PyDO2 Manual

Author: Jacob Smullyan <smulloni@smullyan.org>
Revision: 1695
Date: 2005-11-17 14:43:33 -0500 (Thu, 17 Nov 2005)
Status: Draft

Table of Contents

1 Introduction

PyDO is an ORM (Object-Relational Mapper) database access library for Python. This document describes PyDO version 2.0 and later only.

2 Acknowledgements

PyDO was originally written by Drew Csillag in 2000, and released under the GPL license in 2001 as part of SkunkWeb. Several developers have contributed to the codebase since then (see ACKNOWLEDGEMENTS in the source distribution). Jacob Smullyan is responsible for work on the 2.0 series, but while errors and misfeatures are attributable to him, the fundamental design remains Csillag's.

3 Overview

Note

PyDO is a Python package (finally named pydo in 2.0a3 and later; PyDO1 used the package name `PyDO`, and previous alpha releases of PyDO2 used first `PyDO` and then `PyDO2`), but all the public objects within its submodules (with the exception of the drivers) are imported into its top-level namespace. In what follows we will assume that the current namespace has been initialized by:

```
from pydo import *
```

(In particular, this means that when we want to refer to the `pydo.PyDO` class, we shall just write `PyDO`.)

PyDO's basic strategy is to let you define a `PyDO` subclass for every table in a database that you wish to represent. Each `PyDO` instance contains the data for one row in a database table. As `PyDO` is a `dict` subclass, you can access this data by key, and, if the class attribute `use_attributes` is true (the default) also by attribute. In either case, the key or attribute name is the name of the database column:

```
>>> MyPyDOInstance.title
'Fustian Wonders'
>>> MyPyDOInstance['title']
'Fustian Wonders'
```

If you have column names that are Python keywords (such as "pass", "class", etc.) a warning will be raised when the class is defined and an attempt at attribute access of that field will give rise to a `SyntaxError`, but you'll still be able to access it dictionary-style.

Instances are normally obtained, not by directly invoking the `PyDO` class's constructor, but by calling one of various class methods, discussed below, that return single instances or lists thereof.

PyDO is not an attempt to model all of SQL or its relational model. Its focus is on modelling those relations that tend to be both selectable and updateable, namely, tables. Since PyDO does not provide a complete abstraction layer over SQL, it is entirely appropriate for PyDO-generated queries to be mixed with raw SQL queries in real applications (although that may be hidden in your own abstraction layer).

PyDO 2 requires Python 2.4 or later.

4 Defining Table Classes

To model a database table, you define a subclass of `PyDO` and set some class attributes that describe the table:

```
from pydo import PyDO, Sequence, Unique

class Article(PyDO):
    """PyDO class for the Article table"""

    # define a connection alias so that PyDO knows how to
    # connect to the database
    connectionAlias='my_db'

    # the schema name, if applicable, e.g.:
    # schema = 'myschema'

    # the table name.
    table='article'

    # whether we are allowed to update instances of this class;
    # this defaults to True anyway.
    mutable=True

    # whether, after creating a new instance (i.e., performing
    # an insert) the instance should be refreshed to get any
    # default (or other automatic) values.
    refetch=True

    # declare the fields
    fields=(Sequence('id'),
            Unique('title'),
            'slug',
            'author',
            'created',
            'body')
```

The `connectionAlias` attribute must correspond to an alias initialized elsewhere (with the `initAlias()` function) that tells PyDO how to create a database connection.

If the database supports schemas, like later versions of PostgreSQL, the schema name can be specified by setting the `schema` attribute. When PyDO then generates SQL referring to this table, it will qualify it with the schema name. By default, `schema` is `None` and there will be no such qualification. (The method that returns the actual qualified tablename is `getTable(cls, withSchema=True)`; the `withSchema` parameter determines whether the returned value is schema qualified.)

The `table` attribute is simply the name of the table, view, or table-like entity (set function, for instance). By default, you can leave this out if the name of the class is the name of the table; in this case, `cls.table` will be a name made by coercing the class name to lowercase. If you don't want to allow this behavior, you can suppress it by setting the class attribute `guess_tablename` to `False`. (Note that this feature is of limited utility for database systems like mysql whose table/column names may be case-sensitive.)

