## **Database Connection with Python**

### Python’s DB API

A database connectivity interface allows an application to access data from a variety of DBMSs, using a specific driver for a specific DBMS and operating system. This means that the application can be written without depending on a specific DBMS or the operating system.  
  
Writing Python code to access databases is made easier by the presence of the Python DB API. Because it is a pythonic API, it fits nicely into existing Python code and allows Python programmers to easily store and retrieve data from databases. The Python DB API specifies a way to connect to databases and issue commands to them. This gives the advantage that there is a standard way to write code that deals with a database using connections, cursors and transactions. It also defines a standard exception hierarchy that modules must implement.  
  
The DB API specifies a number of different 'fetch' methods that a cursor should provide. They are fetchone, fetchmany and fetchall. They pretty much do what they say fetching one row from the result set, a group of rows or every row that your query will return in one step. Obviously the fetchall method should be avoided when you are likely to have very big result sets as it may take a long time to return any data.

### Database Modules

Modules are available for most of the popular relational databases, be they F/OSS or commercial.

Implementations are available for:

PostgreSQL (psycopg2, txpostgres, ...)

MySQL (mysql-python, PyMySQL, ...)

MS SQL Server (adodbapi, pymssql, mxODBC, pyodbc, ...)

Oracle (cx\_Oracle, mxODBC, pyodbc, ...)

and many more...

 Most DB API packages can be installed using typical pythonic methods:

$ easy\_insall psycopg2

$ pip install mysql-python

### 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.

**Globals**

DB API 2.0 implementations provide the following global values:

**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:

**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.

**paramstyle**

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

**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.

**Error:** Exception that is the base class of all other error exceptions.

**InterfaceError:** Exception raised for errors that are related to the database interface rather than the database itself.

**DatabaseError:** Exception raised for errors that are related to the database.

**DataError:** Exception raised for errors that are due to problems with the processed data like division by zero, numeric value out of range, etc.

**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.

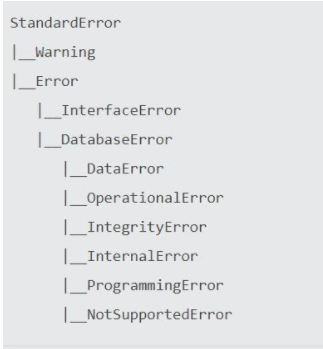
**IntegrityError:** Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails.

**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.

**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.

**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.

 This is the exception inheritance layout:



### Connection Objects

The DB API provides a constructor connect, which returns a Connection object. Connection objects should respond to the following methods.

**Connection methods**

.close(): Close the connection.

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.

 .rollback(): 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.

### 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 cannot be isolated, depending on how the transaction support is implemented.

**Cursor attributes**

.description: This read-only attribute is a sequence of 7-item sequences: name, type\_code, display\_size, internal\_size, precision, scale, null\_ok

Each of these sequences contains information describing one result column.

.rowcount: This read-only attribute specifies the number of rows that the last .execute() produced.

The attribute is -1 in case no .execute() has been performed on the cursor.

**Cursor methods**

.callproc( procname [, parameters ] ): Call a stored database procedure with the given name.

.close(): Close the cursor.

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.

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.

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.

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.

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.

 .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 array size 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 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.

.setinputsizes(sizes): This can be used before a call to .execute() to predefine memory areas for the operation's parameters.

 .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 .execute() method is invoked.

### 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).

 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.

TimeFromTicks(ticks): This function constructs an object holding a time value from the given ticks value.

TimestampFromTicks(ticks): This function constructs an object holding a time stamp value from the given ticks value.

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, BLOBs).

**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.

### Connecting to MS SQL Server from Python with PyODBC

To code along with this tutorial, you will need your own Python environment set up. There are several Python SQL drivers available. However, Microsoft places its testing efforts and its confidence in **PyODBC** driver. PyODBC is an open source Python module that makes accessing ODBC databases simple. PyODBC is community-supported software. Microsoft contributes to the pyODBC open-source community.

**Installation**

The easiest way to install pyodbc library, run the code below. This command automatically downloads and installs the library.

Typically, pyodbc is installed like any other Python package by running:

pip install pyodbc

### Connecting to MS SQL Server

Creating a re-usable functions below is the best practice, so that these can be used again and again with minimum effort.

The first line of the connection function is naming the function (create\_server\_connection) and naming the arguments that that function will take (driver, server, database, user, password).

The next line closes any existing connections so that the server doesn't become confused with multiple open connections.

