PEP 249 – Python Database API Specification v2.0

Author: Marc-André Lemburg <mal at lemburg.com>

Discussions-To: Db-SIG list
Status: Final
Type: Informational
Created: 12-Apr-1999
Post-History:
Replaces: 248

▶ **Table of Contents**

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.

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 Topic Guide.

This document describes the Python Database API Specification 2.0 and a set of common optional extensions. The previous version 1.0 version is still available as reference, in PEP 248. Package writers are encouraged to use this version of the specification as basis for new interfaces.

Module Interface

Constructors

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]

Globals

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 DB-API 1.0 level interface should be assumed.

threadsafety

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

threadsafety	Meaning
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.

Contents

Page Source (GitHub)

paramstyle

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

paramstyle	Meaning
`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`

Exceptions

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

Warning

Exception raised for important warnings like data truncations while inserting, etc. It must be a subclass of the Python `Exception` class [10] [11].

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 `Exception` class [10].

InterfaceError

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

DatabaseError

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

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.

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.

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.

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.

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.

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 [10] [11]:

```
Exception
|__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

Connection objects should respond to the following methods.

Connection methods

.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. Note that closing a connection without committing the changes first will cause an implicit rollback to be performed.

.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 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. Cursors created from the same connection are not isolated, *i.e.*, any changes done to the database by a cursor are immediately visible by the other cursors. Cursors created from different connections can or can not be isolated, depending on how the transaction support is implemented (see also the connection's .rollback() and .commit() methods).

Cursor Objects should respond to the following methods and attributes.

Cursor attributes

.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`

The first two items (`name` and `type_code`) are mandatory, the other five are optional and are set to `None` if no meaningful values can be provided.

This attribute will be `None` for operations that do not return rows or if the cursor has not had an operation invoked via the .execute*() 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 .execute*() produced (for DQL statements like `SELECT`) or affected (for DML statements like `UPDATE` or `INSERT`). [9]

The attribute is -1 in case no .execute*() has been performed on the cursor or the rowcount of the last operation is cannot be determined by the interface. [7]

> **Note**
>
> Future versions of the DB API specification could redefine the latter case to have the object return `None` instead of -1.

Cursor methods

.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 .fetch*() 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 deprecated: .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.

Use of this method for an operation which produces one or more result sets constitutes undefined behavior, and the implementation is permitted (but not required) to raise an exception when it detects that a result set has been created by an invocation of the operation.

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 .execute*() did not produce any result set or no call was issued yet.

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

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 .execute*() 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 .execute*() 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 .execute*() did not produce any result set or no call was issued yet.

.arraysize

This read/write attribute specifies the number of rows to fetch at a time with .fetchmany(). It defaults to 1 meaning to fetch a single row at a time.

Implementations must observe this value with respect to the .fetchmany() method, but are free to interact with the database a single row at a time. It may also be used in the implementation of .executemany().

.setinputsizes(*sizes*)

This can be used before a call to .execute*() 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 .execute*() 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. `LONG`s, `BLOB`s, 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 .execute*() 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 .execute*() 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 type
: This type object is used to describe columns in a database that are string-based (e.g. `CHAR`).

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

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

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

ROWID type
: 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 for Module Authors

- Date/time objects can be implemented as Python datetime module objects (available since Python 2.3, with a C API since 2.4) or using the mxDateTime

package (available for all Python versions since 1.5.2). They both provide all necessary constructors and methods at Python and C level.

- 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 Date(*time.localtime(ticks)[:3])

def TimeFromTicks(ticks):
    return Time(*time.localtime(ticks)[3:6])

def TimestampFromTicks(ticks):
    return Timestamp(*time.localtime(ticks)[:6])
```

- 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 C interface have a look at `Include/bufferobject.h` and `Objects/bufferobject.c` in the Python source distribution.

