

Database objects for Python: dbhelpers



Introduction

The Python language has a well-defined database applications program interface (DBAPI), described at:

<http://www.python.org/topics/database/>

This interface makes it possible to write database applications that can be ported easily across different database platforms such as Sybase, Informix and Oracle.

However, it is still necessary for the user program to interact with this interface using SQL (Structured Query Language). Learning SQL is nontrivial; it is a subtle and complex language.

Another problem with building applications directly on the Python DBAPI is the problem of maintaining consistency between the database definition and the application code. If field names or types or indexes or other details of the database are changed, we must be sure that the application tracks these changes. Finding all the code in an application that depends on the database structure can be tedious and error-prone.

Object-oriented design, especially the idea of encapsulation and information hiding, suggests a way out of this maintenance problem. If we encapsulate all the information about the database schema in a set of objects, that limits the amount of code we need to maintain through the inevitable evolution of the structure of the application's underlying database.

There is no actual module named `dbhelpers`; there are, instead, a series of actual modules, one for each underlying database server. At this writing, there are three implementations:

- `sy_db.py` for Sybase
- `ora_db.py` for Oracle
- `my_db.py` for MySQL

Extending these functions to cover other databases should be a trivial procedure.

Your application will not be completely unchanged when shifting from one implementation to another. You will have to change two lines:

1. Change the `import` statement to refer to the correct version for your database.
2. The exact values of the arguments to the constructor `TheDatabase` will vary across versions. For example, the argument to the `sy_db.py` and `my_db.py` versions are dictionaries, while that to `ora_db.py` is a single string.

You will have to know some SQL, especially the part used to describe field types and sizes, but most of the building of complete SQL statements and their execution will be done for you by the module.

In particular, the objects of `dbhelpers` have methods for testing whether a table or index exist, creating them, and deleting them, as well as more routine operations such as queries and updates. This solves the problem of synchronizing a Python application with an external, non-Python procedure for database creation—we suggest that you do the creation and deletion of tables with the same objects that take care of queries and updates.

Conventions used in this document

The semantics of all the constructors and methods in this module are defined according to the author's interpretation of the Cleanroom methodology as presented in Dr. Allan M. Stavely's *Toward Zero-defect programming* (Addison-Wesley, 1999, ISBN 0-201-38595-3). My formatting conventions for intended functions are described at:

<http://www.nmt.edu/~shipman/soft/clean/i-f.html>

Overview of objects in dbhelpers

These are the classes in the module:

- `class TheDatabase` represents the entire database. You need instantiate this only once in the application.
- `class Table` represents one table in the database.
- `class Column` represents one column in a table.
- `class Row` represents one row in a table.
- `class QueryResult` is a helper object that represents the result of a query.
- `class SortSpec` is a helper object for specifying sort orders in a query.
- `class TableKey` is a helper object for describing multi-column keys on a table.
- `class TableInteg` is a helper object for describing integrity constraints on a table.

In the future, objects for views may be added as well.

The TheDatabase object

Your application connects to the database server by instantiating a `TheDatabase` object. Here is the generic intended function for all implementations:

```
TheDatabase ( options ):  
[ if options specifies a valid set of options for connecting to  
  a database →  
  return a new TheDatabase object representing a connection  
  to that database ]
```

Here is the intended function for the Oracle version, `ora_db.py`:

```
[ if options is a valid connect string of the form "user/pwd@sys"  
  where user is the username, pwd is the password, and  
  sys is the system name →  
  return a new TheDatabase object representing a connection to  
  to that database ]
```

The Sybase version, `sy_db.py`:

```
[ if options is a dictionary that defines keys "user",
  "password", and "server" with appropriate values
  to access a Sybase database →
  return a new TheDatabase object representing a connection
  to that database ]
```

The MySQL version, `my_db.py`:

```
[ if options is a dictionary that defines keys "host",
  "user", "password", and "db" with appropriate
  values to access a MySQL database →
  return a new TheDatabase object representing a connection
  to that database ]
```

Note: The constructor for class `TheDatabase` is a Singleton object (as described in Erich Gamma et al, *Design Patterns*, Addison-Wesley, 1994, ISBN 0-201-63361-2). In effect, this means that the first call to the constructor instantiates a connection to the database, but all further calls to the constructor return that same instance. There is no performance penalty for multiple calls to the constructor.

