mkleehammer / pyodbc

Code Issues 191 Pull requests 12 Actions Projects Wiki Security Insights

Getting started

Edit New Page Jump to bottom

Keith Erskine edited this page on 15 May 2020 · 7 revisions

Connect to a Database

Pass an ODBC connection string to the pyodbc connect() function which will return a Connection. Once you have a connection you can ask it for a Cursor. For example:

```
import pyodbc

# Specifying the ODBC driver, server name, database, etc. directly
cnxn = pyodbc.connect('DRIVER={ODBC Driver 17 for SQL Server};SERVER=localhost;DATABASE=te

# Using a DSN, but providing a password as well
cnxn = pyodbc.connect('DSN=test;PWD=password')

# Create a cursor from the connection
cursor = cnxn.cursor()
```

There are lots of options when connecting, so see the connect() function and the Connecting to Databases section for more details.

Make sure you set the encoding or decoding settings needed for your database and the version of Python you are using:

```
# This is just an example that works for PostgreSQL and MySQL, with Python 2.7.
cnxn.setdecoding(pyodbc.SQL_CHAR, encoding='utf-8')
cnxn.setdecoding(pyodbc.SQL_WCHAR, encoding='utf-8')
cnxn.setencoding(encoding='utf-8')
```

Selecting Some Data

Select Basics

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.

```
cursor.execute("select user_id, user_name from users")
row = cursor.fetchone()
if row:
    print(row)
```

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

```
cursor.execute("select user_id, user_name from users")
row = cursor.fetchone()
print('name:', row[1])          # access by column index (zero-based)
print('name:', row.user_name)   # access by name
```

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

```
while True:
    row = cursor.fetchone()
    if not row:
        break
    print('id:', row.user_id)
```

The fetchall() 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.

```
cursor.execute("select user_id, user_name from users")
rows = cursor.fetchall()
for row in rows:
    print(row.user_id, row.user_name)
```

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

```
cursor.execute("select user_id, user_name from users"):
for row in cursor:
    print(row.user_id, row.user_name)
```

or just:

```
for row in cursor.execute("select user_id, user_name from users"):
    print(row.user_id, row.user_name)
```

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:

```
cursor.execute("""
    select user_id, user_name
      from users
     where last_logon < ?
       and bill_overdue = ?
""", datetime.date(2001, 1, 1), 'y')
```

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:

```
cursor.execute("""
    select user_id, user_name
      from users
     where last_logon < ?
       and bill_overdue = ?
""", [datetime.date(2001, 1, 1), 'y'])
```

Inserting Data

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

```
cursor.execute("insert into products(id, name) values ('pyodbc', 'awesome library')")
cnxn.commit()
```

or, parameterized:

```
cursor.execute("insert into products(id, name) values (?, ?)", 'pyodbc', 'awesome library'
cnxn.commit()
```

Note the calls to cnxn.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:

```
cursor.execute("delete from products where id <> ?", 'pyodbc')
print(cursor.rowcount, 'products deleted')
cnxn.commit()
```

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

```
deleted = cursor.execute("delete from products where id <> 'pyodbc'").rowcount
cnxn.commit()
```

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

Tips and Tricks

Quotes

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

```
deleted = cursor.execute("delete from products where id <> 'pyodbc'").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):

```
cursor.execute("delete from products where name like '%bad\name%'")   # Python will conver
cursor.execute(r"delete from products 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 = cursor.execute("select count(*) as user_count from users").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 p.date_of_birth,
           p.email,
           a.city
    from person as p
    left outer join address as a on a.address_id = p.address_id
    where p.status = 'active'
      and p.name = ?
""")
rows = cursor.execute(sql, 'John Smith').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 = cursor.execute("select max(id) from users").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:

```
cursor.execute("select coalesce(max(id), 0) from users").fetchone()[0]
cursor.execute("select coalesce(count(*), 0) from users").fetchone()[0]
```

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

```
# Careful!
cursor.execute("""
    select create_timestamp
    from photos
    where user_id = 1
    order by create_timestamp desc
    limit 1
""").fetchone()[0]

# Preferred
cursor.execute("""
    select create_timestamp
    from photos
    where user = 1
    order by create_timestamp desc
    limit 1
""").fetchval()
```

The first example will raise an exception if there are no rows for user_id 1. 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.