Note

If you declare the table explicitly in a class, the `guess_tablename` attribute is set to `False` (unless you simultaneously set it also) so that subclasses will inherit the declared table name.

The `fields` attribute should be a tuple or list of either `Field` instances (of which `Sequence` and `Unique` are subclasses), or data -- strings (which should be column names), dicts, or tuples -- that can be passed to a `Field` constructor (e.g., `Field(*fieldTuple)`). You can use your own `Field` subclasses if you wish to store additional information about fields (e.g., data type, validators, etc.), and if you want to customize how strings, tuples, or dicts are turned into `Fields` for you, you can shadow the static factory method `PyDO.create_field()` to do so.

A `Sequence` field is used to represent either an auto-increment column, for databases like MySQL that use that mechanism, or a sequence column, as used in PostgreSQL. These columns are implicitly unique and not null. For PostgreSQL, you can explicitly declare the name of the sequence with the `sequence` parameter of the `Sequence` constructor:

```
Sequence('id', 'chimpanzee_id_seq')
```

If you do not, PyDO will infer the name of the sequence from the name of the table and field, i.e.:

```
$table_$field_seq
```

A `Unique` field is used to represent a column that has a single-column uniqueness constraint and is not null. Multiple-column not-null uniqueness constraints can also be indicated, with the `unique` class attribute:

```
from pydo import PyDO

class ArticleKeywordJunction(PyDO):
"""PyDO class for junction table between Article and Keyword"""
    connectionAlias="my_db"
    table="article_keyword_junction"
    fields=('article_id',
            'keyword_id')
    unique=(('article_id', 'keyword_id'),)
```

The `unique` attribute can be thought of as analogous to how, in SQL, you may declare uniqueness constraints in a separate clause after you have declared the fields; it is optional to do so for single-column uniqueness constraints, which are more conveniently declared inline with the field, but necessary for the multi-column case.

Note

The reason that in PyDO, declaring a uniqueness constraint implies a not null constraint, is that PyDO is only interested in unique constraints as a way of determining precisely to which row in the database a given object corresponds. A nullable uniqueness constraint is of no use to PyDO in this regard, and hence the library doesn't attempt to model it.

It is not necessary to declare any uniqueness constraints in a `PyDO` class at all, either implicitly with the `Unique` field class, or via the `unique` class attribute. However, if you do not, instances of the corresponding `PyDO` class won't be able to identify the unique rows in the database table to which they correspond, and hence the instances will not be mutable. (If the class is mutable, however, it will still be possible to perform inserts and mass updates and deletes.)

The inherited fields, uniqueness constraints, and sequences of a class may be read, but not changed, through the class methods `getFields()`, `getUniquenessConstraints()`, and `getSequences()`, respectively.

If you omit the `fields` and `unique` declarations completely and declare the class attribute `guess_columns`, PyDO will attempt to introspect into the database and build the table description itself at class creation time. The declaration only affects the class in which it is declared; classes that inherit the attribute will not themselves attempt to guess columns. By default, column guessing will require querying the database when the class is initialized for every process in which the class is imported; to mitigate this potential performance hit, the data can be cached to disk if you set `PyDO.guesscache` to an instance of `GuessCache` or a compatible object, or to `1` or `True`, in which case a default `GuessCache` will be used. `GuessCache` stores pickles associated with classes in a cache directory, by default one created with the name `$USERNAME_pydoguesscache` inside `tempfile.gettempdir()`, where `$USERNAME` is the login of the current user; if the schema of one of your objects has changed and you want to refresh the cache, simply delete the corresponding cache object and restart your application.

Finally, if you are writing a quick script and want basic, uncustomized PyDO classes for every table in a schema, the function `autoschema` will generate them for you, and return them to you in a dictionary keyed by class name:

```
globals().update(autoschema(alias='myDBAlias',
                          schema='public',
                          guesscache=True,
                          module=someModule)
```

By default, it will use the default `GuessCache`, and specify no schema; you must give an alias (which should be initialized first with `initAlias`). The `module` parameter has the same effect as the `module` parameter to `PyDO.project()`: it causes the new classes to be associated with the provided module, so they can be pickled and unpickled. While convenient for scripts, using `autoschema` gives you no way of adding methods to your PyDO objects or customizing their attributes, so isn't well suited for PyDO's main purpose, namely, crafting an application's data access layer.