Methods on the `TheDatabase` object

```
.execute ( sql ):
[ if sql is a string →
  if sql is a valid SQL command against self →
  self := self with that command executed
  return whatever the DBAPI returned
  else →
  raise DBHelpersError ]
```

The methods described below are more or less internal to the module. They are used by all the other objects to issue actual SQL to the server.

Your application may not need this method, but it is always there for SQL operations not otherwise supported. Obviously you can get into real trouble if you start issuing such SQL operations as `create table`, so use the class-provided operations whenever possible.

```
.oneRowQuery ( sql ):
[ if (sql is a string) →
  if sql is a valid SQL query that returns one row →
  return a Row object representing that result
  else →
  raise DBHelpersError ]

.multiRowQuery ( sql ):
[ if (sql is a string) →
  if sql is a valid SQL query that returns multiple rows →
  return a list of Row objects representing those results
  else →
  raise DBHelpersError ]

.cursorQuery ( sql ):
[ if (sql is a string) →
```

return a DBAPI cursor object containing the result
of executing `sql`

To allow lazy evaluation of queries, the `.cursorQuery` method instantiates a new cursor in the DBAPI, executes `sql` against it, and returns the DBAPI cursor. The caller can choose to retrieve rows via the DBAPI's `.fetchone()`, `.fetchall()`, or `.fetchmany()` methods. *The caller is responsible for closing the cursor with its `.close()` method.*

The Table object

Although the `dbhelpers` module does a lot of the low-level work of database interaction, you must still design your tables and their indices and the links between them. Objects in class `Table` encapsulate the column structure of a single table.

Here is the constructor:

```
Table ( db, name, columns, keyList=None, tabIntegList=None ):  
[ if (db is a Database object)  
  and (name is a valid table name string)  
  and (columns is a list of Column objects representing  
    one or more columns)  
  and (keyList is a list of TableKey objects representing  
    one or more keys on the table as a whole)  
  and (tabIntegList is a list of TableInteg objects or None) →  
  return a Table object representing that table ]
```

Since each `Table` object encapsulates the complete structure of one database table, you must pass in a complete definition of the table structure each time your applications runs. (Someday we might change this to operate from an external schema, but the current implementation is somewhat lightweight in this respect.)

Here are notes on the arguments to the constructor:

<code>db</code>	The <code>TheDatabase</code> object.
<code>name</code>	The name of the table as a string. The usual SQL naming rules apply to the construction of table names: letters, digits, underscore (<code>_</code>), dollar sign (<code>\$</code>), and pound sign (<code>#</code>).
<code>columns</code>	A list of one or more <code>Column</code> objects that specify the columns of the table <i>in order</i> . See the notes on the <code>Column</code> class below for more details on the description of table columns.
<code>keyList</code>	If provided, this argument must be a list of <code>TableKey</code> objects (see below) defining keys on the table that involve multiple columns. (When a single column is a key, the column's key properties are defined by the <code>Column()</code> constructor).
<code>tabIntegList</code>	If provided, this argument must be a list of <code>TableInteg</code> objects (see below) defining integrity constraints for the table as a whole.

The constructor does not actually create the table; it merely stores the table definition within itself. The various query and update methods assume that the table already exists. Your application can check for the existence of the table using the `.exists()` predicate described below.

Table creation and related methods on the Table object

Members `.db`, `.name`, `.columns`, and `.tabIntegList` are available (read-only!) and have the values of the corresponding arguments to the constructor. Here are the rest of the members and methods.

