PyDO — Python Data Objects

Release 1.0

Drew Csillag

January 29, 2007

CONTENTS

CHAPTER

ONE

What is it?

In short, PyDO allows you to simplify access to databases in a comprehensible way. Ok, now for a bit more detail. SDS, PyDO's predecessor made data modelling and access considerably easier than doing the table design, SQL writing et. al. than doing it yourself. The main problem was is that the database people had a hard time understanding what it actually did under the hood, because, for them (and rightly so) it was really important that it didn't do anything stupid and allowed them to optimize it to death. Since it was difficult to explain to them how it did things, and it did constrain them in meaningful ways, they didn't buy into it (and rightly so). Basically, SDS traded ease of use for understanding, a tradeoff which was it's undoing.

PyDO is meant as a way to give ultimate control to the database people (if they want it) when it comes to database access and still be relatively easy to use, but not as easy as SDS was.

PyDO is also easy to configure, easy to see what goes on under the hood and extremely lightweight (the PyDO.py file comes in currently at 616 lines of not-very-dense code). Because the mapping is quite thin, it is easy to explain how the mythical python expression:

```
SomeObject.getUnique(FOO=3)
```

would yield the SQL query (when using the oracle driver)

```
SELECT COL1, COL2, COL3, FOO from TABLE where FOO = :p1
```

with `:p1` being bound to the integer 3 given it's definition.

Not only that, but you can override the way fetches and/or mutations are done so that they don't necessarily even yeild SQL queries, in the case that you want to do stored procedure access. In general, if you want to go direct to the database connection level to do something, you can, and PyDO doesn't care, very much unlike SDS which potentially could get very confused.

PyDO has no notion of a relation. Relations are handled by the PyDO data developer by using methods. For example, if you have a Users class and a Groups class, one would likely write a getGroups() method on the Users object to fetch the Groups object associated with it. PyDO does however provide convience functions to make implementing relations simpler, specifically joinTable and joinTableSQL methods which make many-to-many relations easier. One-to-one and one-to-many are are typically done making calls to `getUnique()` and `getSome()` methods on the target class.

Unlike SDS, PyDO can also use more than one connection at a time. Each data class defines a connection alias, which maps to a PyDO connection string which subsequently maps to a database interface instance (specific to the database type). The connection alias feature exists because you don't want to have to change all the connect strings in your code to move them from the development environment to the production environment, you just have to change the connect string that the alias points to.

CHAPTER

TWO

How to Define a Data Class

To define a PyDO data class, the first thing to do is inherit from the PyDO base class. From there, you define a series of class attributes to configure the object.

The connectionAlias attribute specifies the connection alias mentioned above to determine which connection to use.

The table attribute specifies what database table this object maps to. Multiple dataclasses may point at the same database table.

The fields attribute is a tuple of two-tuples of column name (or field name)/database type. The case of the field name *is* significant. For all intents and purposes, use upper case unless the documentation for your database driver says otherwise (none of them currently do, or even have docs either for that matter). If you have multiple data classes pointing at the same database table, they need not specify the same field tuples (they can though).

Data class instances are mutable unless you say `mutable = 0` in the definition of your data class.

If you would like for fields in rows to be populated with values from sequences (on databases that have named sequences, i.e. oracle) when creating new rows, this can be done by specifying the sequenced attribute as a dictionary of fieldname:sequence name pairs.

If you would like to fetch fields that are populated via the auto-increment feature of your database (if it has one, like MySQL, oracle doesn't) on insert, this can be done by specifying the auto_increment attribute as a dictionary of fieldname:autoincname pairs. In the case that autoincrement field fetches aren't named (i.e. MySQL), just specify 1 as the autoincname, and beware then that you can have only one item in the dictionary.

For some methods (i.e. `getUnique`, `delete`, `refresh`, etc.) that PyDO has, it requires that it be able to obtain a unique row given a set of column names/values. The way to specify this is to set the unique attribute on your data class to a list of strings or tuples of strings (can mix and match) that identify that either this column (in the case of a string) or this set of columns (tuple of strings) uniquely identifies a row.