4.1 Inheritance Semantics

`PyDO` classes are normal Python classes (subclassing `dict`) which use a metaclass to parse the `field` and `unique` class attribute declarations and store the derived information in private fields. Special inheritance semantics obtain for `field` and `unique`, in that the privately stored parsed values corresponding to those declarations are inherited from superclasses even if `fields` is redeclared in the subclass, shadowing any superclass's declaration. Subclasses therefore may augment the field listing of their base classes. This behavior is applicable not only to cases like PostgreSQL table inheritance, but to defining base or mixin classes (which need not be `PyDO` subclasses themselves) that define groups of fields that are shared by multiple tables.

Normally, if a subclasses redeclares a field declared by a base class, the subclass's declaration overrides that of the base class, but an exception is made for declarations that simply state the fieldname as a string; in that case, any previous, more informative declaration will be inherited.

Warning

This is generally useful (in the case of projections particularly --see below) but if you wished to override a superclass's definition, say, of `Unique('species')`, just to the non-unique `Field('species')`, you would have to explicitly use the `Field` constructor rather than simply `'species'`.

4.2 Projections

An exception is made to the default inheritance behavior -- that subclasses augment, rather than shadow, their superclasses' field listing -- for the case of projection subclasses, in which the local declaration of fields overrides that of superclasses. Projections are useful when you wish to select only a few columns from a larger table. To derive a projection from a `PyDO` class, simply call the class method `project()` on the class, passing in a tuple of fields as positional arguments (or as a single tuple/list) that you wish to include in the projection:

```
myProjection=MyBaseClass.project('id', 'title')
```

The `project()` method also accepts two keyword arguments: `mutable` and `module`, which if provided will respectively override the `mutable` attribute of the base class, and cause the projection to be stored in the namespace of the provided module, also setting the `__module__` attribute of the new class so that it can pickled and unpickled.

The return value is a subclass of `myBaseClass` with the fields `id` and `title`. This class is cached, so if you call `project()` again with the same arguments you'll get a reference to the same class.

Because of the special inheritance semantics for simple string field declarations, if `MyBaseClass` in the above example is defined as follows:

```
class myBaseClass(PyDO):
    fields=(Sequenced('id'),
            Unique('title'),
            'author'
            'ISBN',
            'first_chapter')
```

myProjection will still know that id and title are unique, and that id is sequenced.

5 Making Queries: getSome() and getUnique()

There are two class methods provided for performing SELECTs. getSome returns a list of rows of PyDO instances:

```
 >>> myFungi.getSome()
[{'id' : 1, 'species' : 'Agaricus lilaceps', 'comment' : 'nice shroom!'},
 {'id' : 2, 'species' :  'Agaricus micromegathus', 'comment' : ''}]
```

getUnique returns a single instance. You must provide enough information to getUnique to satisfy one declared uniqueness constraint; this is accomplished by passing in keyword parameters where the keywords are column names corresponding to the columns of a uniqueness constraint declared for the object, and the values are what you are asserting those columns equal for the unique row:

```
>>> myFungi.getUnique(id=2)
{'id' : 2, 'species' :  'Agaricus micromegathus', 'comment' : ''}
>>> myFungi.getUnique(id=55) is None
True
```

getSome is similar, but admits a much wider range of query options, and returns a list of PyDO instances. Assuming that comment is not a unique field above, you could not add selection criteria based on comment to getUnique(), but could to getSome:

```
>>> myFungi.getSome(comment=None)
[{'id' : 2, 'species' :  'Agaricus micromegathus', 'comment' :  ''}]
>>> myFungi.getSome(comment='better than asparagus', id=55)
[]
```

5.1 Operators

In addition to specifying selection criteria by keyword argument, PyDO gives you three other ways:

1. If you supply a string as the first argument to getSome(), it will be placed as-is in a WHERE clause. Remaining positional arguments will be taken to be values for bind variables in the string:

   ```
   >>> myFungi.getSome("comment != %s", 'favorite of frogs')
   ```

   If you use bind variables, the paramstyle you use must be the same as that of the underlying Python DBAPI driver. To support the pyformat and named paramstyles, in which variables are passed in a dictionary, you can pass in a dictionary as the second argument. When using this style with getSome(), you cannot use keyword arguments to express column equivalence.