The `.create()` method causes the actual creation of a table. Creation is not automatic. You will probably want to write a Python script that creates the database. Once you have run the create script, from then on your application programs will not need to invoke the `.create()` method. We also provide methods for testing whether the table exists, deleting the table's contents (but leaving its definition), and deleting the table outright.

```
.create()
[ if self's table does not exist in self.db →
  create self's table in self.db
  else →
    raise DBHelpersError ]

.deleteContents():
[ if self's table exists in self.db →
  delete the contents of that table
  else →
    raise DBHelpersError ]

.drop():
[ if self's table exists in self.db →
  delete self's table from self.db
  else →
    raise DBHelpersError ]

.exists():
[ if self's table exists in self.db →
  return 1
  else →
    return 0 ]
```

Inserting a new row in a Table object

Next, a method for inserting new rows:

```
.insert(row):
[ if row is a Row object →
  if row can be inserted into self's table →
    self's table := self's table with row inserted
  else →
    raise DBHelpersError ]
```

For help in constructing Row objects from scratch, see the `.mapToRow()` method, described below.

To update an existing row in a table, refer to the Row object, below, under its `.update()` method.

Building a Row object: the `.mapToRow()` method

The constructor for a `Row` object (see below) requires a tuple containing all the column values for one row in a particular order: the order of the `.columns` argument to the `Table()` constructor.

It is often much more convenient for the application to refer to columns by name. Accordingly, the `Table` object supports this method for building a `Row` object from a dictionary whose keys are column names and whose values are column values:

```
.mapToRow ( rowMap ):  
[ if rowMap is a dictionary mapping  $N \mapsto V$  where each  
   $N_i$  is the name of a column in self and each corresponding  $V_i$   
  is a value appropriate for that column, and all required columns  
  are defined in rowMap →  
  if self has at least one column that is an index or one key on  
  the table as a whole →  
  return a Row object representing the  $(N_i, V_i)$  pairs from  
  rowMap, with any missing columns set to their default value  
  else →  
  raise KeyError ]
```

Query methods in the `Table` object

There are three query methods. The simplest, `.queryExact()`, is used to retrieve a unique row, and it returns a `Row` object directly.

```
.queryExact ( colPairs ):  
[ if colPairs is a sequence of two-element sequences  $(n_i, v_i)$   
  such that each  $n_i$  is the name of a column in self and each  
  corresponding  $v_i$  is a value appropriate for that column →  
  if there is exactly one row in self, each of whose columns named  $n_i$   
  has the value  $v_i$  →  
  return that row as a Row object  
  else →  
  return None ]
```

The other two query methods work differently. The `.queryWild()` method is used to do wildcard queries, and `.queryAll()` is used when you want all the rows of a table. Since they may potentially return large numbers of rows, rather than returning a list of `Row` objects that might take a lot of memory, instead they return a `QueryResult` object (see below) that has a method for returning the values one at a time. This allows serial processing by the application.

The `.queryWild()` and `.queryAll()` methods are different in another important way from `.queryExact()`: they permit you to specify the order of the rows returned.

```
.queryWild ( columnName, prefix, sortList=None )  
[ if (columnName is the name of a string-valued column in self)  
  and (prefix is a string)  
  and (sortList is a list of SortSpec objects or None) →  
  return a QueryResult object representing the set of rows in self  
  whose values in column columnName start with prefix,  
  ordered according to sortList if given, otherwise random order ]
```

Only string-valued columns may be queried by `.queryWild()`.

For example, suppose you have a table named `phonebook` and it has a field whose name is `"lastname"`. This query:

```
phonebook.queryWild ( "lastname", "Smit" )
```

might return a query object containing rows in which that field has the values `"Smith"`, `"Smithee"`, `"Smithers"`, and so on.

If you need the results in a specific order, the `sortList` argument may contain a list of `SortSpec` objects that enumerate the key or keys for sorting the result. See below for the `SortSpec` object. If no `sortList` is given, the rows are returned in no particular order.

To retrieve all rows from a table:

```
.queryAll ( sortList=None ):
[ if sortList is None or a list of SortSpec objects →
  return a QueryResult object representing all the rows in self,
  ordered according to sortList if given, otherwise in random
  order ]
```

Again, a `QueryResult` object is returned; see below for a discussion of this object.