Other attributes defined in a data class definition are ignored and will not be present in the actual class.

If you want to add attributes into a data class instance, define the `__init__` method (it will have no argument other than `self`) and it can define whatever other instance attributes it likes, although redefining data class attributes will have undefined behavior (it might work *shrug*).

2.1 Methods

PyDO, like SDS supports methods. Unlike SDS, though PyDO also has the notion of static methods, methods that apply to the data class and not an instance of the data class.

Defining a method is the same as defining regular python methods and needs no explanation.

Defining a static method is merely a matter of defining the method with static_ prepended to the method name. In which case, the self argument points to the data class and not the data class instance.

As you would expect, calling `self.method()` where method is static is the same as calling `SelfsClass.method()`.

If you want to get a hold of the static method to be able to call it from, say, a static method in the a subclass and have it be executed in the class context of the subclass (not the superclass), use the full `self.static_method()` form (`static_` prepended to the static method name).

2.2 Inheritance

Inheritance is supported, albeit in a somewhat limited way. Methods (instance and static) are inherited as you would expect. Fields in a super class will be inherited into the subclass, where you can augment the fields tuple or change the database type (by specifying the field, but with a different database type). This second form may or may not be supported in future releases. The unique is inherited, but setting it will override, not augment, the super class' definition. Inheriting from multiple PyDO classes is undefined as to the real behavior. It may work, but no guarantees.

CHAPTER

THREE

The Details

3.1 Connect Strings

When calling `PyDO.PyDBI.DBIInitAlias`, you have to specify a connect string. If you are using PyDO from within the SkunkWeb server, use the caching versions of the connect strings so that connections get rolled back properly in the event of an error. Obviously, if you dont have the pylib modules required for the caching versions, use the direct methods.

For Oracle, they take one of two forms (either can optionally have `|verbose` appended to them to log the sql executed by the connection):

`pydo:oracle:user|cache` uses the connection caching of the `Oracle` pylib that is used by the `oracle` SkunkWeb service.

`pydo:oracle:user/pw@host` use the `DCOracle` module directly.

For PostgreSQL, they also take one of two forms (either can optionally have `:verbose` appended to them to log the sql executed by the connection):

`pydo:postgresql:user:cache` uses the connection caching of the `PostgreSql` pylib that is used by the `postgresql` SkunkWeb service.

`pydo:postgresql:normal_postgresql_connstr` use the `pgdb` module directly. In addition, if, in lieu of the host portion of the normal PostgreSQL connection string you put `host|port` instead, it will connect to the database listening on the port *port* instead of the default port.

For MySQL, they can take one of a few forms:

`pydo:mysql:normal_mysql_connect_string` use the `MySQL` module directly. To use a cached connection (maintained by the `mysql` SkunkWeb Service, just use `pydo:mysql::::name` where `name` is the connection name from the `MySQLConnectParams` configuration variable in `sw.conf`.

3.2 Inheritance

All base class fields (columns) are inherited, subclasses can add fields and can only change inherited field types.

The `unique` and `connectionAlias` attributes are inherited from left most, depth first class which defines them.

Static methods are inherited as static methods.

Instance methods are inherited as instance methods.

PyDO classes cannot inherit from non-PyDO classes.

The `_instantiable`, `sequenced` and `auto_increment` attributes are not inherited.

3.3 Data Class Details

To be instantiable, class must define (or inherit) the `connectionAlias`, table and fields attributes, or, can set the _instantiable attribute to 1. The overridability is there so, in the case where you have your own fetching mechanisms (i.e. stored procs), you can make the object instantiable even though it normally wouldn't be (since no table for instance).

The fields member is tuple of (columnname, dbtype) pairs. The unique attribute is a list of strings and/or tuple of strings. If a string, this says that this field is unique in the table, if a tuple, this says these fields taken together are unique in the table.