2. You can use SQLOperator instances:

   ```
   >>> myFungi.getSome(OR(EQ(FIELD('comment'), 'has pincers'),
   ...                    LT(FIELD('id'), 40),
   ...                    LIKE(FIELD('species'), '%micromega%')))
   [{'id' : 2, 'species' :  'Agaricus micromegathus', 'comment' :  ''}]
   ```

3. You can use tuples that are turned into `SQLOperator` instances for you; this is equivalent to the above:

   ```
   >>> myFungi.getSome(('OR',
   ...                 ('=', FIELD('comment'), 'has pincers'),
   ...                 ('<', FIELD('id'), 40),
   ...                 ('LIKE', FIELD('species', '%micromega%'))))
   [{'id' : 2, 'species' :  'Agaricus micromegathus', 'comment' :  ''}]
   ```

Either operator syntax can be mixed freely with each other and with keyword arguments to express column equivalence.

The basic idea of operators is that they renotate SQL in a prefix rather than infix syntax, which may not be to everyone's taste; you don't need to use them, as they are purely syntactical sugar. One convenient thing about them is that they automatically convert values included in them to bind variables in the style of the underlying DBAPI driver.

To represent an unquoted value, like a fieldname, a constant, or a function, use the `FIELD` or `CONSTANT` classes (actually, they are synonyms). Another helper class is `SET`, for use with the `IN` operator:

```
>>> myFungi.getSome(IN(FIELD('comment'),
...                    SET('nice shroom!', 'has pincers')))
```

5.2 Order, Limit and Offset

`getSome()` accepts three additional keyword arguments:

order a fieldname to order by, with optional ' ASC' or ' DESC' suffix, or a tuple of such fieldname-with-optional-suffix strings.

offset an integer

limit an integer

5.3 Refreshing An Instance

If you have reason to believe that the data you have for an object is inaccurate or out of date, you can refresh it by calling `myObj.refresh()`, as long as the object has uniqueness constraints so it is possible to get the unique row to which it corresponds.

6 Inserts, Updates, and Deletes

To insert a new record in the database and create the corresponding `PyDO` object, use the class method `new()`:

```
>>> subscription=Subscriptions.new(email='alvin@krinst.org',
                                   magazine='NYRB')
>>> subscription
{'email' : 'alvin@krinst.org', 'magazine' : 'NYRB'}
```

If the object has a field which will acquire a default non-null value even though you haven't specified a value for it, PyDO will automatically refetch it for you if you have set `cls.refetch` to a true value:

```
>>> Sonnet.refetch
True
>>> poem=Sonnet.new(title='Anguished Parsnips',
                    body='\n'.join(' '.join(['oy veh!' * 5]) * 14))
>>> poem.created
datetime.datetime(2005, 5, 9, 11, 6, 25, 221004)
```

This is equivalent to calling `refresh()` after `new()`, and also requires that a uniqueness constraint be been declared for the class. You can also explicitly set the refetch behavior on a per-call basis by using the methods `newfetch()` and `newnofetch()`, or (for backwards compatibility with PyDO1) by using a deprecated keyword parameter, `refetch`, to `new()`:

```
>>> dud=Failure.newfetch(name="Charlie Brown")
>>> dud2=Failure.new(refetch=1, name="Oblomov")
```

Usually this isn't necessary, as whether you need to refetch is primarily determined by the characteristics of the table, but sometimes it useful -- for instance, if refetch is true class-wide, but you don't plan on doing anything with the object you are creating, it will be more efficient to use `newnofetch`.

Note

The `refetch` parameter to `new()` is deprecated because it makes it awkward to have a column named `refetch`. In PyDO2, if you have a field named "refetch", the `refetch` keyword argument to `new()` will be interpreted as field data and won't affect refetch behavior.

If you don't specify the value of a column when calling `new()`, and there is no refetch, PyDO will assume that the default value is null and store `None` for that column.