To retrieve some rows from a table:

```
.querySome ( colPairs, sortList=None ):
[ if (colPairs is a sequence of two-element sequences (ni, vi)
  such that each ni is the name of a column in self and each
  corresponding vi is a value appropriate for that column)
  and (sortList is None or a list of SortSpec objects →
  return a QueryResult object representing all the rows in self
  for which all columns ni have the value vi, ordered according
  to sortList if given, otherwise in random order ]
```

The QueryResult helper object

The object returned by the `.queryWild()` and `.queryAll()` methods on `Table` objects has only one method:

```
.next() :
[ if self contains any more rows →
  self := self minus the next row
  return the next row as a Row object
else →
  return None ]
```

The SortSpec helper object

If you want the results of a multi-row query sorted in a particular way, build a list of `SortSpec` objects representing the columns to be sorted (from major to minor order). Here is the constructor:

```
SortSpec ( columnName, desc=0 ):
[ if (columnName is a string) →
  return a new SortSpec object with column name columnName
  and specifying descending order if desc is true, else ascending
  order ]
```

For example, suppose you have a table object named `dancers` two columns in a database named `elan` and `panache`, and you want all the rows from the table, sorted primarily by ascending `elan`, but by descending `panache` for rows whose `elan` is the same. This query would do the job:

```
dancers.queryAll ( [ SortSpec ( "panache" ),
                    SortSpec ( "elan", desc=1 ) ] )
```

Unload and reload mechanism

Sometimes it is desirable to write the entire contents of a table to a text file—for backup, or for export to other environments, for example.

Also, suppose we want to change the structure of the database—to make a field larger, for example. There is no built-in way of doing that. The recommended procedure for table structure changes is:

1. Unload all the tables to text files.
2. Use the `.drop()` method to destroy the database copies of the tables.
3. Modify the table structure in the application objects built on top of `dbhelpers`, and recreate the tables.
4. Modify the text files to conform to the new format.
5. Reload the database tables from the modified text files.

In order to implement this structure, we need methods for unloading and reloading the table. These methods are called `.unload()` and `.reload()`.

To use database unloading and reloading, you must define your own class that inherits from `class Table`, and define two virtual methods called `.flatten()` (for “flattening” a row, that is, converting it to a string) and `.inflate()` (for “inflating” a string, that is, converting it back to a row). These virtual methods are used by the unload and reload methods.

```
.unload ( fileName ) :
[ if fileName is a string →
  if fileName names a file that can be created new →
    that file := self's table contents as a flat file in the same
    format expected by .reload()
  else → raise DBHelpersError ]

.reload ( fileName ) :
[ if fileName is a string →
  if (fileName names a readable, valid flat file) →
    self := self with the table reloaded from that file
  else → raise DBHelpersError ]
```

Here are intended functions for the two virtual methods:

```
.flatten(row) :
[ if row is a Row object for self's table →
  return a string representing row as a string ]

.inflate(s) :
[ if s is a string →
  if s is a valid string representation of a row in self →
    return a Row object in self
  else → raise DBHelpersError ]
```

The TableKey object

When there are keys on the table that involve more than one column, the `TableKey` helper object is used to define each such key.

Here is the constructor:

```
TableKey ( keyType, collist ):  
[ if (keyType is UNIQUE or PRIMARY)  
  and (collist is a list of field names) →  
  return a new TableKey object with key type keyType  
  and using the fields named in collist in order as keys ]
```

The names `UNIQUE` and `PRIMARY` are defined in the module. A table may have any number of unique keys, and either zero or one primary key.

The members `.keyType` and `.collist` are available, read-only, and contain the values passed to the constructor.

The TableInteg object

For integrity constraints on a table as a whole, the `TableInteg` helper object is used to describe columns in the table that must be defined by rows in a foreign table.

For example, a table of inventory locations contains a part number column. The part numbers are defined in the “part kinds” table. We always want to be able to look up part numbers in the inventory table and get a description of the part from the part kinds table—if a part number in the inventory table isn’t in the part kinds table, that’s an error.

So a table integrity constraint describes the foreign table and a list of one or more fields in the two tables that must correspond.