- 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 [10]:

```
class Error(Exception):
    pass
```

```
class Warning(Exception):
    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.

Optional DB API Extensions

During the lifetime of DB API 2.0, module authors have often extended their implementations beyond what is required by this DB API specification. To enhance compatibility and to provide a clean upgrade path to possible future versions of the specification, this section defines a set of common extensions to the core DB API 2.0 specification.

As with all DB API optional features, the database module authors are free to not implement these additional attributes and methods (using them will then result in an `AttributeError`) or to raise a NotSupportedError in case the availability can only be checked at run-time.

It has been proposed to make usage of these extensions optionally visible to the programmer by issuing Python warnings through the Python warning framework. To make this feature useful, the warning messages must be standardized in order to be able to mask them. These standard messages are referred to below as *Warning Message*.

Cursor.rownumber

This read-only attribute should provide the current 0-based index of the cursor in the result set or `None` if the index cannot be determined.

The index can be seen as index of the cursor in a sequence (the result set). The next fetch operation will fetch the row indexed by .rownumber in that sequence.

Warning Message: "DB-API extension cursor.rownumber used"

Connection.Error, Connection.ProgrammingError, etc.

All exception classes defined by the DB API standard should be exposed on the Connection objects as attributes (in addition to being available at module scope).

These attributes simplify error handling in multi-connection environments.

Warning Message: "DB-API extension connection.<exception> used"

Cursor.connection

This read-only attribute return a reference to the Connection object on which the cursor was created.

The attribute simplifies writing polymorph code in multi-connection environments.

Warning Message: "DB-API extension cursor.connection used"

Cursor.scroll(*value* [, *mode='relative'*])

Scroll the cursor in the result set to a new position according to *mode*.

If mode is `relative` (default), value is taken as offset to the current position in the result set, if set to `absolute`, value states an absolute target position.

An `IndexError` should be raised in case a scroll operation would leave the result set. In this case, the cursor position is left undefined (ideal would be to

not move the cursor at all).

> **Note**
>
> This method should use native scrollable cursors, if available, or revert to an emulation for forward-only scrollable cursors. The method may raise NotSupportedError to signal that a specific operation is not supported by the database (e.g. backward scrolling).

Warning Message: "DB-API extension cursor.scroll() used"

Cursor.messages

This is a Python list object to which the interface appends tuples (exception class, exception value) for all messages which the interfaces receives from the underlying database for this cursor.

The list is cleared by all standard cursor methods calls (prior to executing the call) except for the .fetch*() calls automatically to avoid excessive memory usage and can also be cleared by executing `del cursor.messages[:]`.

All error and warning messages generated by the database are placed into this list, so checking the list allows the user to verify correct operation of the method calls.

The aim of this attribute is to eliminate the need for a Warning exception which often causes problems (some warnings really only have informational character).

Warning Message: "DB-API extension cursor.messages used"

Connection.messages

Same as Cursor.messages except that the messages in the list are connection oriented.

The list is cleared automatically by all standard connection methods calls (prior to executing the call) to avoid excessive memory usage and can also be cleared by executing `del connection.messages[:]`.

Warning Message: "DB-API extension connection.messages used"

Cursor.next()

Return the next row from the currently executing SQL statement using the same semantics as .fetchone(). A `StopIteration` exception is raised when the result set is exhausted for Python versions 2.2 and later. Previous versions don't have the `StopIteration` exception and so the method should raise an `IndexError` instead.

Warning Message: "DB-API extension cursor.next() used"

Cursor.__iter__()

Return self to make cursors compatible to the iteration protocol [8].

Warning Message: "DB-API extension cursor.__iter__() used"

Cursor.lastrowid

This read-only attribute provides the rowid of the last modified row (most databases return a rowid only when a single `INSERT` operation is performed). If the operation does not set a rowid or if the database does not support rowids, this attribute should be set to `None`.

The semantics of `.lastrowid` are undefined in case the last executed statement modified more than one row, e.g. when using `INSERT` with `.executemany()`.

Warning Message: "DB-API extension cursor.lastrowid used"

Connection.autocommit

Attribute to query and set the autocommit mode of the connection.

Return `True` if the connection is operating in autocommit (non-transactional) mode. Return `False` if the connection is operating in manual commit (transactional) mode.

Setting the attribute to `True` or `False` adjusts the connection's mode accordingly.

Changing the setting from `True` to `False` (disabling autocommit) will have the database leave autocommit mode and start a new transaction. Changing from `False` to `True` (enabling autocommit) has database dependent semantics with respect to how pending transactions are handled. [12]

Deprecation notice: Even though several database modules implement both the read and write nature of this attribute, setting the autocommit mode by writing to the attribute is deprecated, since this may result in I/O and related exceptions, making it difficult to implement in an async context. [13]

Warning Message: "DB-API extension connection.autocommit used"

Optional Error Handling Extensions

The core DB API specification only introduces a set of exceptions which can be raised to report errors to the user. In some cases, exceptions may be too disruptive for the flow of a program or even render execution impossible.

For these cases and in order to simplify error handling when dealing with databases, database module authors may choose to implement user definable error handlers. This section describes a standard way of defining these error handlers.

Connection.errorhandler, Cursor.errorhandler

Read/write attribute which references an error handler to call in case an error condition is met.

The handler must be a Python callable taking the following arguments:

```
errorhandler(connection, cursor, errorclass, errorvalue)
```

where connection is a reference to the connection on which the cursor operates, cursor a reference to the cursor (or `None` in case the error does not apply to a cursor), *errorclass* is an error class which to instantiate using *errorvalue* as construction argument.

The standard error handler should add the error information to the appropriate `.messages` attribute (Connection.messages or Cursor.messages) and raise the exception defined by the given *errorclass* and *errorvalue* parameters.

If no `.errorhandler` is set (the attribute is `None`), the standard error handling scheme as outlined above, should be applied.

Warning Message: "DB-API extension .errorhandler used"

Cursors should inherit the `.errorhandler` setting from their connection objects at cursor creation time.

Optional Two-Phase Commit Extensions

Many databases have support for two-phase commit (TPC) which allows managing transactions across multiple database connections and other resources.

If a database backend provides support for two-phase commit and the database module author wishes to expose this support, the following API should be implemented. NotSupportedError should be raised, if the database backend support for two-phase commit can only be checked at run-time.

TPC Transaction IDs

As many databases follow the XA specification, transaction IDs are formed from three components:

- a format ID
- a global transaction ID
- a branch qualifier

For a particular global transaction, the first two components should be the same for all resources. Each resource in the global transaction should be assigned a different branch qualifier.

The various components must satisfy the following criteria:

- format ID: a non-negative 32-bit integer.
- global transaction ID and branch qualifier: byte strings no longer than 64 characters.

Transaction IDs are created with the .xid() Connection method:

.xid(***format_id*, *global_transaction_id*, *branch_qualifier***)

Returns a transaction ID object suitable for passing to the .tpc_*() methods of this connection.

If the database connection does not support TPC, a NotSupportedError is raised.

The type of the object returned by .xid() is not defined, but it must provide sequence behaviour, allowing access to the three components. A conforming database module could choose to represent transaction IDs with tuples rather than a custom object.

TPC Connection Methods

.tpc_begin(*xid*)

Begins a TPC transaction with the given transaction ID *xid*.

This method should be called outside of a transaction (*i.e.* nothing may have executed since the last .commit() or .rollback()).

Furthermore, it is an error to call .commit() or .rollback() within the TPC transaction. A ProgrammingError is raised, if the application calls .commit() or .rollback() during an active TPC transaction.

If the database connection does not support TPC, a NotSupportedError is raised.

.tpc_prepare()

Performs the first phase of a transaction started with .tpc_begin(). A ProgrammingError should be raised if this method outside of a TPC transaction.

After calling .tpc_prepare(), no statements can be executed until .tpc_commit() or .tpc_rollback() have been called.

.tpc_commit([*xid*])

When called with no arguments, .tpc_commit() commits a TPC transaction previously prepared with .tpc_prepare().

If .tpc_commit() is called prior to .tpc_prepare(), a single phase commit is performed. A transaction manager may choose to do this if only a single resource is participating in the global transaction.

When called with a transaction ID *xid*, the database commits the given transaction. If an invalid transaction ID is provided, a ProgrammingError will be raised. This form should be called outside of a transaction, and is intended for use in recovery.

On return, the TPC transaction is ended.

.tpc_rollback([*xid*])

When called with no arguments, .tpc_rollback() rolls back a TPC transaction. It may be called before or after .tpc_prepare().

When called with a transaction ID *xid*, it rolls back the given transaction. If an invalid transaction ID is provided, a ProgrammingError is raised. This form should be called outside of a transaction, and is intended for use in recovery.

On return, the TPC transaction is ended.

.tpc_recover()

Returns a list of pending transaction IDs suitable for use with `.tpc_commit(xid)` or `.tpc_rollback(xid)`.

If the database does not support transaction recovery, it may return an empty list or raise NotSupportedError.

Frequently Asked Questions

The database SIG often sees reoccurring questions about the DB API specification. This section covers some of the issues people sometimes have with the specification.

Question:

How can I construct a dictionary out of the tuples returned by .fetch*():

Answer:

There are several existing tools available which provide helpers for this task. Most of them use the approach of using the column names defined in the cursor attribute .description as basis for the keys in the row dictionary.

Note that the reason for not extending the DB API specification to also support dictionary return values for the .fetch*() methods is that this approach has several drawbacks:

- Some databases don't support case-sensitive column names or auto-convert them to all lowercase or all uppercase characters.

- Columns in the result set which are generated by the query (e.g. using SQL functions) don't map to table column names and databases usually generate names for these columns in a very database specific way.

As a result, accessing the columns through dictionary keys varies between databases and makes writing portable code impossible.

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, threadsafety, 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.

Post-publishing additions to the DB API 2.0 specification:

- Additional optional DB API extensions to the set of core functionality were specified.

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 which should be addressed in future versions:

- Define a useful return value for .nextset() for the case where a new result set is available.
- Integrate the decimal module `Decimal` object for use as loss-less monetary and decimal interchange format.

Footnotes

[1]

As a guideline the connection constructor parameters should be implemented as keyword parameters for more intuitive use and follow this order of parameters:

Parameter	Meaning
`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$')
```