Next Python try-except block to handle any potential errors. The first part tries to create a connection to the server using the pyodbc.connect method using the details specified by the user in the arguments. If this works, the function prints a happy little success message.

The except part of the block prints the error which MS SQL Server returns in the unfortunate circumstance that there is an error.

Finally, if the connection is successful, the function returns a connection object.

import pyodbc

def create\_server\_connection(driver, server, database, user, password):

conn = None

try:

conn = pyodbc.connect(DRIVER = driver, SERVER = server, DATABASE = database, UID = user, PWD = password)

print("MS SQL Server Database connection successful")

except Error as err:

print(f"Error: '{err}'")

return conn

driver = '{ODBC Driver 17 for SQL Server}'

server = 'localhost'

database = 'master'

user = 'sa'

password = 'xxx'

conn = create\_server\_connection(driver, server, database, user, password)

# MS SQL Server Database connection successful

### Creating a New Database

Now after establishing a connection, the next step is to create a new database on the server. This function takes two arguments, connection (a connection object) and query (a SQL query). It executes the query in the server via the connection.

def create\_database(conn, create\_database\_query):

csr = conn.cursor()

try:

csr.execute(create\_database\_query)

print("Database created successfully")

except Error as err:

print(f"Error: '{err}'")

conn.autocommit = True

create\_database\_query = 'CREATE DATABASE TestDB'

create\_database(conn, create\_database\_query)

# Database created successfully

### Creating a Query Execution Function

The final function is to create a query execution function. This is going to take SQL queries, stored in Python as strings, and pass them to the cursor.execute() method to execute them on the server.

def execute\_query(conn, query):

csr = conn.cursor()

try:

csr.execute(query)

conn.commit()

print("Query successful")

except Error as err:

print(f"Error: '{err}'")

query ='USE TestDB'

execute\_query(conn, query)

# Query successful

query = 'CREATE TABLE TestA ( \

ID int IDENTITY(1,1) NOT NULL, \

FirstName varchar(255) NOT NULL, \

LastName varchar(255) NOT NULL, \

PRIMARY KEY(ID))'

execute\_query(conn, query)

# Query successful

### Inserting Data

To insert data, pass the insert SQL to Cursor execute(), along with any parameters necessary:

csr = conn.cursor()

csr.execute("INSERT TestA (FirstName, LastName) VALUES ('Aaaa', 'Bbbb')")

conn.commit()

or, parameterized:

csr.execute("INSERT TestA (FirstName, LastName) VALUES (?, ?)", 'Cccc', 'Dddd')

conn.commit()

Note the calls to conn.commit(). You must call commit (or set autocommit to True on the connection) otherwise your changes will be lost!

### Updating and Deleting

Updating and deleting work the same way, pass the SQL to execute.

However, you often want to know how many records were affected when updating and deleting, in which case you can use the Cursor rowcount attribute:

csr.execute("delete from TestA where FirstName = ?", 'Aaaa')

conn.commit()

print(csr.rowcount, 'row(s) deleted')

Since execute() always returns the cursor, you will sometimes see code like this (notice .rowcount on the end).

deleted = csr.execute("delete from TestA where FirstName = 'Cccc'").rowcount

conn.commit()

print(deleted, 'row(s) deleted')

Note the calls to conn.commit(). You must call commit (or set autocommit to True on the connection) otherwise your changes will be lost!

### Reading Data

All SQL statements are executed using the Cursor execute() function. If the statement returns rows, such as a select statement, you can retrieve them using the Cursor fetch functions - fetchone(), fetchall(), fetchmany(). If there are no rows, fetchone() will return None, whereas fetchall() and fetchmany() will both return empty lists.

csr = conn.cursor()

csr.execute('select FirstName, LastName from TestA')

row = csr.fetchone()

if row:

print(row)

Row objects are similar to tuples, but they also allow access to columns by name:

csr.execute('select FirstName, LastName from TestA')

row = csr.fetchone()

print('LastName:', row[1]) # access by column index (zero-based)

print('LastName:', row.LastName) # access by name

The fetchone() function returns None when all rows have been retrieved.

csr.execute('select FirstName, LastName from TestA')

while True:

row = csr.fetchone()

if not row:

break

print('FirstName:', row.FirstName)

The fetchone() function returns all remaining rows in a list. Bear in mind those rows will all be stored in memory so if there a lot of rows, you may run out of memory.

