>>> import psycopg2

# Connect to an existing database

>>> conn = psycopg2.connect("dbname=test user=postgres")

# Open a cursor to perform database operations

>>> cur = conn.cursor()

# Execute a command: this creates a new table

>>> cur.execute("CREATE TABLE test (id serial PRIMARY KEY, num integer, data varchar);")

# Pass data to fill a query placeholders and let Psycopg perform

# the correct conversion (no more SQL injections!)

>>> cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)",

... (100, "abc'def"))

# Query the database and obtain data as Python objects

>>> cur.execute("SELECT \* FROM test;")

>>> cur.fetchone()

(1, 100, "abc'def")

# Make the changes to the database persistent

>>> conn.commit()

# Close communication with the database

>>> cur.close()

>>> conn.close()

PEP 249 -- Python Database API Specification v2.0

**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.

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]

psycopg2.**connect**(dsn=None, connection\_factory=None, cursor\_factory=None, async=False, \*\*kwargs)

Create a new database session and return a new connection object.

The connection parameters can be specified as a libpq connection string using the dsn parameter:

*conn = psycopg2.connect("dbname=test user=postgres password=secret")*

or using a set of keyword arguments:

*conn = psycopg2.connect(dbname="test", user="postgres", password="secret")*

or using a mix of both: if the same parameter name is specified in both sources, the kwargs value will have precedence over the dsn value. Note that either the dsn or at least one connection-related keyword argument is required.

The basic connection parameters are:

* dbname – the database name (database is a deprecated alias)
* user – user name used to authenticate
* password – password used to authenticate
* host – database host address (defaults to UNIX socket if not provided)
* port – connection port number (defaults to 5432 if not provided)

Any other connection parameter supported by the client library/server can be passed either in the connection string or as a keyword. The PostgreSQL documentation contains the complete list of the supported parameters. Also note that the same parameters can be passed to the client library using environment variables.

Using the connection\_factory parameter a different class or connections factory can be specified. It should be a callable object taking a dsn string argument. See Connection and cursor factories for details. If a cursor\_factory is specified, the connection’s cursor\_factory is set to it. If you only need customized cursors you can use this parameter instead of subclassing a connection.

Using async=True an asynchronous connection will be created: see Asynchronous support to know about advantages and limitations. async\_ is a valid alias for the Python version where async is a keyword.

Changed in version 2.4.3: any keyword argument is passed to the connection. Previously only the basic parameters (plus sslmode) were supported as keywords.

**DB API extension:** The non-connection-related keyword parameters are Psycopg extensions to the DB API 2.0.

**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.

psycopg2.**apilevel**

String constant stating the supported DB API level. For psycopg2 is 2.0.

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.

psycopg2.**threadsafety**

For psycopg2 is 2, i.e. threads can share the module and the connection.

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

psycopg2.**paramstyle**

For psycopg2 is pyformat. See also Passing parameters to SQL queries.

psycopg2.**\_\_libpq\_version\_\_**

Integer constant reporting the version of the libpq library this psycopg2 module was compiled with (in the same format of server\_version). If this value is greater or equal than 90100 then you may query the version of the actually loaded library using the libpq\_version() function.

**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 StandardError (defined in the module exceptions).

psycopg2.**Warning**

It is a subclass of the Python StandardError.

**Connection Objects**

Connection objects should respond to the following methods.

class **connection**

Handles the connection to a PostgreSQL database instance. It encapsulates a database session.

Connections are created using the factory function connect().

Connections are thread safe and can be shared among many threads. See Thread and process safety for details.

**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.

**close()**

Close the connection now (rather than whenever del is executed). The connection will be unusable from this point forward; an InterfaceError 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 any pending change to be discarded as if a ROLLBACK was performed (unless a different isolation level has been selected: see set\_isolation\_level()).

Changed in version 2.2: previously an explicit ROLLBACK was issued by Psycopg on close(). The command could have been sent to the backend at an inappropriate time, so Psycopg currently relies on the backend to implicitly discard uncommitted changes. Some middleware are known to behave incorrectly though when the connection is closed during a transaction (when status is STATUS\_IN\_TRANSACTION), e.g. PgBouncer reports an unclean server and discards the connection. To avoid this problem you can ensure to terminate the transaction with a commit()/rollback() before closing.

