pyfirebirdsql documentation

Release 1.0.0

Hajime Nakagami

Sep 03, 2017

Contents

pyfirebirdsql is a Python extension package that implements *Python Database API 2.0*-compliant support for the open source relational database Firebird® . In addition to the minimal feature set of the standard Python DB API, pyfirebirdsql also exposes nearly the entire native client API of the database engine.

pyfirebirdsql is free – covered by a permissive BSD-style license that both commercial and noncommercial users should find agreeable.

This documentation set is not a tutorial on Python, SQL, or Firebird; rather, it is a topical presentation of pyfirebirdsql's feature set, with example code to demonstrate basic usage patterns. For detailed information about Firebird features, see the Firebird documentation, and especially the excellent The Firebird Book written by Helen Borrie and published by APress.

CHAPTER 1

Documentation Contents:

pyfirebirdsql Installation Guide

Dependencies

pyfirebirdsql requires a valid combination of the dependencies in the list below.

Detailed instructions on how to install each dependency are beyond the scope of this document; consult the dependency distributor for installation instructions.

Satisfying the dependencies is not difficult! For mainstream operating systems – including Windows , Macosx and Linux – easily installable binary distributions are available for *all* of pyfirebirdsql's dependencies (see the download links below).

1. Operating System - one of:
    - Windows (32/64)
    - Linux Any Cpu
    - FreeBSD
    - Macosx
    - Other Unix or Unix-like operating system
2. Firebird 2.1 or later server version installation [download here] (Firebird client is not necessary to connect to a server)
3. Python [download here] 2.6 or later (including Python 3.x) It was tested with cpython , ironpython and pypy

How to install

- *Install by pip*
- Install from *FreeBSD ports collection*

- Install from *source distribution*

Install by pip

```
pip instal firebirdsql
```

Installation from FreeBSD ports collection

FreeBSD has it's port now:

```
# cd /usr/ports/databases/py-firebirdsql/
# make install clean
```

Installation from source distribution

Shortcut for the Experienced and Impatient:

```
(decompress pyfirebirdsql into *temp_dir*)
cd *temp_dir*
python setup.py install
python -c "import firebirdsql"
(delete *temp_dir*)
```

Then hit the Usage Guide.

Test your pyfirebirdsql installation

pyfirebirdsql has an extensive test suite, but it is not really intended for routine public use.

To verify that pyfirebirdsql is installed properly, switch to a directory *other than the temporary directory into which you decompressed the source distribution* (to avoid conflict between the copy of firebirdsql in that directory and the copy placed under the standard Python *site-packages* directory), then verify the importability of your pyfirebirdsql installation by issuing the following command:

```
python -c "import firebirdsql as fb; print fb.__version__"
```

If the import attempt does not encounter any errors and the version number is what you expected, you are finished. Next, consider reading the pyfirebirdsql Usage Guide.

You should not encounter any errors at this stage since you have already completed the installation steps successfully. If you do, please report them to the firebird-python support list.

Quick-start Guide / Tutorial

This brief tutorial aims to get the reader started by demonstrating elementary usage of pyfirebirdsql. It is not a comprehensive Python Database API tutorial, nor is it comprehensive in its coverage of anything else.

The numerous advanced features of pyfirebirdsql are covered in another section of this documentation, which is not in a tutorial format, though it is replete with examples.

Connecting to a Database

Example 1

A database connection is typically established with code such as this:

```
import firebirdsql

# The server is named 'bison'; the database file is at '/temp/test.fdb'.
con = firebirdsql.connect(dsn='bison:/temp/test.fdb', user='sysdba', password='pass')

# Or, equivalently:
con = firebirdsql.connect(
    host='bison', database='/temp/test.fdb',
    user='sysdba', password='pass'
  )
```

Example 2

Suppose we want to connect to the database in SQL Dialect 1 and specifying UTF-8 as the character set of the connection:

```
import firebirdsql

con = firebirdsql.connect(
    dsn='bison:/temp/test.fdb',
    user='sysdba', password='pass',
    charset='UTF8' # specify a character set for the connection
  )
```

Executing SQL Statements

For this section, suppose we have a table defined and populated by the following SQL code:

```
create table languages
(
  name               varchar(20),
  year_released      integer
);

insert into languages (name, year_released) values ('C',         1972);
insert into languages (name, year_released) values ('Python',    1991);
```

Example 1

This example shows the *simplest* way to print the entire contents of the *languages* table:

```
import firebirdsql

con = firebirdsql.connect(dsn='/temp/test.fdb', user='sysdba', password='masterkey')

# Create a Cursor object that operates in the context of Connection con:
cur = con.cursor()

# Execute the SELECT statement:
cur.execute("select * from languages order by year_released")
```

```
# Retrieve all rows as a sequence and print that sequence:
print cur.fetchall()
```

Sample output:

```
[('C', 1972), ('Python', 1991)]
```

Example 2

Here's another trivial example that demonstrates various ways of fetching a single row at a time from a *SELECT*-cursor:

```
import firebirdsql

con = firebirdsql.connect(dsn='/temp/test.fdb', user='sysdba', password='masterkey')

cur = con.cursor()
SELECT = "select name, year_released from languages order by year_released"

# 1. Iterate over the rows available from the cursor, unpacking the
# resulting sequences to yield their elements (name, year_released):
cur.execute(SELECT)
for (name, year_released) in cur:
    print '%s has been publicly available since %d.' % (name, year_released)

# 2. Equivalently:
cur.execute(SELECT)
for row in cur:
    print '%s has been publicly available since %d.' % (row[0], row[1])
```

Sample output:

```
C has been publicly available since 1972.
Python has been publicly available since 1991.
C has been publicly available since 1972.
Python has been publicly available since 1991.
```

Example 3

The following program is a simplistic table printer (applied in this example to *languages*):

```
import firebirdsql as fb

TABLE_NAME = 'languages'
SELECT = 'select * from %s order by year_released' % TABLE_NAME

con = fb.connect(dsn='/temp/test.fdb', user='sysdba', password='masterkey')

cur = con.cursor()
cur.execute(SELECT)

# Print a header.
for fieldDesc in cur.description:
    # Description name
    print fieldDesc[0] ,
print # Finish the header with a newline.
print '-' * 78
```

```
# For each row, print the value of each field left-justified within
# the maximum possible width of that field.
fieldIndices = range(len(cur.description))
for row in cur:
    for fieldIndex in fieldIndices:
        fieldValue = str(row[fieldIndex])
        #DESCRIPTION_DISPLAY_SIZE
        fieldMaxWidth = cur.description[fieldIndex][2]

        print fieldValue.ljust(fieldMaxWidth) ,

    print # Finish the row with a newline.
```

Sample output:

```
NAME                 YEAR_RELEASED
--------------------------------------------------------------------------------
C                    1972
Python               1991
```

Example 4

Let's insert more languages:

```
import firebirdsql

con = firebirdsql.connect(dsn='/temp/test.fdb', user='sysdba', password='masterkey')

cur = con.cursor()

newLanguages = [
    ('Lisp',  1958),
    ('Dylan', 1995),
  ]

cur.executemany("insert into languages (name, year_released) values (?, ?)",
    newLanguages
  )

# The changes will not be saved unless the transaction is committed explicitly:
con.commit()
```

Note the use of a *parameterized* SQL statement above. When dealing with repetitive statements, this is much faster and less error-prone than assembling each SQL statement manually. (You can read more about parameterized SQL statements in the section on *Prepared Statements*.)

After running Example 4, the table printer from Example 3 would print:

```
NAME                 YEAR_RELEASED
--------------------------------------------------------------------------------
Lisp                 1958
C                    1972
Python               1991
Dylan                1995
```

Calling Stored Procedures

Firebird supports stored procedures written in a proprietary procedural SQL language. Firebird stored procedures can have *input* parameters and/or *output* parameters. Some databases support *input/output* parameters, where the same parameter is used for both input and output; Firebird does not support this.

It is important to distinguish between procedures that *return a result set* and procedures that *populate and return their output parameters exactly once*. Conceptually, the latter "return their output parameters" like a Python function, whereas the former "yield result rows" like a Python generator.

Firebird's *server-side* procedural SQL syntax makes no such distinction, but *client-side* SQL code (and C API code) must. A result set is retrieved from a stored procedure by *SELECT'ing from the procedure, whereas output parameters are retrieved with an 'EXECUTE PROCEDURE* statement.

To *retrieve a result set* from a stored procedure with pyfirebirdsql, use code such as this:

```
cur.execute("select output1, output2 from the_proc(?, ?)", (input1, input2))

# Ordinary fetch code here, such as:
for row in cur:
    ... # process row

con.commit() # If the procedure had any side effects, commit them.
```

To *execute* a stored procedure and *access its output parameters*, use code such as this:

```
cur.callproc("the_proc", (input1, input2))

# If there are output parameters, retrieve them as though they were the
# first row of a result set.  For example:
outputParams = cur.fetchone()

con.commit() # If the procedure had any side effects, commit them.
```

This latter is not very elegant; it would be preferable to access the procedure's output parameters as the return value of *Cursor.callproc()*. The Python DB API specification requires the current behavior, however.

Python Database API Specification 2.0

pyfirebirdsql is the Python Database API 2.0 compliant driver for Firebird. The *Reference / Usage Guide* is therefore divided into three parts:

- Python Database API 2.0 specification
- pyfirebirdsql Compliance to Python DB 2.0 API specification.
- pyfirebirdsql features beyond Python DB 2.0 API specification.

If you're familiar to Python DB 2.0 API specification, you may skip directly to the next topic.

Note: This is a local copy of the specification. The online source copy is available at http://www.python.org/topics/database/DatabaseAPI-2.0.html

---

Introduction

This API has been defined to encourage similarity between the Python modules that are used to access databases. By doing this, we hope to achieve a consistency leading to more easily understood modules, code that is generally more portable across databases, and a broader reach of database connectivity from Python.

The interface specification consists of several sections:

- Module Interface
- Connection Objects
- Cursor Objects
- Type Objects and Constructors
- Implementation Hints
- Major Changes from 1.0 to 2.0

Comments and questions about this specification may be directed to the SIG for Database Interfacing with Python.

For more information on database interfacing with Python and available packages see the Database Topics Guide on www.python.org.

This document describes the Python Database API Specification 2.0. The previous version 1.0 version is still available as reference. Package writers are encouraged to use this version of the specification as basis for new interfaces.

Module Interface

Access to the database is made available through connection objects. The module must provide the following constructor for these:

connect(*parameters...*)
Constructor for creating a connection to the database. Returns a Connection Object . It takes a number of parameters which are database dependent.[1]

These module globals must be defined:

apilevel
String constant stating the supported DB API level. Currently only the strings *'1.0'* and *'2.0'* are allowed. If not given, a Database API 1.0 level interface should be assumed.

threadsafety
Integer constant stating the level of thread safety the interface supports. Possible values are:

- *0* = Threads may not share the module.
- *1* = Threads may share the module, but not connections.
- *2* = Threads may share the module and connections.
- *3* = Threads may share the module, connections and cursors. Sharing in the above context means that two threads may use a resource without wrapping it using a mutex semaphore to implement resource locking.

Note that you cannot always make external resources thread safe by managing access using a mutex: the resource may rely on global variables or other external sources that are beyond your control.

[1] As a guideline the connection constructor parameters should be implemented as keyword parameters for more intuitive use and follow this order of parameters: *dsn* = Data source name as string *user* = User name as string (optional) *password* = Password as string (optional) *host* = Hostname (optional) *database* = Database name (optional) E.g. a connect could look like this: *connect(dsn='myhost:MYDB',user='guido',password='234$?')*

paramstyle
String constant stating the type of parameter marker formatting expected by the interface. Possible values are[2]:

- *'qmark'* = Question mark style, e.g. '...WHERE name=?'
- *'numeric'* = Numeric, positional style, e.g. '...WHERE name=:1'
- *'named'* = Named style, e.g. '...WHERE name=:name'
- *'format'* = ANSI C printf format codes, e.g. '...WHERE name=%s'
- *'pyformat'* = Python extended format codes, e.g. '...WHERE name=%(name)s'

The module should make all error information available through these exceptions or subclasses thereof:

exception Warning
Exception raised for important warnings like data truncations while inserting, etc. It must be a subclass of the Python StandardError (defined in the module exceptions).

exception Error
Exception that is the base class of all other error exceptions. You can use this to catch all errors with one single 'except' statement. Warnings are not considered errors and thus should not use this class as base. It must be a subclass of the Python StandardError (defined in the module exceptions).

exception InterfaceError
Exception raised for errors that are related to the database interface rather than the database itself. It must be a subclass of Error.

exception DatabaseError
Exception raised for errors that are related to the database. It must be a subclass of Error.

exception DataError
Exception raised for errors that are due to problems with the processed data like division by zero, numeric value out of range, etc. It must be a subclass of DatabaseError.

exception OperationalError
Exception raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, a memory allocation error occurred during processing, etc. It must be a subclass of DatabaseError.

exception IntegrityError
Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails. It must be a subclass of DatabaseError.

exception InternalError
Exception raised when the database encounters an internal error, e.g. the cursor is not valid anymore, the transaction is out of sync, etc. It must be a subclass of DatabaseError.

exception ProgrammingError
Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc. It must be a subclass of DatabaseError.

exception NotSupportedError
Exception raised in case a method or database API was used which is not supported by the database, e.g. requesting a .rollback() on a connection that does not support transaction or has transactions turned off. It must be a subclass of DatabaseError.

This is the exception inheritance layout:

[2] Module implementors should prefer 'numeric', 'named' or 'pyformat' over the other formats because these offer more clarity and flexibility.

```
  StandardError
  |__Warning
  |__Error
     |__InterfaceError
     |__DatabaseError
        |__DataError
        |__OperationalError
        |__IntegrityError
        |__InternalError
        |__ProgrammingError
        |__NotSupportedError

Note: The values of these exceptions are not defined. They should give
the user a fairly good idea of what went wrong though.
```

Connection Objects

Connections Objects should respond to the following methods:

class Connection

close()
Close the connection now (rather than whenever __del__ is called). The connection will be unusable from this point forward; an *Error* (or subclass) exception will be raised if any operation is attempted with the connection. The same applies to all cursor objects trying to use the connection.

commit()
Commit any pending transaction to the database. Note that if the database supports an auto-commit feature, this must be initially off. An interface method may be provided to turn it back on. Database modules that do not support transactions should implement this method with void functionality.

rollback()
This method is optional since not all databases provide transaction support.[3] In case a database does provide transactions this method causes the the database to roll back to the start of any pending transaction. Closing a connection without committing the changes first will cause an implicit rollback to be performed.

cursor()
Return a new Cursor Object using the connection. If the database does not provide a direct cursor concept, the module will have to emulate cursors using other means to the extent needed by this specification.[4]

Cursor Objects

These objects represent a database cursor, which is used to manage the context of a fetch operation. Cursor Objects should respond to the following methods and attributes:

class Cursor

[3] If the database does not support the functionality required by the method, the interface should throw an exception in case the method is used. The preferred approach is to not implement the method and thus have Python generate an *AttributeError* in case the method is requested. This allows the programmer to check for database capabilities using the standard *hasattr()* function. For some dynamically configured interfaces it may not be appropriate to require dynamically making the method available. These interfaces should then raise a *NotSupportedError* to indicate the non-ability to perform the roll back when the method is invoked.