If a class is declared mutable and has a uniqueness constraint, it is possible to mutate an undeleted instance of it by calling:

```
>>> poem['title']='Sayings of the Robo-Rabbi'
```

or, equivalently, if `use_attributes` is true for the class:

```
>>> poem.title='Sayings of the Robo-Rabbi'
```

Multiple updates can be done together via `update()`:

```
MyInstance.update(dict(fieldname=newValue,
                       otherFieldname=otherValue))
```

Each mutation will cause an UPDATE statement to be executed on the underlying database. If you attempt to mutate an immutable `PyDO` instance, a `PyDOError` will be raised. A `PyDOError` will also be raised if the number of rows affected, as returned by the database driver, is not equal to 1. If the driver returns something other than 1 for a successful update in a particular case (for instance, for an updateable view), set the class attribute `_ignore_update_rowcount` to `True`.

It is also possible to update potentially many rows at once with the class method `updateSome()`:

```
>>> Article.updateSome(dict(slug="nonsense"),
...                    LT(FIELD("created"),
...                       CONSTANT("CURRENT_TIMESTAMP")),
...                    author='Smullyan')
6
```

The first argument to `updateSome()` is a dictionary of values to set for affected rows; remains positional and keyword args accept the same arguments as `getSome()` (with the exception of `order`, `limit`, and `offset` which wouldn't make sense in this context). The return value is the number of affected rows.

To delete an instance, call the instance method `delete()`:

```
>>> Article.getUnique(id=44).delete()
```

The method returns nothing; the instance in question is marked as immutable.

To delete many rows at once, use the class method `deleteSome()`:

```
>>> Article.deleteSome(LT(FIELD("created"),
...                       CONSTANT("CURRENT_TIMESTAMP")),
...                    author="Grisby Holloway")
```

The parameters accepted are again the same as for `getSome()`), except for `order`, `limit`, and `offset`, and the return value is the number of affected rows.

6.1 Python and SQL Data Types

The marshalling of SQL datatypes into Python is entirely left to the DBAPI drivers which underlie PyDO. Ideally, the reverse would also be true, but PyDO drivers are able to perform some conversion where the DBAPI drivers fail to (e.g., this is necessary to handle `mx.DateTime` in the `psycopg` driver when using `psycopg1`). PyDO also includes some typewrapper classes -- `DATE`, `TIMESTAMP`, `BINARY`, and `INTERVAL` -- which can be used for updates to coerce data to the appropriate SQL type. The main feature of these wrapper classes is that PyDO knows how to unwrap them, so that after an update the column in question will contain the wrapped value, not the wrapper instance itself.

7 Joins

7.1 Representing Joins Between Tables

To represent a one-to-one join between classes `A` and `B`, you might add instance methods to class `A`, e.g.:

```
def getB(self):
    return B.getUnique(id=self.b_id)

def setB(self, item):
    if item is None:
        self.b_id=None
    else:
        self.b_id=item.id

B=property(getB, setB)
```

PyDO provides an equivalent shortcut:

```
B=ForeignKey('b_id', 'id', B)
```

If the class `B` hasn't been defined yet, but will be defined later in the same module, you can use its name as a string:

```
B=ForeignKey('b_id', 'id', 'B')
```

And if it is defined in a different module, you can give its fully qualified name:

```
B=ForeignKey('b_id', 'id', 'somePackage.someModule.B')
```

When using a string, the actual class is looked up at runtime.

Similarly, to represent a one-to-many join, you could write your own accessor method, calling `B.getSome()`:

```
def getBs(self, *args, **kwargs):
    return B.getSome(a_id=self.id, *args, **kwargs)
```

Again, PyDO provides a terser alternative:

```
getBs=OneToMany('id', 'a_id', B)
```

The result is the same -- `getBs` will be an instance method that takes positional and keyword arguments like `getSome()`. Again, `B` can be either a class or a string representing that class.

To represent a many-to-many join between `A` and `B` through junction table `J`, you either add an instance method that calls `joinTable()`:

```
def getBs(self, *args, **kwargs):
    return self.joinTable('id', 'J', 'a_id', 'b_id',
                          B, 'id', *args, **kwargs)
```

or again use an equivalent shortcut, which is:

```
getBs=ManyToMany('id', 'J', 'a_id', 'b_id', B, 'id')
```

`joinTable()` takes the following arguments:

`thisAttrNames` attribute(s) in current object to join from

`pivotTable` pivot table name

`thisSideColumns` column(s) that correspond to the foreign key column to `thisAttrNames`.