Here is the constructor:

```
TableInteg ( foreignTable, pairList ):  
[ if (foreignTable is a Table object)  
  and (pairList is a list of two-element sequences ( $l_i, f_i$ )  
  where each  $l_i$  is a string and each  $f_i$  is a string) →  
  return a new TableInteg object representing a table integrity  
  constraint with foreignTable as the foreign table and  
  each local field  $l_i$  must be defined by field  $f_i$  in the  
  foreign table ]
```

The elements `.foreignTable` and `.pairList` are available, read-only, and equal to the arguments to the constructor.

The Column *object*

Each object of class `Column` defines one column (field) in a table. Constructor:

```
Column ( name, kind, allowNulls=0, keyType=None, colIntegs=None ):  
[ if (name is a valid name string)  
  and (kind is a valid type string)  
  and (allowNulls is true if null values are allowed, else false)  
  and (keyType is None, UNIQUE, or PRIMARY)  
  and (colIntegs is None or a tuple (tf, cf))  
  return a Column object representing those values ]
```

The `name` argument is the name that will be given to the column. The `kind` argument is a string describing the type of the column using SQL syntax.

If `allowNulls=1` is supplied, null values will be allowed in that column. The default is that nulls are disallowed.

The `keyType`, if given, must be `UNIQUE` (for a unique key) or `PRIMARY` (for a primary key). The default is that the column is not a key.

The `colIntegs` argument is used if there is a database integrity constraint on this column, that is, if every value that appears in this column must exist in some column in a foreign table. The `tf` argument is a `Table` object specifying the foreign table, and the `cf` argument is the name of the column in that table on which this column depends.

The members `.name`, `.kind`, `.allowNulls`, `.keyType`, and `.colInteg` are available, read-only, and contain the values passed to the `Column()` constructor.

Here are some examples of calls to the constructor to give you the general idea:

```
F_X_CODE = "code"           # Name of a field in a foreign table  
xTable = Table ( ... )     # Define the foreign table  
F_DESC = "desc"           # Name of a field in our table  
L_DESC = 30                # Length of that field  
F_IS_TEST = "is_test"     # More field names and lengths  
F_CODE = "code"  
L_CODE = 4  
descCol = Column ( F_DESC, "varchar(%d)" % L_DESC, keyType=PRIMARY )  
isTest = Column ( F_IS_TEST, "tinyint" )  
code = Column ( F_CODE, "varchar(%d)" % L_CODE,  
               colIntegs=(xTable, F_X_CODE))
```

Note the use of manifest constants for field names and lengths. It is good practice to define such entities once and use them everywhere. After formatting, the type strings in the last three examples will amount to:

```
varchar(30) primary key  
tinyint  
varchar(4) references code_table(code)
```

For a complete list of the basic column types, refer to *Appendix A*, under the definition of *datatype*.

The Row object

The purpose of the Row object is to hold rows from tables, either as the result of a retrieval operation, or as an argument passed to an insert or update operation.

The constructor:

```
Row ( table, rowTuple ):  
[ if (table is a Table object  
  and (rowTuple is a tuple containing values for each field in table,  
  in the same order as the columns argument to the Table() constructor) →  
  return a new Row object representing the values from rowTuple ]
```

If it is inconvenient for you to build the rowTuple, see the .mapToRow() method on the Table object, described above.

Members and methods of the Row object

```
.get (name) :  
[ if (name is the name of a column in self's table →  
  return the value of that column of self ]
```

The .get() method retrieves one column from a row by name.

```
.update ( colMap ):  
[ if (colMap is a dictionary mapping  $N \mapsto V$  where each  
   $N_i$  is the name of a column in self's table and each corresponding  
   $V_i$  is a value appropriate for that column's type →  
  if (self's table has at least one unique key) →  
  self's table := self's table with the values of fields  $N$  in  
  self's row replaced by the corresponding values from  $V$  ]  
else →raise DBHelpersError
```

To change the values in a row, use the .update() method on the row. The argument is a dictionary in which each key is a row name and the corresponding value is the new value for that row.