[4] A database interface may choose to support named cursors by allowing a string argument to the method. This feature is not part of the specification, since it complicates semantics of the *.fetchXXX()* methods.

description

This read-only attribute is a sequence of 7-item sequences. Each of these sequences contains information describing one result column: *(name, type_code, display_size, internal_size, precision, scale, null_ok)*. This attribute will be *None* for operations that do not return rows or if the cursor has not had an operation invoked via the *executeXXX()* method yet. The *type_code* can be interpreted by comparing it to the Type Objects specified in the section below.

rowcount

This read-only attribute specifies the number of rows that the last *executeXXX()* produced (for DQL statements like select) or affected (for DML statements like update or insert). The attribute is -1 in case no *executeXXX()* has been performed on the cursor or the rowcount of the last operation is not determinable by the interface.[7]

callproc(*procname*[, *parameters*])

This method is optional since not all databases provide stored procedures.[3] Call a stored database procedure with the given name. The sequence of parameters must contain one entry for each argument that the procedure expects. The result of the call is returned as modified copy of the input sequence. Input parameters are left untouched, output and input/output parameters replaced with possibly new values. The procedure may also provide a result set as output. This must then be made available through the standard *fetchXXX()* methods.

close()

Close the cursor now (rather than whenever __del__ is called). The cursor will be unusable from this point forward; an *Error* (or subclass) exception will be raised if any operation is attempted with the cursor.

execute(*operation*[, *parameters*])

Prepare and execute a database operation (query or command). Parameters may be provided as sequence or mapping and will be bound to variables in the operation. Variables are specified in a database-specific notation (see the module's *paramstyle* attribute for details).[5] A reference to the operation will be retained by the cursor. If the same operation object is passed in again, then the cursor can optimize its behavior. This is most effective for algorithms where the same operation is used, but different parameters are bound to it (many times). For maximum efficiency when reusing an operation, it is best to use the setinputsizes() method to specify the parameter types and sizes ahead of time. It is legal for a parameter to not match the predefined information; the implementation should compensate, possibly with a loss of efficiency. The parameters may also be specified as list of tuples to e.g. insert multiple rows in a single operation, but this kind of usage is depreciated: *executemany()* should be used instead. Return values are not defined.

executemany(*operation*, *seq_of_parameters*)

Prepare a database operation (query or command) and then execute it against all parameter sequences or mappings found in the sequence *seq_of_parameters*. Modules are free to implement this method using multiple calls to the *execute()* method or by using array operations to have the database process the sequence as a whole in one call. The same comments as for *execute()* also apply accordingly to this method. Return values are not defined.

fetchone()

Fetch the next row of a query result set, returning a single sequence, or *None* when no more data is available.[6] An *Error* (or subclass) exception is raised if the previous call to *executeXXX()* did not produce any result set or no call was issued yet.

fetchmany([*size=cursor.arraysize*])

[7] The *rowcount* attribute may be coded in a way that updates its value dynamically. This can be useful for databases that return useable rowcount values only after the first call to a *.fetchXXX()* method.

[5] The module will use the __getitem__ method of the parameters object to map either positions (integers) or names (strings) to parameter values. This allows for both sequences and mappings to be used as input. The term "bound" refers to the process of binding an input value to a database execution buffer. In practical terms, this means that the input value is directly used as a value in the operation. The client should not be required to "escape" the value so that it can be used – the value should be equal to the actual database value.

[6] Note that the interface may implement row fetching using arrays and other optimizations. It is not guaranteed that a call to this method will only move the associated cursor forward by one row.

Fetch the next set of rows of a query result, returning a sequence of sequences (e.g. a list of tuples). An empty sequence is returned when no more rows are available. The number of rows to fetch per call is specified by the parameter. If it is not given, the cursor's *arraysize* determines the number of rows to be fetched. The method should try to fetch as many rows as indicated by the size parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned. An *Error* (or subclass) exception is raised if the previous call to *executeXXX()* did not produce any result set or no call was issued yet. Note there are performance considerations involved with the size parameter. For optimal performance, it is usually best to use the arraysize attribute. If the size parameter is used, then it is best for it to retain the same value from one *fetchmany()* call to the next.

fetchall()
Fetch all (remaining) rows of a query result, returning them as a sequence of sequences (e.g. a list of tuples). Note that the cursor's *arraysize* attribute can affect the performance of this operation. An *Error* (or subclass) exception is raised if the previous call to *executeXXX()* did not produce any result set or no call was issued yet.

nextset()
This method is optional since not all databases support multiple result sets.[3] This method will make the cursor skip to the next available set, discarding any remaining rows from the current set. If there are no more sets, the method returns *None*. Otherwise, it returns a true value and subsequent calls to the fetch methods will return rows from the next result set. An *Error* (or subclass) exception is raised if the previous call to *executeXXX()* did not produce any result set or no call was issued yet.

setinputsizes(*sizes*)
This can be used before a call to *executeXXX()* to predefine memory areas for the operation's parameters. *sizes* is specified as a sequence – one item for each input parameter. The item should be a Type Object that corresponds to the input that will be used, or it should be an integer specifying the maximum length of a string parameter. If the item is *None*, then no predefined memory area will be reserved for that column (this is useful to avoid predefined areas for large inputs). This method would be used before the *executeXXX()* method is invoked. Implementations are free to have this method do nothing and users are free to not use it.

setoutputsize(*size*[, *column*])
Set a column buffer size for fetches of large columns (e.g. LONGs, BLOBs, etc.). The column is specified as an index into the result sequence. Not specifying the column will set the default size for all large columns in the cursor. This method would be used before the *executeXXX()* method is invoked. Implementations are free to have this method do nothing and users are free to not use it.

Type Objects and Constructors

Many databases need to have the input in a particular format for binding to an operation's input parameters. For example, if an input is destined for a DATE column, then it must be bound to the database in a particular string format. Similar problems exist for "Row ID" columns or large binary items (e.g. blobs or RAW columns). This presents problems for Python since the parameters to the *executeXXX()* method are untyped. When the database module sees a Python string object, it doesn't know if it should be bound as a simple CHAR column, as a raw BINARY item, or as a DATE. To overcome this problem, a module must provide the constructors defined below to create objects that can hold special values. When passed to the cursor methods, the module can then detect the proper type of the input parameter and bind it accordingly. A Cursor Object's *description* attribute returns information about each of the result columns of a query. The *type_code* must compare equal to one of Type Objects defined below. Type Objects may be equal to more than one type code (e.g. DATETIME could be equal to the type codes for date, time and timestamp columns; see the Implementation Hints below for details). The module exports the following constructors and singletons:

Date(*year*, *month*, *day*)
This function constructs an object holding a date value.

Time (*hour*, *minute*, *second*)
This function constructs an object holding a time value.

Timestamp (*year*, *month*, *day*, *hour*, *minute*, *second*)
This function constructs an object holding a time stamp value.

DateFromTicks (*ticks*)
This function constructs an object holding a date value from the given ticks value (number of seconds since the epoch; see the documentation of the standard Python time module for details).

TimeFromTicks (*ticks*)
This function constructs an object holding a time value from the given ticks value (number of seconds since the epoch; see the documentation of the standard Python time module for details).

TimestampFromTicks (*ticks*)
This function constructs an object holding a time stamp value from the given ticks value (number of seconds since the epoch; see the documentation of the standard Python time module for details).

Binary (*string*)
This function constructs an object capable of holding a binary (long) string value.

STRING
This type object is used to describe columns in a database that are string-based (e.g. CHAR).

BINARY
This type object is used to describe (long) binary columns in a database (e.g. LONG, RAW, BLOBs).

NUMBER
This type object is used to describe numeric columns in a database.

DATETIME
This type object is used to describe date/time columns in a database.

ROWID
This type object is used to describe the "Row ID" column in a database.

SQL NULL values are represented by the Python *None* singleton on input and output. Note: Usage of Unix ticks for database interfacing can cause troubles because of the limited date range they cover.

Implementation Hints

- The preferred object types for the date/time objects are those defined in the mxDateTime package. It provides all necessary constructors and methods both at Python and C level.
- The preferred object type for Binary objects are the buffer types available in standard Python starting with version 1.5.2. Please see the Python documentation for details. For information about the the C interface have a look at Include/bufferobject.h and Objects/bufferobject.c in the Python source distribution.
- Here is a sample implementation of the Unix ticks based constructors for date/time delegating work to the generic constructors:

```
import time

def DateFromTicks(ticks):

    return apply(Date,time.localtime(ticks)[:3])

def TimeFromTicks(ticks):

    return apply(Time,time.localtime(ticks)[3:6])
```

```
def TimestampFromTicks(ticks):

    return apply(Timestamp,time.localtime(ticks)[:6])
```

- This Python class allows implementing the above type objects even though the description type code field yields multiple values for on type object:

```
  class DBAPITypeObject:
      def __init__(self,*values):
      self.values = values
      def __cmp__(self,other):
      if other in self.values:
          return 0
      if other < self.values:
          return 1
      else:
          return -1

The resulting type object compares equal to all values passed to the
constructor.
```

- Here is a snippet of Python code that implements the exception hierarchy defined above:

```
  import exceptions

  class Error(exceptions.StandardError):
      pass

  class Warning(exceptions.StandardError):
      pass

  class InterfaceError(Error):
      pass

  class DatabaseError(Error):
      pass

  class InternalError(DatabaseError):
      pass

  class OperationalError(DatabaseError):
      pass

  class ProgrammingError(DatabaseError):
      pass

  class IntegrityError(DatabaseError):
      pass

  class DataError(DatabaseError):
      pass

  class NotSupportedError(DatabaseError):
      pass

In C you can use the `PyErr_NewException(fullname, base, NULL)` API to
create the exception objects.
```

Major Changes from Version 1.0 to Version 2.0

The Python Database API 2.0 introduces a few major changes compared to the 1.0 version. Because some of these changes will cause existing DB API 1.0 based scripts to break, the major version number was adjusted to reflect this change. These are the most important changes from 1.0 to 2.0:

- The need for a separate dbi module was dropped and the functionality merged into the module interface itself.
- New constructors and Type Objects were added for date/time values, the RAW Type Object was renamed to BINARY. The resulting set should cover all basic data types commonly found in modern SQL databases.
- New constants (apilevel, threadlevel, paramstyle) and methods (executemany, nextset) were added to provide better database bindings.
- The semantics of .callproc() needed to call stored procedures are now clearly defined.
- The definition of the .execute() return value changed. Previously, the return value was based on the SQL statement type (which was hard to implement right) – it is undefined now; use the more flexible .rowcount attribute instead. Modules are free to return the old style return values, but these are no longer mandated by the specification and should be considered database interface dependent.
- Class based exceptions were incorporated into the specification. Module implementors are free to extend the exception layout defined in this specification by subclassing the defined exception classes.

Open Issues

Although the version 2.0 specification clarifies a lot of questions that were left open in the 1.0 version, there are still some remaining issues:

- Define a useful return value for .nextset() for the case where a new result set is available.
- Create a fixed point numeric type for use as loss-less monetary and decimal interchange format.

Footnotes

Compliance to Python Database API 2.0

Unsupported Optional Features

`Cursor.`**`nextset`**`()`

This method is not implemented because the database engine does not support opening multiple result sets simultaneously with a single cursor.

Nominally Supported Optional Features

class `firebirdsql.`**`Cursor`**

`arraysize`

As required by the spec, the value of this attribute is observed with respect to the *fetchmany* method. However, changing the value of this attribute does not make any difference in fetch efficiency because the database engine only supports fetching a single row at a time.

setinputsizes()

Although this method is present, it does nothing, as allowed by the spec.

setoutputsize()

Although this method is present, it does nothing, as allowed by the spec.

Extensions and Caveats

pyfirebirdsql offers a large feature set beyond the minimal requirements of the Python DB API. Most of these extensions are documented in the section of this document entitled *Native Database Engine Features and Extensions Beyond the Python DB API*.

This section attempts to document only those features that overlap with the DB API, or are too insignificant to warrant their own subsection elsewhere.

firebirdsql.**connect**()

This function supports the following optional keyword arguments in addition to those required by the spec:

Role For connecting to a database with a specific SQL role.

Example:

```
firebirdsql.connect(dsn='host:/path/database.db', user='limited_user',
   password='pass', role='MORE_POWERFUL_ROLE')
```

Charset For explicitly specifying the character set of the connection. See Firebird Documentation for a list of available character sets, and *Unicode Fields and pyfirebirdsql* section for information on handling extended character sets with pyfirebirdsql.

Example:

```
firebirdsql.connect(dsn='host:/path/database.db', user='sysdba',
    password='pass', charset='UTF8')
```

Timeout (*Optional*) Dictionary with timeout and action specification. See section about Connection Timeouts for details.

class firebirdsql.**Connection**

charset

(read-only) The character set of the connection (set via the *charset* parameter of *firebirdsql.connect()*). See Firebird Documentation for a list of available character sets, and *Unicode Fields and pyfirebirdsql* section for information on handling extended character sets with pyfirebirdsql.

server_version

(read-only) The version string of the database server to which this connection is connected. For example, a connection to Firebird 1.0 on Windows has the following *server_version*: *WI-V6.2.794 Firebird 1.0*

execute_immediate()

Executes a statement without caching its prepared form. The statement must *not* be of a type that returns a result set. In most cases (especially cases in which the same statement – perhaps a parameterized statement – is executed repeatedly), it is better to create a cursor using the connection's *cursor* method, then execute the statement using one of the cursor's execute methods.

Arguments:

Sql String containing the SQL statement to execute.

commit(*retaining=False*)

rollback(*retaining=False*)

The *commit* and *rollback* methods accept an optional boolean parameter *retaining* (default *False*) that indicates whether the transactional context of the transaction being resolved should be recycled. For details, see the Advanced Transaction Control: Retaining Operations section of this document. The *rollback* method accepts an optional string parameter *savepoint* that causes the transaction to roll back only as far as the designated savepoint, rather than rolling back entirely. For details, see the Advanced Transaction Control: Savepoints section of this document.

class firebirdsql.**Cursor**

description

pyfirebirdsql makes absolutely no guarantees about *description* except those required by the Python Database API Specification 2.0 (that is, *description* is either *None* or a sequence of 7-element sequences). Therefore, client programmers should *not* rely on *description* being an instance of a particular class or type. pyfirebirdsql provides several named positional constants to be used as indices into a given element of *description* . The contents of all *description* elements are defined by the DB API spec; these constants are provided merely for convenience.

```
DESCRIPTION_NAME
DESCRIPTION_TYPE_CODE
DESCRIPTION_DISPLAY_SIZE
DESCRIPTION_INTERNAL_SIZE
DESCRIPTION_PRECISION
DESCRIPTION_SCALE
DESCRIPTION_NULL_OK
```

Here is an example of accessing the *name* of the first field in the *description* of cursor *cur*:

```
nameOfFirstField = cur.description[0][firebirdsql.DESCRIPTION_NAME]
```

For more information, see the documentation of Cursor.description in the DB API Specification.

rowcount

Although pyfirebirdsql's *Cursor's implement this attribute, the database engine's own support for the determination of "rows affected"/"rows selected" is quirky. The database engine only supports the determination of rowcount for 'INSERT*, *UPDATE*, *DELETE*, and *SELECT* statements. When stored procedures become involved, row count figures are usually not available to the client. Determining rowcount for *SELECT* statements is problematic: the rowcount is reported as zero until at least one row has been fetched from the result set, and the rowcount is misreported if the result set is larger than 1302 rows. The server apparently marshals result sets internally in batches of 1302, and will misreport the rowcount for result sets larger than 1302 rows until the 1303rd row is fetched, result sets larger than 2604 rows until the 2605th row is fetched, and so on, in increments of 1302. As required by the Python DB API Spec, the rowcount attribute "is -1 in case no executeXX() has been performed on the cursor or the rowcount of the last operation is not determinable by the interface".

fetchone()

fetchmany()

fetchall()

pyfirebirdsql makes absolutely no guarantees about the return value of the *fetchone* / *fetchmany* / *fetchall* methods except that it is a sequence indexed by field position. pyfirebirdsql makes absolutely no guarantees about the return value of the *fetchonemap* / *fetchmanymap* / *fetchallmap* methods (documented below)

except that it is a mapping of field name to field value. Therefore, client programmers should *not* rely on the return value being an instance of a particular class or type.

fetchonemap()
This method is just like the standard *fetchone* method of the DB API, except that it returns a mapping of field name to field value, rather than a sequence.

fetchmanymap()
This method is just like the standard *fetchmany* method of the DB API, except that it returns a sequence of mappings of field name to field value, rather than a sequence of sequences.

fetchallmap()
This method is just like the standard *fetchall* method of the DB API, except that it returns a sequence of mappings of field name to field value, rather than a sequence of sequences.

iter()

itermap()
These methods are equivalent to the *fetchall* and *fetchallmap* methods, respectively, except that they return iterators rather than materialized sequences. *iter* and *itermap* are exercised in this example.