If there are no rows, an empty list is returned.

csr.execute('select FirstName, LastName from TestA')

rows = csr.fetchall()

for row in rows:

print(row.FirstName, row.LastName)

If you are going to process the rows one at a time, you can use the cursor itself as an iterator:

csr.execute('select FirstName, LastName from TestA')

for row in csr:

print(row.FirstName, row.LastName)

#or just:

for row in csr.execute('select FirstName, LastName from TestA'):

print(row.FirstName, row.LastName)

Parameters

ODBC supports parameters using a question mark as a place holder in the SQL.

You provide the values for the question marks by passing them after the SQL:

csr.execute("""select FirstName, LastName from TestA

where FirstName = ?""", 'Aaaa')

row = csr.fetchone()

if row:

print(row)

This is safer than putting the values into the string because the parameters are passed to the database separately, protecting against SQL injection attacks.

It is also be more efficient if you execute the same SQL repeatedly with different parameters. The SQL will be "prepared" only once.

(pyodbc keeps only the last prepared statement, so if you switch between statements, each will be prepared multiple times.)

The Python DB API specifies that parameters should be passed as a sequence, so this is also supported by pyodbc:

csr.execute(""" select FirstName, LastName from TestA

where FirstName = ? and LastName = ? """, ['Aaaa', 'Bbbb'])

row = csr.fetchone()

if row:

print(row)

To simplify the process, a function def read\_query function can be created.

def read\_query(conn, query):

csr = conn.cursor()

results = None

try:

csr.execute(query)

results = csr.fetchall()

# Close and delete cursor

csr.close()

del csr

return results

except Error as err:

print(f"Error: '{err}'")

query = 'select FirstName, LastName from TestA'

execute\_query(conn, query)

results = read\_query(conn, query)

print(results)

conn.close()

import pandas as pd

from\_db = []

for result in results:

result = list(result)

from\_db.append(result)

columns = ['FirstName', 'LastName']

df = pd.DataFrame(from\_db, columns=columns)

display(df)

### Tips and Tricks

**Quotes**

Since single quotes are valid in SQL, use double quotes to surround your SQL:

deleted = csr.execute("delete from TestA where FirstName = 'Aaaa'").rowcount

It's also worthwhile considering using 'raw' strings for your SQL to avoid any inadvertent escaping (unless you really do want to specify control characters):

csr.execute("delete from TestA where FirstName like '%bad\name%'") # Python will convert \n to 'new line'!

csr.execute(r"delete from TestA where name like '%bad\name%'") # no escaping

**Naming Columns**

Some databases (e.g. SQL Server) do not generate column names for calculated fields, e.g. COUNT(\*).

In that case you can either access the column by its index, or use an alias on the column (i.e. use the "as" keyword).

row = csr.execute("select count(\*) as user\_count from TestA").fetchone()

print('%s users' % row.user\_count)

### Formatting Long SQL Statements

Long SQL statements are best encapsulated using the triple-quote string format. Doing so does create a string with lots of blank space on the left, but whitespace should be ignored by database SQL engines. If you still want to remove the blank space on the left, you can use the dedent() function in the built-in textwrap module. For example:

import textwrap

sql = textwrap.dedent("""select FirstName, LastName from TestA

where FirstName = 'Aaaa' and LastName = ?

""")

rows = csr.execute(sql, 'Bbbb').fetchall()

**fetchval**

If you are selecting a single value you can use the fetchval convenience method.

If the statement generates a row, it returns the value of the first column of the first row. If there are no rows, None is returned:

maxid = csr.execute("select max(ID) from TestA").fetchval()

Most databases support COALESCE or ISNULL which can be used to convert NULL to a hardcoded value, but note that this will not cause a row to be returned if the SQL returns no rows.

That is, COALESCE is great with aggregate functions like max or count, but fetchval is better when attempting to retrieve the value from a particular row:

csr.execute("select coalesce(max(ID), 0) from TestA where FirstName = 'Eeee'").fetchone()[0]

csr.execute("select coalesce(count(\*), 0) from TestA where FirstName = 'Eeee'").fetchone()[0]

However, fetchval is a better choice if the statement can return an empty set:

# Careful!

csr.execute("""select FirstName from TestA where FirstName = 'Eeee'

""").fetchone()[0]

# Preferred

csr.execute("""select FirstName from TestA where FirstName = 'Eeee'

""").fetchval()

 The first example will raise an exception if there are no rows.

The fetchone() call returns None. Python then attempts to apply [0] to the result (None[0]) which is not valid.

The fetchval method was created just for this situation - it will detect the fact that there are no rows and will return None.