You can make the dataclass instances immutable by defining the `mutable` attribute as a false value (`None`, 0, empty string, etc.)

For databases with named sequences, you can populate the value of an field by defining the sequenced member as a dict of {`fieldname:  sequence_name`} pairs, whereby if, on a call to `new()`, the fields specified in sequenced are not present, the values are fetched from the sequence(s) before insert and subsequently inserted.

For databases with auto-increment fields, you can populate the value of an field by defining the auto_increment member with a dict of {`fieldname:  auto_increment_name`} pairs and the values will be populated into the object after the insert is executed. In the case of MySQL, there can only be one auto-increment field per table, so the auto_increment_name is needed, but it's value is irrelevant.

To define a static method (one that applies to the dataclass) define the method as

`def static_`*realmethodname*

Others are instance methods.

To get an unbound instance method, get *data_class*.*instance_method*, to get a static method, unbound from it's original data class (presumably called from a sub-data-classe), use *data_class*.static_*static_method_name*.

Attributes (static methods, data members, etc.) on data classes are accessible from instances.

3.3.1 Data Class Attributes

`_klass`	name of the data class
`_baseClasses`	tuple of super classes
`_staticMethods`	dict of static methods
`_instanceMethods`	dict of instance methods
`_rootClass`	is the `_PyDOBase` root class metaclass instance
`_instantiable`	is this instantiable
`connectionAlias`	connection alias string
`table`	string naming the table
`mutable`	are instances of this mutable?
`fieldDict`	the dict of columnname: dbtype
`unique`	list of candidate keys

`sequenced` dict of attrname: seq_name

`auto_increment` dict of attrname: auto_increment_name

3.3.2 Data Class Instance Attributes

`_dataClass` the class which I'm an instance of

`_dict` dict of current row

3.3.3 Static Methods

`getDBI()` gets database interface (see conn.readme)

`getColumns(qualified= None)` get column names (with table name if qualified)

`getTable()` get table name

`_baseSelect(qualified = None)` get SELECT fragment to get rows of object

`_matchUnique(kw)` returns an eligible candidate key based on contents of `_dict`

`_uniqueWhere(conn, kw)` generate a where clause from output of `_matchUnique`

`getUnique(**kw)` get a unique obj based on keyword args

`getSome(**kw)` get some objs based on keyword args

`getSomeSQL(**kw)` given the attribute/value pairs in kw, return sql statement, values to be used in a call to `conn.execute`. If kw is empty, the WHERE text in the sql statement will still be preseverved. Basically useful for constructing ad-hoc queries on a table.

`getSomeWhere(*args, **kw)` Allows you to use the operator objects in `PyDO.operators` to be able to use sql operators other than the implicit `AND` as used by the other static get methods. The `**kw` argument is the same as the other static get methods. The `*args` argument however allows you to combine operators to do operations like OR, NOT, LIKE, etc. For example, the following would get all rows where the last name field was LIKE `Ingers%`.

```
obj.getSomeWhere(LIKE(FIELD('last_name'), ('Ingers%')))
```

The complete list of operators is in (see section 5, page 19).

`getTupleWhere(opTuple, **kw)` Allows you to use a somewhat Lispish notation for generating SQL queries, like so:

```
obj.getTupleWhere(('OR',
                   ('LIKE', FIELD('last_name'), 'Ingers%'),
                   ('OR', ('<>', FIELD('id'), 355),
                          ('=', FIELD('id'), 356))))
```

Strings are used to represent operators rather than the SQLOperator class wrappers used in getSomeWhere(), but the FIELD and SET classes are still useful. The `kw` argument is treated the same as in getSome() and getSomeWhere().

`getSQLWhere(sql, values=())` executes a sql statement to fetch the object type where you supply the where clause (without the `WHERE` keyword) and values in the case that you bind variables.

`scatterFetchSQL(objlist)` do a scatter fetch (a select from more than one table) based on the relation information in objs which is of the form:

```
[
   (object, attributes, destinationObject, destinationAttributes),
   (object, attributes, destinationObject, destinationAttributes)
]
```

This basically states that objects attributes are a foreign key to destinationObject's destinationAttributes.