Also see [13] regarding planned future additions to this list.

[2]

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

[3] *(1, 2, 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 .fetch*() methods.

[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.

[7]

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

[8]

Implementation Note: Python C extensions will have to implement the `tp_iter` slot on the cursor object instead of the `.__iter__()` method.

[9]

The term *number of affected rows* generally refers to the number of rows deleted, updated or inserted by the last statement run on the database cursor. Most databases will return the total number of rows that were found by the

corresponding `WHERE` clause of the statement. Some databases use a different interpretation for `UPDATE`s and only return the number of rows that were changed by the `UPDATE`, even though the `WHERE` clause of the statement may have found more matching rows. Database module authors should try to implement the more common interpretation of returning the total number of rows found by the `WHERE` clause, or clearly document a different interpretation of the `.rowcount` attribute.

[10] *(1, 2, 3, 4)*

In Python 2 and earlier versions of this PEP, `StandardError` was used as the base class for all DB-API exceptions. Since `StandardError` was removed in Python 3, database modules targeting Python 3 should use `Exception` as base class instead. The PEP was updated to use `Exception` throughout the text, to avoid confusion. The change should not affect existing modules or uses of those modules, since all DB-API error exception classes are still rooted at the `Error` or `Warning` classes.

[11] *(1, 2)*

In a future revision of the DB-API, the base class for `Warning` will likely change to the builtin `Warning` class. At the time of writing of the DB-API 2.0 in 1999, the warning framework in Python did not yet exist.

[12]

Many database modules implementing the autocommit attribute will automatically commit any pending transaction and then enter autocommit mode. It is generally recommended to explicitly .commit() or .rollback() transactions prior to changing the autocommit setting, since this is portable across database modules.

[13] *(1, 2)*

In a future revision of the DB-API, we are going to introduce a new method `.setautocommit(value)`, which will allow setting the autocommit mode, and make `.autocommit` a read-only attribute. Additionally, we are considering to add a new standard keyword parameter `autocommit` to the Connection constructor. Modules authors are encouraged to add these changes in preparation for this change.

Acknowledgements

Many thanks go to Andrew Kuchling who converted the Python Database API Specification 2.0 from the original HTML format into the PEP format in 2001.

Many thanks to James Henstridge for leading the discussion which led to the standardization of the two-phase commit API extensions in 2008.

Many thanks to Daniele Varrazzo for converting the specification from text PEP format to ReST PEP format, which allows linking to various parts in 2012.

Copyright

Source: https://github.com/python/peps/blob/main/peps/pep-0249.rst

Last modified: 2023-09-09 17:39:29 GMT