls with the database.

This method is called by ptype.__init__. If the class was not previously registered, a table will be created.

get_instance(*cls, *args, **kwargs*)

Invoked when creating an object

If 'oid' is in the kwargs dict, return a known instance, else create a new instance.

commit(*instance*)

Commit an instance to the database.

If the instance is known, update the contents of the database, else insert the instance in the database.

query_generator(*cls, query*)

Yield all instances from

In addition, the following methods are available for simple database operations.

drop_table(*class_name*)

Remove the table for the class with name class_name.

PyORQ does not support schema evolution yet. You can use this method to remove the old table before modifying a persistent class definition

CAUTION. This may leave the database in an inconsistent state with dangling references.

empty_table(*class_name*)

Remove the contents of the table for the class with name class_name.

CAUTION. This may leave the database in an inconsistent state with dangling references.

5.2 pyorq.interface.postgresql_db — for PostgreSQL

This module imports libpq from pyPgSQL

class postgresql_db(*dsn=None, user=None, password=None, host=None, database=None, port=None, options=None, tty=None, client_encoding=None, unicode_results=None*)

Create an interface to a PostgreSQL database.

All parameters are used to create the argument for the function libpq.PQconnectdb, which creates a connection.

dsn is a string of the form 'host:port:database:user:passwd:options:tty', that can be used as an alternative to the named arguments.

See Also:

pyPgSQL
(http://pypgsql.sourceforge.net/)

For the Python interface to PostgreSQL by Billy G. Allie.

5.3 pyorq.interface.mysql_db — for MySQL

This module imports _mysql, the low-level interface to MySQL, that is part of MySQL-Python.

class `mysql_db`(*host=None, user=None, passwd=None, db=None, port=None*)
Create an interface to a MySQL database.

All parameters are used to create the argument for the function `_mysql.connect`, which creates a connection.

See Also:

MySQL-Python
(http://sourceforge.net/projects/mysql-python)
For the Python interface to MySQL by Andy Dustman.

5.4 `pyorq.interface.sqlite_db` — for SQLite

This module imports `_sqlite`, the low-level interface to SQLite, that is provided by PySQLite.

class `sqlite_db`(*database, mode=0755*)
Create an interface to an SQLite database.

`database` (the filename of the database) and `mode` (the permissions for the file that constitutes the database) are passed to `_sqlite.connect`

See Also:

PySQLite
(http://sourceforge.net/projects/pysqlite)
For the Python interface to SQLite by Michael Owens and Gerhard Haring.