For example, if you have User and UserResidence classes, a scatter fetch may be simply:

```
userObj.scatterFetchSQL( [ (UserResidence, 'USER_OID', User, 'OID') ] )
```

which if executed would return a list of tuples of: (userObj, userResObj) where `userObj.['OID'] == userResObj['USER_OID']`

This function returns sql, baseColumnNames, and a modified version of the objs argument.

`scatterFetchPost(objs, sql, vals, cols)` handle the execution and processing of a scatter fetch to produce a result list – returns the list of tuples referred to in the docstring of `scatterFetchSQL`.

`scatterFetch(objs, **kw)` see `scatterFetchSQL` for format of `objs` and `getSome` for format of `**kw`.

`new(refetch = None, **kw)` get new object based on kw args, if refetch is true, refetch obj after insert

`_validateFields(dict)` does simple validation of fields on insert

`commit()` causes the database connection of this object to commit.

`rollback()` causes the database connection of this object to roll back.

3.3.4 Instance Methods

`__init__()` can be used to prepopulate data object instance attributes. Can have no arguments other than self.

`dict()` returns copy of dict representing current row

`updateValues(dict)` make the values in current dict "stick"

`delete()` delete current row

`refresh()` reload current object

`joinTable(thisAttrNames, pivotTable, thisSideColumns, thatSideColumns, thatObject, thatAtt`
do cool m2m join

`joinTableSQL(thisAttrNames, pivotTable, thisSideColumns, thatSideColumns, thatObject, that`
returns `sql` and `value` list for `conn.execute` to do a m2m join but doesn't execute it so you can do ordering or other stuff.

3.3.5 Dict-type Instance Methods

PyDO objects also obey a good majority of the dictionary interface. They are:

```
__getitem__(item)
__setitem__(item, val)
items()
copy()
has_key(key)
key()
values()
get(item, default = None)
update(dict)
```

CHAPTER

FOUR

The Big Example

An Oracle example:

Ok, for starters you’ve got a simple users table:

```
CREATE TABLE USERS (
    OID NUMBER,
    USERNAME VARCHAR(16),
    PASSWORD VARCHAR(16),
    CREATED DATE,
    LAST_MOD DATE
);
```

And a sequencer for the `OID`:

```
CREATE SEQUENCE USERS_OID_SEQ;
```

And you want to do stuff with it using PyDO.

```
from PyDO import *
DBIInitAlias(’drew’, ’pydo:oracle:drew/drew@drew’)
class Users(PyDO):
    connectionAlias = ’drew’
    table = ’USERS’
    fields = (
        (’OID’     , ’NUMBER’),
        (’USERNAME’, ’VARCHAR(16)’),
        (’PASSWORD’, ’VARCHAR(16)’),
        (’CREATED’ , ’DATE’),
        (’LAST_MOD’, ’DATE’)
        )
    sequenced = {
        ’OID’: ’USERS_OID_SEQ’
        }
    unique = [ ’OID’, ’USERNAME’ ]
```

Ok, line-by-line, this is what this all means:

```
> from PyDO import *
```

Import the contents of PyDO into your module namespace. PyDO is pretty clean and shouldn't pollute the namespace significantly as it was designed to be imported this way, but if that irks you, doing a regular `import PyDO` will also work (but you'll need to adequately qualify things, obviously).

```
> DBIInitAlias('drew', 'pydo:oracle:drew/drew@drew')
```

PyDO has a database driver library thingy. It's not really meant for use outside of PyDO, but you can use it if you like, it's mainly there so the main PyDO code doesn't have to care so much about the underlying database so much in terms of things like: whether it support bind variables or not and etc.