Native Database Engine Features and Extensions Beyond the Python DB API

Programmatic Database Creation and Deletion

The Firebird engine stores a database in a fairly straightforward manner: as a single file or, if desired, as a segmented group of files.

The engine supports dynamic database creation via the SQL statement *CREATE DATABASE*.

The engine also supports dropping (deleting) databases dynamically, but dropping is a more complicated operation than creating, for several reasons: an existing database may be in use by users other than the one who requests the deletion, it may have supporting objects such as temporary sort files, and it may even have dependent shadow databases. Although the database engine recognizes a *DROP DATABASE* SQL statement, support for that statement is limited to the *isql* command-line administration utility. However, the engine supports the deletion of databases via an API call, which pyfirebirdsql exposes to Python (see below).

pyfirebirdsql supports dynamic database creation and deletion via the module-level function *firebirdsql.create_database()* and the method *drop_database()*. These are documented below, then demonstrated by a brief example.

firebirdsql.**create_database**()
Creates a database according to the supplied *CREATE DATABASE* SQL statement. Returns an open connection to the newly created database.

Arguments:

Sql string containing the *CREATE DATABASE* statement. Note that this statement may need to include a username and password.

Connection.**drop_database**()
Deletes the database to which the connection is attached.

This method performs the database deletion in a responsible fashion. Specifically, it:

- raises an *OperationalError* instead of deleting the database if there are other active connections to the database

•deletes supporting files and logs in addition to the primary database file(s)

This method has no arguments.

Example program:

```
import firebirdsql

con = firebirdsql.create_database(
      "create database '/temp/db.db' user 'sysdba' password 'pass'"
      )
con.drop_database()
```

Advanced Transaction Control

For the sake of simplicity, pyfirebirdsql lets the Python programmer ignore transaction management to the greatest extent allowed by the Python Database API Specification 2.0. The specification says, "if the database supports an auto-commit feature, this must be initially off". At a minimum, therefore, it is necessary to call the *commit* method of the connection in order to persist any changes made to the database. Transactions left unresolved by the programmer will be 'rollback'ed when the connection is garbage collected.

Remember that because of ACID, every data manipulation operation in the Firebird database engine takes place in the context of a transaction, including operations that are conceptually "read-only", such as a typical *SELECT*. The client programmer of pyfirebirdsql establishes a transaction implicitly by using any SQL execution method, such as `execute_immediate()`, `Cursor.execute()`, or `Cursor.callproc()`.

Although pyfirebirdsql allows the programmer to pay little attention to transactions, it also exposes the full complement of the database engine's advanced transaction control features: transaction parameters, retaining transactions, savepoints, and distributed transactions.

Explicit transaction start

In addition to the implicit transaction initiation required by Python Database API, pyfirebirdsql allows the programmer to start transactions explicitly via the *Connection.begin* method.

`Connection.`**`begin`**`(`*`tpb`*`)`

Starts a transaction explicitly. This is never *required*; a transaction will be started implicitly if necessary.

Tpb Optional transaction parameter buffer (TPB) populated with *firebirdsql.isc_tpb_** constants. See the Firebird API guide for these constants' meanings.

Transaction Parameters

The database engine offers the client programmer an optional facility called *transaction parameter buffers* (TPBs) for tweaking the operating characteristics of the transactions he initiates. These include characteristics such as whether the transaction has read and write access to tables, or read-only access, and whether or not other simultaneously active transactions can share table access with the transaction.

Connections have a `default_tpb` attribute that can be changed to set the default TPB for all transactions subsequently started on the connection. Alternatively, if the programmer only wants to set the TPB for a single transaction, he can start a transaction explicitly via the `begin()` method and pass a TPB for that single transaction.

For details about TPB construction, see the Firebird API documentation. In particular, the `ibase.h` supplied with Firebird contains all possible TPB elements – single bytes that the C API defines as constants whose names begin with *isc_tpb_*. pyfirebirdsql makes all of those TPB constants available (under the same names) as module-level constants in the form of single-character strings. A transaction parameter *buffer* is handled in C as a character array;

pyfirebirdsql requires that TPBs be constructed as Python strings. Since the constants in the *firebirdsql.isc_tpb_** family are single-character Python strings, they can simply be concatenated to create a TPB.

Warning: This method requires good knowledge of *tpc_block* structure and proper order of various parameters, as Firebird engine will raise an error when badly structured block would be used. Also definition of *table reservation* parameters is uncomfortable as you'll need to mix binary codes with table names passed as Pascal strings (characters preceded by string length).

The following program uses explicit transaction initiation and TPB construction to establish an unobtrusive transaction for read-only access to the database:

```
import firebirdsql

con = firebirdsql.connect(dsn='localhost:/temp/test.db', user='sysdba', password='pass
↪')

# Construct a TPB by concatenating single-character strings (bytes)
# from the firebirdsql.isc_tpb_* family.
customTPB = (
      firebirdsql.isc_tpb_read
    + firebirdsql.isc_tpb_read_committed
    + firebirdsql.isc_tpb_rec_version
  )

# Explicitly start a transaction with the custom TPB:
con.begin(tpb=customTPB)

# Now read some data using cursors:
...

# Commit the transaction with the custom TPB.  Future transactions
# opened on con will not use a custom TPB unless it is explicitly
# passed to con.begin every time, as it was above, or
# con.default_tpb is changed to the custom TPB, as in:
#   con.default_tpb = customTPB
con.commit()
```

For convenient and safe construction of custom *tpb_block*, pyfirebirdsql provides special utility class *TPB*.

class `firebirdsql.`**`TPB`**

`access_mode`
Required access mode. Default *isc_tpb_write*.

`isolation_level`
Required Transaction Isolation Level. Default *isc_tpb_concurrency*.

`lock_resolution`
Required lock resolution method. Default *isc_tpb_wait*.

`lock_timeout`
Required lock timeout. Default *None*.

`table_reservation`
Table reservation specification. Default *None*. Instead of changing the value of the table_reservation object itself, you must change its *elements* by manipulating it as though it were a dictionary that mapped "TABLE_NAME": (sharingMode, accessMode) For example:

```
tpbBuilder.table_reservation["MY_TABLE"] =
  (firebirdsql.isc_tpb_protected, firebirdsql.isc_tpb_lock_write)
```

render()
Returns valid *transaction parameter block* according to current values of member attributes.

```
import firebirdsql

con = firebirdsql.connect(dsn='localhost:/temp/test.db', user='sysdba', password='pass
↪')

# Use TPB to construct valid transaction parameter block
# from the firebirdsql.isc_tpb_* family.
customTPB = TPB()
customTPB.access_mode = firebirdsql.isc_tpb_read
customTPB.isolation_level = firebirdsql.isc_tpb_read_committed
    + firebirdsql.isc_tpb_rec_version

# Explicitly start a transaction with the custom TPB:
con.begin(tpb=customTPB.render())

# Now read some data using cursors:
...

# Commit the transaction with the custom TPB.  Future transactions
# opened on con will not use a custom TPB unless it is explicitly
# passed to con.begin every time, as it was above, or
# con.default_tpb is changed to the custom TPB, as in:
#   con.default_tpb = customTPB.render()
con.commit()
```

If you want to build only *table reservation* part of *tpb* (for example to add to various custom built parameter blocks), you can use class *TableReservation* instead *TPB*.

class firebirdsql.**TableReservation**
This is a *dictionary-like* class, where keys are table names and values must be tuples of access parameters, i.e. "TABLE_NAME": (sharingMode, accessMode)

render()
Returns propely formatted table reservation part of *transaction parameter block* according to current values.

Conenction object also exposes two methods that return infromation about current transaction:

class firebirdsql.**Connection**

trans_info(*request*)
Pythonic wrapper around `transaction_info()` call.

Request One or more information request codes (see transaction_info for details). Multiple codes must be passed as tuple.

Returns decoded response(s) for specified request code(s). When multiple requests are passed, returns a dictionary where key is the request code and value is the response from server.

transaction_info(*request*, *result_type*)
Thin wrapper around Firebird API *isc_transaction_info* call. This function returns information about active transaction. Raises *ProgrammingError* exception when transaction is not active.

Request One from the next constants:

- isc_info_tra_id
- isc_info_tra_oldest_interesting
- isc_info_tra_oldest_snapshot
- isc_info_tra_oldest_active
- isc_info_tra_isolation
- isc_info_tra_access
- isc_info_tra_lock_timeout

See Firebird API Guide for details.

Result_type String code for result type:

- 'i' for Integer
- 's' fro String

Retaining Operations

The *commit* and *rollback* methods of *firebirdsql.Connection* accept an optional boolean parameter *retaining* (default *False*) to indicate whether to recycle the transactional context of the transaction being resolved by the method call.

If *retaining* is *True*, the infrastructural support for the transaction active at the time of the method call will be "retained" (efficiently and transparently recycled) after the database server has committed or rolled back the conceptual transaction.

In code that commits or rolls back frequently, "retaining" the transaction yields considerably better performance. However, retaining transactions must be used cautiously because they can interfere with the server's ability to garbage collect old record versions. For details about this issue, read the "Garbage" section of this document by Ann Harrison.

For more information about retaining transactions, see Firebird documentation.

Savepoints

Firebird 1.5 introduced support for transaction savepoints. Savepoints are named, intermediate control points within an open transaction that can later be rolled back to, without affecting the preceding work. Multiple savepoints can exist within a single unresolved transaction, providing "multi-level undo" functionality.

Although Firebird savepoints are fully supported from SQL alone via the *SAVEPOINT 'name'* and *ROLLBACK TO 'name'* statements, pyfirebirdsql also exposes savepoints at the Python API level for the sake of convenience.

`Connection.`**`savepoint`**`(`*`name`*`)`
Establishes a savepoint with the specified *name*. To roll back to a specific savepoint, call the `rollback()` method and provide a value (the name of the savepoint) for the optional *savepoint* parameter. If the *savepoint* parameter of `rollback()` is not specified, the active transaction is cancelled in its entirety, as required by the Python Database API Specification.

The following program demonstrates savepoint manipulation via the pyfirebirdsql API, rather than raw SQL.

```
import firebirdsql

con = firebirdsql.connect(dsn='localhost:/temp/test.db', user='sysdba', password='pass
↪')
cur = con.cursor()
```

```
cur.execute("recreate table test_savepoints (a integer)")
con.commit()

print 'Before the first savepoint, the contents of the table are:'
cur.execute("select * from test_savepoints")
print ' ', cur.fetchall()

cur.execute("insert into test_savepoints values (?)", [1])
con.savepoint('A')
print 'After savepoint A, the contents of the table are:'
cur.execute("select * from test_savepoints")
print ' ', cur.fetchall()

cur.execute("insert into test_savepoints values (?)", [2])
con.savepoint('B')
print 'After savepoint B, the contents of the table are:'
cur.execute("select * from test_savepoints")
print ' ', cur.fetchall()

cur.execute("insert into test_savepoints values (?)", [3])
con.savepoint('C')
print 'After savepoint C, the contents of the table are:'
cur.execute("select * from test_savepoints")
print ' ', cur.fetchall()

con.rollback(savepoint='A')
print 'After rolling back to savepoint A, the contents of the table are:'
cur.execute("select * from test_savepoints")
print ' ', cur.fetchall()

con.rollback()
print 'After rolling back entirely, the contents of the table are:'
cur.execute("select * from test_savepoints")
print ' ', cur.fetchall()
```

The output of the example program is shown below.

```
Before the first savepoint, the contents of the table are:
  []
After savepoint A, the contents of the table are:
  [(1,)]
After savepoint B, the contents of the table are:
  [(1,), (2,)]
After savepoint C, the contents of the table are:
  [(1,), (2,), (3,)]
After rolling back to savepoint A, the contents of the table are:
  [(1,)]
After rolling back entirely, the contents of the table are:
  []
```

Using multiple transactions with the same connection

Python Database API 2.0 was created with assumption that connection can support only one transactions per single connection. However, Firebird can support multiple independent transactions that can run simultaneously within single connection / attachment to the database. This feature is very important, as applications may require multiple

transaction openned simultaneously to perform various tasks, which would require to open multiple connections and thus consume more resources than necessary.

pyfirebirdsql surfaces this Firebird feature through new class *Transaction* and extensions to *Connection* and *Cursor* classes.

class firebirdsql.**Connection**

trans(*tpb=None*)
Creates a new Transaction that operates within the context of this connection. Cursors can be created within that Transaction via its .cursor() method.

transactions
read-only property

List of non-close()d *Transaction* objects associated with this *Connection*. An element of this list may represent a resolved or unresolved physical transaction. Once a *Transaction* object has been created, it is only removed from the Connection's tracker if the Transaction's *close()* method is called (*Transaction.__del__* triggers an implicit close() call if necessary), or (obviously) if the Connection itself is close()d. The initial implementation will not make any guarantees about the order of the Transactions in this list.

main_transaction
read-only property

Transaction object that represents the DB-API implicit transaction. The implementation guarantees that the same Transaction object will be reused across all DB-API transactions during the lifetime of the Connection.

prepare()
Manually triggers the first phase of a two-phase commit (2PC). Use of this method is optional; if preparation is not triggered manually, it will be performed implicitly by commit() in a 2PC. See also the **'Distributed Transactions'_** section for details.

class firebirdsql.**Cursor**

transaction
read-only property

Transaction with which this Cursor is associated. *None* if the Transaction has been close()d, or if the Cursor has been close()d.

class firebirdsql.**Transaction**

__init__(*connection*, *tpb=None*)
Constructor requires open *Connection* object and optional *tpb* specification.

connection
read-only property

Connection object on which this Transaction is based. When the Connection's close() method is called, all Transactions that depend on the connection will also be implicitly close()d. If a Transaction has been close()d, its connection property will be None.

closed
read-only property

True if Transaction has been closed (explicitly or implicitly).

n_physical
read-only property (int)

Number of physical transactions that have been executed via this Transaction object during its lifetime.

resolution
read-only property (int)

Zero if this Transaction object is currently managing an open physical transaction. *One* if the physical transaction has been resolved normally. Note that this is an int property rather than a bool, and is named *resolution* rather than *resolved*, so that the non-zero values other than one can be assigned to convey specific information about the state of the transaction, in a future implementation (consider distributed transaction prepared state, limbo state, etc.).

cursors
List of non-close()d Cursor objects associated with this Transaction. When Transaction's close() method is called, whether explicitly or implicitly, it will implicitly close() each of its Cursors. Current implementation do not make any guarantees about the order of the Cursors in this list.

begin(*tpb*)
See *Connection.begin()* for details.

commit(*retaining=False*)
See *firebirdsql.Connection.commit()* for details.

close()
Permanently closes the Transaction object and severs its associations with other objects. If the physical transaction is unresolved when this method is called, a rollback() will be performed first.

prepare()
See *Connection.prepare()* for details.

rollback(*retaining=False*)
See *firebirdsql.Connection.rollback()* for details.

savepoint()
See *Connection.savepoint()* for details.

trans_info()
See *Connection.trans_info()* for details.

transaction_info()
See *Connection.transaction_info()* for details.

cursor()
Creates a new Cursor that will operate in the context of this Transaction. The association between a Cursor and its Transaction is set when the Cursor is created, and cannot be changed during the lifetime of that Cursor. See *Connection.cursor()* for more details.

If you don't want multiple transactions, you can use implicit transaction object associated with *Connection* and control it via transaction-management and cursor methods of the *Connection*.

Alternatively, you can directly access the implicit transaction exposed as *main_transaction* and control it via its transaction-management methods.

To use additional transactions, create new *Transaction* object calling *Connection.trans()* method.

Prepared Statements

When you define a Python function, the interpreter initially parses the textual representation of the function and generates a binary equivalent called bytecode. The bytecode representation can then be executed directly by the Python interpreter any number of times and with a variety of parameters, but the human-oriented textual definition of the function never need be parsed again.