`thatSideColumns` column(s) that correspond to the foreign key column to `thatAttrNames`.

`thatObject` the destination object (or its class name)

`thatAttrNames` attribute(s) in destination object to join to

In addition, `joinTable()` takes positional and keyword arguments, similar to `getSome()`; it will accept raw SQL and bind values or `SQLOperator` instances as positional arguments, and understands the keyword arguments `order`, `limit`, and `offset` as well as column name keyword arguments. Also, if you wish to pass in additional tables to the select, you can do so with the `extraTables` keyword argument, with which you can pass a single table name, or a list of names.

`ManyToMany` takes the same arguments as `joinTable` in the same order, except for the optional positional and keyword arguments, which can be passed when the bound method that results from using `ManyToMany` is called.

7.2 Getting Data From Multiple Tables At Once

The `fetch` function makes it possible to query multiple tables, use aggregates and obtain other non-table data, while still returning table data coalesced into `PyDO` instances. Its signature is:

```
def fetch(resultSpec, sqlTemplate, *values, **kwargs)
```

`resultSpec`, a result set specification, is a list that may contain:

- `PyDO` classes;
- 2-tuples of (`PyDO` class, alias string), which indicate that the table represented by the `PyDO` class is to be referenced in SQL by the given alias;
- strings, which represent arbitrary SQL expressions that may occur in a SQL column-list specification.

`sqlTemplate` is a string that may contain interpolation variables in the style of `string.Template`. In particular, two variables are supplied to this template automatically:

`$COLUMNS` a list of columns computed from the supplied resultSpec;

`$TABLES` a list of tables similarly computed.

Additional interpolation variables may be passed in as keyword arguments. Bind variables to the SQL may also be passed in, through positional arguments; if there is only one positional argument, and it is a dictionary, it will be used instead of a list of values, under the assumption that either the `pyformat` or `named` paramstyle is being used.

For each element *E* in the resultSpec, the result row contains one element *F*. If *E* is a `PyDO` class, *F* will either be an instance of *E*, or, if all its corresponding columns were null for that row and *E* has a uniqueness constraint (which in PyDO is implicitly a not null constraint), `None`. If *E* is a string, *F* will be whatever the cursor returned for that column.

For example:

```
>>> tmpl='''SELECT $COLUMNS FROM $TABLES WHERE art.creator=auth.id
...         AND art.id=%s'''
>>> res=fetch([(Article.project('title'), 'art'),
...             (Author.project('lastname'), 'auth'),
...             '3-2'], tmpl, 4)
(({'title': 'My Woodchuck Smarts'}, {'lastname' : 'Pydong'}, 1),)
```

8 Managing Database Connections

All that a `PyDO` class knows about its database connection is its `connectionAlias` attribute. Before you use the class, you must call `initAlias()` to associate that alias with the data needed to make an actual database connection:

```
initAlias(alias, driver, connectArgs, pool=False, verbose=False)
```

`driver` must be the name of a driver registered with PyDO; the built-in ones are currently "mysql", "psycopg", "sqlite", "mssql", and "oracle". `connectArgs` are arguments to pass to the underlying DBAPI driver's `connect()` function; you can pass a tuple of positional args, a dictionary of keyword args, or a single object that will be treated like a tuple of length 1. `pool` is an optional connection pool; if you want one, you can either pass a `ConnectionPool` instance or something with a compatible `connect()` method, or a true value, in which case a default `ConnectionPool` will be created. By default no pool is used. `verbose` is whether or not to log the generated SQL; by default no logging is done.

The class method `PyDO.getDBI()` returns a database interface object (an instance of a driver-specific `pydo.dbi.DBIBase` subclass), which in turn uses an underlying DBAPI database connection. The DBAPI connection is stored in thread-local storage and created lazily when an attempt is made to access it, so transactions in different threads will transparently use different connections. By default the connection will live as long as the current thread. If you use a pool, every time a transaction is completed, the connection will be released by the DBI object and returned to the pool. If you aren't using a pool and are using multiple threads, when the thread is finished, its connection will go out of scope and will get closed during garbage collection.