The arguments to `DBIInitAlias` are: a connection alias name (used as connectionAlias in your data classes), and a PyDO connect string. For oracle, the connect strings are of the form `pydo:oracle:`*user/password@inst*.

```
> class Users(PyDO):
```

All dataclasses inherit directly or indirectly from the `PyDO` base class.

```
> connectionAlias = 'drew'
```

Used to select the database connection to use for this object. You can have more than one connection going at a time, so you need to choose one (presumably the one that has the table you're going to use). In this case we're going to use the alias that we initialized previously.

```
> table = 'USERS'
```

PyDO needs to know what table the rows will be coming from if it's going to do anything, so we point it at the previously created `USERS` table.

```
> fields = (
>     ('OID'     , 'NUMBER'),
>     ('USERNAME', 'VARCHAR(16)'),
>     ('PASSWORD', 'VARCHAR(16)'),
>     ('CREATED' , 'DATE'),
>     ('LAST_MOD', 'DATE')
>     )
```

What you need to do here is associate the column names from the table to their database type. The case of the column names *must* be the same as the native case of the database for such things (specifically, the same case as what the database driver returns on a describe of a query). For most databases, this is uppercase, same for the database type.

```
> sequenced = {
>     'OID': 'USERS_OID_SEQ'
> }
```

This says, if on a call to `new()` (described later), `OID` is not specified, then fetch it from the sequence here named.

```
> unique = [ 'OID', 'USERNAME' ]
```

This is a list of candidate keys — columns that uniquely identify a row.

4.1 Using This New Data Class

Assuming the aforecreated `USERS` table is empty, we need to put some something in it before we start.

```
newUser = Users.new(USERNAME = 'drew', PASSWORD = 'foo', CREATED = SYSDATE,
                    LAST_MOD = SYSDATE)
```

The new method inserts a new row into the table. There is an optional parameter, `refetch` which effectively calls the `refresh` method (described below). This is useful in the case where you have a table with default values for columns and you want to make references to the values with the defaults in place.

What this will do is: fetch a new `OID` from `USERS_OID_SEQ` since `OID` wasn't specified above, and subsequently insert a new row into the `USERS` table with the `OID` and the values specified in the call to `new()`.

As you will notice, `SYSDATE` is a variable that translates to the databases' idea of the current date (and time).

Now that we have a Users instance, we can examine it a bit more closely. PyDO subclass instances observe the python dictionary interface.

For example:

```
>>> newUser['USERNAME']
'drew'
>>> newUser['OID']
1
>>> newUser.keys()
('OID', 'USERNAME', 'PASSWORD', 'CREATED', 'LAST_MOD')
```

4.2 Mutating Data Class Instances

If you don't want your data class instances to be mutable (for whatever reason), assign 1 to the `mutable` attribute in your class definition.

To mutate an object, you use the dictionary-style mutation interface. For example, to change the value of `USERNAME` in the current row:

```
>>> newUser['USERNAME'] = 'fred'
```

What this will do is cause the following UPDATE query to be sent to the connection.

```
UPDATE USERS SET USERNAME = :p1 WHERE OID = :p2
```

(bind variables `:p1` = `'fred'` and `:p2 = 1`)

One might ask: "how the hell did that happen?" The answer is this: it got the table from the table specified in the data class description

```
> table = 'USERS'
```

The attribute name is the item you assigned to. The `OID = :p2` part is a bit more interesting. If you look above, you'll see:

```
> unique = ['OID', 'USERNAME']
```

What PyDO does is this: it loops over the unique list and for each item in the list is determines if it is a tuple or string. If it's a string, it's the name of an field that uniquely identifies a row in the table (here `'OID'`). If the current object has that key-value pair, it stops having found an identifying field and so composes the where clause. If it is a tuple, it is a set of fields that uniquely identify the row. If all such fields are populated in the current object, it will stop and compose the where clause from the ANDing check of those fields in the current object. In the case where either there is no unique line specified or the key-value pairs aren't defined in the current object, an exception saying "No way to get unique row!" will be raised.