Database engines perform a similar series of steps when executing a SQL statement. Consider the following series of statements:

```
cur.execute("insert into the_table (a,b,c) values ('aardvark', 1, 0.1)")
...
cur.execute("insert into the_table (a,b,c) values ('zymurgy', 2147483647, 99999.999)")
```

If there are many statements in that series, wouldn't it make sense to "define a function" to insert the provided "parameters" into the predetermined fields of the predetermined table, instead of forcing the database engine to parse each statement anew and figure out what database entities the elements of the statement refer to? In other words, why not take advantage of the fact that the form of the statement ("the function") stays the same throughout, and only the values ("the parameters") vary? Prepared statements deliver that performance benefit and other advantages as well.

The following code is semantically equivalent to the series of insert operations discussed previously, except that it uses a single SQL statement that contains Firebird's parameter marker (*?*) in the slots where values are expected, then supplies those values as Python tuples instead of constructing a textual representation of each value and passing it to the database engine for parsing:

```
insertStatement = "insert into the_table (a,b,c) values (?,?,?)"
cur.execute(insertStatement, ('aardvark', 1, 0.1))
...
cur.execute(insertStatement, ('zymurgy', 2147483647, 99999.999))
```

Only the values change as each row is inserted; the statement remains the same. For many years, pyfirebirdsql has recognized situations similar to this one and automatically reused the same prepared statement in each *Cursor.execute()* call. In pyfirebirdsql 3.2, the scheme for automatically reusing prepared statements has become more sophisticated, and the API has been extended to offer the client programmer manual control over prepared statement creation and use.

The entry point for manual statement preparation is the *Cursor.prep* method.

Cursor.**prep**(*sql*)

> **Sql** string parameter that contains the SQL statement to be prepared. Returns a *PreparedStatement* instance.

class firebirdsql.**PreparedStatement**

PreparedStatement has no public methods, but does have the following public read-only properties:

sql

A reference to the string that was passed to *prep()* to create this *PreparedStatement*.

statement_type

An integer code that can be matched against the statement type constants in the *firebirdsql.isc_info_sql_stmt_** series. The following statement type codes are currently available:

- *isc_info_sql_stmt_commit*
- *isc_info_sql_stmt_ddl*
- *isc_info_sql_stmt_delete*
- *isc_info_sql_stmt_exec_procedure*
- *isc_info_sql_stmt_get_segment*
- *isc_info_sql_stmt_insert*
- *isc_info_sql_stmt_put_segment*
- *isc_info_sql_stmt_rollback*
- *isc_info_sql_stmt_savepoint*

- *isc_info_sql_stmt_select*
- *isc_info_sql_stmt_select_for_upd*
- *isc_info_sql_stmt_set_generator*
- *isc_info_sql_stmt_start_trans*
- *isc_info_sql_stmt_update*

n_input_params
The number of input parameters the statement requires.

n_output_params
The number of output fields the statement produces.

plan
A string representation of the execution plan generated for this statement by the database engine's optimizer. This property can be used, for example, to verify that a statement is using the expected index.

description
A Python DB API 2.0 description sequence (of the same format as *Cursor.description*) that describes the statement's output parameters. Statements without output parameters have a *description* of *None*.

In addition to programmatically examining the characteristics of a SQL statement via the properties of *PreparedStatement*, the client programmer can submit a *PreparedStatement* to *Cursor.execute()* or *Cursor.executemany()* for execution. The code snippet below is semantically equivalent to both of the previous snippets in this section, but it explicitly prepares the *INSERT* statement in advance, then submits it to *Cursor.executemany()* for execution:

```
insertStatement = cur.prep("insert into the_table (a,b,c) values (?,?,?)")
inputRows = [
    ('aardvark', 1, 0.1),
    ...
    ('zymurgy', 2147483647, 99999.999)
  ]
cur.executemany(insertStatement, inputRows)
```

Example Program

The following program demonstrates the explicit use of PreparedStatements. It also benchmarks explicit *PreparedStatement* reuse against pyfirebirdsql's automatic *PreparedStatement* reuse, and against an input strategy that prevents *PreparedStatement* reuse.

```
import time
import firebirdsql

con = firebirdsql.connect(dsn=r'localhost:D:\temp\test-20.firebird',
    user='sysdba', password='masterkey'
  )

cur = con.cursor()

# Create supporting database entities:
cur.execute("recreate table t (a int, b varchar(50))")
con.commit()
cur.execute("create unique index unique_t_a on t(a)")
con.commit()

# Explicitly prepare the insert statement:
```

```
psIns = cur.prep("insert into t (a,b) values (?,?)")
print 'psIns.sql: "%s"' % psIns.sql
print 'psIns.statement_type == firebirdsql.isc_info_sql_stmt_insert:', (
    psIns.statement_type == firebirdsql.isc_info_sql_stmt_insert
  )
print 'psIns.n_input_params: %d' % psIns.n_input_params
print 'psIns.n_output_params: %d' % psIns.n_output_params
print 'psIns.plan: %s' % psIns.plan

print

N = 10000
iStart = 0

# The client programmer uses a PreparedStatement explicitly:
startTime = time.time()
for i in xrange(iStart, iStart + N):
    cur.execute(psIns, (i, str(i)))
print (
    'With explicit prepared statement, performed'
    '\n  %0.2f insertions per second.' % (N / (time.time() - startTime))
  )
con.commit()

iStart += N

# pyfirebirdsql automatically uses a PreparedStatement "under the hood":
startTime = time.time()
for i in xrange(iStart, iStart + N):
    cur.execute("insert into t (a,b) values (?,?)", (i, str(i)))
print (
    'With implicit prepared statement, performed'
    '\n  %0.2f insertions per second.' % (N / (time.time() - startTime))
  )
con.commit()

iStart += N

# A new SQL string containing the inputs is submitted every time, so
# pyfirebirdsql is not able to implicitly reuse a PreparedStatement.  Also, in a
# more complicated scenario where the end user supplied the string input
# values, the program would risk SQL injection attacks:
startTime = time.time()
for i in xrange(iStart, iStart + N):
    cur.execute("insert into t (a,b) values (%d,'%s')" % (i, str(i)))
print (
    'When unable to reuse prepared statement, performed'
    '\n  %0.2f insertions per second.' % (N / (time.time() - startTime))
  )
con.commit()

# Prepare a SELECT statement and examine its properties.  The optimizer's plan
# should use the unique index that we created at the beginning of this program.
print
psSel = cur.prep("select * from t where a = ?")
print 'psSel.sql: "%s"' % psSel.sql
print 'psSel.statement_type == firebirdsql.isc_info_sql_stmt_select:', (
    psSel.statement_type == firebirdsql.isc_info_sql_stmt_select
```

```
  )
print 'psSel.n_input_params: %d' % psSel.n_input_params
print 'psSel.n_output_params: %d' % psSel.n_output_params
print 'psSel.plan: %s' % psSel.plan

# The current implementation does not allow PreparedStatements to be prepared
# on one Cursor and executed on another:
print
print 'Note that PreparedStatements are not transferrable from one cursor to another:'
cur2 = con.cursor()
cur2.execute(psSel)
```

Output:

```
psIns.sql: "insert into t (a,b) values (?,?)"
psIns.statement_type == firebirdsql.isc_info_sql_stmt_insert: True
psIns.n_input_params: 2
psIns.n_output_params: 0
psIns.plan: None

With explicit prepared statement, performed
  9551.10 insertions per second.
With implicit prepared statement, performed
  9407.34 insertions per second.
When unable to reuse prepared statement, performed
  1882.53 insertions per second.

psSel.sql: "select * from t where a = ?"
psSel.statement_type == firebirdsql.isc_info_sql_stmt_select: True
psSel.n_input_params: 1
psSel.n_output_params: 2
psSel.plan: PLAN (T INDEX (UNIQUE_T_A))

Note that PreparedStatements are not transferrable from one cursor to another:
Traceback (most recent call last):
  File "adv_prepared_statements__overall_example.py", line 86, in ?
    cur2.execute(psSel)
firebirdsql.ProgrammingError: (0, 'A PreparedStatement can only be used with the
 Cursor that originally prepared it.')
```

As you can see, the version that prevents the reuse of prepared statements is about five times slower – *for a trivial statement*. In a real application, SQL statements are likely to be far more complicated, so the speed advantage of using prepared statements would only increase.

As the timings indicate, pyfirebirdsql does a good job of reusing prepared statements even if the client program is written in a style strictly compatible with the Python DB API 2.0 (which accepts only strings – not *PreparedStatement* objects – to the *Cursor.execute()* method). The performance loss in this case is less than one percent.

Named Cursors

To allow the Python programmer to perform scrolling *UPDATE* or *DELETE* via the "*SELECT ... FOR UPDATE*" syntax, pyfirebirdsql provides the read/write property *Cursor.name*.

`Cursor.`**`name`**

Name for the SQL cursor. This property can be ignored entirely if you don't need to use it.

Example Program

```
import firebirdsql

con = firebirdsql.connect(dsn='localhost:/temp/test.db', user='sysdba', password='pass
↪')
curScroll = con.cursor()
curUpdate = con.cursor()

curScroll.execute("select city from addresses for update")
curScroll.name = 'city_scroller'
update = "update addresses set city=? where current of " + curScroll.name

for (city,) in curScroll:
    city = ... # make some changes to city
    curUpdate.execute( update, (city,) )

con.commit()
```

Parameter Conversion

pyfirebirdsql converts bound parameters marked with a *?* in SQL code in a standard way. However, the module also offers several extensions to standard parameter binding, intended to make client code more readable and more convenient to write.

Implicit Conversion of Input Parameters from Strings

The database engine treats most SQL data types in a weakly typed fashion: the engine may attempt to convert the raw value to a different type, as appropriate for the current context. For instance, the SQL expressions *123* (integer) and *'123'* (string) are treated equivalently when the value is to be inserted into an *integer* field; the same applies when *'123'* and *123* are to be inserted into a *varchar* field.

This weak typing model is quite unlike Python's dynamic yet strong typing. Although weak typing is regarded with suspicion by most experienced Python programmers, the database engine is in certain situations so aggressive about its typing model that pyfirebirdsql must compromise in order to remain an elegant means of programming the database engine.

An example is the handling of "magic values" for date and time fields. The database engine interprets certain string values such as *'yesterday'* and *'now'* as having special meaning in a date/time context. If pyfirebirdsql did not accept strings as the values of parameters destined for storage in date/time fields, the resulting code would be awkward. Consider the difference between the two Python snippets below, which insert a row containing an integer and a timestamp into a table defined with the following DDL statement:

```
create table test_table (i int, t timestamp)
```

```
i = 1
t = 'now'
sqlWithMagicValues = "insert into test_table (i, t) values (?, '%s')" % t
cur.execute( sqlWithMagicValues, (i,) )
```

```
i = 1
t = 'now'
cur.execute( "insert into test_table (i, t) values (?, ?)", (i, t) )
```

If pyfirebirdsql did not support weak parameter typing, string parameters that the database engine is to interpret as "magic values" would have to be rolled into the SQL statement in a separate operation from the binding of the rest of the parameters, as in the first Python snippet above. Implicit conversion of parameter values from strings allows the consistency evident in the second snippet, which is both more readable and more general.

It should be noted that pyfirebirdsql does not perform the conversion from string itself. Instead, it passes that responsibility to the database engine by changing the parameter metadata structure dynamically at the last moment, then restoring the original state of the metadata structure after the database engine has performed the conversion.

A secondary benefit is that when one uses pyfirebirdsql to import large amounts of data from flat files into the database, the incoming values need not necessarily be converted to their proper Python types before being passed to the database engine. Eliminating this intermediate step may accelerate the import process considerably, although other factors such as the chosen connection protocol and the deactivation of indexes during the import are more consequential. For bulk import tasks, the database engine's external tables also deserve consideration. External tables can be used to suck semi-structured data from flat files directly into the relational database without the intervention of an ad hoc conversion program.

Dynamic Type Translation

Dynamic type translators are conversion functions registered by the Python programmer to transparently convert database field values to and from their internal representation.

The client programmer can choose to ignore translators altogether, in which case pyfirebirdsql will manage them behind the scenes. Otherwise, the client programmer can use any of several standard type translators included with pyfirebirdsql, register custom translators, or set the translators to *None* to deal directly with the pyfirebirdsql-internal representation of the data type. When translators have been registered for a specific SQL data type, Python objects on their way into a database field of that type will be passed through the input translator before they are presented to the database engine; values on their way out of the database into Python will be passed through the corresponding output translator. Output and input translation for a given type is usually implemented by two different functions.

Specifics of the Dynamic Type Translation API

Translators are managed with next methods of *Connection* and *Cursor*.

`Connection.`**`get_type_trans_in`**`()`
Retrieves the inbound type translation map.

`Connection.`**`set_type_trans_in`**`(`*trans_dict*`)`
Changes the inbound type translation map.

`Cursor.`**`get_type_trans_in`**`()`
Retrieves the inbound type translation map.

`Cursor.`**`set_type_trans_in`**`(`*trans_dict*`)`
Changes the inbound type translation map.

The *set_type_trans_[in|out]* methods accept a single argument: a mapping of type name to translator. The *get_type_trans[in|out]* methods return a copy of the translation table.

Cursor's inherit their 'Connection's translation settings, but can override them without affecting the connection or other cursors (much as subclasses can override the methods of their base classes).

The following code snippet installs an input translator for fixed point types (*NUMERIC/ DECIMAL* SQL types) into a connection:

```
con.set_type_trans_in( {'FIXED': fixed_input_translator_function} )
```

The following method call retrieves the type translation table for *con*:

```
con.get_type_trans_in()
```

The method call above would return a translation table (dictionary) such as this:

```
{
  'DATE': <function date_conv_in at 0x00920648>,
  'TIMESTAMP': <function timestamp_conv_in at 0x0093E090>,
  'FIXED': <function <lambda> at 0x00962DB0>,
  'TIME': <function time_conv_in at 0x009201B0>
}
```

Notice that although the sample code registered only one type translator, there are four listed in the mapping returned by the *get_type_trans_in* method. By default, pyfirebirdsql uses dynamic type translation to implement the conversion of *DATE*, *TIME*, *TIMESTAMP*, *NUMERIC*, and *DECIMAL* values. For the source code locations of pyfirebirdsql's reference translators, see the table in the next section.

In the sample above, a translator is registered under the key *'FIXED'*, but Firebird has no SQL data type named *FIXED*. The following table lists the names of the database engine's SQL data types in the left column, and the corresponding pyfirebirdsql-specific key under which client programmers can register translators in the right column. **Mapping of SQL Data Type Names to Translator Keys**

SQL Type(s)	Translator Key
CHAR / VARCHAR	'TEXT' for fields with charsets *NONE*, *OCTETS*, or *ASCII* 'TEXT_UNICODE' for all other charsets
BLOB	'BLOB'
SMALLINT/INTEGER/BIGINT	'INTEGER'
FLOAT/ DOUBLE PRECISION	'FLOATING'
NUMERIC / DECIMAL	'FIXED'
DATE	'DATE'
TIME	'TIME'
TIMESTAMP	'TIMESTAMP'

Database Arrays

pyfirebirdsql converts database arrays *from* Python sequences (except strings) on input; *to* Python lists on output. On input, the Python sequence must be nested appropriately if the array field is multi- dimensional, and the incoming sequence must not fall short of its maximum possible length (it will not be "padded" implicitly–see below). On output, the lists will be nested if the database array has multiple dimensions.

Database arrays have no place in a purely relational data model, which requires that data values be *atomized* (that is, every value stored in the database must be reduced to elementary, non-decomposable parts). The Firebird implementation of database arrays, like that of most relational database engines that support this data type, is fraught with limitations.

Database arrays are of fixed size, with a predeclared number of dimensions (max. 16) and number of elements per dimension. Individual array elements cannot be set to *NULL* / *None*, so the mapping between Python lists (which have dynamic length and are therefore *not* normally "padded" with dummy values) and non-trivial database arrays is clumsy.

Stored procedures cannot have array parameters.

Finally, many interface libraries, GUIs, and even the isql command line utility do not support database arrays.

In general, it is preferable to avoid using database arrays unless you have a compelling reason.