.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.

**commit()**

By default, Psycopg opens a transaction before executing the first command: if commit() is not called, the effect of any data manipulation will be lost.

The connection can be also set in “autocommit” mode: no transaction is automatically open, commands have immediate effect. See Transactions control for details.

Changed in version 2.5: if the connection is used in a with statement, the method is automatically called if no exception is raised in the with block.

.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.

**rollback()**

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.

Changed in version 2.5: if the connection is used in a with statement, the method is automatically called if an exception is raised in the with block.

.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(name=None, cursor\_factory=None, scrollable=None, withhold=False)**

Return a new cursor object using the connection.

If name is specified, the returned cursor will be a server side cursor (also known as named cursor). Otherwise it will be a regular client side cursor. By default a named cursor is declared without SCROLL option and WITHOUT HOLD: set the argument or property scrollable to True/False and or withhold to True to change the declaration.

The name can be a string not valid as a PostgreSQL identifier: for example it may start with a digit and contain non-alphanumeric characters and quotes.

Changed in version 2.4: previously only valid PostgreSQL identifiers were accepted as cursor name.

Warning: It is unsafe to expose the name to an untrusted source, for instance you shouldn’t allow name to be read from a HTML form. Consider it as part of the query, not as a query parameter.

The cursor\_factory argument can be used to create non-standard cursors. The class returned must be a subclass of psycopg2.extensions.cursor. See Connection and cursor factories for details. A default factory for the connection can also be specified using the cursor\_factory attribute.

DB API extension: All the function arguments are Psycopg extensions to the DB API 2.0.

**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.

class **cursor**

Allows Python code to execute PostgreSQL command in a database session. Cursors are created by the connection.cursor() method: they are bound to the connection for the entire lifetime and all the commands are executed in the context of the database session wrapped by the connection.

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 the connections’ isolation level. See also rollback() and commit() methods.

Cursors are not thread safe: a multithread application can create many cursors from the same connection and should use each cursor from a single thread. See Thread and process safety for details.

**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.

*description*

This read-only attribute is a sequence of 7-item sequences.

Each of these sequences is a named tuple (a regular tuple if collections.namedtuple() is not available) containing information describing one result column:

* 1. name: the name of the column returned.
  2. type\_code: the PostgreSQL OID of the column. You can use the pg\_type system table to get more informations about the type. This is the value used by Psycopg to decide what Python type use to represent the value. See also Type casting of SQL types into Python objects.
  3. display\_size: the actual length of the column in bytes. Obtaining this value is computationally intensive, so it is always None unless the PSYCOPG\_DISPLAY\_SIZE parameter is set at compile time. See also PQgetlength.
  4. internal\_size: the size in bytes of the column associated to this column on the server. Set to a negative value for variable-size types See also PQfsize.
  5. precision: total number of significant digits in columns of type NUMERIC. None for other types.
  6. scale: count of decimal digits in the fractional part in columns of type NUMERIC. None for other types.
  7. null\_ok: always None as not easy to retrieve from the libpq.

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\*() methods yet.

Changed in version 2.4: if possible, columns descriptions are named tuple instead of regular tuples.

.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.

*rowcount*

**Cursor methods**

.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]

参数为序列或映射，将与操作中的变量绑定。变量遵循数据库特定记号法。

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).

游标将保存操作指针。若相同的操作再次传入，游标可优化其行为。相同的操作再次执行时仅绑定不同的参数，这是最有效的算法。

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.

*execute(query, vars=None) #Commands execution methods*

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 either with positional (%s) or named (%(name)s) placeholders. See \*Passing parameters to SQL queries\*.

变量被指定为位置或命名占位符。

The method returns None. If a query was executed, the returned values can be retrieved using fetch\*() methods.

.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.

*executemany(query, vars\_list) #Commands execution methods*

Execute a database operation (query or command) against all parameter tuples or mappings found in the sequence vars\_list.

The function is mostly useful for commands that update the database: any result set returned by the query is discarded.

Parameters are bounded to the query using the same rules described in the execute() method.

Warning: In its current implementation this method is not faster than executing execute() in a loop. For better performance you can use the functions described in Fast execution helpers.

**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.