Ideally, if you want to update more than one field in your object in one UPDATE query, you can use the `update` method (from the dictionary interface) to accomplish this.

```
> newUser.update({'USERNAME': 'barney', PASSWORD='iF0rG0t'})
```

This will update both column values in one UPDATE query.

You might now say, well, that's all fine an dandy, but to do this correctly, I want to make sure that the `LAST_MOD` field gets updated appropriately when people change the object! Well, be at rest, we can do that too. Behind the scenes, the `__getitem__` and update methods call an instance method updateValues that actually does the hard work and we can override this to update `LAST_MOD` as appropriate.

4.3 Defining Instance Methods

If we add the following method to the `Users` class definition, this will do the trick:

```
    def updateValues(self, dict):
        if not dict.has_key['LAST_MOD']:
            dict = dict.copy()
            dict['LAST_MOD'] = SYSDATE
        return PyDO.updateValues(self, dict)
```

All it does is say, if they didn't specify a value for `LAST_MOD` (we assume here that if they specified it, they did for good reason), we make a copy of the dict (in the case that they still hold a reference to it, we don't want to screw it up) and set `LAST_MOD` to `SYSDATE`, and subsequently call our baseclasses version of `updateValues`.

This brings us to methods, of which there are two flavors, static and instance. Python itself doesn't have the notion of static methods, but for certain applications (specifically PyDO), they can be made available and incredibly useful.

To create a regular method, just write it as if everything was normal in python land. Nothing big to mention here. It will apply to instances of data classes and you can get an unbound verion by saying *class.method* just as in regular Python.

4.4 Defining Static Methods

For static methods, you define your method as such:

`def static_`*mymethodname*`(self,` *...whatever...*`):`

The `static_` prefix says "this is a static method". The `self` argument will point to the data class itself, not an instance of the dataclass. In the case that you want to call a super classes static method on the current subclass, and in the context of the subclass, you say:

```
        fooresult = MySuperClass.static_barmethod(self, baz, fred, barney)
```

This is useful when you are overloading a static method in a subclass but still want to call the superclass version (such as `new`). This is very much unlike Java, which disallows this. (don't know about C++)

Your newly-defined method can then be called as

`SomeClass.`*mymethodname*`(`*...whatever...*`)` without the `static_` prefix to call the method statically. For example, the call to the `new` method on the `Users` object towards the beginning of this document is a static call (it's defined as `static_new` in the `PyDO` base class).

NOTE: you cannot override a static method with an instance method or vice versa.

Why this is useful is this: You want to make it so that you don't have to specify the `CREATED` and `LAST_MOD` fields when making a call to `new` since the caller shouldn't really have to care and it can be taken care of automatically. You can, if you want to, enforce what fields they can or must set on a call to new. For example: setting the `CREATED` and `LAST_MOD` automatically and enforcing that `USERNAME` and `PASSWORD` only are specified.

```
    def static_new(self, refetch = None, USERNAME, PASSWORD):
        return PyDO.static_new(self, refetch, USERNAME=USERNAME,
                               PASSWORD=PASSWORD, CREATED=SYSDATE,
                               LAST_MOD=SYSDATE)
```

MAKE SURE TO USE THE STATIC UNBOUND VERSION WHEN CALLING YOUR SUPERCLASS OR THE WRONG THINGS WILL LIKELY HAPPEN!

In the cases where PyDO is not your direct superclass, you might call your superclass' `static_new` method instead. On the other hand, you may want to handle the new method entirely yourself.

4.5 Relations and PyDO

The way you do relations with PyDO is with methods. For example, if we had a `Files` class which had an field `OWNER_ID` which was a foreign key to the `USERS` table, we could write a method for the `Users` object like this (a one to many relation):

```
    def getFiles(self):
        return Files.getSome(OWNER_ID = self['OID'])
```

The `getSome` static method, given fields in the object will generate a where clause with those fields and return a list of objects, each of which representing one row.