Example Program

The following program inserts an array (nested Python list) into a single database field, then retrieves it.

```
import firebirdsql

con = firebirdsql.connect(dsn='localhost:/temp/test.db', user='sysdba', password='pass
↪')
con.execute_immediate("recreate table array_table (a int[3,4])")
con.commit()

cur = con.cursor()

arrayIn = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9,10,11,12]
  ]

print 'arrayIn:  %s' % arrayIn
cur.execute("insert into array_table values (?)", (arrayIn,))

cur.execute("select a from array_table")
arrayOut = cur.fetchone()[0]
print 'arrayOut: %s' % arrayOut

con.commit()
```

Output:

```
arrayIn:  [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
arrayOut: [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

Blobs

pyfirebirdsql supports the insertion and retrieval of blobs either wholly in memory ("materialized mode") or in chunks ("streaming mode") to reduce memory usage when handling large blobs. The default handling mode is "materialized"; the "streaming" method is selectable via a special case of Dynamic Type Translation.

In **materialized** mode, input and output blobs are represented as Python *str* objects, with the result that the entirety of each blob's contents is loaded into memory. Unfortunately, flaws in the database engine's C API prevent automatic Unicode conversion from applying to textual blobs in the way it applies to Unicode *CHAR* and *VARCHAR* fields in any Firebird version prior to version 2.1.

Note: pyfirebirdsql 3.3 introduces new type_conv mode 300 that enables automatic type conversion for textual blobs when you're working with Firebird 2.1 and newer.

In **streaming** mode, any Python "file-like" object is acceptable as input for a blob parameter. Obvious examples of such objects are instances of `file` or `StringIO`. Each output blob is represented by a *`firebirdsql.BlobReader`* object.

class `firebirdsql.`**`BlobReader`**

BlobReader is a "file-like" class, so it acts much like a *file* instance opened in *rb* mode.

BlobReader adds one method not found in the "file-like" interface:

chunks()
Takes a single integer parameter that specifies the number of bytes to retrieve in each chunk (the final chunk may be smaller).

For example, if the size of the blob is *50000000* bytes, *BlobReader.chunks(2**20)* will return *47* one-megabyte chunks, and a smaller final chunk of *716928* bytes.

Due to the combination of CPython's deterministic finalization with careful programming in pyfirebirdsql's internals, it is not strictly necessary to close *BlobReader* instances explicitly. A *BlobReader* object will be automatically closed by its *__del__* method when it goes out of scope, or when its *Connection* closes, whichever comes first. However, it is always a better idea to close resources explicitly (via *try...finally*) than to rely on artifacts of the CPython implementation. (For the sake of clarity, the example program does not follow this practice.)

Example Program

The following program demonstrates blob storage and retrieval in both *materialized* and *streaming* modes.

```
import os.path
from cStringIO import StringIO

import firebirdsql

con = firebirdsql.connect(dsn=r'localhost:D:\temp\test-20.firebird',
    user='sysdba', password='masterkey'
  )

cur = con.cursor()

cur.execute("recreate table blob_test (a blob)")
con.commit()

# --- Materialized mode (str objects for both input and output) ---
# Insertion:
cur.execute("insert into blob_test values (?)", ('abcdef',))
cur.execute("insert into blob_test values (?)", ('ghijklmnop',))
# Retrieval:
cur.execute("select * from blob_test")
print 'Materialized retrieval (as str):'
print cur.fetchall()

cur.execute("delete from blob_test")

# --- Streaming mode (file-like objects for input; firebirdsql.BlobReader
#     objects for output) ---
cur.set_type_trans_in ({'BLOB': {'mode': 'stream'}})
cur.set_type_trans_out({'BLOB': {'mode': 'stream'}})

# Insertion:
cur.execute("insert into blob_test values (?)", (StringIO('abcdef'),))
cur.execute("insert into blob_test values (?)", (StringIO('ghijklmnop'),))

f = file(os.path.abspath(__file__), 'rb')
cur.execute("insert into blob_test values (?)", (f,))
f.close()

# Retrieval using the "file-like" methods of BlobReader:
cur.execute("select * from blob_test")

readerA = cur.fetchone()[0]
```

```
print '\nStreaming retrieval (via firebirdsql.BlobReader):'

# Python "file-like" interface:
print 'readerA.mode:    "%s"' % readerA.mode
print 'readerA.closed:   %s'  % readerA.closed
print 'readerA.tell():   %d'  % readerA.tell()
print 'readerA.read(2): "%s"' % readerA.read(2)
print 'readerA.tell():   %d'  % readerA.tell()
print 'readerA.read():  "%s"' % readerA.read()
print 'readerA.tell():   %d'  % readerA.tell()
print 'readerA.read():  "%s"' % readerA.read()
readerA.close()
print 'readerA.closed:   %s'  % readerA.closed

# The chunks method (not part of the Python "file-like" interface, but handy):
print '\nFor a blob with contents "ghijklmnop", iterating over'
print 'BlobReader.chunks(3) produces:'
readerB = cur.fetchone()[0]
for chunkNo, chunk in enumerate(readerB.chunks(3)):
    print 'Chunk %d is: "%s"' % (chunkNo, chunk)
```

Output:

```
Materialized retrieval (as str):
[('abcdef',), ('ghijklmnop',)]

Streaming retrieval (via firebirdsql.BlobReader):
readerA.mode:    "rb"
readerA.closed:   False
readerA.tell():   0
readerA.read(2): "ab"
readerA.tell():   2
readerA.read():  "cdef"
readerA.tell():   6
readerA.read():  ""
readerA.closed:   True

For a blob with contents "ghijklmnop", iterating over
BlobReader.chunks(3) produces:
Chunk 0 is: "ghi"
Chunk 1 is: "jkl"
Chunk 2 is: "mno"
Chunk 3 is: "p"
```

Connection Timeouts

Connection timeouts allow the programmer to request that a connection be automatically closed after a specified period of inactivity. The simplest uses of connection timeouts are trivial, as demonstrated by the following snippet:

```
import firebirdsql

con = firebirdsql.connect(dsn=r'localhost:D:\temp\test.db',
    user='sysdba', password='masterkey',
    timeout={'period': 120.0} # time out after 120.0 seconds of inactivity
  )
```

```
...
```

The connection created in the example above is *eligible* to be automatically closed by pyfirebirdsql if it remains idle for at least 120.0 consecutive seconds. pyfirebirdsql does not guarantee that the connection will be closed immediately when the specified period has elapsed. On a busy system, there might be a considerable delay between the moment a connection becomes eligible for timeout and the moment pyfirebirdsql actually closes it. However, the thread that performs connection timeouts is programmed in such a way that on a lightly loaded system, it acts almost instantaneously to take advantage of a connection's eligibility for timeout.

After a connection has timed out, pyfirebirdsql reacts to attempts to reactivate the severed connection in a manner dependent on the state of the connection when it timed out. Consider the following example program:

```
import time
import firebirdsql

con = firebirdsql.connect(dsn=r'localhost:D:\temp\test.db',
    user='sysdba', password='masterkey',
    timeout={'period': 3.0}
  )
cur = con.cursor()

cur.execute("recreate table test (a int, b char(1))")
con.commit()

cur.executemany("insert into test (a, b) values (?, ?)",
    [(1, 'A'), (2, 'B'), (3, 'C')]
  )
con.commit()

cur.execute("select * from test")
print 'BEFORE:', cur.fetchall()

cur.execute("update test set b = 'X' where a = 2")

time.sleep(6.0)

cur.execute("select * from test")
print 'AFTER: ', cur.fetchall()
```

So, should the example program print

```
BEFORE: [(1, 'A'), (2, 'B'), (3, 'C')]
AFTER:  [(1, 'A'), (2, 'X'), (3, 'C')]
```

or

```
BEFORE: [(1, 'A'), (2, 'B'), (3, 'C')]
AFTER:  [(1, 'A'), (2, 'B'), (3, 'C')]
```

or should it raise an exception? The answer is more complex than one might think.

First of all, we cannot guarantee much about the example program's behavior because there is a race condition between the obvious thread that's executing the example code (which we'll call "UserThread" for the rest of this section) and the pyfirebirdsql-internal background thread that actually closes connections that have timed out ("TimeoutThread"). If the operating system were to suspend UserThread just after the *`firebirdsql.connect()`* call for more than the specified timeout period of 3.0 seconds, the TimeoutThread might close the connection before UserThread had performed any preparatory operations on the database. Although such a scenario is extremely unlikely when more

"realistic" timeout periods such as 1800.0 seconds (30 minutes) are used, it is important to consider. We'll explore solutions to this race condition later.

The *likely* (but not guaranteed) behavior of the example program is that UserThread will complete all preparatory database operations including the *cur. execute ("update test set b = 'X' where a = 2")* statement in the example program, then go to sleep for not less than 6.0 seconds. Not less than 3.0 seconds after UserThread executes the *cur. execute ("update test set b = 'X' where a = 2")* statement, TimeoutThread is likely to close the connection because it has become eligible for timeout.

The crucial issue is how TimeoutThread should resolve the transaction that UserThread left open on *con*, and what should happen when UserThread reawakens and tries to execute the *cur. execute ("select * from test")* statement, since the transaction that UserThread left open will no longer be active.

User-Supplied Connection Timeout Callbacks

In the context of a particular client program, it is not possible for pyfirebirdsql to know the best way for TimeoutThread to react when it encounters a connection that is eligible for timeout, but has an unresolved transaction. For this reason, pyfirebirdsql's connection timeout system offers callbacks that the client programmer can use to guide the TimeoutThread's actions, or to log information about connection timeout patterns.

The "Before Timeout" Callback

The client programmer can supply a "before timeout" callback that accepts a single dictionary parameter and returns an integer code to indicate how the TimeoutThread should proceed when it finds a connection eligible for timeout. Within the dictionary, pyfirebirdsql provides the following entries:

dsn The *dsn* parameter that was passed to *firebirdsql.connect* when the connection was created.

has_transaction A boolean that indicates whether the connection has an unresolved transaction.

active_secs A *float* that indicates how many seconds elapsed between the point when the connection attached to the server and the last client program activity on the connection.

idle_secs A *float* that indicates how many seconds have elapsed since the last client program activity on the connection. This value will not be less than the specified timeout period, and is likely to only a fraction of a second longer.

Based on those data, the user-supplied callback should return one of the following codes:

`firebirdsql.`**`CT_VETO`**
Directs the TimeoutThread not to close the connection at the current time, and not to reconsider timing the connection out until at least another timeout period has passed. For example, if a connection was created with a timeout period of 120.0 seconds, and the user-supplied "before callback" returns *CT_VETO*, the TimeoutThread will not reconsider timing out that particular connection until at least another 120.0 seconds have elapsed.

`firebirdsql.`**`CT_NONTRANSPARENT`**
("Nontransparent rollback")

Directs the TimeoutThread to roll back the connection's unresolved transaction (if any), then close the connection. Any future attempt to use the connection will raise a `firebirdsql.ConnectionTimedOut` exception.

`firebirdsql.`**`CT_ROLLBACK`**
("Transparent rollback")

Directs the TimeoutThread to roll back the connection's unresolved transaction (if any), then close the connection. Upon any future attempt to use the connection, pyfirebirdsql will *attempt* to transparently reconnect to the database and "resume where it left off" insofar as possible. Of course, network problems and the like could prevent pyfirebirdsql's *attempt* at transparent resumption from succeeding. Also, highly state-dependent objects

such as open result sets, *BlobReader*, and *PreparedStatement* cannot be used transparently across a connection timeout.

`firebirdsql.`**`CT_COMMIT`**
("Transparent commit")

Directs the TimeoutThread to commit the connection's unresolved transaction (if any), then close the connection. Upon any future attempt to use the connection, pyfirebirdsql will *attempt* to transparently reconnect to the database and "resume where it left off" insofar as possible.

If the user does not supply a "before timeout" callback, pyfirebirdsql considers the timeout transparent only if the connection does not have an unresolved transaction.

If the user-supplied "before timeout" callback returns anything other than one of the codes listed above, or if it raises an exception, the TimeoutThread will act as though *CT_NONTRANSPARENT* had been returned.

You might have noticed that the input dictionary to the "before timeout" callback does *not* include a reference to the *Connection* object itself. This is a deliberate design decision intended to steer the client programmer away from writing callbacks that take a long time to complete, or that manipulate the *Connection* instance directly. See the caveats section for more information.

The "After Timeout" Callback

The client programmer can supply an "after timeout" callback that accepts a single dictionary parameter. Within that dictionary, pyfirebirdsql currently provides the following entries:

dsn The *dsn* parameter that was passed to *firebirdsql.connect()* when the connection was created.

active_secs A *float* that indicates how many seconds elapsed between the point when the connection attached to the server and the last client program activity on the connection.

idle_secs A *float* that indicates how many seconds elapsed between the last client program activity on the connection and the moment the TimeoutThread closed the connection.

pyfirebirdsql only calls the "after timeout" callback after the connection has actually been closed by the TimeoutThread. If the "before timeout" callback returns *CT_VETO* to cancel the timeout attempt, the "after timeout" callback will not be called.

pyfirebirdsql discards the return value of the "after timeout" callback, and ignores any exceptions.

The same caveats that apply to the "before timeout" callback also apply to the "after timeout" callback.

User-Supplied Connection Timeout Callback Caveats

- The user-supplied callbacks are executed by the TimeoutThread. They should be designed to avoid blocking the TimeoutThread any longer than absolutely necessary.
- Manipulating the *Connection* object that is being timed out (or any of that connection's subordinate objects such as *Cursor*, *BlobReader*, or *PreparedStatement*) from the timeout callbacks is strictly forbidden.

Examples

Example: 'CT_VETO'

The following program registers a "before timeout" callback that unconditionally returns *CT_VETO*, which means that the TimeoutThread never times the connection out. Although an "after timeout" callback is also registered, it will never be called.

```
import time
import firebirdsql

def callback_before(info):
    print
    print 'callback_before called; input parameter contained:'
    for key, value in info.items():
        print '  %s: %s' % (repr(key).ljust(20), repr(value))
    print
    # Unconditionally veto any timeout attempts:
    return firebirdsql.CT_VETO

def callback_after(info):
    assert False, 'This will never be called.'

con = firebirdsql.connect(dsn=r'localhost:D:\temp\test.db',
    user='sysdba', password='masterkey',
    timeout={
        'period': 3.0,
        'callback_before': callback_before,
        'callback_after':  callback_after,
      }
  )
cur = con.cursor()

cur.execute("recreate table test (a int, b char(1))")
con.commit()

cur.executemany("insert into test (a, b) values (?, ?)",
    [(1, 'A'), (2, 'B'), (3, 'C')]
  )
con.commit()

cur.execute("select * from test")
print 'BEFORE:', cur.fetchall()

cur.execute("update test set b = 'X' where a = 2")

time.sleep(6.0)

cur.execute("select * from test")
rows = cur.fetchall()
# The value of the second column of the second row of the table is still 'X',
# because the transaction that changed it from 'B' to 'X' remains active.
assert rows[1][1] == 'X'
print 'AFTER: ', rows
```

Sample output:

```
BEFORE: [(1, 'A'), (2, 'B'), (3, 'C')]

callback_before called; input parameter contained:
  'dsn'               : 'localhost:D:\\temp\\test.db'
  'idle_secs'         : 3.0
  'has_transaction'   : True

AFTER:  [(1, 'A'), (2, 'X'), (3, 'C')]
```

Example: Supporting Module 'timeout_authorizer'

The example programs for *CT_NONTRANSPARENT*, *CT_ROLLBACK*, and *CT_COMMIT* rely on the *TimeoutAuthorizer* class from the module below to guarantee that the TimeoutThread will not time the connection out before the preparatory code has executed.