If you want to manage connections outside of PyDO, you can, by using the DBI object's `swapConnection()` method:

```
oldConnection=myDBI.swapConnection(newConnection)
# do something with PyDO
```

Because the connections are stored thread-locally, this is thread-safe. Using this technique, one could juggle multiple transactions in the same process without using multiple threads.

The dbapi module that underlies a given DBI object is available as `DBIobj.dbapiModule`; the standard dbapi-mandated exceptions defined in that module are available in a dictionary, `DBIobj.exceptions`, keyed by name.

8.1 Transactions

The DBI object's `autocommit` property reports whether the drivers uses transactions (in which case, its value is false). By default, most drivers use transactions (mysql being the outstanding exception). Some drivers support mutating this property, but as a matter of policy transactions are the norm for PyDO.

To commit a transaction, call `commit()` on the DBI object, or, equivalently, on any PyDO class or instance with the corresponding connection alias, which is equivalent to calling `obj.getDBI().commit()`. To rollback, call `rollback()`, again either on the DBI object or on a PyDO object.

Note

Although you may call `commit()` or `rollback()` via a particular class or instance, that is only for convenience and implies no particular isolation of the commit or rollback to that object. If you create six PyDO objects with the same connectionAlias and commit or rollback one of them, all are affected equally.

8.2 Connection Pools

If you are using transactions in multiple threads, a connection pool can reduce the cost of connecting to the database. The constructor has this signature:

```
pool=ConnectionPool(max_poolsize=0,
                    keep_poolsize=1,
                    delay=0.2,
                    retries=10)
```

`max_poolsize` is the maximum number of connections it will permit you to have in the pool at any one time; if 0, there is no upper limit. `keep_poolsize` is the maximum number of connections it will retain in the pool. (In other words, the pool may grow up to `max_poolsize`, but it will keep getting reduced to `keep_poolsize` when connections are released.) `delay` is the number of seconds it will delay if it needs to retry getting a connection, because the pool has reached its maximum size; `retries` is the number of times to retry before giving up and raising a `PyDOError`.

When a connection is returned to a pool, any outstanding transaction is rolled back. Committing or rolling back also causes connections to be returned to the pool, so normally nothing special needs to be done to return it or manage the pool. If transactions are not being used, however, you'll need to manually return the connection to the pool by calling `dbiObj.endConnection()`.

9 A Complete Example

Consider the following toy sqlite database:

```
CREATE TABLE contact (
  id integer not null primary key,
  first_name text,
  last_name text not null,
  address_id1 integer references address,
  address_id2 integer references address,
  email1 text,
  email2 text,
  work_phone text,
  home_phone text,
  mobile_phone text
);
```

```
CREATE TABLE address (
  id integer not null primary key,
  line1 text,
  line2 text,
  town text,
  state text,
  country text,
  postal_code text
);

CREATE TABLE note (
  id integer not null primary key,
  title text,
  body text not null,
  created timestamp
);

CREATE TABLE contact_note (
  contact_id integer references contact,
  note_id integer references note,
  primary key (contact_id, note_id)
);
```

The following Python module wraps its tables in an api:

```
"""
a small example PIM application, using sqlite.

The schema is actually rather ridiculous (why are contact -> notes
many to many?)  but you get the idea.

"""
import os
from mx.DateTime import now
from pydo import *

class Note(PyDO):
    connectionAlias="pim"
    # if created had a default value (which
    # it would if the version of sqlite this was
    # tested on supported anything like CURRENT_TIMESTAMP)
    # I'd set:
    # refetch=True
    fields=(Sequence('id'),
            'title',
            'body',
            'created')

class Address(PyDO):
    connectionAlias='pim'
    fields=(Sequence('id'),
            'line1',
            'line2',
            'town',
            'state',
            'country',
            'postal_code')

    def getContacts(self):
        return Contact.getSome(OR(EQ(FIELD('address_id1'), self.id),
                                  EQ(FIELD('address_id2'), self.id)))


class Contact(PyDO):
    connectionAlias='pim'
    fields=(Sequence('id'),
            'first_name',
            'last_name',
            'address_id1',
            'address_id2',
            'email1',
            'email2',
            'work_phone',
            'home_phone',
            'mobile_phone')

    Address1=ForeignKey('address_id1', 'id', Address)
    Address2=ForeignKey('address_id2', 'id', Address)

    def addNote(self, title, body):
        n=Note.new(title=title, body=body, created=now())
        junction=ContactNote.new(contact_id=self.id,
                                 note_id=n.id)

    getNotes=ManyToMany('id',
                        'contact_note',
                        'contact_id',
                        'note_id',
                        Note,
                        'id')


class ContactNote(PyDO):
    connectionAlias='pim'
    # table is specified here, because the class name
    # is not the same as the table name
    table="contact_note"
    fields=('contact_id',
            'note_id')
    unique=(('contact_id',
             'note_id'),)

DB=os.environ.get('PIMDB', 'pim.db')
initAlias('pim', 'sqlite', DB)
```

```
def initDB():
    joseAddress=Address.new(line1="43 Chestnut Place",
                            line2="Fourth Floor",
                            town="Princeton",
                            state="NJ",
                            country="USA",
                            postal_code="06540")
    jose=Contact.new(first_name='Jose',
                     last_name='Gutenberg',
                     email1='jgutenberg@example.com',
                     address_id1=joseAddress.id,
                     home_phone='609/555-1234')
    jose.addNote('French Tutor',
                 'meet every other Thursday at 7pm in the Annex')
    jose.commit()
```