For example, suppose you have a Row object named invRec, and you want to set the value of the field named F_CODE to "foo" and the value of the field name F_STATUS to 77. This method call would do the job:

```
invRec.update ( {F_CODE: "foo", F_STATUS: 77} )
```

Finally, we need a way to delete rows from a table:

```
.delete ( )  
[ if self's table has at least one column or table key →  
  self's table := self's table with self deleted  
else →  
  raise DBHelpersError
```

This method deletes from the database the row corresponding to a Row object. The table must have at least one unique key (whether on a column or on the table as a whole).

Structuring your application

We assume that you will build a module for all the user-level database functions required by your application. Here is a suggested outline for this module:

- Start by importing the version of `dbhelpers` for your database. For example:

```
from syhelpers import *
```

- Following this, declare global manifest constants for the names of tables and the names and lengths of fields. For example, I like to use names starting with `F_` for field names and names starting with `L_` for field lengths. If you have a four-character column called `code`, you might define that column's name and length with the statements

```
F_CODE = "code"  
L_CODE = 4
```

- You may want to define a class for the database as a whole. This class would be instantiated once at the beginning of your application. In it are methods for all the various retrieval and update functions in it.

One advantage to defining a unified class for the database is that it can define a `.create()` method that takes care of properly sequencing the creation of the component tables and indexes.

- Next, define classes for important entities represented by tables, rows in those tables, and indices. Each table object and index must provide a `.create()` method that creates the corresponding actual table or index.

Appendix A: SQL subset used

Here is the subset of Structured Query Language that will be generated by this package in general. Variants for particular database servers may use extensions for that language, of course.

```
sql-stmt      ::= create-tbl-stmt
              | select-stmt
              | update-stmt
              | delete-stmt

create-tbl-stmt ::= "create table" table-name "(" create-body ")"
table-name      ::= name
name            ::= name-start-char [ name-char ... ]
name-start-char ::= letter | "_"
name-char       ::= name-start-char | "$" | "#"

create-body     ::= column-desc ... [ constraint-desc ... ]
column-desc     ::= col-name datatype col-modifier
col-name        ::= name
datatype        ::= fixed-datatype
                  | v1-datatype "(" integer ")"
                  | v2-datatype "(" integer [ "," integer ] ")"
fixed-datatype  ::= "int" | "integer" | "smallint" | "tinyint"
                  | "real" | "double precision"
v1-datatype     ::= "char" | "varchar" | "text" | "image"
                  | "binary" | "varbinary"
v2-datatype     ::= "decimal" | "numeric"

col-modifier    ::= [ default-clause ] [ null-clause ] [ key-clause ] [ ref-clause ]
default-clause ::= "default" { const-expr | "null" }
null-clause    ::= "identity" | "null" | "not null"
key-clause     ::= "unique" | "primary key"
ref-clause     ::= "references" x-table-name "(" x-col-name ")"
x-table-name   ::= name
x-col-name     ::= name

constraint-desc ::= "constraint" c-name { key-constraint | ref-constraint }
c-name         ::= name
key-constraint ::= { "unique" | "primary key" } "(" col-list ")"
col-list       ::= col-name [ "," col-name ... ]
ref-constraint ::= "foreign key" our-keys "references" their-keys
our-keys       ::= "(" col-list ")"
their-keys     ::= x-table-name "(" col-list ")"

select-stmt     ::= "select * from" table-name [ sel-cond ] [ sort-clause ]
sel-cond        ::= "where" { equal-select | wild-select }
equal-select    ::= { col-name "=" value } [ "and" ... ]
wild-select     ::= col-name "like" "prefix%"
sort-clause     ::= "order by" { col-name [ "desc" ] } [, ...]
```

```
update-stmt ::= "update" table-name set-clause where-update  
set-clause ::= "set" { col-name "=" value } [, ...]  
where-update ::= "where" { col-name "=" value } [, ...]  
  
delete-stmt ::= "delete" table-name "where" equal-select
```

Written by John W. Shipman (tcc-doc@nmt.edu). This version printed 2000-09-24.