```
import threading
import firebirdsql

class TimeoutAuthorizer(object):
    def __init__(self, opCodeWhenAuthorized):
        self.currentOpCode = firebirdsql.CT_VETO
        self.opCodeWhenAuthorized = opCodeWhenAuthorized

        self.lock = threading.Lock()

    def authorize(self):
        self.lock.acquire()
        try:
            self.currentOpCode = self.opCodeWhenAuthorized
        finally:
            self.lock.release()

    def __call__(self, info):
        self.lock.acquire()
        try:
            return self.currentOpCode
        finally:
            self.lock.release()
```

Example: 'CT_NONTRANSPARENT'

```
import threading, time
import firebirdsql
import timeout_authorizer

authorizer = timeout_authorizer.TimeoutAuthorizer(firebirdsql.CT_NONTRANSPARENT)
connectionTimedOut = threading.Event()

def callback_after(info):
    print
    print 'The connection was closed nontransparently.'
    print
    connectionTimedOut.set()

con = firebirdsql.connect(dsn=r'localhost:D:\temp\test.db',
    user='sysdba', password='masterkey',
    timeout={
        'period': 3.0,
        'callback_before': authorizer,
        'callback_after':  callback_after,
      }
  )
cur = con.cursor()

cur.execute("recreate table test (a int, b char(1))")
con.commit()

cur.executemany("insert into test (a, b) values (?, ?)",
```

```
    [(1, 'A'), (2, 'B'), (3, 'C')]
  )
con.commit()

cur.execute("select * from test")
print 'BEFORE:', cur.fetchall()

cur.execute("update test set b = 'X' where a = 2")

authorizer.authorize()
connectionTimedOut.wait()

# This will raise a firebirdsql.ConnectionTimedOut exception because the
# before callback returned firebirdsql.CT_NONTRANSPARENT:
cur.execute("select * from test")
```

Sample output:

```
BEFORE: [(1, 'A'), (2, 'B'), (3, 'C')]

The connection was closed nontransparently.

Traceback (most recent call last):
  File "connection_timeouts_ct_nontransparent.py", line 42, in ?
    cur.execute("select * from test")
firebirdsql.ConnectionTimedOut: (0, 'A transaction was still unresolved when
this connection timed out, so it cannot be transparently reactivated.')
```

Example: 'CT_ROLLBACK'

```
import threading, time
import firebirdsql
import timeout_authorizer

authorizer = timeout_authorizer.TimeoutAuthorizer(firebirdsql.CT_ROLLBACK)
connectionTimedOut = threading.Event()

def callback_after(info):
    print
    print 'The unresolved transaction was rolled back; the connection has been'
    print ' closed transparently.'
    print
    connectionTimedOut.set()

con = firebirdsql.connect(dsn=r'localhost:D:\temp\test.db',
    user='sysdba', password='masterkey',
    timeout={
        'period': 3.0,
        'callback_before': authorizer,
        'callback_after':  callback_after,
      }
  )
cur = con.cursor()

cur.execute("recreate table test (a int, b char(1))")
con.commit()

cur.executemany("insert into test (a, b) values (?, ?)",
```

```
    [(1, 'A'), (2, 'B'), (3, 'C')]
  )
con.commit()

cur.execute("select * from test")
print 'BEFORE:', cur.fetchall()

cur.execute("update test set b = 'X' where a = 2")

authorizer.authorize()
connectionTimedOut.wait()

# The value of the second column of the second row of the table will have
# reverted to 'B' when the transaction that changed it to 'X' was rolled back.
# The cur.execute call on the next line will transparently reactivate the
# connection, which was timed out transparently.
cur.execute("select * from test")
rows = cur.fetchall()
assert rows[1][1] == 'B'
print 'AFTER: ', rows
```

Sample output:

```
BEFORE: [(1, 'A'), (2, 'B'), (3, 'C')]

The unresolved transaction was rolled back; the connection has been
 closed transparently.

AFTER:  [(1, 'A'), (2, 'B'), (3, 'C')]
```

Example: 'CT_COMMIT'

```
import threading, time
import firebirdsql
import timeout_authorizer

authorizer = timeout_authorizer.TimeoutAuthorizer(firebirdsql.CT_COMMIT)
connectionTimedOut = threading.Event()

def callback_after(info):
    print
    print 'The unresolved transaction was committed; the connection has been'
    print ' closed transparently.'
    print
    connectionTimedOut.set()

con = firebirdsql.connect(dsn=r'localhost:D:\temp\test.db',
    user='sysdba', password='masterkey',
    timeout={
        'period': 3.0,
        'callback_before': authorizer,
        'callback_after':  callback_after,
      }
  )
cur = con.cursor()

cur.execute("recreate table test (a int, b char(1))")
con.commit()
```

```
cur.executemany("insert into test (a, b) values (?, ?)",
    [(1, 'A'), (2, 'B'), (3, 'C')]
  )
con.commit()

cur.execute("select * from test")
print 'BEFORE:', cur.fetchall()

cur.execute("update test set b = 'X' where a = 2")

authorizer.authorize()
connectionTimedOut.wait()

# The modification of the value of the second column of the second row of the
# table from 'B' to 'X' will have persisted, because the TimeoutThread
# committed the transaction before it timed the connection out.
# The cur.execute call on the next line will transparently reactivate the
# connection, which was timed out transparently.
cur.execute("select * from test")
rows = cur.fetchall()
assert rows[1][1] == 'X'
print 'AFTER: ', rows
```

Sample output:

```
BEFORE: [(1, 'A'), (2, 'B'), (3, 'C')]

The unresolved transaction was committed; the connection has been
 closed transparently.

AFTER:  [(1, 'A'), (2, 'X'), (3, 'C')]
```

Database Event Notification

What are database events?

The database engine features a distributed, interprocess communication mechanism based on messages called *database events*. A database event is a message passed from a trigger or stored procedure to an application to announce the occurrence of a specified condition or action, usually a database change such as an insertion, modification, or deletion of a record. The Firebird event mechanism enables applications to respond to actions and database changes made by other, concurrently running applications without the need for those applications to communicate directly with one another, and without incurring the expense of CPU time required for periodic polling to determine if an event has occurred.

Why use database events?

Anything that can be accomplished with database events can also be implemented using other techniques, so why bother with events? Since you've chosen to write database-centric programs in Python rather than assembly language, you probably already know the answer to this question, but let's illustrate.

A typical application for database events is the handling of administrative messages. Suppose you have an administrative message database with a *messages* table, into which various applications insert timestamped status reports. It may be desirable to react to these messages in diverse ways, depending on the status they indicate: to ignore them, to

initiate the update of dependent databases upon their arrival, to forward them by e-mail to a remote administrator, or even to set off an alarm so that on-site administrators will know a problem has occurred.

It is undesirable to tightly couple the program whose status is being reported (the *message producer*) to the program that handles the status reports (the *message handler*). There are obvious losses of flexibility in doing so. For example, the message producer may run on a separate machine from the administrative message database and may lack access rights to the downstream reporting facilities (e.g., network access to the SMTP server, in the case of forwarded e-mail notifications). Additionally, the actions required to handle status reports may themselves be time-consuming and error-prone, as in accessing a remote network to transmit e-mail.

In the absence of database event support, the message handler would probably be implemented via *polling*. Polling is simply the repetition of a check for a condition at a specified interval. In this case, the message handler would check in an infinite loop to see whether the most recent record in the *messages* table was more recent than the last message it had handled. If so, it would handle the fresh message(s); if not, it would go to sleep for a specified interval, then loop.

The *polling-based* implementation of the message handler is fundamentally flawed. Polling is a form of busy-wait; the check for new messages is performed at the specified interval, regardless of the actual activity level of the message producers. If the polling interval is lengthy, messages might not be handled within a reasonable time period after their arrival; if the polling interval is brief, the message handler program (and there may be many such programs) will waste a large amount of CPU time on unnecessary checks.

The database server is necessarily aware of the exact moment when a new message arrives. Why not let the message handler program request that the database server send it a notification when a new message arrives? The message handler can then efficiently sleep until the moment its services are needed. Under this *event-based* scheme, the message handler becomes aware of new messages at the instant they arrive, yet it does not waste CPU time checking in vain for new messages when there are none available.

How events are exposed to the server and the client process?

1. Server Process ("An event just occurred!")

   To notify any interested listeners that a specific event has occurred, issue the *POST_EVENT* statement from Stored Procedure or Trigger. The *POST_EVENT* statement has one parameter: the name of the event to post. In the preceding example of the administrative message database, *POST_EVENT* might be used from an *after insert* trigger on the *messages* table, like this:

```
create trigger trig_messages_handle_insert
  for messages
    after insert
as
begin
  POST_EVENT 'new_message';
end
```

   Note: The physical notification of the client process does not occur until the transaction in which the *POST_EVENT* took place is actually committed. Therefore, multiple events may *conceptually* occur before the client process is *physically* informed of even one occurrence. Furthermore, the database engine makes no guarantee that clients will be informed of events in the same groupings in which they conceptually occurred. If, within a single transaction, an event named *event_a* is posted once and an event named *event_b* is posted once, the client may receive those posts in separate "batches", despite the fact that they occurred in the same conceptual unit (a single transaction). This also applies to multiple occurrences of *the same* event within a single conceptual unit: the physical notifications may arrive at the client separately.

2. Client Process ("Send me a message when an event occurs.")

Note: If you don't care about the gory details of event notification, skip to the section that describes pyfirebirdsql's Python-level event handling API.

The Firebird C client library offers two forms of event notification. The first form is *synchronous* notification, by way of the function **:cfunc:'isc_wait_for_event()'**. This form is admirably simple for a C programmer to use, but is inappropriate as a basis for pyfirebirdsql's event support, chiefly because it's not sophisticated enough to serve as the basis for a comfortable Python-level API. The other form of event notification offered by the database client library is *asynchronous*, by way of the functions **:cfunc:'isc_que_events()'** (note that the name of that function is misspelled), **:cfunc:'isc_cancel_events()'**, and others. The details are as nasty as they are numerous, but the essence of using asynchronous notification from C is as follows:

(a) Call **:cfunc:'isc_event_block()'** to create a formatted binary buffer that will tell the server which events the client wants to listen for.

(b) Call **:cfunc:'isc_que_events()'** (passing the buffer created in the previous step) to inform the server that the client is ready to receive event notifications, and provide a callback that will be asynchronously invoked when one or more of the registered events occurs.

(c) [The thread that called **:cfunc:'isc_que_events()'** to initiate event listening must now do something else.]

(d) When the callback is invoked (the database client library starts a thread dedicated to this purpose), it can use the **:cfunc:'isc_event_counts()'** function to determine how many times each of the registered events has occurred since the last call to **:cfunc:'isc_event_counts()'** (if any).

(e) [The callback thread should now "do its thing", which may include communicating with the thread that called **:cfunc:'isc_que_events()'**.]

(f) When the callback thread is finished handling an event notification, it must call **:cfunc:'isc_que_events()'** again in order to receive future notifications. Future notifications will invoke the callback again, effectively "looping" the callback thread back to Step 4.

How events are exposed to the Python programmer?

The pyfirebirdsql database event API is comprised of the following: the method *Connection.event_conduit* and the class *EventConduit*.

`Connection.`**`event_conduit`**`()`

Creates a conduit (an instance of *EventConduit*) through which database event notifications will flow into the Python program.

event_conduit is a method of *Connection* rather than a module-level function or a class constructor because the database engine deals with events in the context of a particular database (after all, *POST_EVENT* must be issued by a stored procedure or a trigger).

Arguments:

Event_names A sequence of string event names The *EventConduit.wait()* method will block until the occurrence of at least one of the events named by the strings in *event_names*. pyfirebirdsql's own event-related code is capable of operating with up to 2147483647 events per conduit. However, it has been observed that the Firebird client library experiences catastrophic problems (including memory corruption) on some platforms with anything beyond about 100 events per conduit. These limitations are dependent on both the Firebird version and the platform.

class `firebirdsql.`**`EventConduit`**

__init__()
The *EventConduit* class is not designed to be instantiated directly by the Python programmer. Instead, use the *Connection.event_conduit* method to create *EventConduit* instances.

wait(*timeout=None*)
Blocks the calling thread until at least one of the events occurs, or the specified *timeout* (if any) expires.

If one or more event notifications has arrived since the last call to *wait*, this method will retrieve a notification from the head of the *EventConduit*'s internal queue and return immediately.

The names of the relevant events were supplied to the *Connection.event_conduit* method during the creation of this *EventConduit*. In the code snippet below, the relevant events are named *event_a* and *event_b*:

```
conduit = connection.event_conduit( ('event_a', 'event_b') )
conduit.wait()
```

Arguments:

Timeout *optional* number of seconds (use a *float* to indicate fractions of seconds) If not even one of the relevant events has occurred after *timeout* seconds, this method will unblock and return *None*. The default *timeout* is infinite.

Returns: *None* if the wait timed out, otherwise a dictionary that maps *event_name* -> *event_occurrence_count*.

In the code snippet above, if *event_a* occurred once and *event_b* did not occur at all, the return value from *conduit.wait()* would be the following dictionary:

```
{
 'event_a': 1,
 'event_b': 0
}
```

close()
Cancels the standing request for this conduit to be notified of events.

After this method has been called, this *EventConduit* object is useless, and should be discarded. (The boolean property *closed* is *True* after an *EventConduit* has been closed.)

This method has no arguments.

flush()
This method allows the Python programmer to manually clear any event notifications that have accumulated in the conduit's internal queue.

From the moment the conduit is created by the `Connection.event_conduit()` method, notifications of any events that occur will accumulate asynchronously within the conduit's internal queue until the conduit is closed either explicitly (via the *close* method) or implicitly (via garbage collection). There are two ways to dispose of the accumulated notifications: call *wait()* to receive them one at a time (*wait()* will block when the conduit's internal queue is empty), or call this method to get rid of all accumulated notifications.

This method has no arguments.

Returns: The number of event notifications that were flushed from the queue. The "number of event *notifications*" is not necessarily the same as the "number of event *occurrences*", since a single notification can indicate multiple occurrences of a given event (see the return value of the *wait* method).

Example Program

The following code (a SQL table definition, a SQL trigger definition, and two Python programs) demonstrates pyfirebirdsql-based event notification.

The example is based on a database at *'localhost:/temp/test.db'*, which contains a simple table named *test_table*. *test_table* has an *after insert* trigger that posts several events. Note that the trigger posts *test_event_a* twice, *test_event_b* once, and *test_event_c* once.

The Python event *handler* program connects to the database and establishes an *EventConduit* in the context of that connection. As specified by the list of *RELEVANT_EVENTS* passed to *event_conduit*, the event conduit will concern itself only with events named *test_event_a* and *test_event_b*. Next, the program calls the conduit's *wait* method without a timeout; it will wait infinitely until *at least one* of the relevant events is posted in a transaction that is subsequently committed.

The Python event *producer* program simply connects to the database, inserts a row into *test_table*, and commits the transaction. Notice that except for the printed comment, no code in the producer makes any mention of events – the events are posted as an implicit consequence of the row's insertion into *test_table*.

The insertion into *test_table* causes the trigger to *conceptually* post events, but those events are not *physically* sent to interested listeners until the transaction is committed. When the commit occurs, the handler program returns from the *wait* call and prints the notification that it received.

SQL table definition:

```
create table test_table (a integer)
```

SQL trigger definition:

```
create trigger trig_test_insert_event
  for test_table
    after insert
as
begin
  post_event 'test_event_a';
  post_event 'test_event_b';
  post_event 'test_event_c';

  post_event 'test_event_a';
end
```

Python event *handler* program:

```
import firebirdsql

RELEVANT_EVENTS = ['test_event_a', 'test_event_b']

con = firebirdsql.connect(dsn='localhost:/temp/test.db', user='sysdba', password='pass
↪')
conduit = con.event_conduit(RELEVANT_EVENTS)

print 'HANDLER: About to wait for the occurrence of one of %s...\n' % RELEVANT_EVENTS
result = conduit.wait()
print 'HANDLER: An event notification has arrived:'
print result
conduit.close()
```

Python event *producer* program:

```
import firebirdsql

con = firebirdsql.connect(dsn='localhost:/temp/test.db', user='sysdba', password='pass
↪')
cur = con.cursor()

cur.execute("insert into test_table values (1)")
print 'PRODUCER: Committing transaction that will cause event notification to be sent.
↪'
con.commit()
```

Event producer output:

```
PRODUCER: Committing transaction that will cause event notification to be sent.
```

Event handler output (assuming that the handler was already started and waiting when the event producer program was executed):

```
HANDLER: About to wait for the occurrence of one of ['test_event_a', 'test_event_b']..
↪.