If `Users` had an One to One relationship with `Residence`, we could write a method to get it (presuming `Residence`, again has a foreign key to the `USERS` table in a column/field named `OWNER_ID`):

```
def getResidence(self):
    return Residences.getUnique(OWNER_ID = self['OID'])
```

The `getUnique` static method is similar to `getSome` except that it will return only one row or `None`. It uses the `unique` attribute (here on the `Residences` object) to determine how to get a unique row. If you don't specify any identifying rows, it will raise an exception saying "No way to get a unique row", or in the case that it mysteriously finds more than one row, will raise a similar exception.

To do Many To Many relations, things are a bit more interesting. Since there may or may not be an object that represents the pivot table (or linkage table) that links the two tables together, and you probably wouldn't want to do the work to traverse all of them anyhow, there is a `joinTable` method which simplifies the work.

Say there is a `Groups` entity and a table `USERS_TO_GROUPS` which is the pivot table and has two columns, `USER_ID` and `GROUP_ID` (foriegn keyed as appropriate). You'd write a method `getGroups` as such:

```
def getGroups(self):
    return self.joinTable('OID', 'USERS_TO_GROUPS', 'USER_ID',
                          'GROUP_ID', Groups, 'OID')
```

What this will do is do the join across the `USERS_TO_GROUPS` table to the table that the `Groups` object corresponds to. The parameters (matched up to the arguments supplied above) are:

`thisAttributeNames` `'OID'` attribute(s) in current object to join from

`pivotTable` `'USERS_TO_GROUPS'` pivot table name

`thisSideColumns` `'USER_ID'` column(s) that correspond to the foriegn key column to `myAttributeName`.

`thatSideColumns` `'GROUP_ID'` column(s) that correspond to the foriegn key column to `thatAttributeName`.

`thatObject` `Groups` the destination object.

`thatAttributeNames` `'OID'` see `thatSideColumns`.

If in the case you want to do things like ordering and such on a many to many relation, you can use the `joinTableSQL` function (takes the same arguments) to get the sql and value list to use. From there you can add to the generated sql statement things like `ORDER BY FOOTABLE.BAZCOLUMN` and such. From there you use the dbi's execute function to execute the query and subsequently construct the objects.

4.6 Refreshing an Object

Ok, we've learned that the information in the current row has been altered (by nefarious means - mwahahahah! or not) in the database, but we still hold the old information. By calling the `refresh()` method, it effectively does a `getUnique` on itself and refreshes it's contents. If the object no longer exists, it raises an appropriate error.

4.7 Deleting An object

We're now done with this user object and want to dispose of it from the database. Using the `delete` method it issues an appropriate DELETE query to the database. Since it needs a unique row, the usual things related to uniqueness mentioned above apply.

4.8 Committing and Rollback

Each PyDO object contains the two methods `commit()` and `rollback()`. These, in turn call the corresponding methods on their database connection. Normally there is little room for confusion, but in the case of using multiple database connections simultaneously, this could be a bit more confusing on what exactly is getting committed.

CHAPTER

FIVE

Operators

The `PyDO.operators` module consists of support code to let you do more with the `getSomeWhere` and `getTupleWhere` static methods of PyDO objects. These are no more than SQL-generating routines, and hence fall into the category of syntactic sugar, but they can be convenient in some circumstances.

The below are SQLOperator subclasses for use with the `getSomeWhere` method:

`NOT` takes one argument, the item to negate.

`EQ` takes two arguments, the items to compare for equality.

`NE` takes two arguments, the items to compare for inequality.

`LT` takes two arguments, the items to compare for less than.

`LT_EQ` takes two arguments, the items to compare for less than or equal to.

`GT` takes two arguments, the items to compare for greater than.

`GT_EQ` takes two arguments, the items to compare for greater than or equal to.

`LIKE` takes two arguments, the first a `FIELD`and the second, a string that is the LIKE matcher.

`IN` takes two arguments, the first a `FIELD`, and the second, a `SET` constructed with the items that should be checked for membership. In short, the SQL IN operator.

`AND` Takes two arguments, two sub expressions to logically and.

`OR` Takes two arguments, two sub expressions to logically or.