10 Differences From PyDO 1

This version of PyDO differs in several ways from PyDO version 1.x, most notably:

1. PyDO1 defines fields as a tuple of tuples (fieldname, dbtype); type is required, as PyDO1 drivers take much of the responsibility for marshalling Python data types to database types, and does so as a function of column type. PyDO2 does not need to know about what the database type is of the underlying columns, because DBAPI drivers now largely take care of this themselves.
2. You cannot define uniqueness constraints in the field list in PyDO1.
3. Sequences and auto-increment fields are handled separately in PyDO1, and both have to be declared in separate class attributes, which have been dropped in PyDO2.
4. The `SQLOperator` syntax is now more flexible, and is accepted by `getSome()`. In PyDO1, there are three additional methods that accept different query syntaxes: `getSomeWhere()`, `getTupleWhere()`, and `getSQLWhere()`. These have been dropped. `joinTable()` now accepts this query syntax as well, so it is no longer necessary to override a protected method to add additional criteria to a join query.
5. PyDO1 is not thread-safe, and has no connection pool facilities for multi-threaded use. What connection management facilities it does have are tied in with SkunkWeb. PyDO2 is entirely separate from SkunkWeb.
6. PyDO1 uses a different package structure; the new version does not have the same submodules. However, everything you would normally need is available in the top-level namespace for both versions.
7. PyDO1 does not have projections, and the inheritance semantics, while similar, are not exactly the same.
8. PyDO1 does not use new-style classes (as it predates them), so the metaclass functionality is more elaborate, including its own implementation of class methods (which it calls "static" methods).

9. PyDO1 supports more databases than PyDO2 does at the time of writing.

10. PyDO2 does not implement PyDO1's original `scatterFetch()` method, which returns multiple `PyDO` objects of different types in a single query, but has a new function, `fetch` which has a superset of the same functionality.

11. PyDO1 has a variable `SYSDATE` that means the current datetime, regardless of the underlying db. PYDO2 does not abstract this, as it seems unnecessary now; you can use something database-dependent like `CONSTANT('CURRENT_TIMESTAMP')` or `mx.DateTime.now()`.

12. The package name of PyDO in this version is `pydo`, not `PyDO`. Both versions can be installed simultaneously without any fancy footwork on case-sensitive operating systems; on case-insensitive OSes, it is still possible to install and use both, by putting PyDO1 into a zip file, in which paths are always case-sensitive. (Thanks to Hamish Lawson for suggesting this workaround.)

13. The `newfetch()` and `newnofetch()` methods and the `refetch` class attribute of `PyDO` objects are new in PyDO2; in PyDO1, the `refetch` keyword argument to `new()` was used instead, but was broken for the unlikely case of a column named "refetch".

14. The support of schema-qualified table names and optional guessing of table name from class name is new in PyDO2.

15. The optional guessing of field information at runtime, controlled by the `guess_columns` attribute, is a new feature in PyDO2.

16. `autoschema` is new in PyDO2.

17. `ForeignKey`, `OneToMany` and `ManyToMany` are new in PyDO2.