HANDLER: An event notification has arrived:
{'test_event_a': 2, 'test_event_b': 1}
```

Notice that there is no mention of *test_event_c* in the result dictionary received by the event handler program. Although *test_event_c* was posted by the *after insert* trigger, the event conduit in the handler program was created to listen only for *test_event_a* and *test_event_b* events.

Pitfalls and Limitations

- Remember that if an *EventConduit* is left active (not yet closed or garbage collected), notifications for any registered events that actually occur will continue to accumulate in the EventConduit's internal queue even if the Python programmer doesn't call *`EventConduit.wait()`* to receive the notifications or *`EventConduit.flush()`* to clear the queue. The ill-informed may misinterpret this behavior as a memory leak in pyfirebirdsql; it is not.
- NEVER use LOCAL-protocol connections in a multithreaded program that also uses event handling! The database client library implements the local protocol on some platforms in such a way that deadlocks may arise in bizarre places if you do this. *This no-LOCAL prohibition is not limited to connections that are used as the basis for event conduits; it applies to all connections throughout the process.* So why doesn't pyfirebirdsql protect the Python programmer from this mistake? Because the event handling thread is started by the database client library, and it operates beyond the synchronization domain of pyfirebirdsql at times.

Note: The restrictions on the number of active *EventConduit's in a process, and on the number of event names that a single 'EventConduit* can listen for, have been removed in pyfirebirdsql 3.2.

The *database_info* API

Firebird provides various informations about server and connected database via *database_info* API call. pyfirebirdsql surfaces this API through next methods on Connection object:

Connection.**database_info**(*request*, *result_type*)

This method is a *very thin* wrapper around function **:cfunc:‘isc_database_info()‘**. This method does *not* attempt to interpret its results except with regard to whether they are a string or an integer.

For example, requesting **:cdata:‘isc_info_user_names‘** with the call

```
con.database_info(firebirdsql.isc_info_user_names, 's')
```

will return a binary string containing a *raw* succession of length- name pairs. A more convenient way to access the same functionality is via the *db_info()* method.

Arguments:

Request One of the *firebirdsql.isc_info_** constants.

Result_type Must be either *‘s’* if you expect a string result, or *‘i’* if you expect an integer result.

Example Program

```
import firebirdsql

con = firebirdsql.connect(dsn='localhost:/temp/test.db', user='sysdba', password=
↪'pass')

# Retrieving an integer info item is quite simple.
bytesInUse = con.database_info(firebirdsql.isc_info_current_memory, 'i')

print 'The server is currently using %d bytes of memory.' % bytesInUse

# Retrieving a string info item is somewhat more involved, because the
# information is returned in a raw binary buffer that must be parsed
# according to the rules defined in the Interbase® 6 API Guide section
# entitled "Requesting buffer items and result buffer values" (page 51).
#
# Often, the buffer contains a succession of length-string pairs
# (one byte telling the length of s, followed by s itself).
# Function firebirdsql.raw_byte_to_int is provided to convert a raw
# byte to a Python integer (see examples below).
buf = con.database_info(firebirdsql.isc_info_db_id, 's')

# Parse the filename from the buffer.
beginningOfFilename = 2
# The second byte in the buffer contains the size of the database filename
# in bytes.
lengthOfFilename = firebirdsql.raw_byte_to_int(buf[1])
filename = buf[beginningOfFilename:beginningOfFilename + lengthOfFilename]

# Parse the host name from the buffer.
beginningOfHostName = (beginningOfFilename + lengthOfFilename) + 1
# The first byte after the end of the database filename contains the size
# of the host name in bytes.
lengthOfHostName = firebirdsql.raw_byte_to_int(buf[beginningOfHostName - 1])
host = buf[beginningOfHostName:beginningOfHostName + lengthOfHostName]

print 'We are connected to the database at %s on host %s.' % (filename, host)
```

Sample output:

```
The server is currently using 8931328 bytes of memory.
We are connected to the database at C:\TEMP\TEST.DB on host WEASEL.
```

As you can see, extracting data with the *database_info* function is rather clumsy. In pyfirebirdsql 3.2, a higher-level means of accessing the same information is available: the *db_info()* method. Also, the Services API (accessible to Python programmers via the *firebirdsql.services* module) provides high-level support for querying database statistics and performing maintenance.

Connection.**db_info**(*request*)

High-level convenience wrapper around the *database_info()* method that parses the output of *database_info* into Python-friendly objects instead of returning raw binary uffers in the case of complex result types. If an unrecognized *isc_info_** code is requested, this method raises *ValueError*.

For example, requesting **:cdata:‘isc_info_user_names‘** with the call

```
con.db_info(firebirdsql.isc_info_user_names)
```

returns a dictionary that maps (username -> number of open connections). If *SYSDBA* has one open connection to the database to which *con* is connected, and *TEST_USER_1* has three open connections to that same database, the return value would be *{‘SYSDBA’: 1, ‘TEST_USER_1’: 3}*

Arguments:

Request must be either:

- A single *firebirdsql.isc_info_** info request code. In this case, a single result is returned.
- A sequence of such codes. In this case, a mapping of (info request code -> result) is returned.

Example Program

```
import os.path

import firebirdsql

DB_FILENAME = r'D:\temp\test-20.firebird'
DSN = 'localhost:' + DB_FILENAME

##########################################################################
# Querying an isc_info_* item that has a complex result:
##########################################################################
# Establish three connections to the test database as TEST_USER_1, and one
# connection as SYSDBA.  Then use the Connection.db_info method to query the
# number of attachments by each user to the test database.
testUserCons = []
for i in range(3):
    tCon = firebirdsql.connect(dsn=DSN, user='test_user_1', password='pass')
    testUserCons.append(tCon)

con = firebirdsql.connect(dsn=DSN, user='sysdba', password='masterkey')

print 'Open connections to this database:'
print con.db_info(firebirdsql.isc_info_user_names)

##########################################################################
# Querying multiple isc_info_* items at once:
##########################################################################
# Request multiple db_info items at once, specifically the page size of the
# database and the number of pages currently allocated.  Compare the size
# computed by that method with the size reported by the file system.
# The advantages of using db_info instead of the file system to compute
# database size are:
#   - db_info works seamlessly on connections to remote databases that reside
```

```
#     in file systems to which the client program lacks access.
#   - If the database is split across multiple files, db_info includes all of
#     them.
res = con.db_info(
    [firebirdsql.isc_info_page_size, firebirdsql.isc_info_allocation]
  )
pagesAllocated = res[firebirdsql.isc_info_allocation]
pageSize = res[firebirdsql.isc_info_page_size]
print '\ndb_info indicates database size is', pageSize * pagesAllocated, 'bytes'
print   'os.path.getsize indicates size is ', os.path.getsize(DB_FILENAME), 'bytes
↪'
```

Sample output:

```
  Open connections to this database:
  {'SYSDBA': 1, 'TEST_USER_1': 3}

db_info indicates database size is 20684800 bytes
os.path.getsize indicates size is  20684800 bytes
```

Using Firebird Services API

Database server maintenance tasks such as user management, load monitoring, and database backup have traditionally been automated by scripting the command-line tools **gbak**, **gfix**, **gsec**, and **gstat**.

The API presented to the client programmer by these utilities is inelegant because they are, after all, command-line tools rather than native components of the client language. To address this problem, Firebird has a facility called the Services API, which exposes a uniform interface to the administrative functionality of the traditional command-line tools.

The native Services API, though consistent, is much lower-level than a Pythonic API. If the native version were exposed directly, accomplishing a given task would probably require more Python code than scripting the traditional command-line tools. For this reason, pyfirebirdsql presents its own abstraction over the native API via the *firebirdsql.services* module.

Establishing Services API Connections

All Services API operations are performed in the context of a connection to a specific database server, represented by the *firebirdsql.services.Connection* class.

firebirdsql.services.**connect**(*host='service_mgr'*, *user='sysdba'*, *password=None*)

Establish a connection to database server Services and returns *firebirdsql.services.Connection* object.

Host The network name of the computer on which the database server is running.

User The name of the database user under whose authority the maintenance tasks are to be performed.

Password User's password.

Since maintenance operations are most often initiated by an administrative user on the same computer as the database server, *host* defaults to the local computer, and *user* defaults to *SYSDBA*.

The three calls to *firebirdsql.services.connect()* in the following program are equivalent:

```
from firebirdsql import services

con = services.connect(password='masterkey')
con = services.connect(user='sysdba', password='masterkey')
con = services.connect(host='localhost', user='sysdba', password='masterkey')
```

class firebirdsql.services.**Connection**

close()
Explicitly terminates a *Connection*; if this is not invoked, the underlying connection will be closed implicitly when the *Connection* object is garbage collected.

Server Configuration and Activity Levels

Connection.**getServiceManagerVersion**()
To help client programs adapt to version changes, the service manager exposes its version number as an integer.

```
from firebirdsql import services
con = services.connect(host='localhost', user='sysdba', password='masterkey')

print con.getServiceManagerVersion()
```

Output (on Firebird 1.5.0):

```
2
```

firebirdsql.services is a thick wrapper of the Services API that can shield its users from changes in the underlying C API, so this method is unlikely to be useful to the typical Python client programmer.

Connection.**getServerVersion**()
Returns the server's version string:

```
from firebirdsql import services
con = services.connect(host='localhost', user='sysdba', password='masterkey')

print con.getServerVersion()
```

Output (on Firebird 1.5.0/Win32):

```
WI-V1.5.0.4290 Firebird 1.5
```

At first glance, thhis method appears to duplicate the functionality of the *firebirdsql.Connection.server_version* property, but when working with Firebird, there is a difference. *firebirdsql.Connection.server_version* is based on a C API call (**:cfunc:'isc_database_info()'**) that existed long before the introduction of the Services API. Some programs written before the advent of Firebird test the version number in the return value of **:cfunc:'isc_database_info()'**, and refuse to work if it indicates that the server is too old. Since the first stable version of Firebird was labeled *1.0*, this pre-Firebird version testing scheme incorrectly concludes that (e.g.) Firebird 1.0 is older than Interbase 5.0.

Firebird addresses this problem by making **:cfunc:'isc_database_info()'** return a "pseudo-InterBase" version number, whereas the Services API returns the true Firebird version, as shown:

```
import firebirdsql
con = firebirdsql.connect(dsn='localhost:C:/temp/test.db', user='sysdba',␣
↪password='masterkey')
print 'Interbase-compatible version string:', con.server_version
```

```
import firebirdsql.services
svcCon = firebirdsql.services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
print 'Actual Firebird version string:     ', svcCon.getServerVersion()
```

Output (on Firebird 1.5.0/Win32):

```
Interbase-compatible version string: WI-V6.3.0.4290 Firebird 1.5
Actual Firebird version string:      WI-V1.5.0.4290 Firebird 1.5
```

`Connection.`**`getArchitecture`**`()`

Returns platform information for the server, including hardware architecture and operating system family.

```
from firebirdsql import services
con = services.connect(host='localhost', user='sysdba', password='masterkey')

print con.getArchitecture()
```

Output (on Firebird 1.5.0/Windows 2000):

```
Firebird/x86/Windows NT
```

Unfortunately, the architecture string is almost useless because its format is irregular and sometimes outright idiotic, as with Firebird 1.5.0 running on x86 Linux:

```
Firebird/linux Intel
```

Magically, Linux becomes a hardware architecture, the ASCII store decides to hold a 31.92% off sale, and Intel grabs an unfilled niche in the operating system market.

`Connection.`**`getHomeDir`**`()`

Returns the equivalent of the *RootDirectory* setting from `firebird.conf`:

```
from firebirdsql import services
con = services.connect(host='localhost', user='sysdba', password='masterkey')

print con.getHomeDir()
```

Output (on a particular Firebird 1.5.0/Windows 2000 installation):

```
C:\dev\db\firebird150\
```

Output (on a particular Firebird 1.5.0/Linux installation):

```
/opt/firebird/
```

`Connection.`**`getSecurityDatabasePath`**`()`

Returns the location of the server's core security database, which contains user definitions and such. Interbase® and Firebird 1.0 named this database `isc4.gdb`, while in Firebird 1.5 it's renamed to `security.fdb` and to `security2.fdb` in Firebird 2.0 and later.

```
from firebirdsql import services
con = services.connect(host='localhost', user='sysdba', password='masterkey')

print con.getSecurityDatabasePath()
```

Output (on a particular Firebird 1.5.0/Windows 2000 installation):

```
C:\dev\db\firebird150\security.fdb
```

Output (on a particular Firebird 1.5.0/Linux installation):

```
/opt/firebird/security.fdb
```

Connection.**getLockFileDir**()

The database engine uses a lock file to coordinate interprocess communication; *getLockFileDir()* returns the directory in which that file resides:

```
from firebirdsql import services
con = services.connect(host='localhost', user='sysdba', password='masterkey')

print con.getLockFileDir()
```

Output (on a particular Firebird 1.5.0/Windows 2000 installation):

```
C:\dev\db\firebird150\
```

Output (on a particular Firebird 1.5.0/Linux installation):

```
/opt/firebird/
```

Connection.**getCapabilityMask**()

The Services API offers "a bitmask representing the capabilities currently enabled on the server", but the only availabledocumentation for this bitmask suggests that it is "reserved for future implementation". firebirdsql exposes this bitmask as a Python *int* returned from the *getCapabilityMask()* method.

Connection.**getMessageFileDir**()

To support internationalized error messages/prompts, the database engine stores its messages in a file named `interbase.msg` (Interbase® and Firebird 1.0) or `firebird.msg` (Firebird 1.5 and later). The directory in which this file resides can be determined with the *getMessageFileDir()* method.

```
from firebirdsql import services
con = services.connect(host='localhost', user='sysdba', password='masterkey')

print con.getMessageFileDir()
```

Output (on a particular Firebird 1.5.0/Windows 2000 installation):

```
C:\dev\db\firebird150\
```

Output (on a particular Firebird 1.5.0/Linux installation):

```
/opt/firebird/
```

Connection.**getConnectionCount**()

Returns the number of active connections to databases managed by the server. This count only includes *database* connections (such as open instances of *firebirdsql.Connection*), not *services manager* connections (such as open instances of *firebirdsql.services.Connection*).

```
import firebirdsql, firebirdsql.services
svcCon = firebirdsql.services.connect(host='localhost', user='sysdba', password=
↪'masterkey')

print 'A:', svcCon.getConnectionCount()
```

```
con1 = firebirdsql.connect(dsn='localhost:C:/temp/test.db', user='sysdba',␣
↪password='masterkey')
print 'B:', svcCon.getConnectionCount()

con2 = firebirdsql.connect(dsn='localhost:C:/temp/test.db', user='sysdba',␣
↪password='masterkey')
print 'C:', svcCon.getConnectionCount()

con1.close()
print 'D:', svcCon.getConnectionCount()

con2.close()
print 'E:', svcCon.getConnectionCount()
```

On an otherwise inactive server, the example program generates the following output:

```
A: 0
B: 1
C: 2
D: 1
E: 0
```

Connection.**getAttachedDatabaseNames**()

Returns a list of the names of all databases to which the server is maintaining at least one connection. The database names are not guaranteed to be in any particular order.