`PLUS, MINUS, MULT, DIV` Takes two arguments, both of them either numerical constants or a `FIELD`. The values will be mathematically operated on as appropriate.

Two other classes exist to control the escaping of SQL output. The `FIELD` class is used to indicate that a given string is not a string literal, but the name of a database column. The `SET` class is used to generate a properly formatted list of things to use as the second argument of the `IN` operator.

`getTupleWhere` does not use thes SQLOperator subclasses, but directly inserts the operator strings provided by the user into the generated SQL. The FIELD and SET classes are still needed, however.

CHAPTER

SIX

The *Genscripts

In the PyDO distribution, there are a number of scripts called *something*`genscript.py`. What these scripts do, is login to the database, and ask the database about the schema (read as: grovel over the system catalogs), and ask the user (that is, you) a few questions about what it finds, and what to call things. Then, they output a python module containing premade PyDO classes that contain the knowledge of the schema it produced. Currently genscripts exist for Oracle (`ogenscript.py`, PostgreSQL `pgenscript.py` and SAPDB `sabdbgenscript`. They are a great way to automate something that would otherwise be tedious (at least) and errorprone. Not only that, but occasionally, you wind up discovering relations in your schema that you werent aware of.

In order for these tools to be able to gather relation information, the appropriate referential integrity constraints must be in place.

CHAPTER

SEVEN

Adding Support for Another Database

If you wish to add support for another database that is currently not supported, you have to implement a class that follows the following interface:

```
class interface:
    """don’t actually inherit from me, this is just for documentation
    purposes"""
    def __init__(self, dbconnstr):
        """the dbconnstr is the conn str with the pydo:dbkind: bit
        chopped off"""

        self.bindVariables = 1 | 0 # 1 - I support bind variables, 0 - I don’t
        pass

    def getConnection(self):
        """Get the actual database connection"""
        pass

    def bindVariable(self):
        """if you support bindVariables, return next bind variable name.
        suitable for direct inclusion into a sql query"""

    def sqlStringAndValue(self, val, attributeName, dbtype):
        """Returns a sql string and a value.  The literal is to be put into
        the sql query, the value should is put into the value list that is
        subsequently passed to execute().

        The reason for this is for example, using bind variables in the
        generated sql, you want to return a bind variable string and the
        value to be passed.  If doing such things requires state, you can
        clear it in resetQuery().

        """
        return "LITERAL", val

    def execute(self, sql, values, attributes):
        """Executes the statement with the values and does conversion
        of the return result as necessary.

        result is list of dictionaries, or number of rows affected"""
    def convertResultRows(self, columnNames, attributeKinds, rows):
        """converts the result list into a list of dictionaries keyed
        by column name, and data type conversion specified by the
        attributeKinds dictionary (keyed by attribute, valued by database
        datatype).
        """

    def resetQuery(self):
        """Reset things like bind variable numbers if necessary before a query
        Need only if there is state between sqlLiterals because of bind
        variables, otherwise, don’t need this.  Called before a query is
        executed.
        """

    def getSequence(self, name):
        """If db has sequences, this should return the sequence named name"""
        return 1

    def getAutoIncrement(self, name):
        """if things like mysql where can get the sequence after the insert"""
        return 1

    def typeCheckAndConvert(self, value, attributeName, attrDbType):
        """check values type to see that it is valid and subsequently
        do any conversion to value necessary to talk to the database with
        it, i.e. mxDateTime to database date representation"""

    def postInsertUpdate(self, PyDOObject, dict, isInsert):
        """to do anything needed after an insert or update of the values
        into table.  Specifically to handle cases like blobs where you
        insert/update with a new blob, but have to select for update and
        then deal with the blob post factum

        PyDOObject is the object being affected
        Dict is the dict of new values
        isInsert is a boolean stating whether or not this is an insert (true)
                 or an update (false).
        """
```

Once that is done, an appropriate change to the `_driverConfig` dictionary in the `PyDBI.py` file.