```
import firebirdsql, firebirdsql.services
svcCon = firebirdsql.services.connect(host='localhost', user='sysdba', password=
↪'masterkey')

print 'A:', svcCon.getAttachedDatabaseNames()

con1 = firebirdsql.connect(dsn='localhost:C:/temp/test.db', user='sysdba',␣
↪password='masterkey')
print 'B:', svcCon.getAttachedDatabaseNames()

con2 = firebirdsql.connect(dsn='localhost:C:/temp/test2.db', user='sysdba',␣
↪password='masterkey')
print 'C:', svcCon.getAttachedDatabaseNames()

con3 = firebirdsql.connect(dsn='localhost:C:/temp/test2.db', user='sysdba',␣
↪password='masterkey')
print 'D:', svcCon.getAttachedDatabaseNames()

con1.close()
print 'E:', svcCon.getAttachedDatabaseNames()

con2.close()
print 'F:', svcCon.getAttachedDatabaseNames()

con3.close()
print 'G:', svcCon.getAttachedDatabaseNames()
```

On an otherwise inactive server, the example program generates the following output:

```
A: []
B: ['C:\\TEMP\\TEST.DB']
```

```
C: ['C:\\TEMP\\TEST2.DB', 'C:\\TEMP\\TEST.DB']
D: ['C:\\TEMP\\TEST2.DB', 'C:\\TEMP\\TEST.DB']
E: ['C:\\TEMP\\TEST2.DB']
F: ['C:\\TEMP\\TEST2.DB']
G: []
```

`Connection.`**`getLog`**`()`

Returns the contents of the server's log file (named `interbase.log` by Interbase® and Firebird 1.0; `firebird.log` by Firebird 1.5 and later):

```
from firebirdsql import services
con = services.connect(host='localhost', user='sysdba', password='masterkey')

print con.getLog()
```

Output (on a particular Firebird 1.5.0/Windows 2000 installation):

```
WEASEL (Client) Thu Jun 03 12:01:35 2004
  INET/inet_error: send errno = 10054

WEASEL (Client) Sun Jun 06 19:21:17 2004
  INET/inet_error: connect errno = 10061
```

Database Statistics

`Connection.`**`getStatistics`**`(`*database*, *showOnlyDatabaseLogPages=0...*`)`

Returns a string containing a printout in the same format as the output of the **`gstat`** command-line utility. This method has one required parameter, the location of the database on which to compute statistics, and five optional boolean parameters for controlling the domain of the statistics.

Map of gstat paremeters to getStatistics options

gstat command-line option	*getStatistics* boolean parameter
-header	showOnlyDatabaseHeaderPages
-log	showOnlyDatabaseLogPages
-data	showUserDataPages
-index	showUserIndexPages
-system	showSystemTablesAndIndexes

The following program presents several *getStatistics* calls and their *gstat*-command-line equivalents. In this context, output is considered "equivalent" even if their are some whitespace differences. When collecting textual output from the Services API, firebirdsql terminates lines with *n* regardless of the platform's convention; *gstat* is platform-sensitive.

```
from firebirdsql import services
con = services.connect(user='sysdba', password='masterkey')

# Equivalent to 'gstat -u sysdba -p masterkey C:/temp/test.db':
print con.getStatistics('C:/temp/test.db')

# Equivalent to 'gstat -u sysdba -p masterkey -header C:/temp/test.db':
print con.getStatistics('C:/temp/test.db', showOnlyDatabaseHeaderPages=True)

# Equivalent to 'gstat -u sysdba -p masterkey -log C:/temp/test.db':
print con.getStatistics('C:/temp/test.db', showOnlyDatabaseLogPages=True)
```

```
# Equivalent to 'gstat -u sysdba -p masterkey -data -index -system C:/temp/test.db
↪':
print con.getStatistics('C:/temp/test.db',
    showUserDataPages=True,
    showUserIndexPages=True,
    showSystemTablesAndIndexes=True
  )
```

The output of the example program is not shown here because it is quite long.

Backup and Restoration

pyfirebirdsql offers convenient programmatic control over database backup and restoration via the *backup* and *restore* methods.

At the time of this writing, released versions of Firebird/Interbase® do not implement incremental backup, so we can simplistically define *backup* as the process of generating and storing an archived replica of a live database, and *restoration* as the inverse. The backup/restoration process exposes numerous parameters, which are properly documented in Firebird Documentation to **gbak**. The pyfirebirdsql API to these parameters is presented with minimal documentation in the sample code below.

`Connection.`**`backup`**(*sourceDatabase*, *destFilenames*, *destFileSizes=()*, *<options>*)

Creates a backup file from database content.

Simple Form

The simplest form of *backup* creates a single backup file that contains everything in the database. Although the extension *'.fbk'* is conventional, it is not required.

```
from firebirdsql import services
con = services.connect(user='sysdba', password='masterkey')

backupLog = con.backup('C:/temp/test.db', 'C:/temp/test_backup.fbk')
print backupLog
```

In the example, *backupLog* is a string containing a *gbak*-style log of the backup process. It is too long to reproduce here.

Although the return value of the *backup* method is a freeform log string, *backup* will raise an exception if there is an error. For example:

```
from firebirdsql import services
con = services.connect(user='sysdba', password='masterkey')

# Pass an invalid backup path to the engine:
backupLog = con.backup('C:/temp/test.db', 'BOGUS/PATH/test_backup.fbk')
print backupLog
```

```
Traceback (most recent call last):
  File "adv_services_backup_simplest_witherror.py", line 5, in ?
    backupLog = con.backup('C:/temp/test.db', 'BOGUS/PATH/test_backup.fbk')
  File "C:\code\projects\firebirdsql\Kinterbasdb-3.0\build\lib.win32-2.
↪3\firebirdsql\services.py", line 269, in backup
    return self._actAndReturnTextualResults(request)
  File "C:\code\projects\firebirdsql\Kinterbasdb-3.0\build\lib.win32-2.
↪3\firebirdsql\services.py", line 613, in _actAndReturnTextualResults
    self._act(requestBuffer)
```

```
  File "C:\code\projects\firebirdsql\Kinterbasdb-3.0\build\lib.win32-2.
↪3\firebirdsql\services.py", line 610, in _act
    return _ksrv.action_thin(self._C_conn, requestBuffer.render())
firebirdsql.OperationalError: (-902, '_kiservices could not perform the action:␣
↪cannot open backup file BOGUS/PATH/test_backup.fbk. ')
```

Multifile Form

The database engine has built-in support for splitting the backup into multiple files, which is useful for circumventing operating system file size limits or spreading the backup across multiple discs.

pyfirebirdsql exposes this facility via the *Connection.backup* parameters *destFilenames* and *destFileSizes*. *destFilenames* (the second positional parameter of *Connection.backup*) can be either a string (as in the example above, when creating the backup as a single file) or a sequence of strings naming each constituent file of the backup. If *destFilenames* is a string-sequence with length *N*, *destFileSizes* must be a sequence of integer file sizes (in bytes) with length *N-1*. The database engine will constrain the size of each backup constituent file named in *destFilenames[:-1]* to the corresponding size specified in *destFileSizes*; any remaining backup data will be placed in the file name by *destFilenames[-1]*.

Unfortunately, the database engine does not appear to expose any convenient means of calculating the total size of a database backup before its creation. The page size of the database and the number of pages in the database are available via `database_info()` calls: *database_info(firebirdsql.isc_info_page_size, 'i')* and *database_info(firebirdsql.isc_info_db_size_in_pages, 'i')*, respectively, but the size of the backup file is usually smaller than the size of the database.

There *should* be no harm in submitting too many constituent specifications; the engine will write an empty header record into the excess constituents. However, at the time of this writing, released versions of the database engine hang the backup task if more than 11 constituents are specified (that is, if *len(destFilenames) > 11*). pyfirebirdsql does not prevent the programmer from submitting more than 11 constituents, but it does issue a warning.

The following program directs the engine to split the backup of the database at `C:/temp/test.db` into `C:/temp/back01.fbk`, a file 4096 bytes in size, `C:/temp/back02.fbk`, a file 16384 bytes in size, and `C:/temp/back03.fbk`, a file containing the remainder of the backup data.

```
from firebirdsql import services
con = services.connect(user='sysdba', password='masterkey')

con.backup('C:/temp/test.db',
   ('C:/temp/back01.fbk', 'C:/temp/back02.fbk', 'C:/temp/back03.fbk'),
    destFileSizes=(4096, 16384)
  )
```

Extended Options

In addition to the three parameters documented previously (positional *sourceDatabase*, positional *destFilenames*, and keyword *destFileSizes*), the *Connection.backup* method accepts six boolean parameters that control aspects of the backup process and the backup file output format. These options are well documented so in this document we present only a table of equivalence between **gbak** options and names of the boolean keyword parameters:

gbak option	Parameter Name	Default Value
-T	transportable	True
-M	metadataOnly	False
-G	garbageCollect	True
-L	ignoreLimboTransactions	False
-IG	ignoreChecksums	False
-CO	convertExternalTablesToInternalTables	True

Connection.**restore**(*sourceFilenames*, *destFilenames*, *destFilePages=()*, <*options*>)

Restores database from backup file.

Simplest Form

The simplest form of *restore* creates a single-file database, regardless of whether the backup data were split across multiple files.

```
from firebirdsql import services
con = services.connect(user='sysdba', password='masterkey')

restoreLog = con.restore('C:/temp/test_backup.fbk', 'C:/temp/test_restored.db')
print restoreLog
```

In the example, *restoreLog* is a string containing a *gbak*-style log of the restoration process. It is too long to reproduce here.

Multifile Form

The database engine has built-in support for splitting the restored database into multiple files, which is useful for circumventing operating system file size limits or spreading the database across multiple discs.

pyfirebirdsql exposes this facility via the *Connection.restore* parameters *destFilenames* and *destFilePages*. *destFilenames* (the second positional argument of *Connection.restore*) can be either a string (as in the example above, when restoring to a single database file) or a sequence of strings naming each constituent file of the restored database. If *destFilenames* is a string-sequence with length *N*, *destFilePages* must be a sequence of integers with length *N-1*. The database engine will constrain the size of each database constituent file named in *destFilenames[:-1]* to the corresponding page count specified in *destFilePages*; any remaining database pages will be placed in the file name by *destFilenames[-1]*.

The following program directs the engine to restore the backup file at `C:/temp/test_backup.fbk` into a database with three constituent files: `C:/temp/test_restored01.db`, `C:/temp/test_restored02.db`, and `C:/temp/test_restored03.db`. The engine is instructed to place fifty user data pages in the first file, seventy in the second, and the remainder in the third file. In practice, the first database constituent file will be larger than *pageSize*destFilePages[0]*, because metadata pages must also be stored in the first constituent of a multifile database.

```
from firebirdsql import services
con = services.connect(user='sysdba', password='masterkey')

con.restore('C:/temp/test_backup.fbk',
    ('C:/temp/test_restored01.db', 'C:/temp/test_restored02.db', 'C:/temp/test_
↪restored03.db'),
    destFilePages=(50, 70),
    pageSize=1024,
    replace=True
  )
```

Extended Options

These options are well documented so in this document we present only a table of equivalence between the **gbak** options and the names of the keyword parameters to *Connection.restore*:

gbak option	Parameter Name	Default Value
-P	pageSize	[use server default]
-REP	replace	False
-O	commitAfterEachTable	False
-K	doNotRestoreShadows	False
-I	deactivateIndexes	False
-N	doNotEnforceConstraints	False
-USE	useAllPageSpace	False
-MO	accessModeReadOnly	False
-BU	cacheBuffers	[use server default]

Database Operating Modes, Sweeps, and Repair

`Connection.`**`sweep`**(*database*, *markOutdatedRecordsAsFreeSpace=1*)
Not yet documented.

`Connection.`**`setSweepInterval`**(*database*, *n*)
Not yet documented.

`Conenction.`**`setDefaultPageBuffers`**(*database*, *n*)
Not yet documented.

`Conenction.`**`setShouldReservePageSpace`**(*database*, *shouldReserve*)
Not yet documented.

`Conenction.`**`setWriteMode`**(*database*, *mode*)
Not yet documented.

`Conenction.`**`setAccessMode`**(*database*, *mode*)
Not yet documented.

`Conenction.`**`activateShadowFile`**(*database*)
Not yet documented.

`Conenction.`**`shutdown`**(*database*, *shutdownMethod*, *timeout*)
Not yet documented.

`Conenction.`**`bringOnline`**(*database*)
Not yet documented.

`Conenction.`**`getLimboTransactionIDs`**(*database*)
Not yet documented.

`Conenction.`**`commitLimboTransaction`**(*database*, *transactionID*)
Not yet documented.

`Conenction.`**`rollbackLimboTransaction`**(*database*, *transactionID*)
Not yet documented.

`Conenction.`**`repair`**(*database*, <*options*>)
Not yet documented.

User Maintenance

`Conenction.`**`getUsers`**(*username=None*)
By default, lists all users.

`Conenction.`**`addUser`**(*user*)

User An instance of `User` with *at least* its username and password attributes specified as non-empty values.

`Conenction.`**`modifyUser`**`(`*user*`)`
Changes user data.

User An instance of `User` with *at least* its username and password attributes specified as non-empty values.

`Conenction.`**`removeUser`**`(`*user*`)`
Accepts either an instance of services.User or a string username, and deletes the specified user.

`Conenction.`**`userExists`**`(`*user*`)`
Returns a boolean that indicates whether the specified user exists.

class `firebirdsql.services.`**`User`**
Not yet documented.

pyfirebirdsql Links

python firebird database adapters

- The Python Wiki >> Firebird
  - pyfirebirdsql
  - FDB

Python + Databases

- Python
- distutils Home Page
- distutils Installation Instructions for Generic Packages
- Python Database Topic Guide
- Python DB-API Spec 2.0
  - PEP 249 - Augmented Python DB-API Spec 2.0

Firebird

- Firebird
- IBPhoenix
  - IBPhoenix Tools Downloads Area
- Flamerobin - nice open source and cross platform GUI for Firebird

Help

- Firebird-Python list (Yahoo Group)
- Firebird-Support list (Yahoo Group)

- Python Database SIG mailing list

pyfirebirdsql Changelog

Version 0.6.5

- callproc() is implemented.
- document is prepared.

Version 0.6.6

- fix issue #28 InternalError after commit
- fix issue #30 Incorrect handling empty row for RowMapping

Version 0.7.0

- change parameter name 'explain_plan' in class PrepareStatement.__init__().
- issue #32 don't decode character set OCTETS
- issue #33 op_allocate_statement do not require a transaction handle.
- add event notification

Version 0.7.1

- fix fetchonemap()
- issue #35 add timeout parameter to conduit.wait()

Version 0.7.2

- issue #36 add Cur.itermap() method
- run tests in FB 1.5 (error treatment db_info(), trans_info())
- issue #37 truncate field name from system table in FB 1.5

Version 0.7.3

- issue #38 fix some problems (thanx sergyp)
- issue #39 AttributeError when accessing cursor.description
- issue #41 fix int parameter bug
- issue #42 Implemented Cursor.rowcount() (thanx jtasker)
- setup.py test command

Version 0.7.4

- refactoring
- call socket.close()

Version 0.8.0

- refactoring
- buf fix
- add 'role' parameter to connection
- fetch right striped string if CHAR type
- fetch string if BLOB subtype 1
- boolean type support (Firebird 3)
- connection keep only one transaction
- 'INSERT ... RETURNING ...' statement support

Version 0.8.1

- bug fix (send all packets)

Version 0.8.2

- support 32k bytes over execute() parameter.

Version 0.8.3

- refactoring
- add repair() method in Services
- add is_disconnect() method in Connection

Version 0.8.4

- fix release bug (add insufficient files)

Version 0.8.5

- add bringOnline(), shutdown() methods in Services

Version 0.8.6

- fix exception when fetch after insert. return None

Version 0.9.0

- support Firebird 3 (experimental)

Version 0.9.1

- Refactoring
- bugfixes
- Modify isolation level. Similar to fdb.
- Add Connection.set_autocommit() autocommit mode.

Version 0.9.2

- fix Binary() function
- return recordset as tuple not list

Version 0.9.3

- refactoring
- fix issue #50 alternative to crypt on windows

Version 0.9.4

- fix Cursor.rowcount.
- Cursor.callproc() return out parameters.
- Cursor.execute() return cursor instance itself.

Version 0.9.5

- Protocol version 11 support

Version 0.9.6

- support Firebird 3 (CORE-2897)

Version 0.9.7

- fix null indicator for Firebird 3
- PyCrypto support for Firebird 3

Version 0.9.8

- fix issue #58 wrong logic for handling lage BLOBs.
- update error messages.

Version 0.9.9

- refactoring
- fix issue #60

Version 0.9.10

- fix bug for non posix (windows) environment. issue #62

Version 0.9.11

- fix issue #60 (again)

Version 0.9.12

- Enable Srp authentication and disable Wireprotocol for Firebird 3
- fix a bug about srp authentication
- refactoring and flake8

Version 0.9.13

- PEP 479 issue #66

Version 1.0.0

- refactoring
- Add license file.
- Documents update.

pyfirebirdsql LICENSE

THIS SOFTWARE IS PROVIDED BY HAJIME NAKAGAMI ‘‘AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL HAJIME NAKAGAMI OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CHAPTER 2

Indices and tables

- genindex

Python Module Index

f

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

W